# MTopS: Multi-Query Optimization for Continuous Top-K Query Workloads

by

Avani Shastri

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

---

May 2011

APPROVED:

---

Professor Elke A. Rundensteiner, Thesis Advisor

---

Professor Joseph Beck, Thesis Reader

---

Professor Craig E. Wills, Head of Department

**Abstract**

A continuous top-k query retrieves the $k$ most preferred objects from a data stream according to a given preference function. These queries are important for a broad spectrum of applications from web-based advertising, network traffic monitoring, to financial analysis. Given the nature of such applications, a data stream may be subjected at any given time to multiple top-k queries with varying parameter settings requested simultaneously by different users.

This workload of simultaneous top-k queries must be executed efficiently to assure real time responsiveness. However, existing methods in the literature focus on optimizing single top-k query processing, thus would handle each query independently. They are thus not suitable for handling large numbers of such simultaneous top-k queries due to their unsustainable resource demands.

In this thesis, we present a comprehensive framework, called MTopS for Multiple Top-K Optimized Processing System. MTopS achieves resource sharing at the query level by analyzing parameter settings of all queries in the workload, including window-specific parameters and top-k parameters. We further optimize the shared processing by identifying the minimal object set from the data stream that is both necessary and sufficient for top-k monitoring of all queries in the workload. Within this framework, we design the MTopBand algorithm that maintains the up-to-date top-k result set in the size of O ($k$), where $k$ is the required top-k result set, eliminating the need for any recomputation.

To overcome the overhead caused by MTopBand to maintain replicas of the top-k result set across sliding windows, we optimize this algorithm further by integrating these views into one integrated structure, called MTopList. Our associated top-k maintenance algo-

rithm, also called MTopList algorithm, is able to maintain this linear integrated structure, thus able to efficiently answer all queries in the workload. MTopList is shown to be memory optimal because it maintains only the distinct objects that are part of top-k results of at least one query. Our experimental study, using real data streams from domains of stock trades and moving object monitoring, demonstrates that both the efficiency and scalability in the query workload of our proposed technique is superior to the state-of-the-art solutions.

**Acknowledgements**

I would like to thank my research advisor, Prof. Elke A. Rundensteiner for her tremendous support, encouragement, and guidance all throughout this work. She not only helped me in my academic development but also in my overall development as an engineer and a professional. It is an absolute privilege for anyone to work under her guidance.

My sincere thanks to my Thesis reader, Prof. Joseph Beck for his valuable suggestions and cooperation.

I want to thank Di Yang for his continuous guidance and feedback throughout this work. He not only helped me understand the research domain but also provided me a solid foundation for the work.

I would also like to thank Prof. Matthew O. Ward his valuable guidance and to all DSRG and XMDV group members for their cooperation at all times.

I would like to express my deepest gratitude to my husband, Ankit for his unstinting support and unwavering commitment to my success in my endeavors. I could accomplish this only because of the motivation and tremendous cooperation provided by him throughout the work.

I want to thank my immediate family Ami, Gaurav, Nidhi, and most of all to both my parents and my parents-in-laws - for their continuous encouragement and support. Their involvement always added to my enthusiasm for the work.

I gratefully acknowledge the National Science Foundation for the support under Grant No. IIS-0633930 and CRI-0551584.

# Contents

# List of Figures

# Chapter 1

# Introduction

With the continuous proliferation of web applications and digital devices, the input data rates of streams arriving at a data stream management system (DSMS) have grown by leaps and bounds. Naturally, there is thus critical requirement to process these huge volumes of data so as to generate real time results by reducing the lag between data acquisition and acting on the acquired data.

## 1.1 Motivation

Top-k queries are critical for large number of applications ranging from web advertising, financial analysis to network traffic monitoring. A top-k query returns the $k$ most preferred objects from a dataset $P$ according to a given preference function $F$. Since streaming data is infinite while the notion of top-k can only be defined based on a finite number of objects, window constraints are usually adopted to make top-k queries applicable to data streams [14, 15, 16] Such a window can be time based or tuple-count based. Time based sliding

windows assume that tuples arrive with a time stamp and remain in the buffer as long as their time stamp belongs to a fixed time period covering the most recent time stamps. Tuplecount based sliding windows contain the most recent N records [14].

Thus top-$k$ queries are not only parametrized by the parameter setting $k$ but also window properties such as window type, size and slide. Analysts may be interested in different top-$k$ volatile stocks of the same financial data while imposing customized time windows and refresh rates. For example, a financial analyst may ask for the top-10 most volatile stocks in the last 1 hour with a refresh rate of 10 minutes. Another analyst may want to look at the top-200 most volatile stocks in the last 30 minutes with a refresh rate of 5 minutes.

In fact, even a single analyst may at times submit multiple queries with different parameter settings with the intent to further analyze retrieved result sets so to derive a well supported conclusion. Real time systematic processing of such workloads of top-$k$ queries is essential.

As motivated above, a stream processing system should be able to accomodate a workload of numerous top-k queries, and thus successfully calculate the correct top-k results at the required output moments for each of these queries. In this work, we focus on processing multiple top-k queries with arbitrary query parameter settings, while still achieving real time results for each of these queries as needed by any stream processing system.

