Answering linear optimization queries with an approximate stream index

Gang Luo • Kun-Lung Wu • Philip S. Yu
IBM T.J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532, USA
email: {luog, klwu, psyu}@us.ibm.com

Received: October 26, 2007 / Revised: February 12, 2008 / Accepted: June 23, 2008

Abstract
We propose a SAO index to approximately answer arbitrary linear optimization queries in a sliding window of a data stream. It uses limited memory to maintain the most "important" tuples. At any time, for any linear optimization query, we can retrieve the approximate top-*K* tuples in the sliding window almost instantly. The larger the amount of available memory, the better the quality of the answers is. More importantly, for a given amount of memory, the quality of the answers can be further improved by dynamically allocating a larger portion of the memory to the outer layers of the SAO index.

Keywords — Indexing method, Query processing, Relational database, Stream processing, Linear optimization query

1. Introduction
Data stream applications are becoming popular [1, 3, 9, 35, 36, 37]. Many of such applications use various linear optimization queries [8, 24, 25] to retrieve the (approximate) top-*K* tuples that maximize or minimize the linearly weighted sums of certain attribute values. For example, in environmental epidemiological applications, various linear models that incorporate remotely sensed images, weather information, and demographic information are used to predict the outbreak of certain environmental epidemic diseases, like Hantavirus Pulmonary Syndrome [24]. In oil/gas exploration applications, linear models that incorporate drill sensor measurements and seismic information are used to guide the drilling direction [25]. In financial applications, linear models that incorporate personal credit history, income level, and employment history are used to evaluate credit risks for loan approvals [24].

In all the above applications, data continuously stream in (say, from satellites and sensors) at a rapid rate. Users frequently pose linear optimization queries and want answers back as soon as possible. Moreover, different individuals may pose queries that have divergent weights and *K*'s. This is because the "optimal" weights may vary from one location to another (in oil/gas exploration), the weights may be adjusted as the model is continually trained with historical data collected more recently (in environmental epidemiology and finance), and different users may have differing preferences.

In a read-mostly environment, Chang *et al*. [8] first proposed an onion index to speed up the evaluation of linear optimization queries against a large database relation. An onion index organizes all the tuples in the database relation into one or more convex layers, where each convex layer is a convex hull. For each $i \geq 1$, the $(i+1)$th convex layer is contained within the ith convex layer. For any linear optimization query, to find the top-*K* tuples, we need to search no more than all the vertices of the first *K* outer convex layers in the onion index.

However, due to the extremely high cost of computing precise convex hulls [28, 29], both the creation and the maintenance of the onion index are rather expensive. Moreover, an onion index requires lots of storage because it keeps track of all the tuples in a relation. In a streaming environment, tuples keep arriving rapidly while available memory is limited. Hence, it is impossible to maintain a precise onion index for a data stream, let alone using it to provide exact answers to linear optimization queries.

To address these problems, we propose a SAO (*S*tream *A*pproximate *O*nion-like structure) index for a data stream. The index provides high-quality, approximate answers to arbitrary linear optimization queries almost instantly. Our key observation is that the precise onion index typically contains a large number of convex layers, but most inner layers are not needed for answering linear optimization queries. Hence, the SAO index maintains only the first few outer convex layers. Moreover, each layer in the SAO index only keeps some of

the most "important" vertices rather than all the vertices. As a result, the amortized maintenance cost of a SAO index is rather small because the great majority of the incoming tuples, more than 95% in most cases, do not cause any changes to the index and are quickly discarded, even though individual inserts or deletes might have non-trivial costs.

A key challenge in designing a SAO index is: For a given amount of memory, how do we properly allocate it among the layers so that the quality of the answers can be maximized? To do that, a dynamic, error-minimizing storage allocation strategy is used so that a larger portion of the available memory tends to be allocated to the outer layers than to the inner layers. In this way, both storage and maintenance overheads of the SAO index are greatly reduced. More importantly, the errors introduced into the approximate answers are also minimized.

With limited memory and continually arriving tuples, there are intrinsic errors in any stream application. It is difficult to provide an upper bound on these errors for linear optimization queries because the amount of inaccuracies depends on the specific sequence of tuples in a stream. Similar to what was shown in Yi *et al*. [33], such errors can be substantial in a pathological case where the available memory is not sufficient to hold all the tuples within a sliding window of a stream and the sequence of arriving tuples happens to maximize the errors. However, in practice, the exact errors can be measured based on stream traces. As shown in the experiments conducted in this paper, the actual errors are relatively minor (often less than 1%) even if the SAO index holds only a tiny fraction (less than 0.1%) of the tuples in the sliding window. This is because, statistically, only few tuples cause errors. Moreover, the impact of any error, no matter how large it may be, disappears immediately once the tuple causing the error has moved out of the sliding window.

For some stream applications, the linear optimization queries are known in advance and the entire history, not just a sliding window, of the stream is considered. In this case, for each query, an in-memory materialized view can be maintained to continuously keep track of the top-K tuples. However, if there are many such queries, it may not be feasible to keep all these materialized views in memory and/or to maintain them in real time. As a consequence, the SAO index method is still needed under such circumstances.

We implemented the SAO index by modifying a widely-used Qhull package [5]. Our experimental results on both real and synthetic data sets show that the SAO index can handle high tuple arrival rates, be maintained efficiently in real time, and provide high-quality answers to linear optimization queries almost instantly.

A preliminary version of this paper appeared in ICDE'07 [34]. However, we have provided in the current paper with a significant amount of new technical materials. In Luo *et al*. [34], only the high level idea of the algorithm is described. It omits many important details, such as mathematical derivations, justification of decisions made in the design of the algorithm, and illustrations of how the algorithm works. In contrast, these critical details are elaborated in the current paper. Moreover, in Luo *et al*. [34], only a brief performance study is included with only two initial performance figures, whereas in the current paper, a comprehensive set of additional performance studies is provided including many sensitivity analyses.

The rest of the paper is organized as follows. Section 2 briefly reviews the traditional onion index. Section 3 describes our SAO index. Section 4 presents results from a prototype implementation of our techniques. We discuss related work in Section 5 and conclude in Section 6.

2. Review of the Traditional Onion Index

We briefly review the earlier onion index [8] for linear optimization queries against a large database relation. Suppose each tuple contains $n \geq 1$ numerical *feature* attributes and $m \geq 0$ other non-feature attributes. A top-K *linear optimization query* asks for the top-K tuples that maximize the following linear equation: $\max_{top\ K}\{\sum_{i=1}^{n} w_i a_i^j\}$, where $(a_1^j, a_2^j, ..., a_n^j)$ is the feature attribute vector of the jth tuple and $(w_1, w_2, ..., w_n)$ is the weighting vector of the query. Some w_i's may be zero. Here, $v_j = \sum_{i=1}^{n} w_i a_i^j$ is called the *linear combination value* of the jth tuple. Note that a linear optimization query may alternatively ask for the K minimal linear combination values. In this case, we can turn such a query into a maximization query by switching the signs of the weights. Without loss of generality, we focus on maximization queries in this paper.

A set of tuples S can be mapped to a set of points in an n-dimensional space according to their feature attribute vectors. For a top-K linear optimization query, the top-K tuples are those K tuples with the largest projection values along the query direction. Linear programming theory has the following theorem:

Theorem 1 [14]. Given a linear maximization criterion and a set of tuples S, the maximum linear combination value is achieved at one or more vertices of the convex hull of S.

Utilizing this property, the onion index in Chang *et al*. [8] organizes all the tuples into one or more convex layers. The first convex layer l_1 is the convex hull of all the tuples in S. The vertices of l_1 form a set $S_1 \subseteq S$. For each $i > 1$, the ith convex layer l_i is the convex hull of all the tuples in $S - \bigcup_{j=1}^{i-1} S_j$. The vertices of l_i form a set $S_i \subseteq S - \bigcup_{j=1}^{i-1} S_j$. It is easy to see that for each $i \geq 1$, l_{i+1} is contained within l_i. Figure 1 shows an example onion index in two-dimensional space.

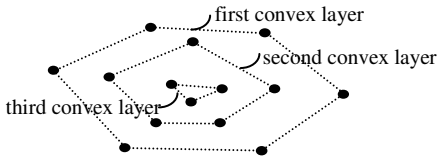


Figure 1. An onion index with three convex layers in two-dimensional space.

From Theorem 1, the maximum linear combination value at each l_i ($i \geq 1$) is larger than all the linear combination values from l_i's inner layers. Also, there may be multiple tuples on l_i whose linear combination values are larger than the maximum linear combination value of l_{i+1}. Hence, we have the following property:

Property 1: For any linear optimization query, suppose all the tuples are sorted in descending order of their linear combination values (v_j). The tuple that is ranked kth in the sorted list is called the kth *largest tuple*. Then the largest tuple is on l_1. The second largest tuple is on either l_1 or l_2. In general, for any $i \geq 1$, the ith largest tuple is on one of the first i outer convex layers.

Given a top-K linear optimization query, the search procedure of the onion index starts from l_1 and searches the convex layers one by one. On each convex layer, all its vertices are checked. Based on Property 1, the search procedure can find the top-K tuples by searching no more than the first K outer convex layers.

