An Optimal Strategy for Monitoring Top-k Queries in Streaming Windows

Di Yang, Avani Shastri, Elke A. Rundensteiner and Matthew O. Ward
Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA, USA.
diyang, avanishastri, rundenst, matt@cs.wpi.edu

ABSTRACT

Continuous top-k queries, which report a certain number (k) of top preferred objects from data streams, are important for a broad class of real-time applications, ranging from financial analysis to network traffic monitoring. Existing solutions for tackling this problem aim to reduce the computational costs by incrementally updating the top-k results upon each window slide. However, they all suffer from the performance bottleneck of periodically requiring a complete recomputation of the top-k results from scratch. Such an operation is not only computationally expensive but also causes significant memory consumption, as it requires keeping all objects alive in the query window. To solve this problem, we identify the "Minimal Top-K candidate set" (MTK), namely the subset of stream objects that is both necessary and sufficient for continuous top-k monitoring. Based on this theoretical foundation, we design the MinTopk algorithm that elegantly maintains MTK and thus eliminates the need for recomputation. We prove the optimality of the MinTopk algorithm in both CPU and memory utilization for continuous top-k monitoring. Our experimental study shows that both the efficiency and scalability of our proposed algorithm is clearly superior to the state-of-the-art solutions.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Query processing

General Terms

Algorithms Performance

Keywords

Streaming Data, Top-k Query, Optimality

1. INTRODUCTION

Motivation. Given a dataset and a preference function, a top-k query returns **k** objects with the highest preference function score among all objects. For example, a financial analyst may submit a top-k query to a stock transaction database asking for the top 100 most significant transactions (with largest $price \times volume$) in the year 2010. While most research efforts focus on supporting top-k queries in conventional databases [4, 8, 19], recently researchers started to look at the problem of tackling top-k query execution in the streaming context [15]. Given the infinite nature of data streams, window constraints are adopted to make top-k queries applicable to data streams. Top-k queries over streaming windows serve many important real-time applications ranging from financial analysis and e-marketing to network security. For example, a financial analyst may submit a continuous top-k query to monitor the 5 most significant transactions within the last ten minutes as potential key indicators of the most recent market trend changes. Or, a bidding-based hotel booking system, such as priceline.com, needs to continuously monitor the top k room bids with highest bidding prices, each from a traveler, where k equals the total number of rooms that its associated hotels have. Also, in network traffic analysis [15], analysts can discover the potential victim of an ongoing Distributed Denial of Service (DDoS) attack by monitoring the nodes with the top-k largest throughput.

In some applications, the top-k results are to be returned in ranked order based on preference scores, as the higher ranked objects may be more preferred. In the previous hotel room bidding example, although the booking system always monitors the top k bids, at any particular time point, only a subset of these k rooms, say k' ($k' \leq k$) rooms, will be available. The availability of the rooms, indicated by k', changes continuously over time, but will never exceed k. At any particular moment, only the top ranked k' bids with highest price from the travelers can be accepted by the system for booking. Therefore, in such scenarios, the top-k result needs to be returned in the ranked order of the preference scores.

Challenges. Efficient execution of continuous top-k queries in streaming environments is challenging. The techniques developed for top-k queries in conventional databases [4, 8, 19] cannot be directly applied nor easily adapted to fit streaming environments. This is because the key problem they solved is, given huge volumes of static data, how to pre-analyze the data to prepare appropriate meta information to subsequently answer incoming top-k queries efficiently [4, 8]. Streaming data however is dynamic with its characteristics dramatically changing over time. Given the real-time response requirement of streaming applications, relying on static algorithms to re-compute the top-k results from scratch for each window is not feasible in practice [15].

Therefore, the key problem to be tackled for continuous top-k query execution is to design a top-k maintenance mechanism that efficiently updates the top-k results even under extremely high input data rates and over huge query windows. The state-of-the-art technique [15] for this problem did not succeed to eliminate the key performance bottleneck of being forced to periodically recompute the top-k results from scratch.

Corresponding to the window sliding process, any incremental top-k maintenance mechanism needs to handle the impact of the *insertion* of new objects and of the *expiration* of existing objects to the top-k result set. The major challenge for designing such a mechanism lies in handling *expirations*, as they may frequently trigger the expensive recomputation process. More specifically, when the existing top-k objects expire from the query window, while the new objects fail to fill all the "openings" left by the expired objects, the recomputation process must now search for the qualified substitutes among *all objects in the window*. This recomputation process represents a serious bottleneck for top-k maintenance in terms of both CPU and memory consumption. Computationally, when searching for qualified substitutes for new top-k results, this process has to look at potentially all objects in the query window. Thus, the computational cost is close to or even equivalent to calculating the full top-k result from scratch. Memory-wise, as current non-top-k objects may qualify as top-k in future windows, such a recomputation process requires keeping and maintaining all objects alive in the window. This can be a huge burden on memory utilization. For queries that have large window sizes, the number of objects required to be stored can easily reach millions or higher, even when the actual number of objects a user is interested in is fairly small, say $k = 10$ or $k = 100$.

Proposed Solution. In this work, we design a solution that achieves optimal CPU and memory complexity for continuous top-k query monitoring. Our solution eliminates the need for any recomputation. It achieves not only incremental computation (recomputation-free), but also, due to its novel state encoding strategy, it realizes minimal memory utilization in the order of parameter k only and independent from the window size.

To tackle the recomputation bottleneck, we observe that, given the query window semantics, the life span of incoming objects is known upon their arrival. Thus, by analyzing the life span as well as the preference scores of the objects in the current window, we can pre-determine which subset of the current objects has the potential to contribute to the top-k results in future windows, and generate the "predicted top-k results" for a sequence of future windows. The objects in those "predicted top-k results" constitute the "Minimal Top-K candidate set" (MTK) for continuous top-k monitoring, which we prove to be the minimal set of objects that any algorithm has to keep for accurate query execution. We also show that the size of MTK is independent from the window size, and we prove it to be only 2k in the average case. As consequence, any object not belonging to MTK can be discarded immediately, resulting in significant savings in both CPU and memory utilization. By incrementally maintaining MTK, we eliminate the need for handling object expiration in top-k monitoring and successfully solve the recomputation bottleneck problem.

However, the straightforward MTK maintenance method is far from an efficient, not to mention optimal, solution for continuous top-k monitoring in many cases. Its performance in both CPU and memory utilization can be significantly affected by repeatedly storing overlapped top-k candidates for adjacent windows. Our experimental studies (Section 6) confirm such insufficiency of the straightforward MTK maintenance method.

To solve this problem, we design a compact MTK encoding method that enables the integrated representation of top-k candidates for all future windows. The integrated representation strategy does not only eliminate the redundant object references that need to be stored by the straightforward MTK maintenance method, but also guarantees constant time maintenance effort for each future window at the arrival of each new object. Based on this compact encoding, we present our proposed algorithm, MinTopk. In general, for each new object, MinTopk updates the top-k results in only logarithmic time in the size of MTK, $O(log(MTK.size))$ (Section 5). In the average case, such cost is only $O(log(k))$. We prove that MinTopk achieves optimal memory complexity for top-k monitoring in sliding windows. We also prove that MinTopk achieves optimal CPU complexity for this problem, when the top-k results are returned in a ranked order.

Our experimental studies on real data streams from stock transaction and moving object domains demonstrate that MinTopk saved at least 85% of the CPU time compared with the state-of-the-art solution [15], while using almost negligible memory space, in all our test cases. For example, when processing a top-k query with 1M-tuple window size and k equal to 1K tuple, our method only takes 10 seconds to update the query result for each window slide (100K new tuples), while [15] takes more than 2 minutes, which can hardly fulfill the requirement of real-time analysis (See Figure 11 in our Experimental Section). Put differently, our system can comfortably handle a 10K/sec data rate, while the state-of-the-art technique can barely catch up with a 1K/sec data rate.

The contributions of this work include: 1) We identify and formalize the "Minimal Top-K candidate set" (MTK), which is the minimal but sufficient set for monitoring top-k objects in sliding windows over data streams. We prove that in the average case the size of MTK is independent from the window size and linear in *k*. 2) We present a novel algorithm MinTopk, which eliminates a key performance bottleneck suffered by prior state-of-the-art solutions, namely full recomputation from scratch for the whole window. 3) We prove that MinTopk achieves optimal complexity in memory utilization for monitoring top-k objects in sliding windows. We also prove that the MinTopk algorithm achieves optimal CPU complexity for continuous top-k queries, when the top-k results are returned in a ranked order on preference scores. 4) Our experimental studies based on real streaming data confirm the clear superiority of MinTopk over all existing methods.

For better readability, the proofs for all lemmas and theorems, except the final proof for the optimality of our proposed algorithm, have been put into the Appendix.

2. PRELIMINARIES

2.1 Problem Definition

Given a dataset D and a preference function F, a top-k query $Q(D, F, k)$ returns **k** objects o_1, o_2, ..., o_k from D that have the highest $F(o_i)$ score among all objects in D. The preference function $F(o_i)$ can be a complex function on one or more attributes of o_i.

