MTopS: Scalable Processing of Continuous Top-K Multi-Query Workloads

Avani Shastri, Di Yang, Elke A. Rundensteiner and Matthew O. Ward
Dept. of Computer Science, WPI
Worcester, MA, USA
avanishastri, diyang, rundenst, matt@cs.wpi.edu

ABSTRACT

A continuous top-k query retrieves the k most preferred objects in a data stream according to a given preference function. These queries are important for a broad spectrum of applications ranging from web-based advertising to financial analysis. In various streaming applications, a large number of such continuous top-k queries need to be executed simultaneously against a common popular input stream. To efficiently handle such top-k query workload, we present a comprehensive framework, called MTopS. Within this MTopS framework, several computational components work collaboratively to first analyze the commonalities across the workload; organize the workload for maximized sharing opportunities; execute the workload queries simultaneously in a shared manner; and output query results whenever any input query requires. In particular, MTopS supports two proposed algorithms, MTopBand and MTopList, which both incrementally maintain the top-k objects over time for multiple queries. As the foundation, we first identify the minimal object set from the data stream that is both necessary and sufficient for accurately answering all top-k queries in the workload. Then, the MTopBand algorithm is presented to incrementally maintain such minimum object set and eliminate the need for any recomputation from scratch. To further optimize MTopBand, we design the second algorithm, MTopList which organizes the progressive top-k results of workload queries in a compact structure. MTopList is shown to be memory optimal and also more efficient in terms of CPU time usage than MTopBand. Our experimental study, using real data streams from domains of stock trades and moving object monitoring, demonstrates that both the efficiency and scalability of our proposed techniques are clearly superior to the state-of-the-art solutions.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms, Management, Performance

Keywords

Topk Queries, Stream, Multi-Query Optimization

1. INTRODUCTION

Motivation. Continuous top-k queries over streaming windows has been studied by several previous works [14, 15, 16, 17]. Such query semantics, which require continuous reporting of the k most preferred objects from the most recent portion of the stream, has been shown to be able to serve a broad range of real-time applications. For example, a web-site manager may want to continuously monitor the most "clicked" products within the last 1 hour on their web-site, and thus learn the trend of customers' interest. Also, financial analysts may want to continuously monitor the largest transactions in NYSE within the last 10 minutes, as those transactions may have significant impact on the future market change.

However, existing methods mainly focused on designing an efficient execution strategy for a single continuous top-k query. Little effort has been made to solve the problem of simultaneous execution of a large number of top-k queries submitted to the same input stream. In real-world applications, the need for handling a workload composed of multiple continuous top-k queries is clear for the following reasons: 1) Continuous top-k queries are parameterized, namely, when specifying such a query, an analyst needs to provide parameters, including the number of top preferred objects k, the window size and the refresh rate. 2) Thus, multiple analysts monitoring the same input stream may submit multiple continuous top-k queries yet using different parameter settings. For example, the stock transaction stream from NYSE is constantly monitored by thousands of financial analysts. Although many of them may be interested in monitoring the top-k largest transactions that happened most recently, They may submit top-k queries with different parameter settings due to their different analytical tasks. Some of them may want a relatively large k setting, say top 100 or 200, to capture many high volume transactions, while others may be interested in fewer top objects, say top 5 or 10. Also, some of them may specify a high refresh rate, say every 5 or 10 seconds, as they need to conduct frequent trades that need the most updated query results as their decision making support. Others, of course, may be satisfied with a moderate refresh rate, say every 5 or 10 minutes, as their applications require less frequent updates. 3) In fact, even a single analyst may submit multiple queries with different parameter settings with the intent to further analyze retrieved result sets so as to derive a well supported conclusion. For example, the web-site manager mentioned in the above example may want to know the most clicked products within last 3 hours vs. those within last 10 minutes, which may together better reveal the customers' interest changes. Therefore, a stream processing system should be able to accommodate a workload of large number of top-k queries.

Challenges. However, handling a large number of continuous top-k queries in a single system under high input rate is a challenging problem. The naive method of executing each of the queries independently for a huge workload has prohibitively high demands on both computational and memory resources. The state-of-the-art single continuous top-k query processing method [16] may take 10s to update the query result for a window containing 1M tuples tuples. If one simply uses this method to independently execute 100 continuous top-k queries in single system, the response time of each query suddenly increases to around 1000s, which is definitely unaccpetable for streaming applications. Thus, given the real time responsiveness requirement, we need to design efficiently shared execution strategies, which are able to effectively share the both the memory and CPU utilization among multiple queries.

Proposed Solution. To solve this problem, We present a comprehensive framework, called MTopS for **M**ulti **T**op-k **O**ptimized **P**rocessing **S**ystem. Within this framework, we introduce several innovations for optimizing multiple top-k query processing by effectively sharing the available CPU and memory resources.

1) First, a Meta Query Analyzer (MQA) is proposed in MTopS to analyze the workload at the compilation stage. It generates a single meta query to represent the query logic for all workload queries, and thus succeeds to remodel the problem of executing multiple top-k queries into the execution of a single query meta query for our system. Technical details of MQA algorithms are discussed in Section 4.

2) A Runtime Infrastructure (RINF) is designed in MTopS to physically store the progressive top-k results for the meta query (generated by MQA) in highly compact formats. Our RINF designs guarantee that it only holds the minimum necessary set for multiple top-k query execution, and thus achieve optimal memory usage. Infrastructures' design is discussed in Section 5 and 6.

3) A Runtime Multiple Query Scheduler (RMQS) in MTopS manages the timing for updating the progressive top-k results in RINF and generating query results for individual queries.

4) Novel execution strategies are provided in a Meta Query Executor (MQE) to efficiently update the progressive top-k results in RINF over time. MQE algorithms are also discussed in Section 5 and 6.

5) Finally, a Query Result Extractor (QRE) extracts top-k results for individual queries from RINF whenever needed.

Our experimental studies using real streaming data from moving object monitoring and stock trades domains show that our system uses at least 271 times less CPU processing time and 175.4 folds less memory space as compared to the State-of-the-art independent execution solution[16] for handling 1000 queries. In fact, our MTopS system comfortably handles a workload in the order of 1000 queries with the average processing time ranging between 4-30 ms/object depending on the query parameter settings, which satisfies the needs for most real-time applications.

.

2. PROBLEM DEFINITION

Top-k Queries in Sliding Windows. In a sliding window scenario, the continuous top-k query Q (S,*win*,*slide*,*k*) returns top-k objects within each query window W_i on the data stream S. We use the term 'object' to denote a multi-dimensional tuple in the input data stream. The objects that participate in the top-k results of a given window are referred to as the 'top-k elements' of that window. A query window is a sub stream of objects from stream S that can be either count-based or time-based. The window *win* periodically slides after a fixed amount of objects have arrived (count-based) or a fixed time has passed (time-based) to include new objects from S and to remove expired objects from the previous window W_{i-1}. The top-k results are always generated based on the objects that are alive in the current window W_i.