During a tuple insertion or deletion, one or more convex layers may need to be reconstructed in order to maintain the onion index. (The detailed onion index maintenance procedure is available in Chang *et al*. [8]. We do not review it here.) Both the creation and the maintenance of the onion index require computing convex hulls. This is expensive, as given N points in n-dimensional space, the worst-case computational complexity of constructing the convex hull is $O(N \ln N + N^{\lfloor n/2 \rfloor})$ [28].

3. SAO Index

The original onion index [8] keeps track of all the tuples, requiring lots of storage. Maintaining the original onion index is also computationally costly, making it difficult to meet the real-time requirement of data streams. Actually typical tuple arrival rates are often several orders of magnitude higher than the speed that the original onion index can be maintained.

To address these problems, we propose a SAO index for linear optimization queries against a data stream. Our key idea is to reduce both the index storage and maintenance overheads by keeping only a subset of the tuples in a data stream in the SAO index. We focus on the count-based sliding window model for data streams, with W denoting the sliding window size. That is, the tuples under consideration are the last W tuples that we have seen. Our techniques can be easily extended to the case of time-based sliding windows or the case that the entire history of the stream is considered.

Suppose the available memory can hold $M+1$ tuples. In the steady state, no more than M tuples are kept in the SAO index. That is, the storage budget is M tuples. In a transition period, $M+1$ tuples can be kept in the SAO index temporarily. Our techniques can be extended to the case where memory is measured in bytes. In general, a tuple contains both feature and non-feature attributes. We are interested in finding all the attributes of the top-K tuples. Hence, all the attributes of those tuples in the SAO index are kept in memory. Even if the

convex hull for feature attributes occupy only a small amount of space, the non-feature attributes may still dominate the storage requirement. For example, in the earlier-mentioned, environmental epidemiology application, each tuple has a large non-feature image attribute, which is also kept in memory. Note that the image cannot be stored on disk, even if we like to do so, because the tuple arrival rate can be too high for even the fastest disk to keep up with the rapidly arriving tuples. For example, satellite image transfer rate can easily become close to 1Gbps [6].

Our design principle is as follows. To provide high-quality answers to linear optimization queries, the SAO index carefully controls the number of tuples on each layer. It dynamically allocates proper amount of storage to individual layers so that a larger portion of the available memory tends to be allocated to the outer layers. As such, the quality of the answers can be maximized without increasing the storage requirement. In case of overflow, the SAO index keeps the most "important" tuples and throws away the less "important" ones. Moreover, to minimize the computation overhead, the creation and maintenance algorithms of the SAO index are optimized.

The rest of Section 3 is organized as follows. Section 3.1 provides some background on approximate answers. Section 3.2 describes the SAO index organization. Sections 3.3 and 3.4 discuss memory allocation strategies. Sections 3.5 and 3.6 show how to create and how to maintain the SAO index, respectively. Section 3.7 presents the query evaluation method. Section 3.8 addresses parallel processing.

3.1 A Little Background on Approximate Answers

Users submitting linear optimization queries against data streams generally must accept approximate answers. If $W \le M$, all W tuples in the sliding window can be kept in memory. Then for any linear optimization query, the exact answer can always be computed by checking the last W tuples. However, if $W > M$, which is common in practice, it is impossible to keep the last W tuples in memory. Then for any linear optimization query, the return of exact answers cannot always be guaranteed. The reason is similar to what has been shown in Yi *et al.* [33]: if all the tuples arrive in such an order that their linear combination values decrease monotonically, the memory cannot always hold those K "valid" tuples with the largest linear combination values. Consequently, users have to accept approximate answers.

In the rest of this paper, we focus on the case of $W > M$. In this case, it is impossible to keep the precise onion index in memory. Rather, we propose a SAO index, which can provide approximate answers to linear optimization queries almost instantaneously.

3.2 Index Organization

The SAO index is based on a key observation: An onion index typically contains a large number of convex layers, but most inner layers are not needed for answering the majority of linear optimization queries. For example, as mentioned in Section 2, to answer a top-K linear optimization query, at most the first K outer convex layers need to be searched. Hence, the SAO index keeps only the first few outer convex layers rather than all the convex layers. More specifically, the user who creates the SAO index will specify a number L. The SAO index keeps only the first L outer convex layers.

Intuitively, if most linear optimization queries use a large K (say, 20), L could be smaller than that K (say, $L=10$). However, if most linear optimization queries use a very small K (say, 1), L should be a little larger than that K (say, $L=2$). The reason is as follows. As will be shown in Section 3.3 below, when K is very small, a few backup convex layers are preferred. This is to prevent the undesirable situation where a few tuples on the first K outer convex layers expire and large errors are introduced into the approximate answers to some linear optimization queries. On the other hand, when K is large, for a top-K linear optimization query, it is likely that the top-K tuples can be found on the first J outer convex layers, where $J<K$. In this case, if a few tuples on these J convex layers expire, the other convex layers can serve as backups automatically. Hence, L does not need to be larger than K.

Since M is limited, the SAO index cannot always keep the precise first L outer convex layers. For example, in the worst case, all W tuples in the sliding window may reside on the first convex layer rather than spread over multiple convex layers. Therefore, for each of the first L outer convex layers, the SAO index may only keep some of the most "important" tuples rather than all the tuples belonging to that layer. In other words, each layer in the SAO index is an *approximate convex layer* (ACL) in the sense that it is an approximation to

the corresponding precise convex layer in the onion index. For each i ($1 \le i \le L$), l_i is used to denote the ith ACL.

The SAO index maintains the following properties. Each ACL is the convex hull of all the tuples on that layer. For each i ($1 \le i \le L-1$), l_{i+1} is contained within l_i. Also, the total number of tuples on all L ACLs is no more than M. (Recall that in a transition period, $M+1$ tuples can be kept in the SAO index temporarily.)

All the tuples in the SAO index are kept as a sorted, doubly-linked list L_{dl}. The sorting criterion is a tuple's remaining lifetime. The first tuple in L_{dl} is going to expire the soonest. In this way, we can quickly check whether any tuple in the SAO index expires, which is needed at Step 2 of Section 3.6 below. Also, we can easily delete tuples that are in the middle of L_{dl}, which is necessary when the available memory is exhausted and a tuple needs to be deleted from the SAO index (see Section 3.4 below).

For each ACL, a standard convex hull data structure [29] is maintained. The vertices of the convex hull point to tuples in L_{dl}. Also, each tuple t in L_{dl} has a label indicating the ACL to which tuple t belongs. This label is used when a tuple expires and needs to be removed from the corresponding ACL (see Section 3.6 below).

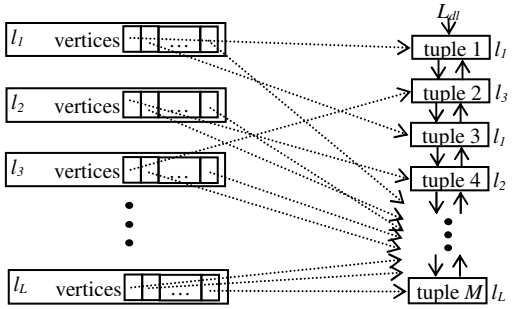


Figure 2. Sample data structure of a SAO index.

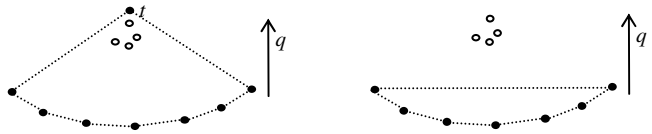
3.3 Allocating Proper Memory to Each Layer

A key challenge in designing a SAO index is: For a given amount of memory, how do we properly allocate the memory to each layer so that the quality of the answers can be maximized? In the following sections, we first illustrate why such careful allocation is needed, then we describe three allocation strategies: a simple, uniform strategy; a static, error-minimizing strategy; and a dynamic, error-minimizing strategy.

Necessity of a Storage Allocation Strategy

The SAO index needs to control the number of tuples on each ACL. Otherwise, one or a few ACLs may use up all the storage budget M. As a result, the SAO index may not provide good-quality answers to certain queries.

For example, suppose l_1, the first ACL, uses up all the storage budget M and all the other $L-1$ ACLs are empty, as shown in Figure 3. (We adopt the convention in Chang *et al.* [8] of using dotted polygons to represent ACLs.) In this case, the information about all the tuples inside l_1 is lost. These tuples are represented by the hollow circles in Figure 3 and thus called hollow tuples.

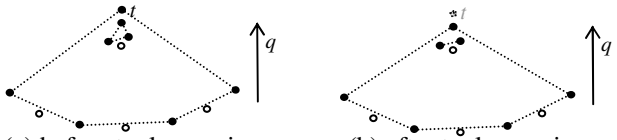


(a) before tuple t expires (b) after tuple t expires

Figure 3. A SAO index in two-dimensional space (2D) with l_1 using up all the storage budget.

Consider a top-1 linear optimization query q whose direction is represented by the arrow in Figure 3. When tuple t expires from the sliding window, the SAO index cannot provide a good-quality answer to q. This is because the linear combination values of those hollow tuples are all much larger than the maximal linear combination value of the remaining tuples on l_1. However, those hollow tuples are not kept in the SAO index.