## 1.2 State-of-Art

Top-$k$ query processing has been extensively studied in conventional databases [2, 8, 20]. These techniques cannot be directly applied nor easily adapted to fit streaming environ-

ments. This is because the key problem they solve 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 [2, 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].

In the streaming scenario, research has primarily focussed on single top-k query processing [14, 15, 16, 17]. These methods focus on only one query registered in the system at a time. However, simultaneous processing of large numbers of top-k queries, as would be experienced by applications as motivated above, remains a challenging open problem to date.

## 1.3 Challenges

One major challenge associated with multi top-k query processing is to support workload of queries with possibly arbitrary parameter settings. More specifically, the parameters of the queries in the query group may be arbitrary, thus not allowing any obvious sharing of computations among distinct queries. We thus set out to analyze characteristics of these queries so as to identify t subprocesses as well as what system resources amongst these queries may be shared.

Given the real time response requirement of the top-k query processing, serving a workload with possibly arbitrary parameters in a single system is highly resource intensive. 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 optimal state-

of-art top-k query processing method may take 10 s to update the query result for each window slide (100K new tuples) for 1M- tuple window slide and k equal to 1K tuples[16]. In this scenario 1000 such queries with refresh rates of 5 minutes each were to be executed one after the another, it may take time more than two hours to generate the top-k result for all queries; obviously failing to answer most of the queries at the required refresh rate.

## 1.4 Proposed Solution

We present a comprehensive framework 'MTopS' for " **M**ulti **Top-K O**ptimized **P**rocessing **S**ystem, to achieve simultanous execution of a workload of parameterized top-k queries with arbitrary window parameter settings, namely (*win* and *slide*) and the top-k parameter *k*.

Within this framework we introduce several innovations essential for optimizing multi-query top-k processing by effictively sharing the available CPU and memory resources.

1. First, we carefully analyze the workload so as to generate a single meta query to represent all the workload queries. As discussed in Section V, as a first processing step we successfully remodel the problem of maintaining multiple queries into the execution of a single query.

2. We propose an execution strategy that drives the single meta query to process the complete workload under the high speed input data rate so as to generate real time top-k results required by each of the queries. As discussed in Section VI, our execution strategy achieves not only completely incremental computation but also memory utilization in the order of query parameter k.

We identify the minimum object set that is both necessary and sufficient for generating

accurate and timely rop-k results. We maintain this top-k result set in our proposed data structure MTopBand. The key idea is to precompute and maintain metadata namely, the set of objects that have potential to belong to the top-k result in one or more of the future output windows. This is determined based on expiration times of objects and keeping sufficiently many objects for each future expiration moment. We introduce MTopBand maintenance algorithm for updating the top-k result setsin real time with the arrial of new objects at the system and expiration of objects in the existing top-k result sets

3. We further improve the performace by analyzing interrelationships among consecutive top-k result sets. We observe that majority of the objects in the adjacent future windows tend to overlap due to which our first proposed algorithm MTopBand usually maintains some multiple copies of one top-k object. We thus design an integrated maintenance mechanism that maintains only the distinct copy of an object in a linear data structure, MTopList across all queries and all window time slices. This mechanism avoids storing overlapping results multiple times and also enables us to design algorithms to update them linearly rather than updating them indiidually for each window. Futhermore, we also provide a detailed complexity analysis for our techniques.

4. Lastly, we utilize seperate algorithms to generate runtime instructions for meta query execution and exact result extraction for each of the queries in the query group at a required output moment.

5. We conduct extensive experiments on both real and synthetic data sets to demonstrate the efficiency and the scalability of our techniques. Experimental evaluation(Section XI) shows that the MTopS 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. We demonstrate that resource consumption in our approach is not proportional to the size of workload as is the case with the state-of-art approach of optimally monitoring top-k queries[16]. Hence our approach achieves far superior performance with increasing workload.

# Chapter 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 reuired 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.

# Chapter 3

# The MTopS Framework

We now introduce the architecture of the MTopS framework shown in Figure 3.1, while details of the techniques used in each block are discussed in Chapters 4, 5, 6, 7, and 8.