Multiple Top-k Queries. Given a query workload *WL* with n top-k queries Q_1(S,win_1,$slide_1$,k_1), Q_2(S,win_2,$slide_2$,k_2),..., Q_n(S,win_n,$slide_n$,k_n) querying the same input data stream S while all the other query parameters, i.e, *win*, *slide*, *k* may differ.

We focus on executing all the registered queries simultaneously such that each query is answered accurately at their respective output moments. More specifically, we continuously output the required top-k results for each query at their corresponding slide sizes. Our goal is to minimize both the average processing time for each object and the peak memory space needed by the system.

3. MTOPS FRAMEWORK

We now introduce the architecture of the MTopS framework shown in Figure 1, while details of the techniques used in each block are discussed in the subsequent sections 4 to 8.

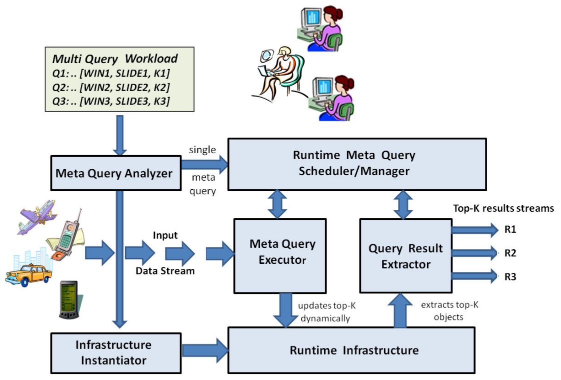

Figure 1: MTopS System Architecture

Multi Query Analyzer (MQA). The functionality of MQA is to analyze the similarity among the member queries in the workload, and thus organize them at the compilation stage with the goal of maximizing the resource-sharing for the later runtime execution. In particular, we propose to use a "meta query strategy", which builds a single meta query Q_{meta} to integrate all the member queries in the given workload. Namely, the input of MQA is a workload of top-k queries with arbitrary parameter settings, and the output of MQA is single meta query.

The meta query Q_{meta} has the following key characteristics. 1)The query window of Q_{meta} always covers all objects in the stream that are necessary to answer every member query. 2) The slide size of Q_{meta} is no longer fixed but rather adaptive during the execution, depending on the nearest time point that any member query needs to output or to conduct a new window addition or expired window removal. The specific algorithm of building such a meta query is discussed in Section V.

Runtime Infrastructure (RINF) and Its Instantiator (IINS). To execute the meta query generated by MQA, we need an infrastructure to physically hold the meta data, namely the top-k candidates, during the meta query execution.

In this work, we propose two runtime infrastructure designs, which not simply collect the top-k candidates, but also encode them into efficiently updatable formats. These two designs are the MTopBand and MTopList structures. We prove that by using those carefully designed structures, we maintain the minimum object set that is necessary and sufficient for answering all member queries, while any unnecessary object can be discarded immediately when it arrives at the system.

RINF is instantiated by its Instantiator (IINS) at the compilation stage.

Runtime Meta Query Scheduler (RMQS). As we discussed earlier in Multi-Query Analyzer, Q_{meta} needs to adapt its slide size to meet the time points for output, to build new windows or delete expired windows, for member queries.

RMQS sends instructions to MQE and QRE at scheduled window-addition/deletion time points or output time points, and thus tells them to conduct the corresponding operations at proper time. Such instructions guarantee that the RINF is properly updated and the top-k results of all member queries are output as the queries demand.

Meta Query Executor (MQE). MQE is the key online computation module which executes the meta query Q_{meta} by incrementally updating the top-k candidates held in RINF as the input stream passing by. Such update process include two aspects, namely handling the newly arrived objects and purging the expired objects.

When handling newly arrived objects, for each new object o_{new}, MQE first evaluates whether it has the potential to appear in the output of Q_{meta}. o_{new} is inserted only if it is eligible to participate as top-k result, otherwise discarded immediately to avoid unnecessary computation and storage.

When purging expired objects, MQE checks which objects are “completely expired" for the meta query, meaning that they are no longer in the query window of any workload queries. Such objects are physically removed from RINF immediately, while those expired for only some of the queries will still be kept in RINF. In this case MQE updates the properties of RINF.

Query Result Extractor (QRE). The functionality of QRE is to extract the top-k results from RINF for each member query at the moment when the output of this particular query is needed. This result extraction process is non-trivial, because the top-k candidates for all member queries are encoded in a single data structure in RINF.

4. ANALYZING THE MULTI TOP-K QUERY WORKLOAD

We now discuss our queries' parameters analysis that transforms the workload of many queries into single meta query.

Notion of Predicted Views. It is well recognized that in the sliding window scenario, query windows tend to partially overlap (Q_{meta}.*slide* < Q_{meta}.*win*). Therefore, if an object participates in the top-k result of window W_i, it may also participate in the top-k results of some of the future windows W_{i+1}, W_{i+2}, , W_{i+n} until the end of its life span. Thus based on our knowledge at time W_i, and the slide size *slide*, we can exactly ŞpredictŤ the specific subset of the objects in the current window that will participate in each of the future windows. We call these predicted subsets of future windows as “predicted views”.

With this knowledge, we can predetermine (partial) top-k results for each of these future windows based on the objects in the current window after considering the object expiration. We call these partial results as “predicted top-k results”. Thus, these predicted top-k results need to be updated only if a new object that arrives to the system is eligible to participate as a top-k result. Otherwise, these predicted top-k result sets will be the actual result sets for future windows. Figure 2 (left) shows the current window W_0 and predicted views of two future windows W_1 and W_2 with window size *win* = 12 and the slide size *slide* = 4. The predicted view W_1 contains those objects from W_0 those are still alive after the window W_0 slides. Likewise, predicted view of W_2 contains all objects that are still alie after window W_1 slides. In Figure 2 (left), the numbers shown in the white circles represent the objects' scores. Figure 2 (right) shows the corresponding current and predicted top-k result sets for windows W_0, W_1, and W_2 respectively. Evidently, each of these result sets contain k objects, with highest preference scores, that are still alive.

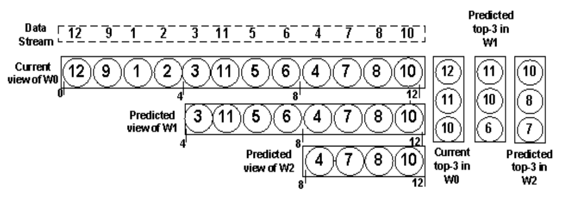

Figure 2: Predicted views of three consecutive windows at W_0

We exploit this “predictability property” of sliding windows to analyze the parameters of all the workload queries and finally integrate them. Next, we handle the cases where only one of the parameters among all the queries is varying. Later, we discuss more general cases, where two or more query parameters are varying.