Now suppose the SAO index controls the number of tuples on each ACL. For example, the storage budget M is divided among all L ACLs in a more balanced way, as shown in Figure 4(a). This has the effect that some of the information contained in l_1 is lost while some other information can be kept in the other $L-1$ ACLs.



(a) before tuple t expires (b) after tuple t expires

Figure 4. A SAO index in 2D with the storage budget divided between l_1 and l_2 in a balanced way.

Then after tuple t expires, l_1 can be "recovered" by using the information contained in l_2 (the recovery procedure is described in Section 3.6), as shown in Figure 4(b). As a result, the SAO index can still provide a good-quality answer to the linear optimization query q.

A Simple, Uniform Storage Allocation Strategy

A simple storage allocation strategy is to divide the storage budget M evenly among all L ACLs. Each ACL cannot keep more than M/L tuples. However, this simple, uniform method is far from being optimal. The reason is as follows. In the precise onion index, according to Property 1, for a linear optimization query, we tend to find more of the top-K tuples on the outer convex layers than on the inner convex layers. For example, consider a top-20 linear optimization query. The precise onion index may find the largest ten tuples on the first convex layer, the next largest six tuples on the second convex layer, and the remaining largest four tuples on the third convex layer. This is consistent with the effect observed in Chang *et al*. [8]: to retrieve the top-K tuples, typically we only need to access a few outer convex layers rather than all the first K outer convex layers in the precise onion index.

Similar to the precise onion index, we tend to find more of the top-K tuples on the outer ACLs than on the inner ACLs. Intuitively, the more tuples allocated to an ACL l_i ($1 \le i \le L$), the closer l_i is to the corresponding precise convex layer and thus the more precise the top tuples we will find on l_i. Moreover, as discussed below, compared to the top tuples that are found on the inner ACLs, the top tuples that are found on the outer ACLs are ranked higher and thus more important. Therefore, in order to provide high-quality answers to linear optimization queries, the SAO index should allocate more tuples to the outer ACLs than to the inner ACLs.

Static, Error-Minimizing Storage Allocation

Now we describe a static, error-minimizing storage allocation strategy when resource is limited. We determine the optimal numbers of tuples the SAO index should allocate to the L ACLs. By resource being limited, we mean that each ACL needs more tuples than can be actually allocated to it. We will describe next a dynamic, error-minimizing storage allocation strategy that is based on the results derived in this section. For each i ($1 \le i \le L$), let N_i denote the optimal number of tuples that should be allocated to l_i. Then $\sum_{i=1}^{L} N_i = M$. (1)

In general, the values of N_i's depend on the exact data distribution. Since the data distribution is usually not known in advance, N_i's cannot be precisely determined. In our derivation, a few simplified assumptions are made. This makes our derived N_i's heuristic in nature rather than exactly optimal. In the performance section 4 below, we show that our heuristics indeed work well in practice.

Consider a top-L linear optimization query. For each i ($1 \le i \le L$), let t_i represent the exact ith largest tuple, and t_i' represent the ith largest tuple that is found in the SAO index. Here, v_i is the linear combination value of t_i, and v_i' is the linear combination value of t_i'. The relative error of t_i' is defined as $e_i = |(v_i - v_i')/v_i|$. (2)

For the top-L tuples (t_i') that are returned by the SAO index, a weighted mean of their relative errors is used as the performance metric e: $e = \sum_{i=1}^{L} u_i e_i / \sum_{i=1}^{L} u_i$, (3)

where u_i is the weight of e_i. Intuitively, the higher the rank of a tuple t, the more important t's relative error. Hence, u_i should be a non-increasing function of i. We would like to minimize the mean of e for all top-L linear optimization queries, which is how N_i's are derived. Our idea is to represent the mean of e as a function

of N_i's and find its minimal value under condition (1). According to our derivation whose details are in the Appendix, we can show that $N_i \propto \sqrt{C_i}$, where $C_j = \sum_{i=j}^{L} 1/i^2$. (4)

3.4 Dynamic, Error-Minimizing Storage Allocation

If for each i ($1 \le i \le L$), l_i, the ith ACL, always needs more than N_i tuples, then the SAO index can use a static storage allocation strategy so that l_i gets a fixed storage quota of N_i tuples. However, the real world is more dynamic. At any time, some ACLs may need more than N_i tuples while other ACLs may need fewer than N_i tuples. As tuples keep entering and leaving the sliding window, the storage requirements of different ACLs change continuously. If the SAO index sticks with the static storage allocation, the total storage quota of M tuples cannot always be fully utilized. For example, this is the case if some ACLs do not use up their storage quota N_i. This will hurt the quality of the answers.

To ensure the best quality of the answers, the SAO index needs to fully utilize the storage budget M as much as possible. Therefore, instead of static storage allocation, it does dynamic storage allocation. In this way, the ACLs that need extra storage quota can "borrow" some quota from those ACLs that have spare quota.

Now we describe the dynamic storage allocation strategy. Our design principle is: Whenever possible, the storage budget M is used up. At the same time, the SAO index tries its best to maintain the condition that the number of tuples on l_i is proportional to $\sqrt{C_i}$.

The concrete method is as follows. For each i ($1 \le i \le L$), let M_i denote the number of tuples on l_i. The SAO index continuously monitors these M_i's. At any time, there are two possible cases. In the first case, $\sum_{i=1}^{L} M_i \le M$. This is a normal case and nothing needs to be done, as the storage budget M has not been used up. In the second case, $\sum_{i=1}^{L} M_i = M + 1$. (According to Section 3.6, $\sum_{i=1}^{L} M_i$ can never be larger than $M+1$.) This is an overflow case, as the storage budget M is exceeded by one. We need to pick a candidate ACL and delete one tuple from it.

Note that the dynamic storage allocation strategy is of a fine granularity. Each time when memory is exhausted, one tuple is deleted from the SAO index. The reader might wonder if we could use a coarser granularity. That is, multiple tuples (rather than a single tuple) are deleted at once from the SAO index. Then it will take longer before memory is exhausted again. However, such a method is not desirable in our environment. This is because our storage budget is precious, as M may be small. We want to fully utilize the limited storage budget as much as possible so that the SAO index can provide the best-quality answers to linear optimization queries. Moreover, as can be seen from Section 3.6 (Step 1), the insertion of a new tuple into the SAO index may cause multiple tuples to be expelled from l_L and then some storage becomes available automatically.

Choosing a Candidate Approximate Convex Layer

We first discuss how to choose the candidate ACL. For each i ($1 \le i \le L$), let $r_i = M_i / N_i$. We pick j such that $r_j = \max\{r_i \mid r_i > 1, 1 \le i \le L\}$. This j must exist. Otherwise for each i ($1 \le i \le L$), $r_i \le 1$. This leads to $\sum_{i=1}^{L} M_i \le \sum_{i=1}^{L} N_i = M$, which conflicts with the condition that $\sum_{i=1}^{L} M_i = M + 1$. l_j is chosen as the candidate ACL.

This method is based on the intuition that the candidate ACL l_j should satisfy the following two conditions. First, l_j has used up its fixed quota N_j. Second, among all the ACLs that have used up their fixed quota N_i, l_j exceeds its fixed quota (by the ratio r_j) the most. In this way, we can be fair to those ACLs that have not used up their fixed quota N_i. Also, the SAO index can maintain the condition $M_i \propto \sqrt{C_i}$ as much as possible.

Choosing a Candidate Tuple

Now one candidate tuple needs to be deleted from the candidate ACL l_j. Intuitively, this candidate tuple t should have a close neighbor so that deleting t will have little impact on the shape of l_j. Two tuples on an ACL are neighbors if they are connected by an edge.

For any tuple t on l_j, let R_t denote the Euclidean distance between tuple t and its nearest neighbor on l_j. The candidate tuple is chosen to be the tuple that has the smallest R_t (usually there are a pair of such tuples and the older one, i.e., the sooner-to-expire one, is picked). Note that R_t is not the smallest distance between tuple t and any other tuple on l_j. Rather, in computing R_t, only tuple t's neighbors are considered.

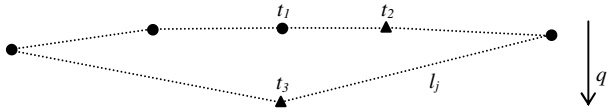


Figure 5. One approximate convex layer of a SAO index in two-dimensional space.

We use an example to illustrate the reasoning. Consider the candidate ACL l_j shown in Figure 5. If R_t denotes the smallest distance between tuple t and any other tuple on l_j, then tuples t_1 and t_3 have the smallest R_t. Suppose t_3 is older than t_1. In this case, t_3 is picked as the candidate tuple and deleted from l_j. This greatly influences the shape of l_j. There are two possible cases, and we can run into trouble in either case:

(1) $j = L$. Consider a top-K linear optimization query q whose direction is shown in Figure 5. Suppose the Kth largest tuple of q comes from l_L. Then the SAO index cannot provide a good-quality answer for the Kth largest tuple of q, since the information about all the tuples inside l_L is lost.

