SAO: A Stream Index for Answering Linear Optimization Queries

Gang Luo Kun-Lung Wu Philip S. Yu
IBM T.J. Watson Research Center
{luog, klwu, psyu}@us.ibm.com

Abstract

Linear optimization queries retrieve the top-K tuples in a sliding window of a data stream that maximize/minimize the linearly weighted sums of certain attribute values. To efficiently answer such queries against a large relation, an onion index was previously proposed to properly organize all the tuples in the relation. However, such an onion index does not work in a streaming environment due to fast tuple arrival rate and limited memory. In this paper, we propose a SAO index to approximately answer arbitrary linear optimization queries against a data stream. It uses a small amount of memory to efficiently keep track of the most "important" tuples in a sliding window of a data stream. The index maintenance cost is small because the great majority of the incoming tuples do not cause any changes to the index and are quickly discarded. At any time, for any linear optimization query, we can retrieve from the SAO index 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. We evaluate the effectiveness of this SAO index through a prototype implementation.

1. Introduction

Data stream applications are becoming popular. Many of such applications use various linear optimization queries [2, 3, 4] 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 [3]. In oil/gas exploration applications, linear models that incorporate drill sensor measurements and seismic information are used to guide the drilling direction [4]. In financial applications, linear models that incorporate personal credit history, income level, and employment history are used to evaluate credit risks for loan approvals [3].

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. [2] 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 *i*th 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, 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 the specific sequence of tuples in a stream. 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 [1]. Our experimental results 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.

2. Review of the Traditional Onion Index

We briefly review the earlier onion index [2] 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.

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.

The onion index in [2] 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.

According to linear programming theory, we have the following property [2]:

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.

3. SAO Index

The original onion index [2] 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. 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.

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.

3.1 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. The SAO index keeps only the first L outer convex layers, where L is specified by the user creating the index.

Since M is limited, the SAO index cannot always keep the precise first L outer 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.

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 3.5 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 3.3 below).

For each ACL, a standard convex hull data structure 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 3.5 below).

3.2 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?

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. 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, as outer ACLs contain the largest tuples.

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. 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)

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 [5], we can show that $N_i \propto \sqrt{C_i}$, where $C_j = \sum_{i=j}^{L} 1/i^2$. (4)

3.3 Dynamic, Error-Minimizing Storage Allocation

The real world is not static. 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. To ensure the best quality of the answers, the SAO index needs to fully utilize the storage budget M as much as possible by doing dynamic storage allocation.

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 3.5, $\sum_{i=1}^{L} M_i$ 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.

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 $\forall i$ ($1 \le i \le L$), $r_i \le 1$. 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.

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).

Deleting Candidate Tuple

Finally, after choosing the candidate tuple t, we use the method that is described in Step 2 of Section 3.5 below to delete t from L_j and then adjust the affected ACLs.

3.4 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 is used to create the L ACLs in batch. From now on, each time a new tuple arrives, we use the method in Section 3.5 to incrementally maintain the SAO index.

3.5 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. 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 3.2 and 3.3, the inner ACLs tend to have fewer tuples than the outer ACLs. From computational geometry literature, 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: Tuple Insertion

All ACLs are checked one by one, starting from L_L. 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.

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 in a way similar to that the onion index is maintained [2].

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}, which 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 in a way similar to that the onion index is maintained [2].

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. 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 3.3 to delete one tuple from the SAO index.

3.6 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 [2].

4. Performance Evaluation

We implemented our techniques by modifying the widely used Qhull (version 2003.1) software package [1], 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 the real data set in the 2005 UC data mining competition [6]. 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.

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 3.2, 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 $u_i = 1/i$ ($1 \le i \le K$). 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.

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.

We varied the storage budget M from 200 tuples to 500 tuples. We used the dynamic storage allocation strategy described in Section 3.3 and compared the following two methods of computing N_i's:

(1) **Error-minimizing method**: This is the method described in Section 3.2. N_i's are computed according to (4).
(2) **Uniform method**: $N_i = M / L$.

Figure 1 shows the impact of M/W ratio on the weighted relative error. For any M/W ratio, both the max error and the avg error are much smaller for the error-minimizing method. Namely, the error-minimizing method works better than the uniform one.

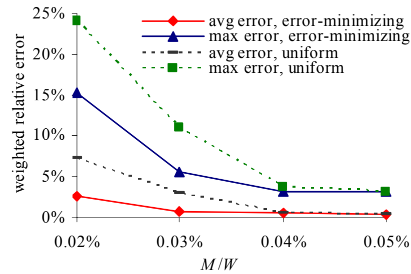

Figure 1. Weighted relative error vs. storage budget.

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.

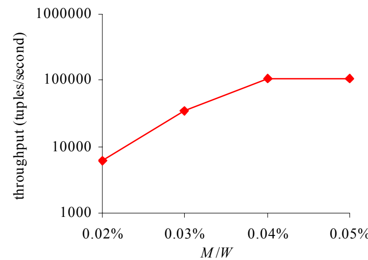

Figure 2. Throughput vs. storage budget.

Figure 2 shows the impact of M/W ratio on the throughput. The y-axis uses logarithmic scale. When M/W=0.05%, the throughput is over 100,000 tuples/second.

[5] is a full version of this paper that includes additional results and algorithm details. In summary, in all our 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.

References

[1] C.B. Barber, D.P. Dobkin, and H. Huhdanpaa. The Quickhull Algorithm for Convex Hulls. ACM Trans. Math. Softw. 22(4): 469-483, 1996.

[2] Y.C. Chang, L.D. Bergman, and V. Castelli et al. The Onion Technique: Indexing for Linear Optimization Queries. SIGMOD Conf. 2000: 391-402.

[3] C.S. Li, Y.C. Chang, and L.D. Bergman et al. Model-Based Multi-modal Information Retrieval from Large Archives. ICDCS International Workshop of Knowledge Discovery and Data Mining in the World-Wide Web, 2000.

[4] C.S. Li, Y.C. Chang, and J.R. Smith et al. SPIRE/EPI-SPIRE Model-Based Multi-modal Information Retrieval from Large Archives. MMCBIR 2001.

[5] G. Luo, K. Wu, and P.S. Yu. SAO: A Stream Index for Answering Linear Optimization Queries, full version. Available at http://www.cs.wisc.edu/~gangluo/onion_full.pdf, 2006.

[6] 2005 UC Data Mining Competition Homepage. http://mill.ucsd.edu, 2005.