4.1 Varying Top-k Parameters - k

Consider the window parameters, i.e., *win* and *slide* are same for all queries in the input workload WL, while the top-k parameter *k* is different. This implies that all queries share windows and require output at the same time, while the number of objects to be output by each query differs.

Lemma 4.1. *Given a workload WL with all member queries having same slide size - slide and same window size - win but arbitrary top-k parameters k, Q_i.k maintained in each of the predicted view will be sufficient to answer each query such that Q_i.k is the query with largest top-k parameter among WL.*

Proof. Lemma 4.1 holds because the predicted views built for the different queries in the workload are overlapped as the win and $slide$ values are same for all the queries. This means that the life time of an object and the output schedules for all queries are same.

Thus if objects equivalent to the largest top-k parameter are maintained in a predicted view , it is sufficient to answer the queries with smaller top-k parameter as well.

Thus, the number of predicted views that need to be built for processing full workload which might include thousands of input queries is equal to predicted views needed to be built for processing any one query in WL. Clearly, full sharing is achieved.

4.2 Varying Slide Sizes - *slide*.

Consider *win* and *k* for all queries in the workload WL are same, while their slide sizes - *slide* may differ. For ease of explanation, let us assume that all the queries start simultaneously. Since their window sizes are equal , at any given time they are querying the same portion of the input data stream. The only difference then is each query needs to generate output at different moments.

Example 4.1. *Given three queries Q_1, Q_2, Q_3 such that Q_1.win = Q_2.win = Q_3.win = 8s; Q_1.slide = 6s, Q_2.slide = 2s, and Q_3.slide = 3s; and Q_1.k = Q_2.k = Q_3.k = 3. Thus, each query requires its output, i.e., top-k result set at every 6, 2, and 3 seconds respectively.*

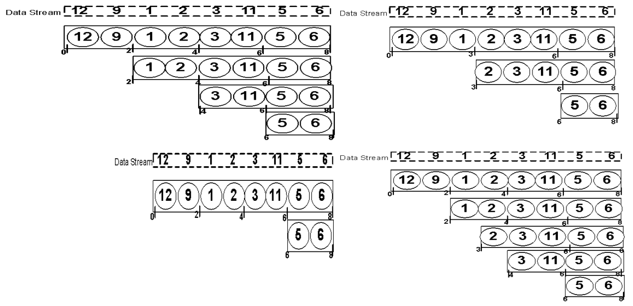

Figure 3: Predicted views needed for processing query Q_1 (top left), Q_2 (top right), Q_3 (bottom left) independently and combined view for meta query Q_{meta} (bottom right)

As consequence, each of these queries may need to maintain different predicted views so as to generate output at their respective slides. Instead, MQA builds a single meta query Q_{meta} that integrates all member queries in workload *WL* to avoid maintaining separate set of predicted views for each query. Figure 3. shows the predicted views that need to be maintained for each of these three queries independently, versus those by the meta query at wall clock time 00:00:08. Q_{meta} has the same window size as all the member queries in *WL* while its slide size is no longer fixed but rather adaptive during the execution. The slide size of Q_{meta} at a particular moment is the nearest moment at which at least one of the queries need to be answered.

For three member queries, MQA builds a meta query Q_{meta} with *WIN* = 8s. At wall clock time 00:00:08, the slide size of Q_{meta} will be 2s as 00:00:10 will be the nearest time at which the member query Q_2 is to be answered. At 00:00:10, its slide sizes are adapted to 1s, 1s and 2s so to output at 00:00:11 (Q_3), at 00:00:12 (Q_2), and at 00:00:14 (Q_1 and Q_2) respectively. Thus, we can now build up all predicted views at 00:00:08 with distinct output points as determined by the meta query. That is we build 6 predicted views starting at 00:00:02,:03,:04,:06, :08 respectively, most of which sere multiple queries.

4.3 Varying Window Sizes - *win*

Consider the slide sizes - *slide* and the top-k parameter - *k* of all queries are same, while their window sizes *win* differs. Here, we first use the simplifying assumption that all the window sizes of the member queries are multiples of their common slide size. We now observe an important characteristic as below.

Lemma 4.2. *Given a workload WL with queries having the same slide but arbitrary win (multiples of slides), the predicted views maintained for Q_i with Q_i.win the largest window size among WL will be sufficient to answer all input queries in WL.*

This is because the predicted views maintained for Q_i will cover all the predicted windows that need to be maintained for all the other queries.

Discussion. If the window sizes of the queries are not in multiples of their common slide size, the predicted views maintained for Q_i will still cover all the other queries. For example, if the slide sizes of each of the queries are the same as above (2s) while the window sizes are Q_1.*win* = 6s, Q_2.*win* = 7s, and Q_3.*win* = 8s. The predicted views built at moment 00:00:08 will be sufficient to answer all these queries. These windows will start from 00:00:00 (serving Q_3), 00:00:01 (serving Q_2), 00:00:02 (serving Q_1 and Q_3), 00:00:03 (serving Q_2), and 00:00:04 (serving Q_1 and Q_3) and so on.

4.4 Varying Window sizes-win and Varying Slide Sizes-$slide$

Next we consider, when both the window sizes *win* and the slide sizes *slide* of all the member queries are arbitrary. Here, we utilize a combination of previously introduced techniques and show that a single meta query with window size equal to the largest window size amongst all the member queries and adaptive slide sizes is sufficient to answer all such queries.

Example 4.2. *Consider, Q_1.win = 8s, Q_2.win = 6s and Q_3.win = 4 s; Q_1.slide=4s, Q_2.slide=3s, Q_3.slide = 2s; and Q_1.k = Q_3.k = Q_3.k = 2. Assuming that all the predicted views for the queries end at the largest window size, we build a meta query Q_{meta} such that Q_{meta}.WIN = 8 and Q_{meta}.SLIDE = ADAPTIVE, Q_{meta}.K= 2 (same for all queries).*

Thus, in this meta query setup, the window size and top-k parameter are now fixed while the slide size of the meta query is adaptively adjusted. At wall clock time 00:00:08, 5 predicted views are created, starting from 00:00:00 (serving Q_1), 00:00:02 (serving Q_2), 00:00:04 (serving both Q_1 and Q_3), 00:00:05 (serving query Q_2), and 00:00:06 (serving query Q_3). Clearly, only 5 windows (current and predicted) need to be maintained instead of the 9 windows that would here been needed if each query were to be executed independently.

4.5 The Most General Case

Finally, we consider the general case with all queries with arbitrary parameter settings. In this case, we build a meta query with window size Q_{meta}.*WIN*= Q_i.*win*, the largest window size among WL; Q_{meta}.*SLIDE* = *ADAPTIVE*, as explained in the previous subsection. Lastly, Q_{meta}.*K* = *ADAPTIVE* as explained below.