(2) $j < L$. Due to the dramatic shape change of l_j, it is likely that after deleting t_3, l_j will overlap with l_{j+1}. In this case, as will be described in Section 3.6 below, the SAO index needs to adjust l_{j+1} and maybe some ACLs inside l_{j+1}. This is rather time-consuming.

In contrast, if R_t is the distance between tuple t and its nearest neighbor on l_j, then tuples t_1 and t_2 have the smallest R_t. Irrespective of whether t_1 or t_2 is deleted from l_j, there is only a minor change to the shape of l_j and thus we are not likely to run into the trouble described above.

Deleting Candidate Tuple

Finally, after choosing the candidate tuple t, we use the method that is described in Step 2 of Section 3.6 below to delete t from l_j and then adjust the affected ACLs.

3.5 Index Creation

At the beginning, the SAO index is empty. We keep receiving new tuples until there are M tuples. Then a standard convex hull construction algorithm, such as the quickhull method whose worst-case computational complexity is $O(N \ln N + N^{\lfloor n/2 \rfloor})$ given N points in n-dimensional space [5], is used to create the L ACLs in batch. This is mainly for efficiency purposes, as creating convex hulls in batch is less expensive than constructing convex hulls incrementally (i.e., each time adding one new tuple) [5]. Note that it is possible that some of the innermost ACLs are empty. From now on, each time a new tuple arrives, we use the method in Section 3.6 to incrementally maintain the SAO index.

3.6 Index Maintenance

In a typical data streaming environment, we expect that $W >> M$, i.e., only a small fraction of all W tuples in the sliding window are stored in the SAO index. Intuitively, this means that tuples on the ACLs can be regarded as anomalies with extreme feature attribute values. The smaller the i ($1 \le i \le L$) is, the more anomalous the tuples on l_i are. As a result, we have the following heuristic (not exact) property:

Property 2: Most new tuples are "normal" tuples and thus inside l_L. Moreover, for a new tuple t, it is most likely to be inside l_L. Less likely is tuple t between l_{L-1} and l_L, and even less likely is tuple t between l_{L-2} and l_{L-1}, etc.

According to our storage allocation strategy described in Sections 3.3 and 3.4, the inner ACLs tend to have fewer tuples than the outer ACLs. From computational geometry literature [28, 29], it is known that given a point p, the complexity of checking whether p is inside a convex polytope P increases with the number of vertices of P. Therefore, we have the following property:

Property 3: For a tuple t, it is typically faster to check whether t is inside an inner ACL than to check whether t is inside an outer ACL.

Upon the arrival of a new tuple t, Properties 2 and 3 are used to reduce the SAO index maintenance overhead. We proceed in three steps. Step 1 checks whether tuple t needs to be inserted into the SAO index. Step 2 checks whether any tuple in the SAO index expires. Step 3 handles memory overflow.

Step 1: Tuple Insertion

All ACLs are checked one by one, starting from l_L. That is, our checking direction is from the innermost ACL to the outermost ACL. From Properties 2 and 3 together with the procedure described below, it can be seen that this checking direction is the most efficient one.

There are two possible cases. In the first case, tuple t is inside l_L. According to Property 2, this is the mostly likely case. Also, according to Property 3, it can be discovered quickly whether tuple t is inside l_L. In this first case, tuple t will not change any of the L ACLs and thus can be thrown away immediately. Since no new tuple is introduced into the SAO index, there will be no memory overflow. Hence, Step 3 can be skipped, although Step 2 still needs to be performed. Note: If l_L is empty, we think that tuple t is outside of l_L because t needs to be inserted into the SAO index.

In the second case, a number k ($1 \le k \le L$) can be located such that tuple t is inside l_{k-1} but outside of l_k. (If $k=1$, tuple t is outside of all L ACLs.) In this case, tuple t needs to be inserted into the SAO index. This insertion will affect l_k and maybe some ACLs inside l_k. However, none of the first $k-1$ ACLs will be affected.

This insertion is done in the following way. The new l_k is computed by considering both tuple t and all the tuples on the existing l_k, using any standard incremental convex hull maintenance algorithm, like the beneath-beyond method whose update computational complexity is $O(N^{\lfloor n/2 \rfloor})$ given a convex hull with N points in n-dimensional space [28, 29]. This may cause one or more tuples to be expelled from l_k. If that happens, the expelled tuples need to be further inserted into the next layer l_{k+1}. In other words, the new l_{k+1} is computed by considering both the expelled tuples and all the tuples on the existing l_{k+1}. This may again expel some tuples from l_{k+1}. The iteration continues until either l_L is reached or no more tuples are expelled. Figure 6 shows an example of inserting a new tuple t into a SAO index. The insertion procedure is described below in pseudo code.

```
Let set S = {t};
i = k;
while (|S|>0 && i≤L) {
   S = S ∪ {tuples on l_i}; // insert expelled tuples into the current layer
   l_i = convex hull of S; // construct a new convex hull
   S = S - {tuples on l_i}; // obtain expelled tuples from l_i
   i++; // go to the next layer
}
```

(a) before inserting tuple t (b) after inserting tuple t

Figure 6. An illustration of inserting a new tuple *t* into a SAO index.

Step 2: Tuple Expiration

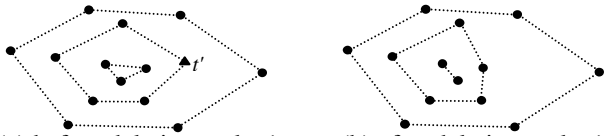
The arrival of tuple t will cause at most one tuple in the SAO index to expire from the sliding window. Let t' denote the first tuple in the doubly-linked list L_{dl}. Recall that all the tuples in L_{dl} are sorted in ascending order of their remaining lifetimes. Hence, only tuple t' needs to be checked, as t' is the only tuple in the SAO index that may expire from the sliding window.

There are two possible cases. In the first case, tuple t' has not expired. We proceed to Step 3 directly.

In the second case, tuple t' has expired and thus needs to be deleted from the SAO index. Suppose tuple t' is on l_k ($1 \le k \le L$). The deletion of tuple t' will affect l_k and maybe some ACLs inside l_k. However, none of the first $k-1$ ACLs will be affected.

This deletion is done in the following way. The new l_k is computed by considering both all the tuples on the existing l_k (except for tuple t') and all the tuples on l_{k+1}. If one or more tuples on l_{k+1} are moved up to the new l_k, the new l_{k+1} needs to be further computed by considering both the remaining tuples on l_{k+1} and all the tuples on l_{k+2}. The iteration continues until either l_L is reached or no more tuples are moved up. Since this iteration procedure reduces the number of tuples in the SAO index by one, there will be no memory overflow and thus Step 3 can be skipped. Figure 7 shows an example of deleting a tuple t' from a SAO index. The deletion procedure is described below in pseudo code.

```
Let set S = {t'};
i = k;
while (|S|>0 && i≤L) {
   S1 = {tuples on li} - S; // obtain remaining tuples on the current layer
   S2 = S1 ∪ {tuples on li+1}; // merge with the next layer
   li = convex hull of S2; // construct a new convex hull
   S = {tuples on li} – S1; // obtain tuples that are moved up to the current layer
   i++; // go to the next layer
}
```



(a) before deleting tuple t' (b) after deleting tuple t'

Figure 7. An illustration of maintaining a SAO index when tuple t' expires.

Step 3: Handling Memory Overflow

In the above two steps, at most one new tuple is introduced into the SAO index while one or more tuples may be deleted (e.g., tuples may get expelled from l_L in Step 1). Now we check whether condition $\sum_{i=1}^{L} M_i \le M$ still holds. (Recall that M_i denotes the number of tuples on l_i.) If not, $\sum_{i=1}^{L} M_i = M + 1$ must be true. In this case, we use the procedure that is described in Section 3.4 to delete one tuple from the SAO index.

Discussion

From the above description, we can see that it is computationally expensive to either insert a new tuple into the SAO index or delete an existing tuple from the SAO index, as multiple ACLs may need to be reconstructed. Fortunately, upon the arrival of a new tuple, the amortized overhead of maintaining the SAO index is not that high. The reason is as follows.

First, according to Property 2, in most cases, the new tuple will be inside the innermost ACL l_L and thus can be thrown away immediately. Also, the number of tuples in the SAO index is at most $M+1$, which is usually much smaller than the sliding window size W. On average, after approximately W/M new tuples are received, one tuple in the SAO index expires. Therefore, we rarely need to either insert a new tuple into the SAO index or delete a tuple from the SAO index.

Second, M is typically not very large. Then for each i ($1 \le i \le L$), M_i, the number of tuples on l_i, is also not very large. This reduces the reconstruction overhead of ACLs, and also the overhead of checking whether the new tuple is inside an ACL.

Third, our SAO index maintenance algorithm has been optimized. For example, an efficient checking direction is used in Step 1.