In the sliding window scenario [1, 17, 14], a continuous top-k query $Q(S, win, slide, F, k)$ returns the top-k objects within each query window W_i on the data stream S. This query window can be either time- or count-based. In either case, the query window has a fixed window size $Q.win$ and a fixed slide $Q.slide$ (either a time interval or an object count). The window periodically slides by $Q.slide$ from W_i to W_{i+1} (a certain amount of objects have arrived or a certain amount of time has passes) to include new objects from S and to remove the expired objects from the previous window W_{i-1}. The top-k results will be generated for the new window W_i only based on the objects alive in that window.

2.2 The Recomputation Bottleneck

Any incremental top-k monitoring algorithm over sliding windows needs to handle the impact of both including new objects and expiring old objects at each window slide. It is easy to see that handling the impact of a new object o_{new} on the current top-k results is straightforward. We can simply compare $F(o_{new})$ with $F(o_{min\_topk})$, where o_{min_topk} is the object in the current top-k set with the smallest F score. If $F(o_{new}) < F(o_{min\_topk})$, this new object will not change the top-k result, otherwise we include it in the top-k result set and remove o_{min_topk} from the top-k set.

However, handling the impact of expired objects on the top-k results is more expensive in terms of resource utilization. Specifically, when existing top-k objects are expired from the window, the incoming new objects may fail to re-fill all the "openings" in the top-k set. In such cases, the state-of-the-art technique [15] needs to recompute the qualified substitutes based on all objects in the window. To do this, two alternative strategies could be adopted: 1) Keep the whole window content sorted by F, and continuously maintain this sorted list with each insertion and expiration of all stream objects. 2) Recompute the top-k from scratch, while using index structure to relieve the computational intensity of recomputation. The state-of-the-art technique [15] adopts the second strategy.

No matter which implementation strategy is chosen, this top-k recomputation process clearly constitutes the key performance bottleneck in terms of both CPU and memory resource utilization. Computationally, when searching for the substitutional top-k candidates, it requires us to re-consider potentially huge numbers of objects. More precisely, all objects in the window need to be re-examined if no index structure has been deployed, or expensive index maintenance costs are needed to load and purge every stream object into and from the index . Alternatively, one could maintain a completely sorted list upon the whole window. As windows could potentially be huge in size, this would further increase the cost of the update process.

Memory-wise, as all objects in the window may later be reconsidered by the recomputation processes, the recomputation process requires complete storage of the whole window. Such memory consumption for queries with large window sizes (millions of objects or even more) can be huge, even when a query only asks for a small k, say 10 or 100. To the best of our knowledge, this recomputation bottleneck remains an open research problem, as no existing technique is able to completely eliminate it from the top-k monitoring process.

3. THEORETICAL FOUNDATIONS AND AN INITIAL APPROACH

3.1 Minimal Top-k Candidate Set

To solve the recomputation problem, we study the predictability property of sliding window semantics. In the sliding window scenarios, query windows tend to partially overlap ($Q.slide < Q.win$) [1, 15, 17]. For example, an analyst may monitor the top 10 significant transactions within the last one hour ($Q.win = 3600(s)$) with a result refresh rate of every minute ($Q.win = 60(s)$). Therefore, an object that falls into the window W_i will also participate in the sequence of the future windows W_{i+0}, W_{i+1} ... W_{i+n} until the end of its life span. Based on our knowledge about objects in the current window and the slide size, we can pre-determine the specific subset of the current objects that will participate (be alive) in each of the future windows. This enables us to pre-generate the partial query results for future windows, namely the "predicted" top-k result for each of the future windows. They would be based on the objects in the current window but have already taken the expiration of these objects in future windows into consideration. While this overlap property of window semantics has been considered for aggregation [14] and clustering [15, 18] queries, here we apply it for top-k computation.

Figure 1 shows an example of a top-k result for the current window and predicted top-k results for the next three future windows ($Q.win = 16$, $Q.slide = 4$). Each object is depicted as a circle labeled with its object id. The position of an object on the Y-axis indicates its F score, while the position on the X-axis indicates the time it arrived at the system.

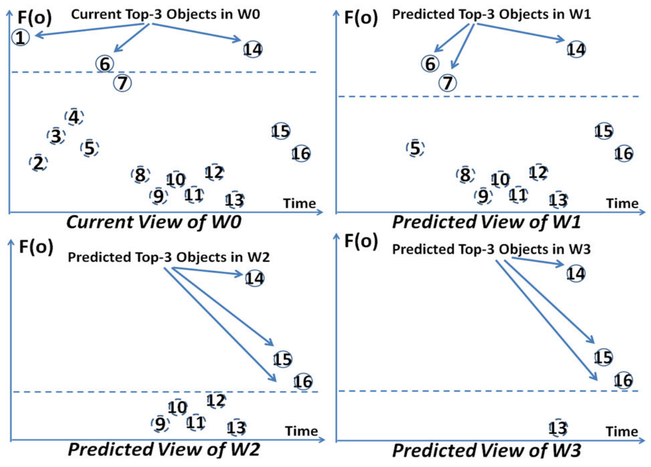

Figure 1: (Predicted) Top-k results four consecutive windows at time of W_0 (slide size = 4 objects)

Based on the objects in W_0, we not only calculate the top-k (k=3) result in W_0, but we also pre-determine the potential top-k results for the next three future windows, namely W_1, W_2 and W_3, until the end of life spans of all objects in W_0. In particular, the top-k results for the current window W_0 are generated based on all 16 objects in W_0, namely objects o_1 to o_{16}. While the predicted top-k results for future windows are calculated based on smaller and smaller subsets of objects in W_0, namely the predicted top-k results for W_1, W_2 and W_3 are calculated based on object o_5 to o_{16}, o_9 to o_{16} and o_{13} to o_{16} respectively. All other objects belonging to W_0 but determined to not fall into the (predicted) top-k results for any of these (future) windows (from W_0 to W_3 in this case) can be discarded immediately.

In the example shown in Figure 1, only the 7 objects within the predicted top-k results for the current and the three future windows (depicted using circles with solid lines) are kept in our system, while the other 9 objects (depicted using circles with dashed lines) are immediately discarded. The latter are guaranteed to have no chance to become part of top-k result throughout their remaining life spans. On the left of Figure 3, we list the predicted top-k results maintained for these four windows. Although these pre-generated top-k results are only "predictions" based on our current knowledge of the objects that have already arrived so far, meaning that they may need to be adjusted (updated) when new objects come in, they guarantee an important property as described below.

Theorem 3.1. *At any time, the objects in the predicted top-k result constitute the **"Minimal Top-K candidate set" (MTK)**, namely the minimal object set that is both necessary and sufficient for accurate top-k monitoring.*

The proof for Theorem 3.1 as well as all subsequent Lemmas and Theorems can be found in the appendix.

3.2 An Initial Approach: PreTopk

Now we first introduce the first step toward solving this problem, which is based on incremental maintenance of MTK. We call this approach PreTopk. When the window slides, the following two steps update the predicted top-k results. At step 1, we simply purge the view of the expired window. For example, as shown in Figure 3, the top-k result of W_0 in Figure 2 is removed, and W_1 becomes the new current view. This simple operation is sufficient for handling the object expiration. At step 2, we create an empty new predicted top-k result for the newest future window to cover the whole life span of the incoming objects. Using our example in Figure 3, the newest future window in this case is W_4. Therefore, each new object will fall into the current window and all future windows that we are currently maintaining. We thus update these predicted top-k results by simply applying the addition of each new object.

In particular, when a new object o_{new} comes in, we attempt to insert it into the predicted top-k result of each window. If the predicted top-k result of a window has not reached the size of k yet, we simply insert o_{new} into it. Otherwise we also remove the existing top-k object with the smallest F score once o_{new} is inserted. If it fails, namely the predicted top-k result sets of all future windows maintained have reached size k and $F(o_{new})$ is no larger than the F score of any object in them, o_{new} will be discarded immediately. Again, such computation is straightforward. Figure 2 shows the updated predicted top-k results of our running example (Figure 1) after the insertion of four new objects.