We now introduce an adaptive *k* strategy to achieve memory efficient processing. To be more precise, in a particular window we maintain the top-k objects such that k is equal to Q_i.*k* where Q_i is the query served by that window. In case one window serves more than one query then *k* for that window is equivalent to the largest top-k parameter among the queries served by his window.

5. THE MTOPBAND STRUCTURE

Once the single meta-query Q_{meta}, has been designed that logically encapsulates a full workload of queries, we instantiate a runtime infrastructure for managing the meta data needed for execution of Q_{meta}. We call this infrastructure the MTopBand.

5.1 MTopBand Design

The MTopBand data structure stores only the top-k objects for the current and those for each of the predicted views, as generated by the meta query Q_{meta}. These predicted views, as discussed in the Section III, are generated based on the meta query logic and thus represent all the member queries in the workload WL.

For each predicted view only a list of top-k objects is maintained, while all other objects that have no chance of participating in the top-k results of current or any of the future views are discarded immediately. We recall that top-k parameter $Q_{meta}.K$ is adaptive based on the query/queries that require output at the moment when a particular window ends. Thus, for each window we maintain only those top-k tuples eligible to be the output for one or more queries at the time point when the window slides. This means, each window may have different number of tuples as the top-k result sets, depending on the the query in the workload that outputs when the window slides. Each of these result sets are sorted based on the object scores F_{scores}.

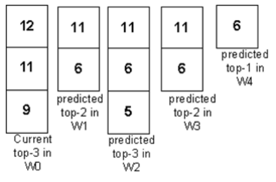

Figure 4: Physical view of MTopBand structure

Figure 4. shows the MTopBand structure based on the workload WL of three queries Q_1, Q_2, and Q_3 introduced in Example 4.2.

Theorem 5.1. *At any time, the top-k result set maintained in the MTopBand structure constitute the minimal object set that is necessary and sufficient for accurate top-k monitoring.*

Proof. We first prove the sufficiency of the objects in the predicted top-k result sets for monitoring the real time top-k results for each of the queries in the workload WL. 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 $Q_{meta}.W_i.K$ 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 $W_i.K$ objects with larger F scores than it in W_i. So, they donŠt 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 ($Q_{meta}.W_i.K$) objects to answer the query Q_i 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 accurate top-k monitoring. This would imply that this object set is the minimal set that any algorithm needs to maintain for correctly answering all the top-k queries in the given workload WL. Any object in the predicted top-k result for a window W_i may eventually be a part of its actual top-k results for one of the queries 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 each of these objects. Based on the sufficiency and necessity we have just proved, the objects in the predicted top-k results constitute namely the minimal object set that is necessary and sufficient for accurate top-k monitoring of all queries in the workload WL.

5.2 MTopBand Maintenance.

The dynamic maintenance of the MTopBand structure requires updating the top-k results for each of the current and predicted views, that include all the queries in the WL, in two scenarios. First, when a new object arriving at the system is eligible to participate in the top-k result sets for one or more queries being served by one or more windows W_i . Secondly, when a window slides some of the objects in the existing top-k result sets may expire and thus require updating the MTopBand data structure. Next, we discuss the proposed algorithms to update the MTopBand structure in the above two scenarios. If a window slides, we update the MTopBand top-k

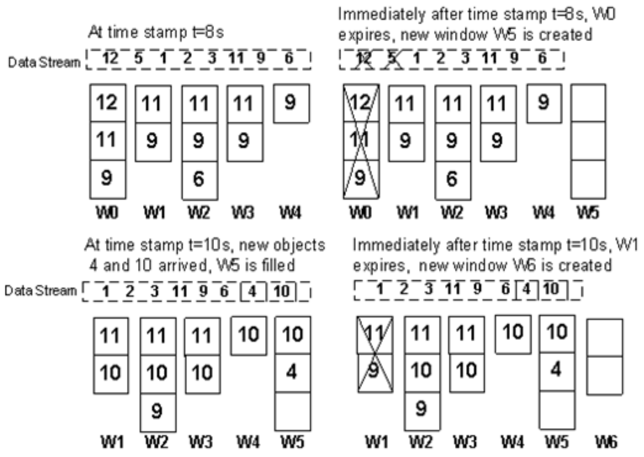
At time stamp t=8s
Immediately after time stamp t=8s, W0 expires, new window W5 is created
Data Stream
At time stamp t=10s, new objects 4 and 10 arrived, W5 is filled
Immediately after time stamp t=10s, W1 expires, new window W6 is created
W0 W1 W2 W3 W4 W5 W6

Figure 5: Updating the multi top-k results in MTopBand

result sets in the following two steps.

At step 1, we remove the top-k result set corresponding to the expired window. For example, Figure 5. depicts the MTopBand structure maintenance based on our running Example 4.2. After time t = 8s, when current window W_0 expires, top-k results of W_0 are purged, and W_1 is the new current window. It is easy to see that the effect of window expiration was already taken into account while building the predicted views/ predicted future windows.

At step 2, we create a new empty MTopBand top-k result set corresponding to the newest predicted view (W_5 in figure 5. (top right)) for the next future window to cover the whole life span of the incoming objects.

After the window slide is taken care of, we attempt to insert the newly arrived object O_{new} in each of the current and future window. The F_{score} of each O_{new} is compared with the object with minimum F_{score}, called as O_{min} henceforth, in each of the current and predicted top-k result set. If the F_{score} of O_{new} is larger than O_{min} of any of the current and future windows, this object is inserted as one of the top-k results of that particular window. Before inserting the O_{new} into any of these result sets, we must find the correct position of this new object, as each top-k result sets/lists are sorted by F_{scores} in the MTopBand structure. This is a simple operation, we continue comparing the F_{score} of O_{new} with each of the top-k results within a particular list till we find an object with F_{score} larger than F_{score} of O_{new}. O_{new} is inserted just below this object in the top-k result list. Now, the O_{min} is deleted from this particular list as O_{min} is no more a part of top-k results for this window. The object immediately above the O_{min} in the result set/ list becomes the new O_{min}. Any new object arriving at the system will now be compared with this new O_{min}. Every arriving object, regardless of its F_{score}, is inserted in the newly created window W_i until the window has not reached the size of $Q_{meta}.W_i.K$.

5.3 Complexity Analysis.

Memory Costs. The memory costs of MTopBand structure depend mainly on two factors, the number of top-k result sets/lists which depends on the number of active predicted views at a given moment and the size of each result set/list. Complexity wise, the memory requirement of the MTopBand structure is in

O(N_{act}*$Q_{meta}.W_i.K$), where N_{act} is the number of active windows at a given time and $Q_{meta}.W_i.K$ is the adaptive K for a given window in the query workload.