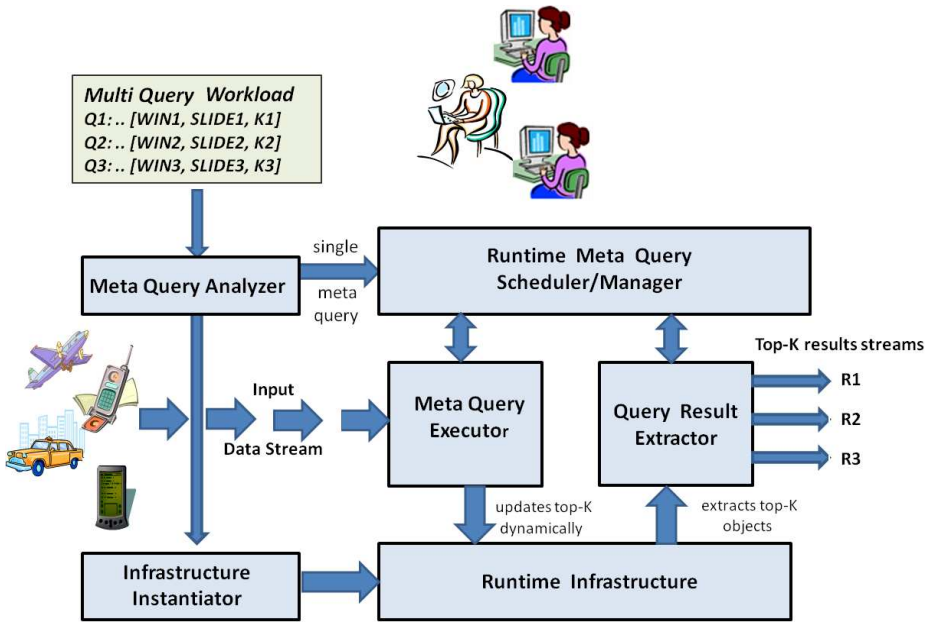


Figure 3.1: MTopS System Architecture

**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 Chapter 4.

**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. We call this infrastructure as Runtime Infrastructure (RINF) in our system.

In this work, we propose two data structure designs for RINF, which do not simply collect the top-k candidates, but also encode them into efficiently updatable formats. These two designs are the independent window representation and integrated window representation respectively. We prove that by using those carefully designed data structures, our RINF maintains 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. At the runtime execution, RINF will be continuously updated as the input stream arrives by Meta Query Executor (MQE) (will be discussed later). Also the query results for each member query will be extracted from RINF by Query Result Extractor (QRE) whenever they are needed.

**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. To guide this slide adaption process of $Q_{meta}$, we build a Runtime Meta Query Scheduler (RMQS) to calculate the nearest time point that is needed next by either of those three operations.

Such schedule information will decide the behavior of other query execution modules, namely MQE and QRE. In particular, 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 $Q_{meta}$, in other words, whether it is possible for $o_{new}$ to make the top-k result of any member queries. If yes, $o_{new}$ will be used to update RINF. Otherwise, $o_{new}$ will be 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 member queries. Those “completely expired” objects will be purged from RINF immediately, while those are expired for some queries but still valid for at least one member query will still be kept in 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. During the result extraction process, by analyzing the specific top-k candidate encoding used by RINF, QRE in our system guarantees that it only touch the objects that will be output for at least one query.

# Chapter 4

# Analyzing the Multi Top-K Query Workload

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

## 4.1 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*). This is because usually the life time of an object is much larger than the arrival rate of new objects. For example, in a typical scenario, an analyst interested in objects arriving in last 24 hours but retrieves the output after every 5 minutes. 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 predict the specific subset of the current objects 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) query results for each of these future windows based on the objects in the current window that already accounted for the object expiration. Thus, these predicted top-k results will have to be updated only if any new object that arrives to the system will be capable of being a part of the top-k result. Otherwise, these predicted top-k result sets can be the actual result sets for future windows.

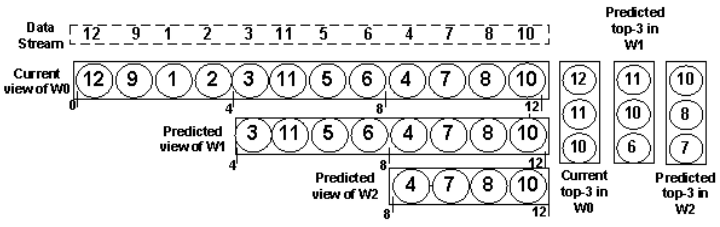


Figure 4.1: Predicted views of three consecutive windows at $W_0$

Figure 4.1 (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 slides. In Figure 4.1 (left), the numbers shown in the white circles represent the objects' scores. when a window slides, following updates are done: 1: a new window is created, 2: a new object is inserted only if the new object is eligible to make it to the top-k of already full window, 3: an old window is deleted. As the window expires, the top-k result of the expired window $W_0$ are no longer valid and is updated based on the new current window ( now $W_1$).

Figure 4.2 (left and right) shows the updated predicted views just after $W_0$ has expired.

At time t = 12 s, the top-3 result is extracted from the 12 objects active in the current window $W_0$, namely, $o_1$-$o_{12}$.

In every window the k objects with the highest preference scores will make the top-k objects for that particular window. Based on the objects in $W_0$, we cannot only calculate the top-k result in $W_0$ for k=3, but also predetermine the potential top-k results for the future windows $W_1$ and $W_2$, until the end of the lifespan of all objects in $W_0$.

Figure 4.2 (right) shows the three top-3 results calculated for $W_0$, $W_1$ and $W_2$ respectively. The predicted top-k results for current window are generated based on objects active in the current window $W_0$, namely, $o_1$-$o_{12}$, future windows are calculated based on smaller and smaller subsets of the objects belonging to $W_0$ that are known not to expire yet in $W_1$ nor in $W_2$, namely, $o_5$-$o_{12}$ in $W_1$ and $o_8$-$o_{12}$ in $W_2$. As the window expires, the top-k result of the expired window $W_0$ are no longer valid and is updated based on the new current window ( now $W_1$). Figure 4.2 (left and right) shows the updated predicted views just after $W_0$ has expired. At time t = 12 s, the top-3 result is extracted from the 12 objects active in the current window $W_0$, namely, $o_1$-$o_{12}$.