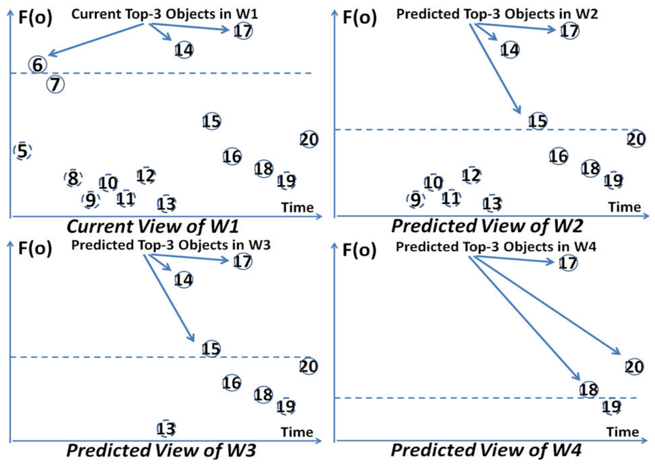

Figure 2: Updated predicted top-k results of four consecutive windows at time of W_1 (slide size = 4 objects)

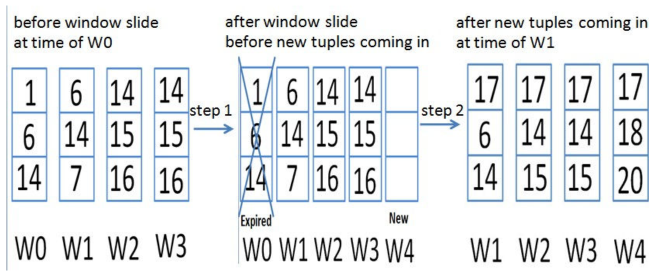

Figure 3: Update process of predicted top-k sets from time of W_0 to W_1

3.3 Cost Analysis and Limitations of PreTopk

The predicted top-k result for each future window can be organized using different data structures, such as a sorted list supported by a tree-based index structure or a min-heap organized on the F score of the objects. No matter which data structure is chosen, the best possible CPU costs for inserting a new object into a top-k object set and keeping the size of the top-k object set unchanged has complexity $O(log(k))$. More precisely, $log(k)$ for positioning the new object in the top-k object set, and $log(k)$ for removing the previous top-k object with the lowest F score. Thus the overall processing costs for handling all new objects for each window slide is $O(N_{new} * C_{nw\_topk} * log(k))$, with N_{new} the number of new objects coming to the system at this slide, and C_{ave_topk} [1] the average number of windows each object is predicted to make top-k when it arrives at the system. As the object expiration process is trivial, this constitutes the total cost for updating the top-k result at each window slide,

Memory-wise, PreTopk maintains predicted top-k results for C_{nw} [2] windows. The memory consumption of PreTopk is composed of two parts: first, the number of distinct objects stored in memory; second, the number of references to the objects in the predicted top-k results of all future windows. An object may appear in the predicted top-k results for multiple windows and thus needs multiple references. In the example shown in Figure 4, object 14 is predicted to be part of the top-k results in four windows, and thus four references to it are needed.

For the first part, PreTopk achieves the minimal number of objects to maintain for continuous top-k monitoring (Theorem 3.1). The size for this minimal set in the average case is analyzed below.

Theorem 3.2. *In the average case* [3]*, the number of distinct objects in predicted results for all future windows is* $2k$*.*

For the second part, the number of references stored by PreTopk is simply $C_{nw} * k$, as there are C_{nw} windows and k objects in each of them. The size of an object reference (Ref_{size}) is typically significantly smaller than the size of the actual object (Obj_{size}), especially when the object contains a large number of attributes. In summary, the average memory cost for PreTopk is $2k * Obj_{size} + C_{nw} * k * ref_{size}$.

Conclusion. PreTopk solves the recomputation bottleneck suffered by the state-of-the-art solutions [15]. Memory-wise, it only keeps the minimal number of objects necessary for top-k query monitoring, which is shown to be independent of the potentially very large window size (in Theorem 3.2). Computation-wise, the processing costs for generating the top-k result in each window are no longer related to the window size. This is a significant improvement over the state-of-the-art solution [15], because both the processing and memory costs of any solution that involves recomputation are related to the window size, which is usually an overwhelming factor compared to k or the slide size.

However, the **limitations** of PreTopk are obvious. The above cost analysis reveals that the performance of the PreTopk algorithm is affected by a constant factor, namely C_{nw}, the number of predicted top-k results to be maintained. More precisely, since PreTopk maintains the predicted top-k result for each window independently, both its CPU and memory costs increase linearly with

[1]When data is uniformly distributed, $C_{ave\_top-k} = \frac{2k}{3slide}$. We omit the derivation process due to the page limitations

[2]$C_{nw} = \lceil \frac{Q_i.win}{Q_i.slide} \rceil$ which is equal to the maximum number of windows a new object can be alive.

[3]Data is uniformly distributed on $F(o)$, indicating that the objects with different F scores have equal opportunity to expire after the window slides.

the number of predicted top-k results to be maintained (C_{nw}). Although C_{nw} is a constant, as it is fixed given the query specification and will never change during query execution, it can be large in some cases. For example, if a query Q has a window size $Q.win = 10000$ and a slide size $Q.slide = 10$, PreTopk maintains predicted top-k results for 1000 different windows. Our experimental studies in Section 6 confirm this inefficiency of PreTopk as the ratio between $Q.win$ and $Q.slide$ increases.

4. PROPOSED SOLUTION: MINTOPK

4.1 Properties of Predicted Top-k Results

To design a solution whose CPU and memory costs are independent not only from the window size but also the number of future windows to be maintained, we analyze the interrelationships among the predicted top-k results maintained by PreTopk.

Property 1: Overlap. We observe that the predicted top-k results in adjacent windows tend to partially overlap, or even be completely identical, especially when the number of predicted top-k results to maintain (C_{nw}) is large. We now explain the overlap property of the predicted top-k results across multiple windows.

Lemma 4.1. *At any given time point, the predicted top-k result for a future window W_i is composed of two parts: 1) $K_{inherited}$, a subset of predicted top-k objects inherited from the previous window W_{i-1} ; 2) K_{new}, the "new" top-k objects which qualify as top-k in W_i but not in W_{i-1}. $|K_{inherited}| \cap |K_{new}| = \emptyset$ and $|K_{inherited}| + |K_{new}| = k$. Then the following property holds:* ***For any object*** $o_i \in K_{new}$ ***and*** $o_j \in K_{inherited}$, $F(o_i) < F(o_j)$.

In the earlier example in Figure 1, at time of window W_0, objects 6 and 14 belong to $K_{inherited}$ of W_1, while object 7 belongs to K_{new} of W_1.

When C_{nw} is large, implying that the window moves a small step (compared to the window size) at each slide, only a small percentage of the objects will expire after each window slide. Then, the majority of the predicted top-k result of a window come from $K_{inherited}$. In the previous example, where $Q.win = 10000$ and $Q.slide = 100$, if k=500 objects, at least 80 percent of the predicted top-k objects in a window will be the same as those in the previous window (worst case). On average, this percentage will be even higher, as the expired top-k objects should be only a small portion of all the expired objects. In the example shown earlier in Figure 3, at W_0, the current top-k of W_0 and predicted top-k results of W_1 only differ in one object, and those of W_2 and W_3 are in fact exactly the same.

Property 2: Fixed Relative Positions. The relative positions between any two objects o_i and o_j in the predicted top-k result sets of different windows remain the same.

Since the F score for any object is fixed and the predicted top-k objects in any window are organized by F scores, o_i will always have a higher rank than o_j in any window in which they both participate, if $F(o_i) > F(o_j)$.

4.2 Solution: Integrated View Maintenance

Given the two properties identified in Section 4.1, we now propose an integrated maintenance mechanism for the sequence of predicted top-k results in future windows. As shown in the cost analysis in Section 3.3, the major processing costs for PreTopk to maintain top-k results lie in positioning each new object into the predicted top-k results of all future windows. Thus, our objective is to share the computation for the positioning each new object into multiple predicted top-k results (multiple future windows).

To achieve this goal, instead of maintaining C_{nw} independent predicted top-k result sets, namely one for each window, we propose to use a single integrated structure to represent the predicted top-k result sets for all windows. We call this structure the *super-top-k list*. At any given time point, this *super-top-k list* includes all distinct objects in the predicted top-k results of the current as well as all future windows. The *super-top-k list* is sorted by $F(o)$. Figure 4 shows an example of the *super-top-k list* containing the objects in predicted top-k results for four windows.

Next, we tackle the problem of how to distinguish among and maintain top-k results for multiple windows in this single *super-top-k list* structure. As a straightforward solution, for each object, we could maintain a window mark (a window Id) for each of the windows in which the object is part of its predicted top-k result set. We call this the *complete window mark strategy*. Using the example in Figure 1, at the time of W_0, object 14 needs to maintain four window marks, namely W_0, W_1, W_1 and W_3, as it is in the predicted top-k results for all these four windows. When an object is qualified for or disqualified from the predicted top-k result set of a window, we would respectively need to add a new or remove an existing window mark from it for the corresponding window.

This solution suffers from a potentially large number of window marks being maintained for each object. Thus, both the addition and removal process of window marks may require traversing the complete window mark lists. This is clearly not desirable.

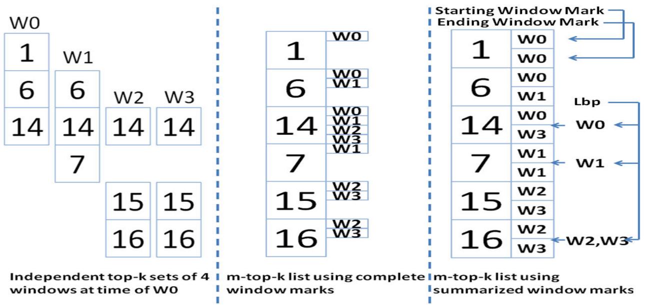
W0
W1
W2
W3
Starting Window Mark
Ending Window Mark
Lbp
W2,W3
Independent top-k sets of 4 windows at time of W0
m-top-k list using complete window marks
m-top-k list using summarized window marks

Figure 4: Independent top-k result sets vs. *super-top-k* structure using complete and summarized window marks

4.3 Optimal Integration Strategy based on Continuous Top-k Career

To overcome this shortcoming, we observe the following.