Lemma 5.1. *MTopBand maintains expected O($Q_{meta}.W_i.K$*N_{act})objects.*

Since an object may participate as a top-k result for its complete life time, it usually participates in multiple subsequent active windows. we maintain only one physical copy and multiple references of any objects which participates in multiple windows. As proved in Theorem 5.1, we maintain minimal set in the MTopBand structure.

Computational Costs. Computationally, there are two major actions that contribute to the cost of updating top-k results in the MTopBand structure. We recall that, we first search if the newly arrived object belongs to any of the top-k result sets. This a constant cost operation, that is a total of N_{act} comparisons in the worst case. Second, the cost for positioning new object in the top-k result set, if it makes into this result set, is O(log(k)) in the best case. The cost of inserting this object into top-k result set and deleting the smallest score object from the existing top-k result set is in O(log(k)) again.

Thus, the overall processing costs for handling all new objects for each window slide is O(N_{new} * N_{act_new} * $log(k)$), with N_{new} the number of new objects coming to the system at this slide, and N_{act_new} is the 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.

Conclusion. As discussed above, MTopBand structure maintains a minimal object set and also achieves absolute incremental computation. Evidently, we do not need to hold the number of tuples equivalent to the complete window size at any stage for computing the top-k results, rather all the computation is done incrementally. This is a clear win over the existing methods for top-k query computation that need to recompute top-k results from scratch periodically [15, 17].

However, we observe that the resource requirements of MTopBand structure grows with N_{act}, the number of predicted views to be maintained. More specifically, since M-MTopBand stores top-k result sets for each of the predicted views independently/individually, its memory and CPU consumption grows with the number of predicted top-k result sets to be maintained.

We confirm this inefficiency of MTopBand structure when the number of predicted views grow large in the experimental study discussed in Section 7. Next, we discuss various properties of the MTopBand structure and utilizing these observations, we further design the optimized integrated compact structure MTopList structure. We then discuss the maintenance and cost analysis of our proposed structure MTopList.

6. OPTIMIZED TOP-K RESULT-SETS INFRASTRUCTURE: MTOPLIST

To tackle these shortcomings, we now analyze the properties of MTopBand to further design a data structure with resource requirements independent of not only the size of the workload WL and the window size of the meta query Q_{meta}, but also the number of future windows. Next, we discuss various properties of the MTopBand structure and utilizing these observations, we further design the optimized integrated compact structure MTopList structure. We then discuss the maintenance and cost analysis of our proposed structure MTopList.

Observation 1. The MTopBand's top-k results in adjacent predicted views tend to partially overlap, or even be completely identical.

Explanation. Top-k results for the current window are computed based on the scores of the objects within the complete window. Yet, the top-k results of the first predicted view are computed based on exactly the same set of objects except for those few objects that will expire with the first slide. This means that the subsequent predicted views inherit subsets of top-k results from their previous windows.

The top-k result sets of the adjacent predicted views will be identical when 1. the objects that expired after the slide were never a part of the top-k result set, $O_{exp}.F_{score}$ < $O_{curr}.F_{score}$; 2. All the newly arriving object in the current window have an object score smaller than objects that are alive from previous window, $O_{exp}.F_{score}$ < $O_{curr}.F_{score}$.

Observation 2. An object may disappear first and then may reappear later in the top-k result sets of subsequent predicted views in its life time.

Explanation. By Theorem 5.1, top-k results for multiple queries are maintained concomitantly in the MTopBand structure, such that only the minimal object sets that may participate as top-k results for one or more queries are kept. We also recall that the predicted views in the MTopBand structure are built such that each view ends at an output moment of one or more top-k queries.(Section 5)

Observation 3. If object o_i and o_j both participate in the predicted top-k result sets of more than one windows, then the relative positions between o_i and o_j remains the same in each of the predicted top-k result set .

Explanation. First, the F_{score} for any object is fixed. Second, the top-k objects in any predicted view are sorted by their F_{scores}. Thus, 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*)*.

6.1 Integrated Infrastructure: MTopList

Given these properties, we now develop an integrated data structure to represent MTopBand top-k result sets for all predicted views. Our goal is to share the (1.) memory space among views by maintaining by maintaining only distinct objects each of which may participate in the predicted top-k results of possibly many queries; (2.) computation of positioning each new object into the predicted top-k results of all predicted views. This sharing leads us to remarkable savings in CPU and memory resources as discussed below.

To achieve this goal, instead of maintaining N_{act} 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 MTopList.

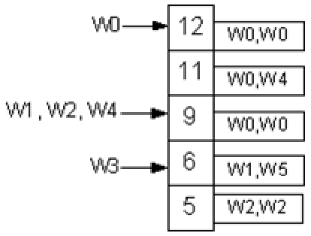

Figure 6: Physical view of MTopList

MTopList is sorted by F_{scores} of these distinct objects. Figure

6. shows the MTopList structure based on the workload WL of the three queries Q1, Q2, and Q3 introduced in Example 4.2. Note that Figure 4 depicts the MTopBand structure for the same example.

MTopList shown in Figure 6 includes all the predicted top-k results in the MTopBand structure. At time stamp t = 8s, a list of only 5 distinct objects with F_{scores} 12, 11, 9 and 6, and 5 are maintained instead of 5 independent top-k result sets for each of the current and future windows with redundant objects between the windows as maintained by MTopBand structure (Figure 4).

Clearly, in the MTopList structure an object may participate in more than one window, and it is usually a part of the top-k results for more than one query. Next, we tackle the problem of how to distinguish among and maintain top-k results for multiple windows and multiple queries in this integrated MTopList structure.

Lemma 6.1. *If top-k parameter k for all queries in WL is equal, then at the output time of the window W_i, the object with the smallest F_{score}, say O_{min_topk} of the predicted top-k results in any future window $W_{i+n}(n > 0)$ has a smaller than or equal F_{score} to that of any window $W_{i+m}(0 \leq m < n)$, i.e. $O_{min\_topk} \leq W_{i+m}.O_{min\_topk}$.*

Proof. When the top-k parameter for all queries is same, the number of predicted top-k results maintained in each current and future window is exactly same. After a window slides, some of the objects from the top-k result set in the current window may expire. The objects in the current window W_i that also participate in W_{i+n}, D_{W_i+n}, is a subset of those will participate in W_{i+m}, D_{Wi+m} (m < n). Thus, the minimal F score of the top-k objects selected from the object set D_{W_i+m} 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}.

Based on Lemma 6.1, we now introduce the first step to distinguish between the objects participating in different windows and in the top-k results of different queries. We call this as *window mark representation*. More specifically, we represent two window marks (window id) for each object in the MTopList, namely the start window mark and the end window mark, which respectively represent the windows in which an object makes its first and its last occurrence to be predicted as the part of top-k result respectively.