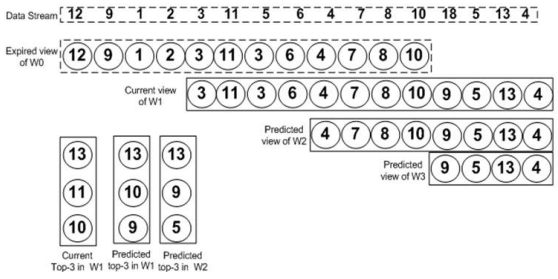


Figure 4.2: Updated predicted views of three consecutive windows at $W_1$

## 4.2 Sharing with Varying Top-k Parameters $k$

Consider all the window parameters, i.e., *win* and *slide* are same for all queries while the top-k parameter *k* is different. This implies only the number of objects to be output by each query differs.

**Lemma 4.2.1** *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 all each query such that $Q_i$.k is the query with largest top-k parameter among WL.*

**Proof.** Lemma 4.2.1 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.

**Example 4.2.1** *if $Q_1$.win = $Q_2$.win = $Q_3$.win = 8s; $Q_1$.slide = $Q_2$.slide = $Q_3$.slide = 2s; and $Q_1$.k = 4, $Q_2$.k = 3, and $Q_3$.k = 2. In this case, MQA builds the meta query such that $Q_{meta}$.WIN = 8, $Q_{meta}$.SLIDE = 2 and $Q_{meta}$.K = 4. Thus, MQA builds only 4 predicted views in total; starting at moments 00:00:00, 00:00:02, 00:00:04, 00:00:06 respectively; instead of 16 predicted views as would have been needed if the each of these queries were executed independently.*

Thus, the number of predicted views that need to be built by the meta query are independent of the number of queries in the WL. Clearly, in this scenario a full sharing is achieved compared to the independent execution of individual queries.

## 4.3 Varying Slide Sizes *slide*.

In this scenario, all queries in the workload WL have the same window sizes *win* and same top-k parameter *k* but their slide sizes *slide* may differ. For ease of explanation, let us assume that all the queries start simultaneously. Since their window sizes of all queries are equal , at any given time they are querying the same portion of the input data stream. The only difference between the queries is that they need to generate output at different moments.

**Example 4.3.1** *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. Each query are required to output their result, i.e., top-k set at every 6, 2, and 3 seconds respectively. As consequence, each of these queries will need to maintain different predicted views so as to generate output at different slides. Figure 4.3 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.*

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. $Q_{meta}$ 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.

**Example 4.3.2** *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*

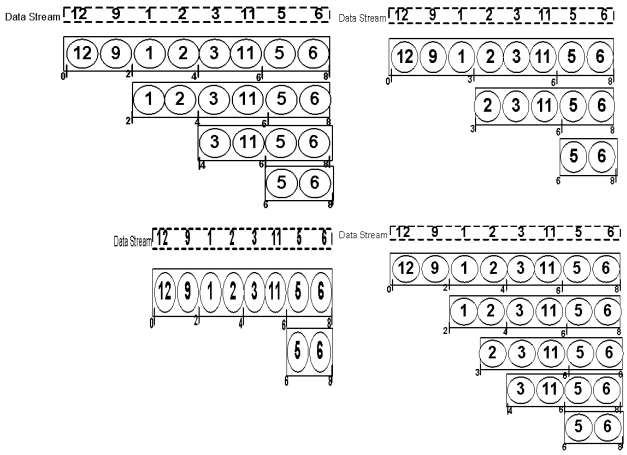


Figure 4.3: Predicted views needed for processing query $Q_1$ (top left), $Q_2$ (top right), $Q_3$ (bottom left) independently and combined view for meta query $Q_{meta}$ (bottom right)

*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 $Q_2$).*

*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, 00:00:03, 00:00:04, 00:00:06, and 00:00:08 respectively, many of which serve multiple queries. For example a the predicted view starting at 00:00:06 is serving all the member queries ($Q_1$, $Q_2$ and $Q_3$). Since the top-K result set to be output by any of the queries would be exactly the same, $Q_{meta}$.K= $Q_1$.k = $Q_2$.k = $Q_3$.k = 3. We thus maintain only the 3 top ranking objects in each of the predicted views.*

## 4.4 Varying Window Sizes *win* and Top-k parameter *k*

In this case, the window sizes *win* vary while the corresponding *slide* value and thus the moments to produce output for each query remain identical. Here we first use the simpli-

fying 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 4.4.1** *Given a query group QG with member queries having the same slide size slide but arbitrary window sizes 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 member 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.

**Example 4.4.1** *If slide sizes and top-k parameter k are equal for the three queries , $Q_1$.slide = $Q_2$.slide = $Q_3$.slide = 2s; $Q_1$.k = $Q_2$.k = $Q_3$.k = 3 while $Q_1$.win = 4s, $Q_2$.win = 6s and $Q_3$.win = 8s. At wall clock time 00:00:08, the predicted views built by $Q_3$ start from 00:00:00, 00:00:02, 00:00:04 and 00:00:06 respectively; those for $Q_2$ start from 00:00:00, 00:00:02, and 00:00:04; and those for $Q_3$ from 00:00:00 and 00:00:02. Clearly, the predicted views needed by $Q_1$ and $Q_2$ overlap with those built by $Q_3$.*

**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.

In summary, even if the window sizes of the queries in the workload *WL* are not multiples of their common slide sizes, the predicted views generated for $Q_i$ (query with largest window size) are sufficient to answer all the queries. Clearly, full sharing is achieved.