3.7 Query Evaluation

To provide approximate answer to a top-K linear optimization query (K can be larger than L), we use the onion index search procedure that is described in Chang *et al.* [8]. We start from l_1 and search the ACLs one

by one. This search terminates when one of the following two conditions are satisfied: (1) all L ACLs have been searched (in this case, all L ACLs are treated as previous ACLs), (2) the Kth largest tuple on the previous ACLs has larger linear combination value than the largest tuple on the current ACL. Then the top-K tuples on the previous ACLs are returned to the user. According to Theorem 1, these K tuples are the top-K tuples in the SAO index.

3.8 Parallel Processing

The above discussion assumes that there is only one computer. If tuples arrive so rapidly that one computer cannot handle all of them, multiple (say, C) computers can be used. The concrete method is as follows. All the tuples are partitioned into C sets (say, using round-robin partitioning [15]). Each computer maintains a SAO index and handles a different set of tuples. When the user submits a top-K linear optimization query, the local top-K tuples are obtained on each computer. All these local top-K tuples are merged together to get the global top-K tuples. This is our answer to the top-K query.

4. Performance Evaluation

We implemented our techniques by modifying the widely used Qhull (version 2003.1) software package [5], which implements efficient constructs for the creation of convex hulls. Our measurements were performed on a computer with one 1.6GHz processor, 1GB main memory, one 75GB disk, and running the Microsoft Windows XP.

Our evaluation used both real and synthetic data sets. The **real data set** came from the 2005 UC data mining competition [31]. Among all the attributes, we used the three (n=3) attributes that carried the most information (i.e., had the largest number of distinct values) as the feature attributes. The purpose for using this real data set is to show that our techniques work well for at least one real data distribution (in addition to several synthetic data distributions) rather than trying to draw any specific conclusion from this data set. Also, this data set is freely available for verification.

The synthetic data sets contain only feature attributes and are described as follows:

Drifting Gaussian distribution data set: Each attribute value follows a Gaussian distribution with variance=1 and mean oscillating between $-D$ and D at a uniform speed. The oscillation period is $4W$, where W is the sliding window size.

Skewed uniform distribution data set: For each attribute value, with probability p, it is uniformly distributed between -0.1 and 0.1; with probability $(1-p)/2$, it is uniformly distributed between -0.5 and -0.1; with probability $(1-p)/2$, it is uniformly distributed between 0.1 and 0.5. This p controls the degree of distribution skew of the data set.

Seven experiments were performed. Each was repeated twenty times (twenty runs). Unless otherwise specified, all the reported numbers are averaged over these runs. In each experiment, after the system has run for enough time and reaches a steady state, we posed 100 top-K linear optimization queries, whose weights were uniformly distributed between -1 and 1. As in Section 3.3 and the Appendix, for each top-K linear optimization query, the weighted relative error e is defined as $e = \sum_{i=1}^{K} u_i e_i / \sum_{i=1}^{K} u_i$, where $u_i = 1/i$ $(1 \le i \le K)$. (Other choices of u_i can be used. The results are similar and thus omitted here.) The following three performance metrics were used:

(1) **Max error**: The maximum e observed for the 100 top-K linear optimization queries.
(2) **Avg error**: The average e observed for the 100 top-K linear optimization queries.
(3) **Throughput**: In the steady state, the average number of tuples that can be processed per second. This is a proxy to the maintenance cost of the SAO index. A high throughput means a low maintenance cost, and vice versa.

In the rest of this section, by default the sliding window size is W=1,000,000. The SAO index contains L=4 ACLs. The number of top tuples is K=10. The storage budget M is 500 tuples. The dimensionality n (i.e., the number of feature attributes) of the data set is 3.

Storage Budget Size

In this experiment, the real data set was used. We varied the storage budget M from 200 tuples to 700 tuples. We compared the following four storage allocation methods:

(1) **Dynamic error-minimizing method**: We used the dynamic storage allocation strategy described in Section 3.4. N_i's are computed according to (4).

(2) **Dynamic uniform method**: We used the dynamic storage allocation strategy described in Section 3.4. N_i's are computed as $N_i = M / L$.

(3) **Static error-minimizing method**: We used the static storage allocation strategy described in Section 3.3. N_i's are computed according to (4).

(4) **Static uniform method**: We used the static storage allocation strategy described in Section 3.3. N_i's are computed as $N_i = M / L$.

Figure 8 shows the impact of *M/W* ratio on the weighted relative error. For any *M/W* ratio, among all four storage allocation methods, the dynamic error-minimizing method achieves both the smallest max error and the smallest avg error. The error-minimizing storage allocation strategy always works better than the uniform one. This is consistent with the explanation we gave in Section 3.3 about why more tuples should be allocated to the outer ACLs than to the inner ACLs. Moreover, the dynamic storage allocation strategy always works better than the static one. This is consistent with the explanation we gave in Section 3.4 about why storage quota needs to be allocated dynamically. For the rest of Section 4, we focus on the dynamic error-minimizing method.

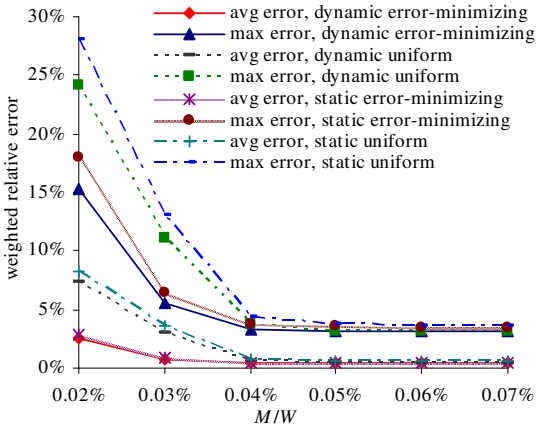


Figure 8. Weighted relative error vs. storage budget.

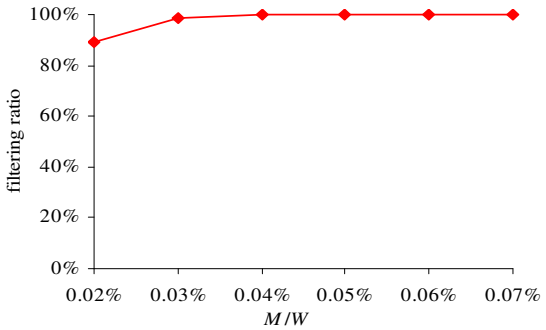


Figure 9. Filtering ratio vs. storage budget.

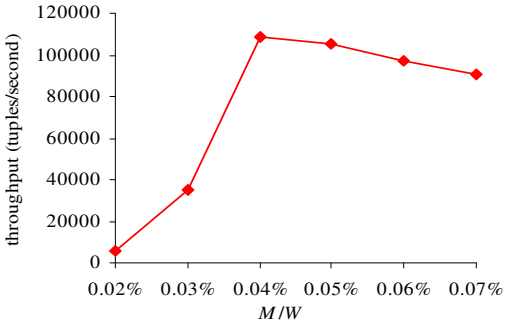


Figure 10. Throughput vs. storage budget.

The larger the *M/W* ratio is, the more information can be kept in the SAO index and thus the better the quality of the answers is. Hence, both the max error and the avg error decrease as *M/W* ratio increases. When *M/W*=0.05%, the avg error is 0.5% while the max error is 3%, both fairly small. In other words, even with a storage budget that is only a very small fraction of the sliding window size, the SAO index can provide fairly accurate answers.

The filtering ratio is defined as the probability that a new tuple is inside the innermost ACL l_L and thus can be thrown away immediately. The higher the filtering ratio, the better it is in terms of reducing the cost of maintaining a SAO index. Figure 9 shows the impact of *M/W* ratio on the filtering ratio. As *M/W* ratio increases, more tuples are stored in the SAO index over all the layers, including the innermost one, l_L.

Incoming tuples are more likely to be filtered as the size of l_L increases. More importantly, in all of the cases, the filtering ratios are always very close to 1. In other words, most new tuples are thrown away rather than being inserted into the SAO index. Note that even when *M/W* is only as small as 0.02%, the filtering ratio is as high as 89%. When *M/W*=0.05%, 99.8% of newly arriving tuples will be dropped immediately.

Figure 10 shows the impact of *M/W* ratio on the throughput. The throughput first increases with *M/W* ratio. After *M/W* ratio reaches a threshold 0.04%, the throughput decreases as *M/W* ratio increases. This is because the throughput depends on two factors: the filtering ratio and the tuple expiration rate (i.e., the speed that tuples in the SAO index expire). Both factors increase with *M/W* ratio. If a new tuple is not inside l_L, expensive convex hull computation needs to be performed to insert this tuple into the SAO index. Similarly, when a tuple in the SAO index expires, expensive convex hull computation needs to be performed to maintain the SAO index. Before *M/W* ratio reaches a threshold, the filtering ratio increases faster than the tuple expiration rate does. Thus, the throughput first increases with *M/W* ratio. After *M/W* ratio reaches this threshold, the volume of l_L starts to increase slowly. Then the tuple expiration rate increases faster than the filtering ratio. As a result, the throughput decreases as *M/W* ratio increases further.

From Figures 8, 9, and 10, we can see that even with a very small storage budget (*M/W*=0.05%), the SAO index can filter out 99.8% of new tuples, support a throughput of over 100,000 tuples/second, and achieve fairly accurate estimates with the avg error of 0.5%.