Lemma 6.2. *For given windows W_{i+m} serving a query Q_m with top-k parameter $Q_m.k = X$, W_{i+n} with $Q_n.k = Y$, and W_{i+p} with $Q_p.k = X$ such that $0 < m < n < p$ and $X > Y > 0$; top-k elements participating in W_{i+m} with rank greater that Y (based on F_{score}) will not participate in W_{i+n} if the objects from rank 1 through Y in W_{i+m} are still alive at the time of window W_{i+n}. The objects with rank greater than Y will again participate in W_{i+p} if they are all still alive.*

Based on the Lemma 6.2, it can be seen that an object during its life time may participate as part of predicted top-k results in windows $Q_n.k = X$ and disappear for windows with top-k parameter $Q_n.k = Y$ then reappear for the windows with parameter $Q_n.k = X$, such that $Q_n.k = Y < Q_n.k = X$.

Thus, we observe that for all top-k results with rank greater than the top-k parameter $Q_i.k$ such that $Q_i.k$ is the smallest *k* parameter among all queries in the workload, there is a possibility of discontinuity in their participation as top-k results in the subsequent windows.

Hence, simply maintaining the first occurrence (start window mark) and the last occurrence (end window mark) would be insufficient to track in which windows among that range, a particular object actually participates. To tackle this, we maintain a separate pointer for each window at the lowest top-k object in the top-k result set so as to identify the actual top-k results in any particular window. We now introduce a minimum F_{score} pointer, FP_{min} for each window in the MTopList. The FP_{min} mark points to the object with smallest F_{score} in a particular window. Thus the number of FP_{min} marks maintained within MTopList is equal to N_{act}, namely one for each active window. We further utilize this pointer for updating the MTopList with each newly arriving object in the data stream as discussed in the next subsection.

Lemma 6.3. *At any given time, utilizing the start window mark and the end window mark of an object in the MTopList structure along with the FP_{min} mark for each window is sufficient to generate the top-k result for any query Q_i in the workload WL.*

6.2 The MTopList Maintenance

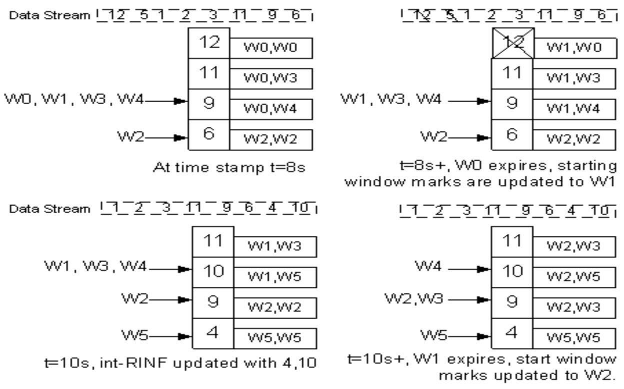

Figure 7: Updating the multi top-k results in integrated structure MTopList

Updating MTopList after expiration of existing objects. A careful mechanism is needed for updating MTopList every time a window slides. As discussed before, each object in the integrated structure may participate as a top-k result in more than one active window. So, if the oldest window W_0 expires the corresponding objects in W_0 cannot simply be deleted from the list. We develop a strategy that uses the starting and ending window marks to decide if an object needs to be physically removed from the MTopList altogether after the window slides.

We observe that, the top-k objects of the current-to-be expired window are the first $Q_{meta}.K$ objects in the MTopList. We recall(Section 4) that $Q_{meta}.K$ is equivalent to $Q_i.k$ where Q_i is the query that needs output when the current window expires. If the window serves more than one member query then $Q_{meta}.K$ is the maximum top-k-parameter of all the queries served by any window. The MTopList is sorted by objects' F_{score}. So, the current-to-expired window being the oldest window will contain $Q_{meta}.K$ objects with highest F_{score} as compared to the other objects in the list.

After the window expires, we increment the starting window mark of all objects in that window by 1. This indicates that window has expired as none of the objects in the list participate in that particular window. After incrementing the starting window marks if any of the objects in the list has a starting window mark larger than the ending window mark then we physically delete this object from the list because this object was participating only in the window that already expired and thus is not needed any more.

Updating MTopList after inserting newly arriving objects. Every time a new object, namely o_{new} is eligible to participate as a top-k result(decided based on o_{new}'s F_{score}) we take the following steps to update MTopList. At step 1, we find the correct position of o_{new} in MTopList. At step 2, we update o_{new}'s starting and ending window mark. Finally, 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 positioning each object into the MTopList, if the predicted top-k result set of any future window represented by the MTopList has not reached the size of k yet, or if its F score is larger than that of any object in the MTopList, we insert it into the MTopList. Otherwise it will be discarded immediately.

The position of O_{new} is easy to find utilizing the min_{FP} marks. $O_{new}.F_{score}$ (F_{score} of the new object O_{new}) is compared each of the min_{FP} starting from the lowest until an object with F_{score} greater than O_{new} is found.

If O_{new} is inserted at its correct position in the MTopList, it is in the predicted top-k results of at least the one window in its life span, its ending window mark is set to be the newest window Id the MTopList such that $O_i.min_{FP}.F_{score} < O_{new}.\ F_{score}$, where $O_i.min_{FP}.F_{score}$ is the F_{score} of the object marked by min_{FP}. The starting window mark of a new object is simply the oldest window on the MTopList, $O_i.min_{FP}.F_{score} < O_{new}\ F_{score}$.

Once we have O_{new}'s updated the starting window mark and ending window mark, we remove the objects pointed by min_{FP} marks from all those windows in which O_{new} is predicted to participate. We note that, here is that O_{new} may not participate as a top-k result in all windows from starting window mark to ending window mark(Observation 2). Thus, only those objects are removed whose $O_i.min_{FP}.F_{score}$ is smaller than $O_{new}\ F_{score}$; $O_i.start_{mark}$ is greater than or equal to $O_{new}.start_{mark}$ and $O_i.start_{mark}$ is smaller than or equal to $O_{new}.end_{mark}$.

7. EXPERIMENTAL STUDY

Our experiments are conducted on a Sony VIAO laptop with Intel Centrino Duo 2.6GHz processor and 1GB memory, running Windows Vista. All the algorithms are implemented in Eclipse IDE using C++.

Real Datasets. We used two real streaming data sets. The first data set, GMTI (Ground Moving Target Indicator) data [18], records the real-time information of moving objects gathered by 24 different data ground stations or aircrafts in 6 hours from Joint STARS. It has around 100,000 records regarding the information of vehicles and helicopters (speed ranging from 0-200 mph) moving in a certain geographic region. The second dataset is the Stock Trading Traces data (STT) from [19], which has one million transaction records throughout the trading hours of a day.

Alternative Algorithms. We compare our proposed algorithm MTopLists performance with two alternative methods, namely, 1. state-of-the-art single top-k query solution MinTopK [16] that processes each query independently in the workload WL. 2. MTopBand, the basic algorithm we first presented in this work (Section 5.).