## 4.5 Varying Window sizes and Varying Slide Sizes

Next we consider, when both the window sizes *win* and the slide sizes *slide* of all the member queries are arbitrary. Here, we 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 4.5.1** *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 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 need to be maintained instead of the 9 windows that would here been needed if each query were to be executed independently.

## 4.6 The Most General Case

Finally, we consider the general case with all parameters with arbitrary 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 save 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 view serves more than one query then *k* for that window is equivalent to the maximum of the top-k of the queries served by this view.

**Example 4.6.1** *For a workload of three queries with arbitrary window and top-k parameter settings, $Q_1$.win = 8s, $Q_2$.win = 6s and $Q_3$.win = 4 s; $Q_1$.slide = 4s, $Q_2$.slide = 3s, and $Q_3$.slide = 2s; and $Q_1$.k = 3 , $Q_2$.k = 2, $Q_3$.k = 1. The meta query builds 5 windows at time t = 8s,namely current window $W_0$ starting at 00:00:00 (for $Q_1$) and predicted views $W_1$, $W_2$, $W_3$ and W4 at 00:00:02 (for $Q_2$), 00:00:04 (serving $Q_1$ and $Q_3$), 00:00:05 (for $Q_2$), and 00:00:06 (serving $Q_3$) respectively. Thus, k = 3 in $W_0$ ($Q_1$.k = 3), and k = 2 in $W_1$($Q_2$.k = 2), and k = 3 objects in $W_2$ etc.*

# Chapter 5

# Independent Top-k Result-sets Infrastructure: Design and Maintenance

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 The MTopBand Structure 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 $F_{scores}$.

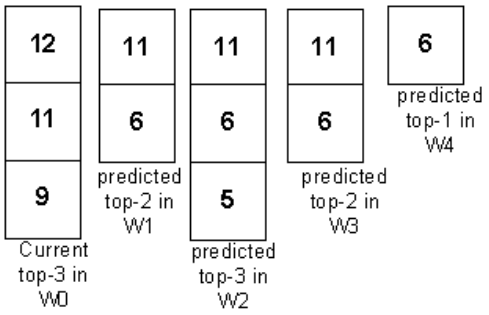


Figure 5.1: Physical view of MTopBand structure

Figure 5.1 shows the MTopBand structure based on the workload $WL$ of three queries $Q_1$, $Q_2$, and $Q_3$ introduced in Example 4.2.1.

We maintain the corresponding top-k results sets, $Q_{meta}.W_0.K$ = 3, $Q_{meta}.W_1.K$ = 2,$Q_{meta}.W_2.K$ = 3, $Q_{meta}.W_3.K$ = 2, and $Q_{meta}.W_4.K$ = 1, for current window $W_0$, and each of the predicted views. Thus, in this example only five objects with the scores 12, 11, 9, 6, and 5 are maintained in the MTopBand structure, while the other three objects, in the input data stream, with the scores 1, 2, and 3 were discarded immediately.

In practice, the window sizes could be orders of magnitude higher than K. For example, a window size $WIN$ = 1,000,000 and $K$ = 10 would be typical. But the set of top-k result

set maintained in the MTopBand structure is minimal and is independent of the potentially very large window size of the queries.

**Theorem 5.1.1** *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 dont 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 The 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.

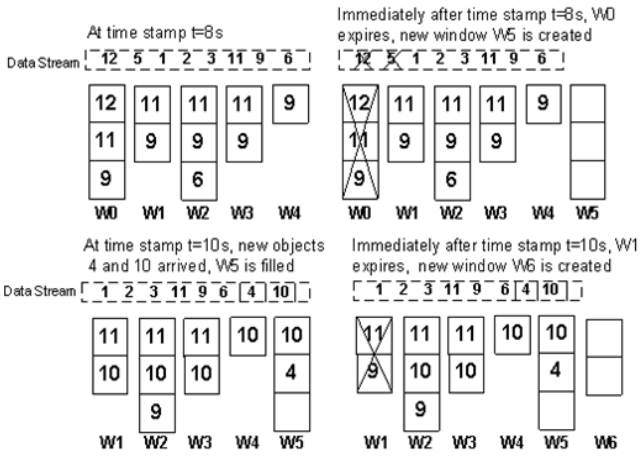


Figure 5.2: Updating the multi top-k results in MTopBand

When a new object arrives at the system, we first check if any of the queries in the

workload needs output. This verification can be easily done by utilizing the meta-query logic. We recall that, a window slides at the next nearest output moment, based on the adaptive slides as logically defined by $Q_{meta}$ to capture the timely output for one or more queries in the workload $WL$. If a window slides, we update the MTopBand top-k 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 5.2 depicts the MTopBand structure maintenance based on our running Example 4.2.1. 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 5.2 (top right)) for the next future window to cover the whole life span of the incoming objects. The number of predicted top-k objects maintained by new window will depend on the top-k parameter ($Q_{meta}.W_i.K$) of the query that needs output at the moment when this newly created window will output. Once the empty window is created , each new object coming in will participate in the current window and all future windows that are currently listed in the MTopBand structure.