Lemma 4.2. *At the time of the current window W_i, the minimal F score of the predicted top-k objects in a future window $W_{i+n}(n > 0)$ is smaller than or equal to that of any window $W_{i+m}(0 \leq m < n)$, $W_{i+n}.F(o_{min\_topk}) \leq W_{i+m}.F(o_{min\_topk})$.*

Based on Lemma 4.2 we derive the lemma below.

Lemma 4.3. *At any given moment, if an object is part of the predicted top-k result for a window W_i, then at that moment it is guaranteed to be in the predicted top-k results for all later windows W_{i+1}, W_{i+2} ... W_{i+j} ($j \geq 0$), until the last window in its life span.*

This continuous top-k career property in Lemma 4.3 establishes the theoretical foundation for an innovative design of the *super-top-k* list. Namely, we design a more compact encoding for window marks of each object. In particular, for each object, as its "top-k career" is continuous, we simply maintain a *starting* and an *ending window mark*, which respectively represent the first and the last windows in which it is predicted to belong to top-k. As shown on the right of Figure 4, the first (upper) window mark maintained by each object is its starting window mark and the second (lower) one is its ending window mark. Clearly, the number of window marks needed for each object is now constant, namely 2, no matter in how many windows it is predicted to belong to the top-k result.

To enable us to efficiently decide in which windows a new object is predicted to make top-k result set when it arrives at the system, we also maintain one *Lower Bound Pointer (lbp)* for each window pointing at the top-k object with the smallest $F(o)$. When a new object arrives, we simply need to compare it with the object pointed by the *lbp* of each window. In the example shown in Figure 4, the *lbp* of W_0 and W_1 point to objects 14 and 7 respectively, while those of W_2 and W_3 both point to object 16. We call this the *summarized window mark strategy*. This is not only an important improvement in terms of memory usage but it significantly simplifies the top-k update process, as demonstrated below.

4.4 Super-Top-K List Maintenance

Handling Expiration. Logically, we simply need to purge the top-k result for the expired window from the *super-top-k* list. This task seems to be no longer as trivial as in the independent view storage solution (PreTopK), because now the top-k objects for different windows are interleaved within one and the same *super-top-k* list. We may need to search through the list to locate the top-k objects of the expired window. However, the following observation bounds such cost, indicating that searching is not needed.

Lemma 4.4. *At any given time, the top-k objects of the current to-be-expired window are guaranteed to be the first k objects in the super-top-k list, namely the ones with the highest F scores.*

Thus we can "purge" the first k objects on our *super-topk-list* without search. Note that purging here is only a logical concept to convey that these objects will no longer serve as top-k for this to-be-expired window. However, some of the top-k objects for the expired window may continue to be part of the predicted top-k results in future windows. We cannot simply delete them from the *super-top-k list* without considering their future role.

Instead, when the window slides, we implement purging of the expired window by updating the window marks of the first k objects in *super-top-k* list. More specifically, we increase the starting window mark by 1 for each of these objects. As the "top-k careers" of an object is continuous (Lemma 4.3), such update conveys that these objects will no longer serve as top-k for the expired window and their "top-k career" are predicted to start at the next window. If the starting window mark of an object becomes larger than its ending window mark, with the ending wondiw mark the latest window in which it can survive, we know that this object will have no chance to be part of the top-k result in its remaining life-span. We can thus physically remove it from the *super-top-k* list.

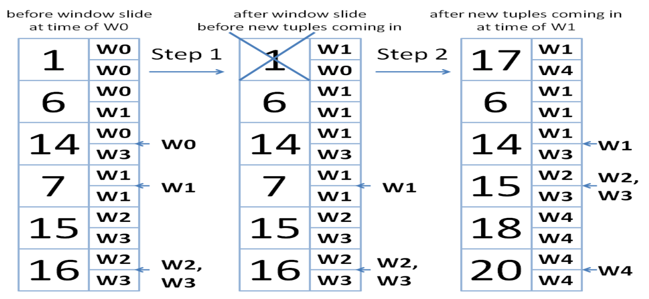

Figure 5: Update process of *super-top-k list* from W_0 to W_1

Handling Insertion. For the insertion of a new object o_{new}, we take two steps to update the *super-top-k* list. First, we position it into the *super-top-k* list. Second, we remove the object with the smallest F score from the windows that the new object is predicted to be part of their top-k results. For the first step, the positioning process has become fairly simple due to the support from the summarized window marks. In particular, for each object, if the predicted top-k result set of any future window represented by the *super-top-k list* has not reached the size of k yet, or if its F score is larger than that of any object in the *super-top-k* list, we insert it into the *super-top-k list* based on its F score. Otherwise it will be discarded immediately. If the new object is inserted into *super-top-k* list, which indicates that it is in the predicted top-k results of at least the last window in its life span, its ending window mark is set to be the window Id of this last window. The starting window mark of a new object is simply the oldest window on the *super-top-k list* whose $F(o_{min\_topk})$ (the F score of the object pointed by its lower bound pointer) is smaller than $F(o_{new})$. We find this window by comparing $F(o_{new})$ with $F(o_{min\_topk})$ of the oldest window on the *super-top-k* list, and keep comparing $F(o_{new})$ with that of the younger ones (with larger window Ids), until we find the oldest window whose $F(o_{min\_topk})$ is smaller than $F(o_{new})$, ($F(o_{min\_topk})$ monotonically decreases as window Id increases in Lemma 4.2). In the example shown in Figure 5, the F score of the new object 17 is larger than $F(o_{min\_topk})$ of all windows, from W_1 to W_4. Thus its starting window mark is set to W_1, indicating that its "top-k career" is predicted to start at W_1.

Second, for each window in which the new object is inserted into its predicted top-k result, one previous top-k object becomes disqualified and thus must be removed. Given that we have an *lbp*, for each window pointing to its top-k object with smallest F score, locating such disqualified object is trivial. For such a disqualified object, as it now serves in one less window as a top-k object, we simply increment its starting window mark by 1. Same as in the purging process, if its starting window mark now becomes larger than its ending window mark, we physically remove it from *super-top-k* list. Objects 7 and 16 are such examples in Figure 5.

4.5 Final Move Towards Optimality

Now we have the last but also the most challenging maintenance task left. We must design a strategy to efficiently redirect the lower bound pointer (*lbp*) of each window from which the object with the smallest $F(o)$ has just been removed. To do this, we need to locate the object that currently has the smallest $F(o)$ for each of those affected windows on the *super-top-k list*. On first sight, this task seems to require at least one complete search through the *super-top-k list* for each affected window. This is because the objects belonging to different windows are now interleaved in this integrated structure. Thus, we would have to search and locate the objects whose $F(o)$ scores used to be the second smallest one in each window. If so, the searches would make the redirecting process very expensive computationally and thus would significantly affect the overall performance of the MinTopk algorithm. To solve this problem, we carefully analyze the characteristics of the *super-top-k list* and discover the following important property.

Lemma 4.5. *For each window W_i whose predicted top-k result is represented by the super-top-k list, the object with the second smallest F score in its predicted top-k result, $W_i.o_{sec\_min\_topk}$, is always directly in front of (adjacent to) the object with the smallest F score in its predicted top-k result, $W_i.o_{min\_topk}$.*

Using Lemma 4.5, we can now conduct the redirection procedure effortlessly. We simply move the lower bound pointer of each affected window by one position up in the *super-top-k* list. Lastly, after the insertion of all new objects, the first k objects on the *super-top-k list* correspond to the top-k results for the current window and can be output directly. We call this proposed algorithm MinTopK. The pseudo code of MinTopK can be found in Figure 6.

4.6 Cost Analysis of MinTopK

CPU and Memory Costs in the General Case. The CPU processing costs of MinTopk to handle object expiration are $O(k)$, as we simply need to update the window marks of the first k objects on the super-top-k list. For handling the new objects, the cost for each object p_{new} is $P^{intopk} * (log(MTK.size) + C_{nw\_topk}) + (1 - P^{intopk}) * 1$ ($0 \leq P^{intopk} \leq 1$), with $MTK.size$ the size of MTK (the number of objects maintained in the *super-top-k list*) and P^{intopk} the probability that p_{new} will make the MTK set. In general, when p_{new} makes the MTK set (with P^{intopk} probability), the cost for positioning p_{new} into the *super-top-k list* is $log(MTK.size)$ with the support of any tree-based index structure. The cost for redirecting the lower bound pointers is simply equal to $C_{nw\_top\_k}$ [4], the number of windows that are affected by its insertion, because we only need to move that pointer for each affected window by one position (Lemma 4.5). Otherwise, with $1 - P^{intopk}$ probability, it will be discarded immediately with the cost of just a single check (comparing its F score with the minimal F score on the *super-top-k* list).

Lemma 4.6. *The CPU complexity for MinTopk to handle each new object is $O(log(MTK.size))$.*

Therefore, the CPU complexity of MinTopk to process each window is $O(N_{new} * (log(MTK.size))$ in the general case, with N_{new} the number of new objects coming in that window slide.

Memory-wise, MinTopk only needs a constant memory size to maintain each object in the MTK set.

Lemma 4.7. *The memory size required by MinTopk to maintain each object p_i in super-top-k list is of constant size, in particular, it is $(Obj_{size} + 2Ref_{size})$.*

Therefore, the memory complexity of MinTopk is $O(MTK.size)$ in the general case.