Experimental Methodologies. We measure two common metrics for stream processing algorithms, namely average processing time for each tuple (CPU time) and memory footprint.

We perform the scalability tests to verify the performance of the proposed algorithms with the increasing number of queries in the input workload. We first evaluate at a time two test cases, each varying on only one of the three query parameters. Then, we test the more general cases of varying two parameters, and finally all three query parameters are set to be arbitrary. For each experiment, we vary *win* in the range of 100K to 1 M, *slide* between 10K to 100K, and *k* in the range of 10-1000.

7.1 Scalability Evaluation

Scalability tests with one arbitrary parameter. For each test case, we prepare three workloads with 10, 100, and 1000 queries respectively by randomly generating one input parameter (in a certain range) for each member query, while using common parameter settings for the other two query parameters.

Fixed win, Fixed $slide$, and Arbitrary k. In this experiment, we evaluate performance of our proposed algorithms as compared to the state-of-art algorithm [16]. We use fixed *win* = 1M and *slide* = 100K, while varying *k* from 10 to 1000. We randomly generate *k* between the range 10 - 1000 for each query.

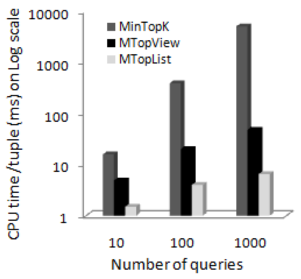

Figure 8: CPU time used by three algorithms with different k values

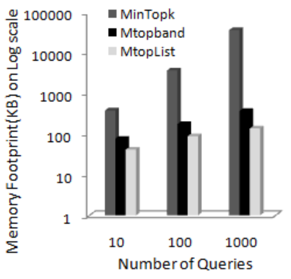

Figure 9: Memory space used by three algorithms with different k values

Figures 8 and 9 show the CPU time and memory space, on logarithmic scale, used by the three algorithms. Clearly, performance of our proposed methods is in order of magnitude better than MinTopK algorithm. Amongst all three compared algorithms , MinTopK's [16] CPU time increases as a direct multiple of the size of workload. The increase in the CPU time is around 100 times when the number of queries increases from 10 to 1000. Put differently, it does not scale well with the cardinality of the workload.

For the memory space used, MTopList has even better performance as its utilization of memory space only increases 2.5 times when the number of queries increases from 10 to 1000, while such increase for MTopBand and MinTopK are 6 times and 99 times respectively.

We note that in this case only top-k parameter *k* is arbitrary. Thus, this is the best possible case for our proposed algorithm as maximum sharing is achieved here. Next, we discuss the experimental evaluation for the cases when *k* is fixed, while other query parameters, *win* or *slide* are varying.

Varying *slide sizes*. In this experiment, we use *win* = 1M and *k* = 1000 , while we randomly generate *slide* values between the range 100K - 1M.

As shown in Figures 10 and 11, both the CPU and memory usage of MTopList is still significantly less than those utilized by the state-of-the-art algorithm MinTopK [16]. In particular, for processing 100 queries , MTopList only takes 0.0066 s to process each object on average, while MinTopK needs 0.712 s for each object. 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 of our basic algorithm MTopBand can be affected by the *win*/*slide* ratio. We recall that MTopBand maintains the predicted top-k results for each future window indepen-

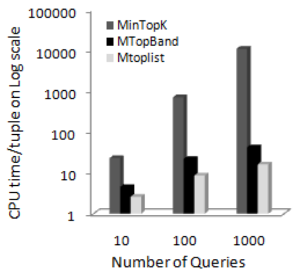

Figure 10: CPU time used by three algorithms with different $slide$ values

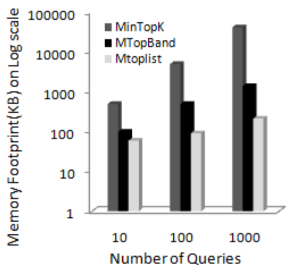

Figure 11: Memory space used by three algorithms with different $slide$ values

dently, thus its resource utilization is expected to increase with the number of future windows maintained, which is equal to $win/slide$.

In this experiment, *win* is fixed to 1M while *slide* is randomly picked, resulting in a large value of $win/slide$ ratio for some of the queries in the workload. On the other hand, the performance of MTopList remains unaffected with increase in the $win/slide$ ratio. This is because MTopList only maintains distinct top-k objects which are not dependent on number of predicted views maintained at a given time.

Scalability tests with more than one arbitrary query parameters For each test case, we prepare three workloads with 10, 100, and 1000 queries respectively by first randomly generating two input parameters (in a certain range) for each member query, while using common parameter settings for just one query parameters. Finally, we evaluate the most general case by randomly generating all three query parameters.

Arbitrary win, Arbitrary $slide$, and fixed k.

In this case, we use *k* = 100, while varying *win* from 100K to 1M and *slide* from 10K to 100K.

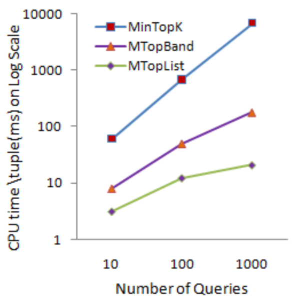

Figure 12: CPU time used by three algorithms with different win and $slide$ values

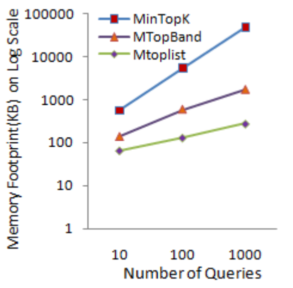

Figure 13: Memory space consumed by three algorithms with different win and $slide$ values

As shown in Figures 12 and 13, the CPU time consumed by MTopList per tuple increases 4.1 times, when the number of queries increase from 10 to 100 and it further increases around 2.3 times when number of queries increase from 100 to 1000. Whereas for the basic proposed algorithm MTopBand, the CPU time increases around 7 times from 10 to 100 queries and around 4.5 times from 100 to 1000 queries. MinTopK utilizes 99 and 91 times more CPU and memory respectiely when number of queries are increase from 10 to 1000. Clearly, this increase in CPU consumption time of the proposed algorithm with increase in the number of queries is modest as compared to the alternative algorithms.

The ratio of increased CPU consumption time is 1.5 times more as compared to the previous only one arbitrary parameter case. This is because two arbitrary query parameters lead to decrease in the sharing amongst different queries, and thus increases the maintenance costs of both MTopList and MTopBand.

General case: All Arbitrary Parameters. Finally, we evaluate the general case with all three parameters win, $slide$, and k being varied arbitrarily.