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 $Q_{meta}.W_i.K$.

**Example 5.2.1** *In Figure 5.2 (bottom left), two new objects with $F_{scores}$ 4 and 10 arrive, while the object with $F_{score}$ 10 replaces objects with $F_{score}$ 9, 6 and 9 in $W_1$, $W_2$ and $W_3$ respectively. However, it is discarded by $W_4$ as its $O_{min}$ has $F_{score}$ = 11 which is larger than 10. The object with $F_{score}$ 4 is discarded immediately by each of the active windows for the same reason. Figure 5.2 (bottom right) shows the updated MTopBand structure. Both the objects with $F_{scores}$ 4 and 10 are inserted as top-k results for the new window $W_5$.*

## 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 5.3.1** *M-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 5.1.1, 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.

**Example 5.3.1** *Consider a three query workload with window sizes $Q_1$.win = 100, $Q_2$.win = 1000 and $Q_3$.win = 1000000; slide sizes $Q_1$.slide = 10, $Q_2$.slide = 10 and $Q_3$.slide = 100; and $Q_1$.k = 10, $Q_2$.k = 100 and $Q_3$.k = 200. Here, the meta-query $Q_{meta}$ generates approximately 10020 predicted views and amongst them around 1000 future windows will maintain 200 objects ($Q_3$.k = 200) each. Thus, around 2 million objects needs to be maintained in the M-MTopBand for generating accurate top-k results when maximum $K$ required is only 200.*

We confirm this inefficiency of MTopBand structure when the number of predicted views grow large in the experimental study discussed in Chapter 9. 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.

# Chapter 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 5.1.1, 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 [Theorem 5.1.1]. 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 4.1].

Since the top-k parameter $k$ of each of these queries may differ, the number top-k objects maintained in each predicted view may also differ. This implies that if an object $O_i$'s rank in the top-k result set is greater than $k_{smallest}$, the smallest top-k parameter of any query $Q_i$ in the workload, it will disappear from the top-k result sets of all those predicted views that end at the output moment of the query $Q_i$. $O_i$ may reappear in the subsequent predicted views that end at time points when other queries with top-k parameter greater than $k_{smallest}$ need output. Object $O_i$ will reappear only if it is still alive and no other new object with an $F_{score}$ greater than $O_i$ has arrived at the system.

**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.

The idea is to only maintain one copy of each of the distinct objects among the MTop-Band top-k result sets across the current window and future windows in an integrated list MTopList, rather than saving the overlapping results multiple times namely one for each future window they participate in More specifically, each object in the MTopList may participate in the top-k results of the current window and one or more future windows.

MTopList is sorted by $F_{scores}$ of these distinct objects. Figure 6.1 shows the MTopList structure based on the workload WL of the three queries Q1, Q2, and Q3 introduced in Example 4.2.1. Note that Figure 5.1 depicts the MTopBand structure for the same example.

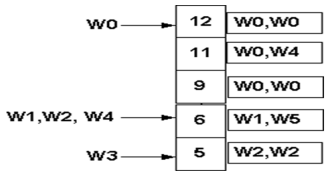


Figure 6.1: Physical view of MTopList

MTopList shown in Figure 6.1 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 5.1).

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 6.1.1** *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 6.1.1, 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.

**Example 6.1.1** *Based on our running example (Example 4.2.1), Figure 6.1 shows the window marks associated with each object in the MTopList at time t = 8s. Object with $F_{score}$ 12 participates in only $W_0$, so both the start window mark and end window mark for 12 are [$W_0$, $W_0$]. Similarly the window marks for objects with $F_{scores}$ 11, 9 and 6 are [$W_0$, $W_3$], [$W_0$, $W_4$], and [$W_2$, $W_2$] respectively. Clearly, the number of window marks needed for each object is always a constant, only 2. Interestingly it is not dependent on the number of windows an object is participating in.*

**Lemma 6.1.2** *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 6.1.2, 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 6.1.3** *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

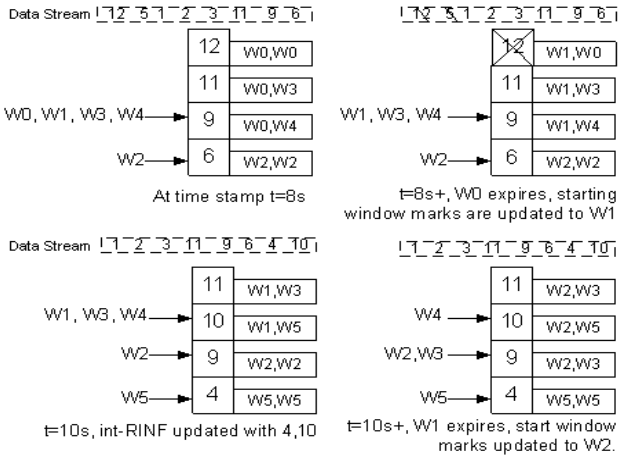


Figure 6.2: Updating the multi top-k results in integrated structure MTopList

**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(Chapter 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.

**Example 6.2.1** *As shown in Figure 6.2 (top right), immediately after time t = 8s, $W_0$ expires and the starting window marks of objects with $F_{scores}$ 12, 11, 9 and 6 are incremented from $W_0$ to $W_1$. Now for object 12, the starting window mark is $W_1$ and the ending window mark is $W_0$. This means that this object is not needed in any of the future windows and can be physically deleted from the list. The list after deleting 12 contains only 11, 9, and 6.*