We repeated the storage budget size experiment by replacing the real data set with the synthetic drifting Gaussian distribution data set, where *D=0*. The results for the Gaussian distribution data set are shown in Figures 11, 12, and 13. They are similar to the results for the real data set.

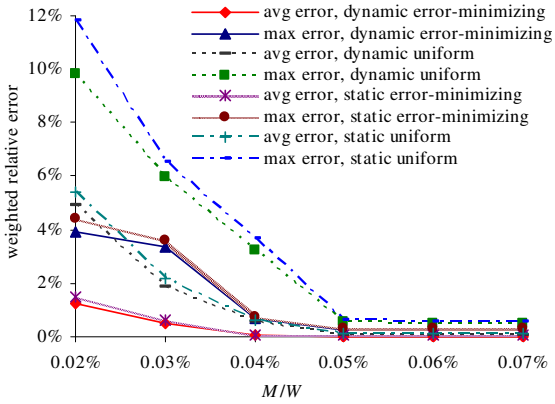


Figure 11. Weighted relative error vs. storage budget (Gaussian distribution data set).

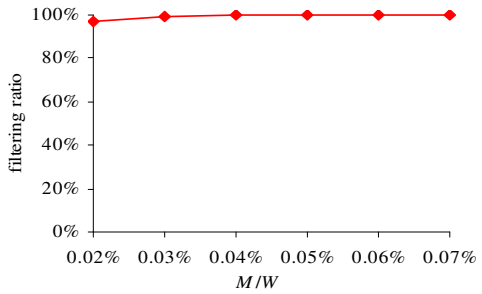


Figure 12. Filtering ratio vs. storage budget (Gaussian distribution data set).

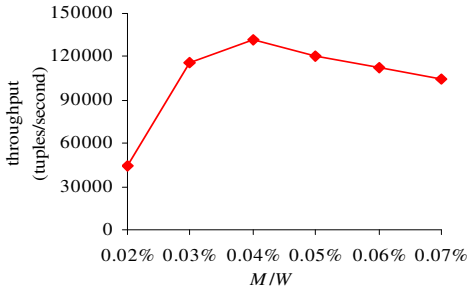


Figure 13. Throughput vs. storage budget (Gaussian distribution data set).

Sliding Window Size

In this experiment, the real data set was used. We varied the sliding window size *W* from 250,000 to 1,500,000. Figure 14 shows the impact of *W* on the weighted relative error. The larger the *W* is, the smaller *M/W* ratio becomes and thus relatively less information is kept in the SAO index. Therefore, when *W*≥750,000, both the avg error and the max error increase slowly with *W*. However, when *W*<750,000, not so many tuples are in the sliding window and these tuples are scattered in the entire data space. If one top tuple is missed for a query, we may have a large weighted relative error. The smaller the *W* is, the larger the chance of having a large weighted relative error. Hence, when *W*<750,000, both the avg error and the max error increase as *W* decreases.

Figure 15 shows the impact of W on the throughput. The larger the W is, the smaller the tuple expiration rate. Therefore, the throughput increases with W.

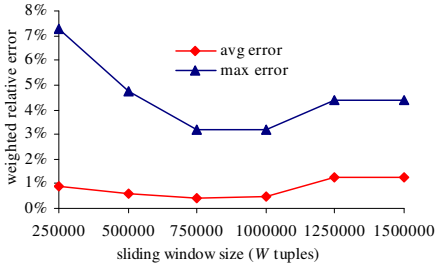


Figure 14. Weighted relative error vs. sliding window size.

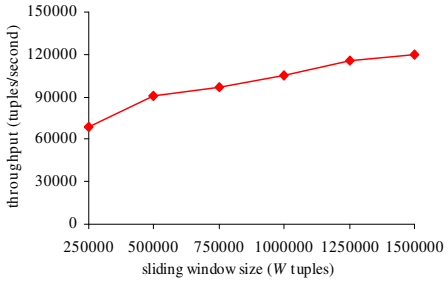


Figure 15. Throughput vs. sliding window size.

Number of Layers

In this experiment, the real data set was used. We varied the number of layers L in the SAO index from 1 to 7. Figure 16 shows the impact of L on the weighted relative error. The more layers in the SAO index, the more likely we can find the exact top-K tuples. Hence, both the avg error and the max error decrease as L increases.

Figure 17 shows the impact of L on the throughput. The larger the L is, the more ACLs need to be re-computed during the SAO index maintenance. In other words, the SAO index maintenance overhead increases with L. As a result, the throughput decreases as L increases.

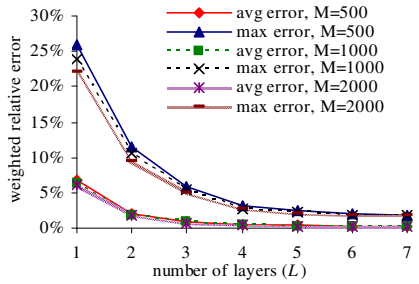


Figure 16. Weighted relative error vs. number of layers.

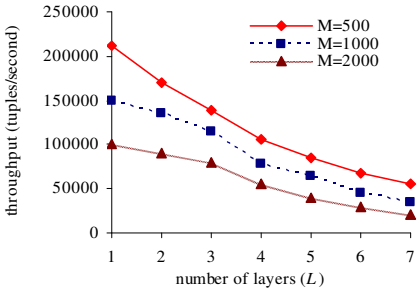


Figure 17. Throughput vs. number of layers.

Number of Top Tuples

In this experiment, the real data set was used. We varied the number of top tuples K from 6 to 14. Figure 18 shows the impact of K on the weighted relative error. The larger the K is, the less likely we can find the exact top-K tuples. Therefore, both the avg error and the max error increase with K.

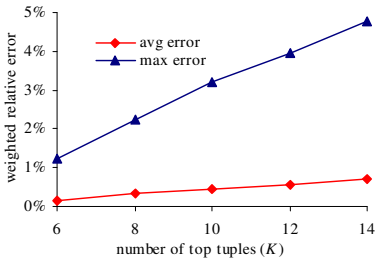


Figure 18. Weighted relative error vs. number of top tuples.

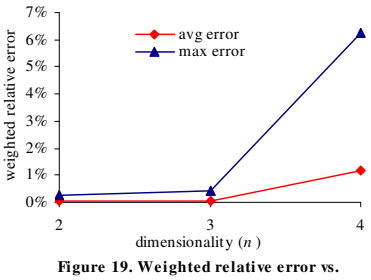


Figure 19. Weighted relative error vs. dimensionality.

Dimensionality

In this experiment, the synthetic Gaussian distribution data set was used. We varied the dimensionality n (i.e., the number of feature attributes) of the data set from two to four. Figure 19 shows the impact of n on the weighted relative error. The larger the n is, the more scattered tuples are distributed in the entire data space

and the larger the penalty of missing one top tuple becomes for a linear optimization query. Therefore, both the avg error and the max error generally increase with n.

Figure 20 shows the impact of n on the throughput. Note that the y-axis uses logarithmic scale. Given the same number of points, the larger the n, the more expensive the overhead of constructing a convex hull becomes. Hence, the throughput decreases as n increases.

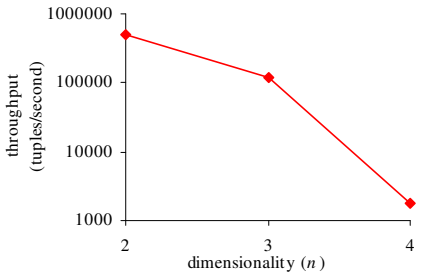


Figure 20. Throughput vs. dimensionality.

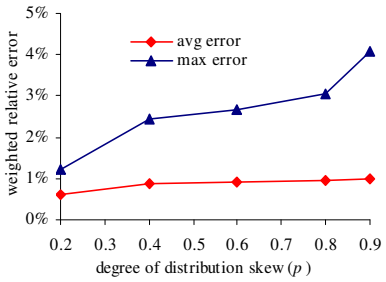


Figure 21. Weighted relative error vs. degree of distribution skew.

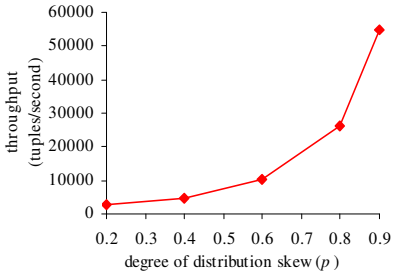


Figure 22. Throughput vs. degree of distribution skew.

Distribution Skew

In this experiment, the synthetic skewed uniform distribution data set was used. Recall that p controls the degree of distribution skew of that data set. We varied p from 0.2 to 0.9. When $p=0.2$, attribute values are uniformly distributed between 0 and 1. The larger the p is, the more skewed the distribution of attribute values becomes.

Figure 21 shows the impact of p on the weighted relative error. The more skewed the distribution of attribute values is, the more scattered the largest tuples are distributed in the entire data space and the larger the penalty of missing one top tuple becomes for a linear optimization query. As a result, the max error increases with p, while the avg error remains fairly stable and shows the robustness of the SAO index approach.