From the analysis above, we can observe that the size of the MTK, $MTK.size$, is a key factor affecting both CPU and memory costs of MinTopk. In the **best case**, $MTK.size$ equals to k. This would happen when the predicted top-k results for all future windows are identical. In the **worst case**, it is equal to the max size of each predicted top-k result set (k) times the number of windows maintained (C_{nw}), namely $C_{nw} * k$. This would mean that all predicted top-k objects expire after each window slide. However, this special case is highly unlikely in real streams. It could only happen when the F scores of the objects in a stream monotonically decrease across time and the slide size is at least as large as k.

Clearly, the average case is the most important one. In the **average case**, the size of the *super-top-k* is only $2k$, as we have proven that the average number of distinct objects in MTK is $2k$ in Lemma 3.2. This is comparable to the size of the final top-k result, which is equal to k. Thus, the average-case CPU complexity of MinTopk for generating the top-k results at each window is $O(N_{new} * log(k))$. The average-case memory complexity of MinTopk is $O(k)$.

[4]$C_{nw\_top\_k}$ is bounded by the constant C_{nw}, namely it is at most equal to C_{nw}, the total number of windows maintained.

5. OPTIMALITY OF MINTOPK

In this section, we prove the optimality of our proposed MinTopk algorithm in both CPU and memory utilization.

Theorem 5.1. *MinTopk achieves optimal memory complexity for continuous top-k monitoring in sliding windows. MinTopk also achieves optimal CPU complexity for continuous top-k monitoring in sliding windows, when the top-k results are returned in a ranked order based on preference scores.*

```
$o_i$: an object. $o_i.T$: object $o_i$'s time stamp.
$o_i.start\_w/.end\_w$: starting/ending window mark of $o_i$.
$W_i.T_{end}$ : ending time of a window $W_i$.
$W_{exp}$: the window just expired.
$W_{new}$: the newest future window.
$W_i.lbp$: lower bound pointer of $W_i$.
$W_i.tkc$: top-k object counter of $W_i$.
$o_{w_{i.lbp}}$: object pointed by lower bound pointer of $W_i$.
$o_{min\_suptopk}$ : object with smallest $F$ score on super-top-k list.

MinTopk ($S, win, slide, F, k$)
1 For each new object $o_{new}$ in stream $S$
2   if $o_{new}.T > W_{cur}.T_{end}$
      //slide window
3     OutputTopKResults();
4     PurgeExpiredWindow();
    // super-top-k list maintenance
5   UpdateSuperTopk ($o_{new}$)

OutputTopKResults()
1   output first k objects on super-top-k list;

PurgeExpiredWindow()
1   For first k objects ($o_{exp}$) on super-top-k list
2     $o_{exp}.start\_w++$;
3     If $o_{exp}.start\_w > o_{exp}.end\_w$
4       remove $o_{exp}$ from super-top-k list;
5   remove $W_{exp}$;
6   create a new future window $W_{new}$;
7   $W_{new}.tkc := 0$;
8   $W_{new}.lbp := o_{min\_suptopk}$ ;

UpdateSuperTopk ($o_i$)
1 If $F(o_i) < F(o_{min\_suptopk})$ AND All $W_i.tkc == k$
2   discard $o_i$ immediately;
3 Else position $o_i$ into super-top-k list;
4   For each $W_i$ that $F(o_{w_{i.lbp}}) < F(o_i)$
5     If $W_i.tkc < k$
6       $W_i.tkc++$;
5     Else $o_{w_{i.lbp}}.start\_w++$;
6     If $o_{w_{i.lbp}}.start\_w > o_{w_{i.lbp}}.end\_w$ ;
9       remove $o_{w_{i.lbp}}$ from super-top-k list;
10      move $W_i.lbp$ by one position in super-top-k list;
```

Figure 6: Proposed Solution: MinTopk Algorithm

Proof: **Memory-Optimality:** We have proven that the MTK set is the the minimal object set that is necessary for any algorithm to accurately monitor top-k objects in sliding windows in Lemma 3.1. We emphasize that this minimality holds in the general case, namely, given any unknown arrival rate distribution and preference score distribution of the input stream. Thus the optimal memory complexity of any top-k monitoring algorithm in sliding windows is at least $O(MTK.size)$.

Now we show that MinTopk achieves this optimal memory complexity. First, MinTopk only maintains one reference for each object in MTK set. Then, in Lemma 4.7, we have shown that the memory space needed by MinTopk for each object in the *super-top-k list* is $Obj_{size} + 2 * Ref_{size}$. Denoting the size of MTK by $MTIK.size$, then the memory cost of MinTopk is $MTK.size * (Obj_{size} + 2 * Ref_{size})$. Since Obj_{size} and Ref_{size} are both of constant size, the memory complexity of MinTopk is $O(MTK.size)$. This proves that MinTopk has optimal memory complexity in the general case.

CPU-Optimality. To prove that MinTopk algorithm achieves the optimal CPU complexity for generating the ranked top-k results at each window slide, we formalize this problem as P_{newk}.

Problem P_{newk}***:*** *Given two datasets D_{new} and D, which respectively represent the new object set for a window slide and the objects inherited from the previous window, $|D_{new}| = N_{new}$ and $|D| = N$. Each object o_i in D_{new} or D has a unique $F(o_i)$ score* [5]*. The objects in D_{new} and D are unsorted on $F_(o)$ score. The goal is to return a dataset K which is composed of k objects from $D_{new} \cup D$ which have the largest $F(o)$ scores in $D_{new} \cup D$ in ranked order of $F(o)$.*

Next we show that the problem P_{newk} is at least as hard as the following problem $P_{newk'}$.

Problem $P_{newk'}$***:****Given two datasets D_{new} and D_k, which respectively represent the new object set coming with a window slide and the existing top-k object set of D ($D_k \subseteq D$), $|D_{new}| = N_{new}$ and $|D_k| = k$. Each object o_i in D_{new} or D_k has a unique $F(o_i)$ score. The objects in D_{new} are unsorted on $F_(o)$ score. The goal is to return a dataset K which is composed of k objects from $D_{new} \cup D_k$ which have the largest $F(o)$ scores in $D_{new} \cup D_k$ in ranked order of $F(o)$.*

The problem P_{newk} is at least as hard as $P_{newk'}$, because any algorithm that solves P_{newk} has to consider any object $o_i \in D$, as any of them may be part of the new top-k results. However, given that D_k is the top-k object set in D, thus for any object o_i, if $o_i \in D$ but $o_i \notin D_k$, it cannot be in the new top-k results, because there are already k objects in D_k having larger $F_(o)$ scores than this o_i. Thus, any algorithm solving P_{newk} can solve $P_{newk'}$ also, indicating that P_{newk} is at least as hard as $P_{newk'}$.

Now we prove the lower bound of $P_{newk'}$ by showing that $P_{newk'}$ can be reduced to the *sorting based on comparison problem* [6]. In particular, first, Let A denote any algorithm that solves this problem. Then we give the following inputs to A, namely the input datasets D_{new} and D_k, in which for any $o_i \in D_{new}$ and $o_j \in D_k$, $F(o_i) > F(o_j)$ and $|D_{new}| = |D_k|$. It is easy to see that if A solves P_{newk} with the inputs above, A sorts D_{new}. This implies that $P_{newk'}$ can be reduced to *sorting based on comparison for D_{new}*, namely any algorithm A solving $P_{newk'}$ can be used to solve *sorting based on comparison problem.* It is well known that the lower bound for *sorting based on comparison problem* on a dataset of size n is $O(n * log(n))$ [6]. Therefore the lower bound of the $P_{newk'}$ is $O(N_{new} * log(k))$.

Since we have shown that any algorithm that solves the top-k problem in sliding windows with ranked top-k results returned is dealing with a problem at least as hard as $P_{newk'}$, we now have proven that the lower bound for any top-k monitoring algorithm for generating the top-k results in ranked order for each window is $O(N_{new} * log(k))$.

As we have shown in Lemma 4.6, MinTopk takes only $O(log(MTK.size))$ to process each new object that arrives within a window slide. Its CPU complexity to process each window is $O(N_{new} *(log(MTK.size))$. Now we must determine what is the size of MTK in the general case. In Section 4.6, we have shown that in the average case $MTK.size = 2k$, and even in the worst case, $MTK.size$ is bounded by a constant factor $C_{nw} = \frac{win}{slide}$. This indicates that no matter what the input rate and preference score distributions of the input stream are, the design of MTK guarantees that it contains at most $C_{nw} * k$ objects. Namely, $MTK.size = C_{nw} * k$ in the worst case. Therefore the CPU cost of MinTopk is $O(N_{new} * log(C_{nw} * k))$ in the worst case. Since C_{nw} is a constant that is known and will not change once the query is specified, the CPU complexity of MinTopk is $O(N_{new} * log(k))$. This proves that MinTopk achieves the optimal CPU complexity for generating ranked top-k objects at each window in the general case. ■

6. EXPERIMENTAL STUDY

Experimental Setup. We conducted our experiments on an HP G70 Notebook PC with an Intel Core(TM)2 Due T6400 2.00GHz processor and 3GB memory, which runs Windows Vista operating system. We implemented the algorithms in C++ using Eclipse.