**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 $O_{new}.end_{mark}$.

**Example 6.2.2** *As shown in Figure 6.2 (bottom left), object with $F_{score}$ 4 is compared with object with $F_{score}$ 6 ($min_{FP}$ of the newest window),4 being smaller than object with $F_{score}$ 6 could not be a part of top-k results for windows [$W_1$-$W_4$]. Thus the starting and ending window marks of this newly inserted object with $F_{score}$ 4 are updated to [$W_5$, $W_5$]. When object with $F_{score}$ 10 arrives, it is larger than the $min_{FP}$ objects in each of the windows. Thus, the starting and ending window marks of object 10 are updated to $W_1$ and $W_5$ respectively. Finally, we remove the object with $F_{score}$ 9(marked by $min_{FP}$ in $W_1$, $W_3$,*

*$W_4$ ) from windows $W_1$, $W_3$, $W_4$ and object with $F_{score}$ 6 (marked by $min_{FP}$ in $W_2$) from window $W_2$ respectively.*

Next, we present the pseudo-code for MTopList maintenance algorithms.

$O_{new}$: newly arriving object.
$WL$: input multi query workload. $RMQS_{ins}$: runtime instructions from scheduler.
$Q_i$: member query in the workload WL that needs output when RMQS$_{ins} = TimetoOutput$.
k$_i$: top-k parameter of query $Q_i$ OR maximum top-k parameter amongst all queries that need output.
$O_{curr}$: current object count.
$min_{FP}$: minimum score mark of each window $W_i$.
$O_{min}$: MTopList.$O_i.min_{FP}.F_{score}$.
$O_{max}$: MTopList.$O_i.F_{maxscore}$.
$W_{oldest/newest}$:oldest/newest $W$ in MTopList.
$start_{mark}$: window starting mark for an object.
$end_{mark}$: window ending mark for an object.
$N_{act}$:Number of active windows

***MTopList*(**$S, RMQS_{ins}$**)**

**1 for** each new object $O_{new}$
**// if time to slide, purge window**
**2 if** $RMQS_{ins} = TimetoOutput$
**3 OutputTopK(**$Q_i$**)**;
**4 PurgeExpiredWindow(**$W_{oldest}$**)**;
**5 AddNewWindow(**$W_{newest}$**)**;
**6 UpdateMTopList(**$O_{new}$**)**;

**OutputTopK(**$W_i$**)**;
**1** output first $k_i$ top-k objects from MTopList;

**PurgeExpiredWindow(**$W_{oldest}$**)**
**1 for** first $k_i$ objects in the MTopList;
**2** $O_i.start_{mark} = O_i.start_{mark} + 1$;
**3 if** $O_i.start_{mark} > O_i.end_{mark}$;
**4 remove** $O_i$ from the MTopList;
**5 set** $W_{oldest}.O_{curr} = zero$;

**AddNewWindow(**$W_{newest}$**)**
**1 if** $W_{oldest}$ is expired;
**2** create newest future window $W_{newest}$;
**3 set** $W_{newest}.O_{curr} = zero$;

Figure 6.3: MTopList Algorithm - *part 1*

**UpdateMTopList(**$O_{new}$**)**

**01** **if** $W_{all}.O_{curr} == k_i$
**02** **if** F($O_{new}$) $<$ F($O_{min}$)
**03** discard $O_{new}$;
**04** **if** F($O_{new}$) $>$ F($O_{max}$)
**05** Add $O_{new}$ to MTopList as $O_{max}$;
**06** Update $O_{new}.start_{mark}$ = $W_i$ and $end_{mark}$ = $W_{N_{act+i}}$;
**07** Update $min_{FP}$ of each active window $W_i$;
**08** **if** F($O_{new}$) $>$ F($O_{min}$) F($O_{new}$) $<$ F($O_{max}$).
**09** position $O_{new}$ into MTopList.
**10** Update $O_{new}.start_{mark}$ = $W_i$ and $end_{mark}$ = $W_{N_{act+i}}$;
**11** Update $min_{FP}$ of each active window $W_i$;

Figure 6.4: MTopList Algorithm - *part 2*

# Chapter 7

# Extracting output for each member query - Query Result Extractor(QRE)

QRE extracts the exact top-k results from the data structure for each member query in the query group at a given moment. QRE picks the $Q_{meta}$ output from integrated data structure each time it is triggered by instructions from the dynamic scheduler RMQS. These instructions are primarily 1. time to output, 2. query/queries that need output at the time when the current window ends.

In the single query top-k processing the slide size is fixed which means query result at any given output moment corresponds simply to the objects that are purged from the system at the time of window slide. On the other hand, MTopList algorithm maintains all the top-k results for many queries together. At the time of output, the current-to-output window may contain the top-k results for more than one query (each with different top-k parameters). Thus the top-k results of different member queries are completely interleaved.