Figure 22 shows the impact of p on the throughput. The more skewed the distribution of attribute values is, the more likely a new tuple is inside l_L and thus can be thrown away immediately. Hence, the throughput increases with p.

Degree of Drifting

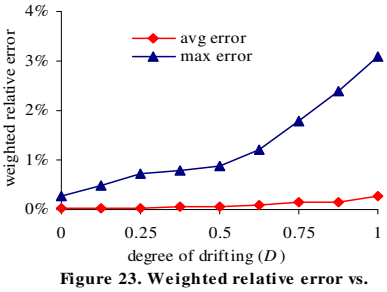


Figure 23. Weighted relative error vs. degree of drifting.

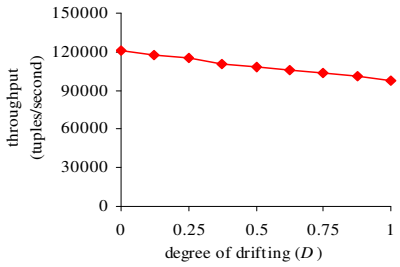


Figure 24. Throughput vs. degree of drifting.

In this experiment, the synthetic drifting Gaussian distribution data set was used. Recall that D controls the degree of drifting of that data set. We varied D from 0 to 1. Figure 23 shows the impact of D on the weighted relative error. The larger the degree of drifting is, the more likely some top tuples are missed for a linear optimization query (the reason is similar to that in Section 3.1). As a result, the max error increases with D, while the avg error remains fairly stable and shows the robustness of the SAO index approach.

Figure 24 shows the impact of D on the throughput. The larger the degree of drifting is, the more likely a new tuple is outside l_1 and thus the SAO index needs to be maintained. Hence, the throughput decreases as D increases.

In summary, in all the above seven experiments, using the SAO index to answer a top-*K* linear optimization query always takes less than 0.00002 second. Using a fairly small amount of memory space, the SAO index provides good approximate answers to top-*K* linear optimization queries, and the quality of approximate answers improves with the amount of available memory. Even under a rapid tuple arrival rate, the SAO index can still be maintained efficiently in real time. Hence, the SAO index can provide good support for answering linear optimization queries against a data stream.

5. Related Work

Hristidis *et al*. [20] proposed using materialized views to answer linear optimization queries. The method in Hristidis *et al*. [20] keeps multiple materialized views, each of which contains a copy of the entire data set. Hence, that method does not work in a data streaming environment, where available storage space is limited.

Yi *et al*. [33] proposed keeping extra tuples to reduce the maintenance overhead of top-*K* materialized views. In their environment, the query, which is defined by the top-*K* materialized view, is known in advance. In our case, the linear optimization queries may not be known in advance.

Top-*K* query evaluation algorithms have been proposed for a variety of scenarios: RDBMS [4, 11, 16, 23], expensive predicates [10], multimedia [17, 18], and web [26]. None of these works addresses the data streaming environment, where available storage space is limited.

Recently, various methods [2, 12, 22] have been proposed for maintaining approximate convex hulls. The method in Agarwal *et al*. [2] cannot handle tuple deletion. The storage space required by the method in Cormode *et al*. [12] has no upper bound. The method in Hershberger *et al*. [22] only works in the two-dimensional case and cannot handle tuple deletion. Therefore, none of these methods applies to our environment, where available space is limited and both tuple insertion and deletion need to be handled.

Böhm *et al*. [7] proposed two methods for computing the precise convex hull that were based on multi-dimensional index structures. Clarkson *et al*. [13] proposed a method for maintaining the precise convex hull under insertion and deletion of data points. All those methods require keeping all data points and thus do not work in our environment, where available storage space is limited.

Mouratidis *et al*. [27] uses multi-layer skybands to answer top-*K* monotonic queries over sliding window against a data stream. Mouratidis *et al*. [27] assumes that (1) the memory is large enough to hold all the tuples in the sliding window, and (2) all the attributes have minimal and maximal values that are known beforehand. These two assumptions are invalid in our environment.

Recently, various synopsis data structures have been proposed for data streams [19]. Our SAO index can be regarded as one kind of synopsis. However, none of the previously proposed synopsis data structures can be used to answer linear optimization queries.

Proportional-share resource management has been studied in the operating system literature [30, 32]. There, the resource under consideration is usually continuous (e.g., CPU, network bandwidth). If one server increases its share of resources, typically multiple other servers will be affected and decrease their shares of resources simultaneously. In our case, the resource under consideration (tuples) is discrete. If memory is exhausted and one ACL requires an extra tuple, our dynamic storage allocation strategy only forces one other ACL to give up one of its tuples.

6. Conclusion

This paper proposes a SAO index to support the answering of arbitrary linear optimization queries against a data stream. With a limited amount of memory, a dynamic, error-minimizing storage allocation strategy is used to maximize the quality of the approximate answers by allocating more memory to the outer layers of the SAO index. The maintenance of the SAO index is very efficient, as most of the newly arriving tuples are thrown away immediately without causing any changes to the SAO index. The efficiency of the SAO index is evaluated through a prototype implementation.

Like that of the original onion index, the maintenance cost of the SAO index increases rapidly with the number of feature attributes. In the case that there are many candidate feature attributes, we need to use an appropriate feature selection method [38] to control the number of used feature attributes so that the SAO index can still be maintained in real time while good approximate answers can be provided to linear optimization queries.

Acknowledgements

We would like to thank Michail Vlachos for helpful discussions.

References

1. Abadi DJ, Carney D, and Çetintemel U *et al*. (2003) Aurora: a New Model and Architecture for Data Stream Management. VLDB J. 12(2): 120-139.
2. Agarwal PK, Har-Peled S, and Varadarajan KR (2004) Approximating Extent Measures of Points. JACM 51(4): 606-635.
3. Babcock B, Babu S, and Datar M *et al*. (2002) Models and Issues in Data Stream Systems. PODS, pp 1-16.
4. Bruno N, Chaudhuri S, and Gravano L (2002) Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. TODS 27(2): 153-187.
5. Barber CB, Dobkin DP, and Huhdanpaa H (1996) The Quickhull Algorithm for Convex Hulls. ACM Trans. Math. Softw. 22(4): 469-483.
6. Bangolae SL, Jayasumana AP, and Chandrasekar V (2003) Gigabit Networking: Digitized Radar Data Transfer and Beyond. ICC, pp 684-688.
7. Böhm C, Kriegel HP (2001) Determining the Convex Hull in Large Multidimensional Databases. DaWaK, pp 294-306.
8. Chang YC, Bergman LD, and Castelli V *et al*. (2000) The Onion Technique: Indexing for Linear Optimization Queries. SIGMOD, pp 391-402.
9. Chandrasekaran S, Cooper O, and Deshpande A *et al*. (2003) TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. CIDR.
10. Chang KC, Hwang SW (2002) Minimal Probing: Supporting Expensive Predicates for Top-k Queries. SIGMOD, pp 346-357.
11. Carey MJ, Kossmann D (1997) On Saying "Enough Already!" in SQL. SIGMOD, pp 219-230.
12. Cormode G, Muthukrishnan S (2003) Radial Histograms for Spatial Streams. Technical Report 2003-11 DIMACS.
13. Clarkson KL, Mehlhorn K, and Seidel R (1993) Four Results on Randomized Incremental Constructions. Comput. Geom. 3: 185-212.
14. Dantzig GB (1963) Linear Programming and Extensions, Princeton University Press.
15. DeWitt DJ, Gray J (1992) Parallel Database Systems: The Future of High Performance Database Systems. CACM 35(6): 85-98.
16. Donjerkovic D, Ramakrishnan R (1999) Probabilistic Optimization of Top N Queries. VLDB, pp 411-422.
17. Fagin R (1996) Combining Fuzzy Information from Multiple Systems. PODS, pp 216-226.
18. Fagin R, Lotem A, and Naor M (2001) Optimal Aggregation Algorithms for Middleware. PODS, pp 102-113.
19. Gibbons PB, Matias Y (1999) Synopsis Data Structures for Massive Data Sets. SODA, pp 909-910.
20. Hristidis V, Koudas N, and Papakonstantinou Y (2001) PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. SIGMOD, pp 259-270.
21. Hardy GH, Littlewood JE, and Polya G (1934) Inequalities, Cambridge University Press.
22. Hershberger J, Suri S (2004) Adaptive Sampling for Geometric Problems over Data Streams. PODS, pp 252-262.
23. Ilyas IF, Aref WG, and Elmagarmid AK (2004) Supporting Top-k Join Queries in Relational Databases. VLDB J. 13(3): 207-221.
24. Li CS, Chang YC, and Bergman LD *et al*. (2000) Model-Based Multi-modal Information Retrieval from Large Archives. ICDCS International Workshop of Knowledge Discovery and Data Mining in the World-Wide Web.
25. Li CS, Chang YC, and Smith JR *et al*. (2001) SPIRE/EPI-SPIRE Model-Based Multi-modal Information Retrieval from Large Archives. MMCBIR.
26. Marian A, Bruno N, and Gravano L (2004) Evaluating Top-k Queries over Web-Accessible Databases. TODS 29(2): 319-362.