Streaming Datasets. The first dataset we used is the Stock Trading Traces data (STT) from [11], which has one million transaction records throughout the trading hours of a day from NYSE. For this data, we use the amount of each transaction, namely the price times the volume, as the preference function F.

The second dataset, GMTI (Ground Moving Target Indicator) is provided by MITRE Corp. modeling troop movement in a certain area. It captures the information of moving objects gathered by different ground stations or aircraft in a 6-hour time frame. It has around 100,000 records regarding the information on vehicles and helicopters (speed ranging from 0-200 mph) moving in a certain geographic region. For this dataset, the preference function F is each target's distance from a stationary ground station calculated based on their latitude and longitude.

For the experiments that involve data sets larger than the sizes of these two datasets, we augment them to the required sizes by appending similar data after them. In particular, we append multiple rounds of the original data varied by setting random differences on all attributes, until it reaches the desired size.

Alternative Algorithms. We compare our proposed algorithm MinTopk's performance with two alternative methods, namely the state-of-the-art solution SMA [15] (Section 2.2), and the basic algorithm we presented in this work, PreTopk (Section 3).

Experimental Methodologies. For all alternatives, we measure two common metrics for stream processing, namely average processing time for each object (CPU time) and the memory footprint, indicating the maximum memory space required by an algorithm. We run each experiment 10 times (runs). Within each run, we process each query for 10K windows (slides for 10K times). The statistics results are averages from the 10 runs. To thoroughly evaluate the alternative algorithms, we compare their performance under a broad range of parameter settings and observe how these settings affect their performance.

Evaluation for Different k Cases. This experiment is to evaluate how the number of preferred objects, k, affects the performance of the three algorithms. We use the STT data. We fix the window size at 1M and slide size at 100K, while varying k from 10 to 10K. As shown in Figures 7 and 8, both the CPU and memory usage of all three alternatives increases as k increases. This is expected,

[5]The assumption on uniqueness of preference scores is a common assumption for top-k and most sorting related problems [6]. It is mainly for simplifying the problem definition. While our proposed techniques do not require such uniqueness of preference scores in query processing.

because the sizes of the key meta-data, namely the predicted top-k results for PreTopk and MinTopk (organized differently though) and the skyband for SMA, all increase linearly with k. However, both the CPU and memory usage of PreTopk and MinTopk are significantly less than those utilized by SMA.

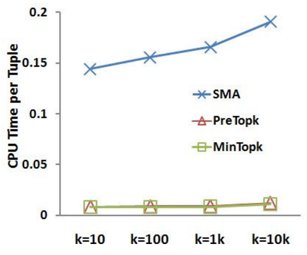

Figure 7: CPU time used by three algorithms with different k values

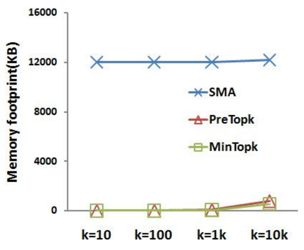

Figure 8: Memory space used by three algorithms with different k values

CPU-wise, both PreTopk and MinTopk saved at least 85% of the processing time for each object compared with that used by SMA in the four test cases. By further analyzing the specific components of the processing time, we found that SMA used a large portion (around 60%) of its processing time on loading and purging large numbers of objects from the grid file. Such cost is completely eliminated by both PreTopk and MinTopk, as they do not maintain any index for the whole window and the "useless" objects are discarded immediately upon their arrival at the system. Also, we found that SMA used around $10 - 30\%$ of the processing time for top-k recomputation in different cases. More specifically, its recomputation rate (number of recomputations divided by the number of window slides) increases from 23% to 56%, as k increases from 10 to 10K. This indicates that the recomputation process is more frequently needed in SMA when the ratio between k to the slide size increases. This is because, when k is large, it is more likely that the same amount of new objects cannot re-fill the skyband to reach the size of at least k. Again, such recomputation process is needed by neither PreTopk nor MinTopk. In general, the huge CPU time savings of PreTopk and MinTopk are achieved by eliminating the need for expensive index maintenance and top-k recomputation.

The memory consumption of both PreTopk and MinTopk is negligible compared with that used by SMA in all test cases, and especially when k is small. The reason is obvious. Namely, both PreTopk and MinTopk only keep the "necessary" objects (MTK), whose size is only $2k$ on average (see Theorem 3.2 in Section 3), while SMA needs to keep all 1M objects alive in the window. Our measurement of the size of MTK, namely the length of *super-top-k list*, also confirms Theorem 3.2, as in all four test cases, the size of MTK never exceeds $3.5k$ and is $2.4k$ on average.

The performance of MinTopk is also better than PreTopk in all these test cases. In particular, MinTopk uses on average 23% less processing time and 33% less memory. Such comparable performance of these two algorithms is caused by the relatively large $slide/win$ rate adopted in this experiment, which makes the number future windows maintained by both algorithms small (only 10 for all cases). We will further analyze this issue in experiment 3.

Evaluation for Different win Cases. Next, we evaluate the effect of the window size win on the algorithms. We use the GMTI data for this experiment. We fix the value of k at $1K$ and the *slide/window* rate at $\frac{1}{10}$, while varying win from $1K$ to $1M$. As shown in Figures 9 and 10, both the CPU and memory usage of PreTopk and MinTopk are significantly less than those utilized by SMA in all test cases, and especially when win is large.

In particular, both the CPU and memory usage of SMA increase dramatically as the window size increases. This is expected, because it requires full storage of all objects alive in the window and thus its memory usage increases almost linearly with the window size. The increase of the CPU time for SMA is also obvious, while less sharp than that of its memory utilization. This is because to process the same number of objects, no matter what the window size is, SMA needs to load and purge each object once into the index. Thus this part of the processing time is the same for all test cases. The increase of CPU time for SMA is primarily caused by the increasing cost of top-k recomputation in larger windows.

Both the CPU and memory usage of PreTopk and MinTopk are not affected by the window size. This confirms our cost analysis in Sections 3 and 4, namely the costs of PreTopk and MinTopk are independent from the window size.

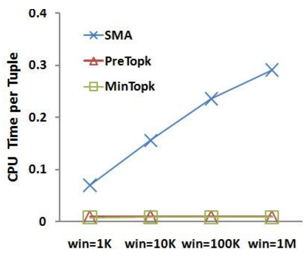

Figure 9: CPU times for varying window sizes

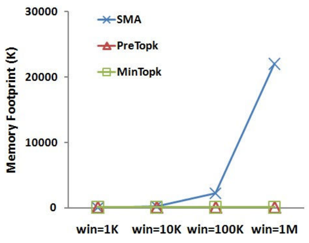

Figure 10: Memory space for varying window sizes

Evaluation for Different $slide$ Cases. In this case, we evaluate the effect of the window size win on the algorithms. We use the STT data for this experiment. We fix the value of k at $1K$ and window size at 1M, while varying the slide/window ratio from $0.01\% - 10\%$, namely the slide size from 100 to $100K$.

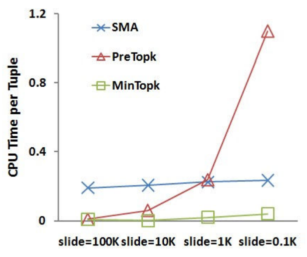

Figure 11: CPU times for varying slide sizes

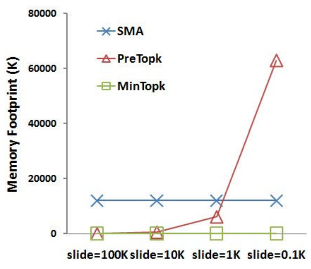

Figure 12: Memory space for varying slide sizes

As shown in Figures 11 and 12, both the CPU and memory usage of MinTopk are still significantly less than those utilized by SMA in all test cases. In particular, in the $slide = 100k$ case, MinTopk only takes 0.097 ms to process each object on average, while SMA needs 0.172 ms for each object. In terms of the response time needed for processing each window, this means that our method only takes around 10 seconds to update the query result for each window slide (100K new objects), while SMA needs more than 2 minutes, This is as expected and can be explained by the same reasons as in the previous test cases. However, an important observation made from this experiment is that the performance of PreTopk can be strongly affected by the *slide/win* rate. In particular, both the CPU and memory usage of PreTopk increase dramatically as the slide/window rate decreases. Its performance is comparable with MinTopk when the $slide/window$ rate is 10%, while it gets even worse than SMA when it decreases to 0.01%. This is as expected. Since PreTopk maintains the predicted top-k results for each future window independently, its resource utilization increases linearly with the number of future windows maintained, which is equal to $\lceil \frac{win}{slide} \rceil$ (see cost analysis in Section 3).

The performance of both SMA and MinTopk are not affected by the change of *slide/win* rate. Their average processing time even drops a little bit, because to process the same amount of objects, the larger slide size will cause less frequent output and thus requires less output cost.

In conclusion, although PreTopk shows comparable performance with MinTopk in the cases in which the *slide/win* rate is modest, its performance can be very poor when the *slide/win* rate is small. In short, the performance of MinTopk has been shown to be stable under any parameter settings.

Evaluation for Non-uniform Arrival Rate Cases. In this experiment, we evaluate the algorithms' performance under non-uniform arrival rate. Namely, the number of objects that arrive at each window slide varies in this case. We use the STT dataset and use the real time stamp of each transaction to present the arrival rates of the stream. The average arrival rate of the transactions in this data is around 400 transactions each second, while the actual arrival rate at each window slide varies significantly depending on the choice of slide size and the particular time period.

