Efficient Continuous Skyline Computation*

Michael Morse Jignesh M. Patel
University of Michigan
{mmorse, jignesh}@eecs.umich.edu

William I. Grosky
University of Michigan – Dearborn
wgrosky@umich.edu

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 sets is to produce a continuous and valid skyline *summary. To the best of our knowledge, this problem has not been addressed before. In this paper we introduce an operator called the* continuous time-interval skyline *operator for evaluating this computation. We also present a new algorithm called LookOut for evaluating the* continuous time-interval skyline *efficiently, and empirically demonstrate the scalability of this algorithm.*

1. Introduction

The skyline operator is an elegant summary method over multi-dimensional data sets [3]. Given a data set P containing data points p_1, p_2, ..., p_n, the skyline of P is the set of all p_i in P such that no p_j *dominates* p_i. All existing skyline algorithms assume that the data set is static, i.e. the data elements have no temporal element associated with it. 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 DBMS is to *continuously* compute a skyline for the data points that are valid at any given time. To the best of our knowledge, this problem has not been addressed before.

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 hotel prices that are valid for certain periods with fixed expiration times.

*This research was supported in part by the National Science Foundation under grant IIS-0414510.

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 a iterative algorithm based on currently known methods by at least an order of magnitude in most cases!*

2. Related Work

Borzsonyi et al. [2] introduced the $skyline$ operation in a database context and showed how B-trees and R-trees can be used to evaluate skyline queries. An algorithm to obtain the skyline based on a nearest neighbors approach was introduced by Kossmann et al. [3]. The branch and bound technique for skyline computation (BBS) was proposed by Papadias et al. in [6, 7], and is more efficient than previous skyline computation methods. Lin et al. [5] focus on computing the skyline against the most recent n of N elements in a data stream, which significantly differs from our continuous time-interval skyline operation. Their operator always has a constant number of elements to consider (n), and data elements are not associated with an explicit time-interval.

3. The LookOut Algorithm

The Model: Each data point in the data set has an interval of time during 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 expires, or b) A new data point j is introduced into the data set.

In the case of an expiration, the data set must be checked for new skyline points that previously may have 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 the 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.

Description: When a new data point arrives, $LookOut$ first stores the item in a spatial index. Each data element is also inserted into a binary heap that is ordered based 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, using the $isSkyline$ algorithm (described below). If so, the new point is added to the skyline and the 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, and if it has, it is deleted from the index. The skyline points themselves are maintained in a list. (The skyline points can also be stored in an index, but the skyline is small and the 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 can be quickly removed. The points that get removed from the skyline 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 any hole in the new skyline. Some, and possibly all, of the points found by $MINI$ may be dominated by other skyline points. Before adding them to the skyline, $LookOut$ tests if each point is in fact a new skyline point using the $isSkyline$ algorithm.

The $isSkyline$ algorithm uses a best-first search paradigm to test if a point P_{new} is a skyline point. The index nodes are inserted into a heap based on their distance from the origin. When expanding a node, $isSkyline$ discards any index node, n, whose lower left corner does not dominate P_{new}. If the upper right corner of the child node (which isn't empty) dominates P_{new}, the algorithm terminates and returns $false$. If the node is a leaf, the elements are compared against P_{new} for dominance. If any such element dominates P_{new}, the algorithm returns $false$. If the heap is ever empty, the algorithm returns $true$.

$MINI$ 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 works 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. If the point is dominated by the growing mini skyline, it is ignored; else, it is added to the mini skyline. If the upper right corner of any internal node is not dominated by P_{old}, then it is discarded. If the top of the heap is a leaf, its local skyline is added to the heap. 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.

4. Experimental Evaluation

In this section, we examine the performance of *LookOut* relative to a naïve method of executing the BBS algorithm to compute the skyline whenever a new data point arrives or a existing skyline point expires. This method is referred to as *BBS*, and can be considered the best alternative method for computing continuous skylines.

Following well established methodology set by previous research on skyline algorithms [5, 6], we present data that follows both the independent and anti-correlated 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 data sets, we use the skyline generator generously provided by the authors of [2]; using this, we created a number of data sets varying in cardinalities from 100K to 500K in two dimensions.

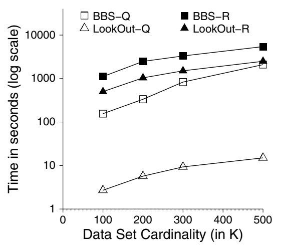
We have implemented both a quadtree [8] and a R*-tree [1] indexing method to use as underlying spatial indexing structures. To maintain consistency with the previous approach by Papadias et al. [6], and for ease of comparison, we set the R*-tree node size at 4KB. For the BBS implementation, we followed the algorithm described in [6], and added the local skyline optimizations described in [7].