27. Mouratidis K, Bakiras S, and Papadias D (2006) Continuous Monitoring of Top-k Queries over Sliding Windows. SIGMOD, pp 635-646.
28. O'Rourke J (1998) Computational Geometry in C, Second Edition, Cambridge University Press.
29. Preparata FP, Shamos MI (1985) Computational Geometry - An Introduction, Springer.
30. Sullivan DG, Seltzer MI (2000) Isolation with Flexibility: A Resource Management Framework for Central Servers. USENIX General Track, pp 337-350.
31. 2005 UC Data Mining Competition Homepage. http://mill.ucsd.edu.
32. Waldspurger CA, Weihl WE (1994) Lottery Scheduling: Flexible Proportional-Share Resource Management. OSDI, pp 1-11.
33. Yi K, Yu H, and Yang J *et al*. (2003) Efficient Maintenance of Materialized Top-k Views. ICDE, pp 189-200.
34. Luo G, Wu K, and Yu PS (2007) SAO: A Stream Index for Answering Linear Optimization Queries. ICDE, pp 1302-1306.
35. Gedik B, Wu K, and Yu PS *et al*. (2007) CPU Load Shedding for Binary Stream Joins. KAIS 13(3): 271-303.
36. Cho M, Pei J, and Wang K (2007) Answering Ad Hoc Aggregate Queries from Data Streams Using Prefix Aggregate Trees. KAIS 12(3): 301-329.
37. Agarwal D (2007) Detecting Anomalies in Cross-classified Streams: A Bayesian Approach. KAIS 11(1): 29-44.
38. Kalousis A, Prados J, and Hilario M (2007) Stability of Feature Selection Algorithms: A Study on High-dimensional Spaces. KAIS 12(1): 95-116.

Appendix

Let p_{ij} ($1 \le i \le L$, $1 \le j \le L$) represent the probability that for a top-L linear optimization query, tuple t_i is on the jth convex layer in the onion index. We assume that in this case, tuple t_i' is also on l_j, the jth ACL in the SAO index. Furthermore, the mean of e_i for all the top-L linear optimization queries is $1/N_j$, based on the intuition that the larger the N_j, the closer l_j is to the jth precise convex layer in the onion index and thus the smaller the e_i.

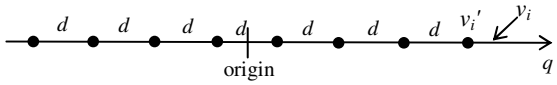


Figure 25. Projection of tuples along the direction of query *q*.

A heuristic justification for the assumption of $1/N_j$ is as follows. It is not mathematically rigorous. Rather, it is only used to illustrate our reasoning process. We project all the N_j tuples on l_j along the query direction, as shown in Figure 25. Each projection is a point. For these N_j points, let d denote the average distance between two adjacent points. Assume that on average, half of the N_j points are to the left of the origin, and the other half of the N_j points are to the right of the origin. The projection of tuple t_i' is point v_i'. We have $E(v_i') = dN_j/2$, where $E(x)$ represents the expectation of x. Note that v_i, the projection of tuple t_i, is to the right of point v_i'. Suppose the average distance between v_i' and v_i is $d/2$. Then for a specific j, the mean of e_i is

$$E\left(\left|\frac{v_i - v_i'}{v_i}\right|\right) \approx E\left(\left|\frac{v_i - v_i'}{v_i'}\right|\right) \approx \frac{E(|v_i - v_i'|)}{E(|v_i'|)} = \left|\frac{d/2}{dN_j/2}\right| = \frac{1}{N_j}.$$

Now we return to the goal of minimizing $\bar{e}$, the mean of e. For each i ($1 \le i \le L$), tuple t_i must be on one of the L convex layers in the onion index. Hence, $\bar{e}_i$, the mean of e_i, is a weighted average over all j's ($1 \le j \le L$):

$$\bar{e}_i = \sum_{j=1}^{L} p_{ij} \frac{1}{N_j}.$$

From (3), $\bar{e} \sum_{i=1}^{L} u_i = \sum_{i=1}^{L} u_i \bar{e}_i = \sum_{i=1}^{L} (u_i \sum_{j=1}^{L} p_{ij} \frac{1}{N_j}) = \sum_{j=1}^{L} (\frac{1}{N_j} \sum_{i=1}^{L} u_i p_{ij})$.

Define $C_j = \sum_{i=1}^{L} u_i p_{ij}$. (5)

We have $\bar{e} \sum_{i=1}^{L} u_i = \sum_{j=1}^{L} \frac{1}{N_j} C_j = \sum_{j=1}^{L} \sqrt{C_j} \frac{\sqrt{C_j}}{N_j}$. (6)

From (1), we get $M = \sum_{j=1}^{L} N_j = \sum_{j=1}^{L} \sqrt{C_j} \frac{N_j}{\sqrt{C_j}}$. (7)

To minimize $\bar{e}$, the following weighted arithmetic-harmonic means inequality is used:

Theorem 2 [21]. Given L positive weights w_1, w_2, …, w_L and L positive numbers x_1, x_2, …, x_L, we have weighted arithmetic mean ≥ weighted harmonic mean, with equality only when $x_1 = x_2 = ... = x_L$. That is,

$$\sum_{j=1}^{L} w_j x_j / \sum_{j=1}^{L} w_j \geq \sum_{j=1}^{L} w_j / \sum_{j=1}^{L} w_j \frac{1}{x_j}. \quad (8)$$

After transforming (8), we have

$$\sum_{j=1}^{L} w_j \frac{1}{x_j} \geq (\sum_{j=1}^{L} w_j)^2 / \sum_{j=1}^{L} w_j x_j. \quad (9)$$

Let $w_j = \sqrt{C_j}$ and $x_j = N_j / \sqrt{C_j}$. Using (6), (7), and (9), we know that $\bar{e}$ (or alternatively, the left side of (6)) is minimized when the following condition holds:

$$N_1 / \sqrt{C_1} = N_2 / \sqrt{C_2} = ... = N_L / \sqrt{C_L}. \quad (10)$$

Then from (1), we get $N_j = \sqrt{C_j} M / \sum_{i=1}^{L} \sqrt{C_i}$. (11)

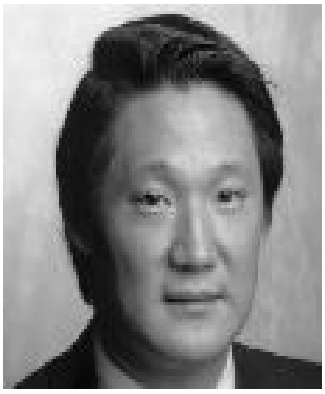
According to Property 1, we know that if $i < j$, $p_{ij} = 0$. If we assume that t_i, the exact ith largest tuple, has equal probability to be on any one of the first i outer convex layers in the onion index, then $p_{ij} = \begin{cases} 1/i & (i \geq j) \\ 0 & (i < j) \end{cases}$. In this paper, for illustration purposes, we pick $u_i = 1/i$. (Other choices of u_i can be used. The results are similar and thus omitted here.) Then from (5), we have

$$C_j = \sum_{i=j}^{L} u_i p_{ij} = \sum_{i=j}^{L} 1/i^2 .$$

Authors Biography

Gang Luo received the BSc degree from Shanghai Jiaotong University, P.R. China, in 1998, and the PhD degree from the University of Wisconsin-Madison in 2004. He is currently a research staff member at IBM T.J. Watson Research Center. His research interests include healthcare informatics, information retrieval, biostatistics, and database.

Kun-Lung Wu received his B.S. in E.E. from the National Taiwan University, Taipei, Taiwan, and his M.S. and Ph.D. in C.S. from the University of Illinois at Urbana-Champaign. He is the Manager of the Data-Intensive Systems and Analytics Group at the IBM T.J. Watson Research Center. The group conducts research and development in various aspects of data-intensive systems and analytics, currently focusing on programming languages and models for streaming applications; advanced analytic algorithms and applications; job scheduling and resource management for data-intensive systems. He is an IBM Master Inventor, and an IEEE Fellow. He was an Associate Editor for IEEE TKDE, 2000-2004. He served as an organizing/program committee member for many international conferences and workshops. He has received several IBM awards, including an IBM Corporate Environmental Affair Excellence Award. He has published extensively in various journals and refereed conferences. He also holds or has applied for many patents.



Philip S. Yu received the M.S. and Ph.D. degrees in E.E. from Stanford University, and the M.B.A. degree from New York University. He is a Professor in the Department of Computer Science at the University of Illinois at Chicago and also holds the Wexler Chair in Information and Technology. He was manager of the Software Tools and Techniques group at the IBM T.J. Watson Research Center. His research interests include data mining, Internet applications and technologies, database systems, multimedia systems, parallel and distributed processing, and performance modeling. Dr. Yu has published more than 500 papers in refereed journals and conferences. He holds or has applied for more than 300 US patents. Dr. Yu is a Fellow of the ACM and a Fellow of the IEEE. He is associate editors of ACM Transactions on the Internet Technology and ACM Transactions on Knowledge Discovery from Data.