As in this experiment the number of objects needed to be processed for each window varies significantly, instead of measuring the average CPU time for processing each tuple, we measure the response time for processing each window, namely the accumulative time for answering the query at each window slide from all objects arrived to results outputted. We evaluate three different cases with slide size $slide$ equal to 1, 10 and 100 seconds respectively, while we fix the window size at 1000 seconds and k equal to 1K for all three cases. For each case, we run the query for 10K windows and we measure the minimum, maximum, average and standard deviation of the response time at each window.

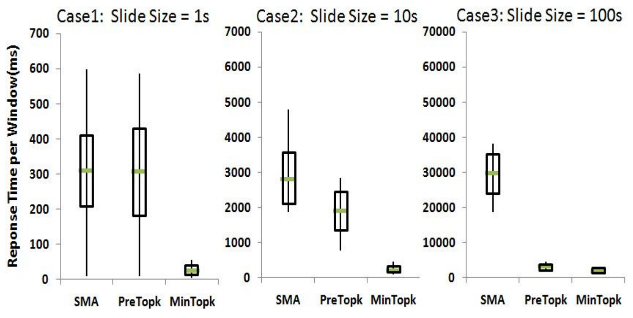

Figure 13: Response time for processing each window given non-uniform arrival rate.

As shown in Figure 13, we observe that given the non-uniform arrival rate, the response time of all three algorithms varies by slide sizes. In general, when the slide size is small, as in the $Slide = 1s$ case, the variations of the response time tend to be very large. This is because given very short time period granularities, the number of objects arriving at each window can vary significantly. As shown in the Case 1 in Figure 13, the minimum response time of all three algorithms are very close to zero. This is because in some windows very few objects arrived (less than 20), and thus they only required limited computation. While when slide size increases, as in the $Slide = 10s$ and $Slide = 100s$ cases, we can observe that the variations of the response time of all algorithms tend to be smaller. This is because the unevenness of the arrival rates in short time periods is now averaged in the larger time frames.

However, no matter what kind of variations exist in the arrival rate, our proposed MinTopk algorithm still shows obvious superiorities to the other two alternatives. Since the overall trends for all three competitor algorithms observed are similar to those trends for the uniform arrival rate cases, the results can again be explained using similar reasoning as given in the previous experiments.

7. RELATED WORK

The problem of top-k query processing was first studied for **static databases**. The well-known top-k algorithms on a single relation, such as $Onion$ [4] and $Prefer$ [8], are preprocessing-based techniques. They focus on preparing the meta-data through a pre-analysis of the data to facilitate subsequent run-time query processing. For example, $Onion$ [4] computes and stores the convex hulls for the data, and later evaluates the top-k query based on these convex hulls.

Previous works regarding top-k result generation based on joins over multiple relations include [9, 10]In these works, to optimize the system resources, top-k processing is usually interleaved with, rather than simply being built on top of, the join operation. For example, [9] proposes a pipelined algorithm which integrates the top-k processing into a hierarchy of join operators. [10] further explores the integration of rank-join operators into conventional database systems.

There is also work in computing the top-k results in distributed data repositories. These works focus on two technical issues: 1) To minimize the communication costs for retrieving the top-k candidates from distributed data repositories [2, 3, 5]. 2) To efficiently combine the top-k candidates reported from distributed data repositories [5, 16].

[19] presents a technique to incrementally maintain the materialized top-k views in static database when updates happen. The basic idea of their methods is simple: instead of maintaining a materialized top-k view, a larger view containing $k' > k$ objects is maintained. Thus the most expensive operation for top-k maintenance, namely recomputing the top-k results from the whole database, is only needed when the $top - k'$ view has less than k members. Thus it happens less frequently. However, as the proposed optimizations are designed to handle a single update or deletion to the database, it does not scale well for streaming applications in which large amounts of objects are inserted and deleted at every window slide.

In general, these techniques designed for the static environment are based on two assumptions. First, all relevant data is a priori available, either locally or on distributed servers, before query execution. Second, the top-k queries are ad-hoc queries executed only one single time. However, both assumptions do not hold for streaming environments in which data are continuously coming in and the top-k queries are continuously re-executed. Thus, clearly, these techniques cannot be directly used to solve our problem, namely continuous top-k monitoring in streaming environments.

More recently, researchers have started to look at the problem of processing **top-k queries in streaming environments** [15, 7, 13, 12]. Among these works, [15] is the closest to our work in that it also tackles the problem of exact continuous top-k query monitoring over a single stream. This work presents two techniques. First, the TMA algorithm computes the new answer of a query whenever some of the current top-k points expire. Second, the SMA algorithm partially precomputes the future changes in the result, achieving better running time at the expense of slightly higher space requirements. More specifically, as the key contributions of this work, SMA maintains a "skyband structure" which aims to contain more than k objects. This idea is similar to the one used in [19] for materialized top-k view maintenance. However, unfortunately, neither of these two algorithms eliminate the recomputation bottleneck (see Section 3) from the top-k monitoring process. Thus, they both require full storage of all objects in the query window. Furthermore, they both need to conduct expensive top-k recomputation from scratch in certain cases, though SMA conducts recomputation less frequently than TMA. While our proposed algorithm eliminates the recomputation bottleneck, and thus realizes completely incremental computation and minimal memory usage.

[7, 12, 13] handle incomplete and probabilistic top-k models respectively in data streams. while we work with a complete and non-probabilistic model. Thus, they target different problems from ours. In particular, the key fact affecting the top-k monitoring algorithm design is the meta information maintained for real-time top-k ranking and the corresponding update methods , which vary fundamentally by specific top-k models. For example, due to the characteristics of the incomplete top-k model, [7] proves that maintaining a object set with its size linear in the size of the sliding window is necessary for incomplete top-k query processing. While in our (complete) top-k model, we maintain a much smaller object set, whose size is independent from the window size but linear in the query parameter k only (see *Lemma* 3.2). Similarly, due to the characteristics of uncertain top-k models, [12, 13] maintain a significantly larger amount of meta information, namely a series of candidate top-k object sets (they call Compact Sets) that contain more objects than the Minimal Top-k Candidate Set (MTK) identified and maintained in this work. These candidate objects are organized and updated in different manners from us to serve their specific models. In general, those specific data structures and the corresponding update algorithms designed for other top-k methods are not optimized for our problem and thus do not achieve the optimal complexities in system resource utilization for our problem.

8. CONCLUSION AND FUTURE WORK

In this work, we present the MinTopk algorithm for continuous top-k query monitoring in streaming environments. MinTopk leverages the "predictability" of sliding window semantics to overcome a key performance bottleneck of the state-of-the-art solutions. By identifying and elegantly updating the minimal object set (MTK) that is necessary and sufficient for top-k monitoring, MinTopk not only minimizes the memory utilization for executing top-k queries in sliding windows, but also achieves optimal CPU complexity when returning the top-k results in a ranked order. Our experimental studies based on real streaming data confirm the clear superiority of MinTopk to the state-of-the-art solution.

In our future work, a major research direction is to study the multiple top-k query sharing problem in the streaming environments. We believe that the techniques proposed in this work, such as the "Minimum Top-k Candidate Set" (Section 3) and the "Integrated Top-k Candidate Set Encoding" (Section 4), can be easily extended to benefit multiple top-k query sharing as well.

9. ACKNOWLEDGEMENT

This work is supported under NSF grants CCF-0811510, IIS-0119276 and IIS-00414380. We thank our collaborators at MITRE, J. Casper and P. Leveille, for providing us the GMTI data stream.

10. REFERENCES

[1] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[2] B. Babcock and C. Olston. Distributed top-k monitoring. In *SIGMOD*, pages 28–39, 2003.

[3] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357, 2002.

[4] Y.-C. Chang, L. D. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: Indexing for linear optimization queries. In *SIGMOD Conference*, pages 391–402, 2000.

[5] S. Chaudhuri, L. Gravano, and A. Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE Trans. Knowl. Data Eng.*, 16(8):992–1009, 2004.

[6] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[7] P. Haghani, S. Michel, and K. Aberer. Evaluating top-k queries over incomplete data streams. In *CIKM*, pages 877–886, 2009.

[8] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *VLDB J.*, 13(1):49–70, 2004.

[9] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Joining ranked inputs in practice. In *VLDB*, pages 950–961, 2002.

[10] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware query optimization. In *SIGMOD Conference*, pages 203–214, 2004.

[11] I. INETATS. Stock trade traces. http://www.inetats.com/.

[12] C. Jin, K. Yi, L. Chen, J. X. Yu, and X. Lin. Sliding-window top-k queries on uncertain streams. *PVLDB*, 1(1):301–312, 2008.

[13] C. Jin, K. Yi, L. Chen, J. X. Yu, and X. Lin. Sliding-window top-k queries on uncertain streams. *VLDB J.*, 19(3):411–435, 2010.

[14] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD Conference*, pages 623–634, 2006.

[15] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD*, pages 635–646, 2006.

[16] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, pages 648–659, 2004.