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

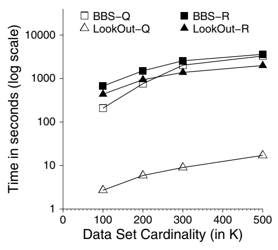
For all experiments, the data structures are entirely memory resident to mimic the application of continuous skylines to streaming applications where such main memory assumptions are common and often the preferred environment (for example [5] also assumes that there is enough main memory). In the naïve BBS 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.

Each point is assigned an arrival and expiration time with a random number generator as follows: we randomly pick an arrival time between 0 and 100K. Then we pick the departure time randomly between the arrival time and 100K. Results are then generated by running each data set from time 0 to 100,001 so that each data point will both enter and be deleted from the system.

In the first experiment we compare the performance of $LookOut$ and BBS. For this experiment, we consider the implementation of BBS and $LookOut$ using the R*-tree and the quadtree. For BBS we use the label BBS-R and BBS-Q for the R*-tree and the quadtree index implementations respectively. Similarly $LookOut$ - R and $LookOut$ - Q is done for the $LookOut$ method. The y axis in all figures uses a log scale to show the workload execution time.



(a)



(b)

Figure 1. (a) Anti.-corr. and (b) indep. with 2 dim., varying cardinality

The results for this experiment for anti-correlated and independent data sets are shown in Figure 1. From this figure, we observe that the execution time for $LookOut$ with a quadtree relative to iterative BBS is almost two orders of magnitude better. $LookOut$ with the R*-tree is about twice as fast as (iterative) BBS. The superior performance of $LookOut$ with respect to BBS 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 BBS does. There is a more marked improvement in $LookOut$ relative to the BBS algorithm with the quadtree than with the R*-tree because the insertions and deletions with the quadtree are very fast, and the quadtree has been shown to be superior to the R*-tree for managing point data [4]. The overhead of the R*-tree limits the amount of performance improvement that $LookOut$ can achieve.

We have also performed experiments evaluating the effect of increasing the data set dimensionality from 2 to 5. (In the interest of space, these figures are omitted in this presentation.) In these experiments, $LookOut$ with an R*-tree outperformed BBS with an R*-tree by an order of magnitude or more in the anti-correlated cases, and a factor of 2-3 better for the independent data set. $LookOut$ with the quadtree continues to be the most efficient method, but the performance advantage of the quadtree over that of the R*-tree becomes less remarkable as dimensionality increases.

From these figures we observe that the execution time for $LookOut$ is less than BBS with each respective data structure. With the quadtree, $LookOut$ is more than an order of magnitude faster than BBS with the quadtree. With the R*-tree, $LookOut$ is faster by a factor of 2 to 3. BBS with the quadtree also outperformed BBS with the R*-tree in all cases.

For the second experiment, we measure 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.) Table 1 presents data on the maximum and average processing delays for $LookOut$ for this experiment. These results indicate that $LookOut$ can process about 104,050 eps for the anti-correlated data set, and about 116,000 eps for the independent data set. (Note 1000/0.00961 = 104,058.)

Card-inality in K	Max Anti-Corr. Delay	Max Indep. Delay	Avg Anti-Corr Delay	Avg Indep. Delay
100	2.08	1.98	0.00961	0.00862
200	4.59	4.34	0.00940	0.00865
300	5.19	4.75	0.00953	0.00880
500	6.20	5.05	0.00983	0.00928

Table 1. Delays in ms for inserts and deletes, varying cardinality, with 2 dim. data.

5. Conclusions

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.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, pages 322–331, 1990.
[2] S. Borzsonyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, pages 421–430, 2001.
[3] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB*, pages 275–286, 2002.
[4] R. K. V. Kothuri, S. Ravada, and D. Abugov. Quadtree and R-tree Indexes in Oracle Spatial: A Comparison Using GIS Data. In *SIGMOD*, pages 546–557, 2002.
[5] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In *ICDE*, pages 502–513, 2005.
[6] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *SIGMOD*, pages 467–478, 2003.
[7] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive Skyline Computation in Database Systems. *TODS*, 30(1):41–82, March 2005.
[8] H. Samet. The Quadtree and Related Hierarchical Data Structures. *Computing Surveys*, 16(4):187–260, 1984.