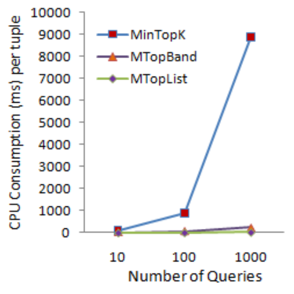

Figure 14: CPU time used by three algorithms with arbitrary win and $slide$ parameters

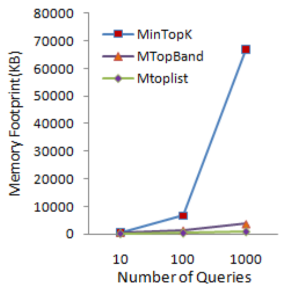

Figure 15: Memory space consumed by three algorithms with arbitrary win an

Figures 14 and 15 show the performance of the three algorithms in terms of CPU and memory utilization. Clearly, MTopList wins over the other two algorithms for both CPU and memory utilization in this case too. MTopList takes around 30-40 times less CPU time to process 1000 queries as compared to MTopBand. Also, MTopList takes around 330 times less CPU time as compared to the state-of-art algorithm MinTopK. This saving is less as compared to the previous cases where only one or at the most two parameters are arbitrary. This is caused by too large variations on the parameter settings. The important observation here is MTopList never performs worse than MTopBand for any workload.

8. RELATED WORK

Top-k queries on a static data set have been well studied in the literature. The top-k algorithms, $Onion$ [2] and $Prefer$ [8], based on preprocessing techniques, require the complete data set to be available in memory for computing the top-k objects.

[10] presents algorithms that reduce the storage and maintenance costs of materialized top-k views in the presence of deletions and updates. Other works in relational databases like [11,12] focus on multidimensional histograms and sampling-based technique to map top-k queries into traditional ranges. [3,4,5] study top-k queries in distributed data repositories. In general, they minimize the communication cost for retrieving the top-k objects from distributed data repositories.

Fagin et al. [13] introduce two methods for processing ranked queries. The TA algorithm is optimal for repositories that support random access, while the NRA algorithm assumes that only sorted access is available. Chang and Hwang [7] introduce MPro, a general algorithm that optimizes the execution of expensive predicates for a variety of top-k queries.

All the above methods are based on the assumption that the relevant data set is available at the compilation stage of query execution either locally or in distributed servers. Also they are designed to report the top-k results only once. Thus these techniques are not suitable for streaming environments where the data are not known in advance, rather they keep changing as new tuples arrive and old ones expire.

More recently researchers have started to look at the problem of top-k queries in streaming environments. Most of this work is focused on single top-k query processing where the assumption is that at a time only one top-k query is registered in the system [6,14,16] . Among these works, [16] presents an optimal technique for top-k query processing both computationally and memory wise. Although optimal for single top-k query processing, this technique does not handle multiple queries simultaneously registered in the system. Our experiments show that our proposed sharing strategy by many orders of magnitude outperforms the solution of executing top-k queries independently for multiple queries.

To the best of our knowledge, [15] and [17] are the only two works that handle simultaneously registered multiple top-k queries in streaming scenario. [15] tackles the problem of exact continuous multiple top-k queries monitoring over a single stream. The proposed techniques share only the indices among different registered queries by maintaining index and bookkeeping structures. They introduce two algorithms. First, the TMA algorithm computes the new answer for a query whenever some of the current top-k points expire. Second, the SMA algorithm maintains a Şsky-band structure" that aims to contain sufficient number of objects so that it need not go back to the full data stream window.

However, unfortunately, neither of these two algorithms eliminates the recomputation bottleneck 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 altogether thus realize complete incremental computation and minimal memory usage.

Experiments conducted by the optimal technique for top-k query processing [16] shows a significant CPU and memory resource saving over [15]. Our experimental results confirm the improvements by many orders of magnitude achieved by our proposed algorithm over [16] for any workload with a size of 2 queries and greater. Thus, our proposed algorithm achieves a clear win over each the state-of-art techniques.

[17] handles multiple top-k queries, but based on the probabilistic top-k model in data streams. While we work with a complete and non-probabilistic model. Also their focus is to achieve sharing among the queries on the preference function while we focus on other important parameters of a continuous top-k query, namely window size, slide size and K. In short, they in large 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 these respective top-k models.

9. CONCLUSION AND FUTURE WORK

In this work, we present the MTopS framework for efficient shared processing of a large number of top-k queries over streaming windows. MTopS achieves significant resource sharing at the query level by analyzing the parameter settings. MTopS further optimizes the shared processing by identifying and maintaining only the minimal object set from the data stream that is both necessary and sufficient for top-k monitoring of all queries in the workload. Our experimental studies based on both real and synthetic streaming data confirm the clear superiority of MTopS to the state-of-the-art solution. We confirm that MTopS exhibits excellent scalability in terms of being able to handle thousands of queries under high speed input streams in our experiments. As future work, this framework can be used to scale-up considering multiple machines and grouping of workloads into sub-workloads to be assigned to different machines.

10. REFERENCES

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

[2] 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 $SIGMOD$, pp 391-402, 2000.

[3] B. Babcock and C. Olston. Distributed top-k monitoring. In $SIGMOD$, pp 28-39, 2003.

[4] K. C.-C. Chang and S. Won Hwang. Minimal probing: supportive expensive probing: supporting expensive predicates for top-k queries. In $SIGMOD$, pp 391-402, 2000.

[5] S. Chaudhuri, L.Gravano, and A. Marian. Optimizing top-k selection queries over multimedia repositories. $IEEETrans./-Knowl.DataEng.$, 16(8):992-1009,2004.

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

[7] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In $SIGMOD$, pp 346-357, 2002.

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

[9] S.Nepal, M.V. Ramakrishnan: Query processing issues in image(multimedia) databases. In $ICDE$ (1999).

[10] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In ICDE, pp 189-200, 2003.

[11] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. TODS,27(2):153-187, 2002.

[12] C.-M. Chen and Y. Ling. A sampling-based estimator for top-k query. In ICDE, pp 617-627, 2002.

[13] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In $PODS$, 2001.

[14] G. Das,D. Gunopulos, N. Koudas, and N. Sarkas. Adhoc Topk Query Answering for Data Streams. In $VLDB$, 2007, pp 23-28.

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

[16] D. Yang, A. Shastri, E.A. Rundensteiner, and M. O. Ward. An Optimal Strategy for Monitoring Top-k Queries in Streaming Windows. In $EDBT$, pp 138-152, 2011.

[17] C. Jin, K. Yi, L. Chen, J. X. Yu, and X. Lin. Sliding-window top-k queries on uncertain streams. PVLDB, 1(1):301Ű312, 2008.

[18] J. N. Entzminger, C. A. Fowler, and W. J. Kenneally. Jointstars and gmti: Past, present and future. IEEE Transactions on Aerospace and Electronic Systems, 35(2):748-762, april 1999.

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

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

11. 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.