[17] D. Yang, E. A. Rundensteiner, and M. O. Ward. Neighbor-based pattern detection for windows over streaming data. In *EDBT*, pages 529–540, 2009.

[18] D. Yang, E. A. Rundensteiner, and M. O. Ward. A shared execution strategy for multiple pattern mining requests over streaming data. *PVLDB*, 2(1):874–885, 2009.

[19] K. Yi, H. Yu, J. Y. 0001, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *ICDE*, pages 189–200, 2003.

APPENDIX

Proof for Theorem 3.1: (MTK) *Proof:* We first prove the *sufficiency* of the objects in the predicted top-k results for monitoring the top-k results. For each of the future windows W_i (the ones that the life span of any object in the current window can reach), the predicted top-k results maintain k objects with the highest F scores for each W_i based on the objects that are in the current window and are known to participate in W_i. This indicates that any other object in the current window can never become a part of the top-k results in W_i, as there are already at least k objects with larger F scores than it in W_i. So, they don't need to be kept. Then, even if no new object comes into W_i in the future or all newly arriving objects have a lower F score, the predicted top-k results would still have sufficient (k) objects to answer the query for W_i. This proves the sufficiency of the predicted top-k results.

Next we prove that any object maintained in the predicted top-k results are *necessary* for top-k monitoring. This would imply that this object set is the minimal set that any algorithm needs to maintain for correctly answering the continuous top-k query. Any object in the predicted top-k result for a W_i may eventually be a part of its real top-k result. This would happen if no new object comes into W_i or all new objects have a lower F score. Thus discarding any of them may cause a wrong result to be generated for a future window. This proves the necessity of keeping these objects.

Based on the sufficiency and necessity we have just proved, the objects in the predicted top-k results constitute the "Minimal Top-K candidate set" (MTK), namely the minimal object set that is necessary and sufficient for accurate top-k monitoring. ■

Proof for Theorem 3.2: (super-top-k size) *Proof:* Since PreTopk maintains predicted top-k results for C_{nw} windows, when data are uniformly distributed, $\frac{1}{C_{nw}}$ of objects in current window W_i will expire after each window slide. For the same reason, the same portion, namely $\frac{1}{C_{nw}}$, of top-k objects will expire after each window slide. Thus $\frac{k}{C_{nw}}$ "new" objects will be stored by the next window W_{i+1} as substitution, while the other top-k objects in the window W_{i+1} overlap with those in the previous window W_i. The same situation holds for each of the later windows until $W_{i+C_{nw}-1}$. As there are $\frac{k}{C_{nw}}$ new (distinct) objects for each window and we maintain C_{nw} windows, there are in total k distinct objects in the predicted top-k results besides the k objects in the current window. They add up to 2k distinct objects. ■

Proof for Lemma 4.1: *Proof:* If there exists an $o_i \in K_{new}$ with $F(o_i)$ larger or equal to $F(o_j)$ of any object $o_j \in K_{inherited}$, o_i will be in the predicted top-k results for the previous window W_{i-1} and thus $o_i \in K_{inherited}$. As $|K_{inherited}| \cap |K_{new}| = \emptyset$ by definition, this is a contradiction. Thus, there cannot exist any $o_i \in K_{new}$ and $o_j \in K_{inherited}$ such that $F(o_i) > F(o_j)$. ■

Proof for Lemma 4.2: *Proof:* Since some objects may expire after each window slide, the objects in the current window W_i that will participate in W_{i+n}, D_W_{i+n}, is a subset of those will participate in W_{i+m}, D_W_{i+m} $(m < n)$. Thus, the minimal F score of the top-k objects selected from the object set D_W_{i+n} in W_{i+n} cannot be larger than the minimal F score of the top-k objects selected from a super set of D_W_{i+n}, namely the object set D_W_{i+m} in W_{i+m}. ■

Proof for Lemma 4.3: *Proof:* The necessary and sufficient condition for an object o_i to appear in the predicted top-k result of a window W_i is that $F(o_i)$ is no less than the minimal F score of the predicted top-k objects in W_i, namely $F(o_i) \geq W_i.F(o_{min\_topk})$. Given o_i appears in the predicted top-k result of W_i, and we have shown that at any given moment $W_i.F(o_{min\_topk}) \geq$ $W_{i+j}.F(o_{min\_topk})$ (Lemma 4.2), we can infer that $F(o_i) \geq W_{i+j}.F(o_{min\_topk})$. This implies that o_i will also appear in the predicted top-k result of any W_{i+j} within o_i's life span. ■

Proof for Lemma 4.4: *Proof:* First, since the objects in the *super-top-k list* are sorted by F scores, the first k objects in *super-top-k list* are those objects with highest F scores within the whole list. Second, the top-k objects of the current window are selected from all the objects in the current window, while the predicted top-k objects for any future window are selected based on a subset of objects in the current window. Therefore, the top-k objects of the current window must be the ones with highest F scores among the current window. Their F scores cannot be lower than those of the other objects belonging to the predicted top-k results of future windows, which constitute the later part of the *super-top-k list*. ■

Proof for Lemma 4.5: *Proof:* This lemma can be proven by an exhaustive examination of all possible scenarios. By Lemma 4.1, we know that, at any given moment, the predicted top-k result set for a future window W_i, $W_i.topk$, is composed of two parts: 1) $K_{inherited}$, a set of inherited top-k objects from the previous window W_{i-1} ; 2) K_{new}, a set of "new" objects that qualify as top-k in W_i but did not in W_{i-1}. By Lemma 4.2, any object $o_i \in W_i.K_{new}$ has a lower F score than any object $o_j \in W_i.K_{inherited}$. For the current window W_i, the proof is straightforward. Since the top-k objects of the current window W_i are always the first k objects in *super-top-k list* (Lemma 4.4), and the objects on the *super-top-k list* are sorted by F scores, $W_i.o_{sec\_min\_topk}$ is in the $(k-1)^{th}$ position of *super-top-k* list, and $W_i.o_{min\_topk}$ is in the k^{th} position.

Now let us consider the next window right after W_i, namely W_{i+1}. There are four possible situations. 1) $W_{i+1}.o_{sec\_min\_topk}$, $W_{i+1}.o_{min\_topk} \in W_{i+1}.K_{inherited}$. It is easy to see that, in this case, $W_{i+1}.K_{new}$ is empty and the top-k objects of W_{i+1} are exactly the same as those in W_i. Thus these two objects are simply the same two top-k objects with the lowest F scores in W_i, and have been shown to be adjacent to each other in the case above. 2) $W_{i+1}.o_{sec\_min\_topk}, W_{i+1}.o_{min\_topk} \in W_{i+1}.K_{new}$. Since any object $o_i \in W_i.K_{new}$ has a lower F score than any object $o_j \in W_i.K_{inherited}$, we know that these two objects with lowest F scores in $W_{i+1}.K_{new}$ definitely have lower scores than any object in $W_{i+1}.K_{inherited}$. Thus no top-k objects in the previous window can be in between of them two. Also, any "new" predicted top-k object in the next window W_{i+2}, namely any object in $W_{i+2}.K_{new}$, must have a smaller F score than these two objects do, otherwise it would have already made top-k in W_{i+1} and thus would not be in $W_{i+2}.K_{new}$ but in $W_{i+2}.K_{inherited}$. So, no predicted top-k object of any later window can be in between of them. This proves the case for the second situation. 3) $W_{i+1}.o_{sec\_min\_topk} \in W_{i+1}.K_{inherited}$ and $W_{i+1}.o_{min\_topk} \in W_{i+1}.K_{new}$. This case is possible only if exactly one top-k object will expire from W_i. In this case, $W_{i+1}.o_{sec\_min\_topk}$ must be the last one in $W_{i+1}.K_{inherited}$, and $W_{i+1}.o_{min_topk}$ must be the only one in $W_{i+1}.K_{new}$. They are thus also adjacent to each other. 4) $W_{i+1}.o_{min\_topk} \in W_{i+1}.K_{new}$ and $W_{i+1}.o_{sec\_min\_topk} \in W_{i+1}.K_{inherited}$. This case is simply impossible, because any object $o_i \in W_i.K_{new}$ has lower F score than any object $o_j \in W_i.K_{inherited}$ (Lemma 4.1), but clearly $F(W_{i+1}.o_{min\_topk}) > F(W_{i+1}.o_{sec\_min\_topk})$. Now we have covered all four possible situations for W_{i+1}. We thus can prove that, at the same moment, $W_{i+j}.o_{sec\_min\_topk}$ and $W_{i+j}.o_{min\_topk}$ $(j > 1)$ in any future window are also in adjacent positions using the same method. ■

Proof for Lemma 4.6: *Proof:* We have shown that the CPU cost of MinTopk to handle each new object is
$P^{intopk} * (log(MTK.size) + C_{nw\_topk}) + (1 - P^{intopk})$ (See Section 4.6). No matter what is the probablity for the new object to make the MTK set (P^{intopk}), $log(MTK.size)$ is the dominant term in this cost expression. ■

Proof for Lemma 4.7: *Proof:* This includes the cost for storing both the raw data and the meta data. In particular, for each object in the *super-top-k list*, the memory cost for storing the object itself is Obj_{size}. Also, MinTopk maintains 2 window marks for it as meta data, which take $2Ref_{size}$ memory space. ■