QRE separates the top-k results required for the given member query at a particular

output moment in the following steps. First, RMQS triggers the QRE each time when one or more member queries need output. The RMQS instructions contain the list of member queries served by the next output window. Based on these instructions, QRE picks the top-k objects from the current-to-expire window from MTopList. Second, based on the top-k parameter of each query that needs output QRE separates the interleaved results and generates the specific top-k results for each member query. Next we continue our running Example 4.1 to explain the result extraction technique.

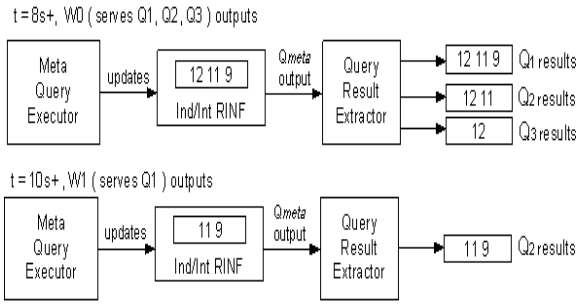


Figure 7.1: Final Top-k results for different member queries

**Example 7.0.3** *As shown in Figure 8.1 (top), RMQS instructs QRE: (1.) t = 8s - time to output $W_0$, (2.) $W_0$ serves $Q_1$ (k=3),$Q_2$ (k=2), and $Q_3$ (k=1). Based on these instructions, QRE picks the current top-k objects with $F_{scores}$ 12, 11, and 9 in $W_0$($Q_{meta}$ results) from MTopList.*

*Three different result sets, namely one for each of the member queries are generated from $Q_{meta}$ result. Object with $F_{score}$ 12, which is top-1 object in the output window $W_0$ is the final result for $Q_3$ ($Q_3$.k=1). Similarly, objects with $F_{scores}$ 12 and 11 ($Q_2$.K=2); and 12,11, and 9 ($Q_1$.k=3) are the final results for queries $Q_2$ and $Q_1$ respectively.*

Figure 8.1 (bottom) shows the next output moment, at time t=10(s), the RMQS triggers

QRE with the instructions: (1.) t=10(s) - time to output $W_1$, (2.) window $W_1$ serves query $Q_2$.

The processing at QRE is straightforward in this case as the output window $W_1$ serves only one query. So the final $Q_2$ result are objects 11, 9 which is exactly same as the objects stored in $W_1$ in the MTopList at time t=10s.

Once the exact query results have been successfully generated and given to the user, QRE acknowledges the dynamic scheduler RQMS so that to send the next set of instructions correctly.

# Chapter 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 skyband 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.

# Chapter 9

# Experimental Evaluation

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 JointSTARS. 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. In our experiment, we used all 14 dimensions of GMTI while detecting clusters based on the targets latitude and longitude. 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 query solution MinTopK [16] for each member query in the workload WL without sharing any of the window or k

parameters amongst these queries. 2. MTopBand, the basic algorithm we presented in this work (Section IV).

**Experimental Methodologies.** We measure two common metrics for stream processing algorithms, namely average processing time for each tuple (CPU time) and memory footprint. Experiments are designed to compare the performance of the proposed algorithm with the alternative algorithms.

First, 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 three test cases, each varying on only one of the three query parameters. Then we test the more general case with varying two parameters arbitrarily. Finally, we test the most general case with all three parameters *win*, *slide*, and *k* being arbitrary. For each experiment, we vary window size *win* in the range of 100K to 1 M, slide size *slide* between 10K to 100K, and top-k parameter *k* in the range of 10-1000.

Second, we conduct overhead evaluation tests to verify the performance of the proposed algorithms while processing a single top-k query or with small workloads. Here, we perform experiments for workload of 1, 2, and 5 queries each and compare the results of MTopList with the state-of-art optimal single query solution[16].

## 9.1 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 [15] while executing the workloads sized from 10 to 1000 queries. We use *win* = 1M and *slide* = 100K, while varying *k* from 10 to 1000. We randomly generate *k* between the range 10 - 1000 for each query.

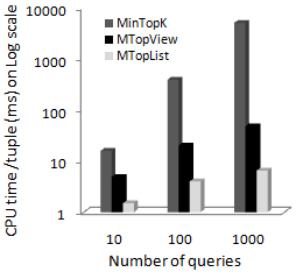


Figure 9.1: CPU time used by three algorithms with different $k$ values

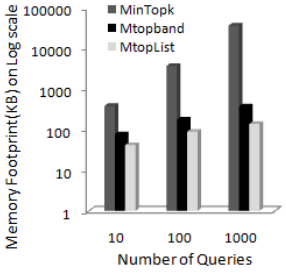


Figure 9.2: Memory space used by three algorithms with different $k$ values

As shown in Figures 11 and 12, both the CPU time and the memory space used by the three algorithms using logarithmic scale. Clearly, performance of our two methods is order of magnitude better. Amongst all three compared algorithms , MinTopK's [15] CPU time increases as a direct multiple of the size of workload. Naturally, 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.

On the other hand , the CPU time required by MTopList to process one tuple increases around 2 times when the number of queries grows one order of magnitude(10 to 100), and then it increases around 1.5 times when number of queries grows another order of magnitude(100 to 1000). Whereas the CPU time for the basic algorithm we presented in this work, MTopBand, increases around 3 times when the number of queries increased from

10 to 100, and it further increases around 2 times with cardinality 100 to 1000.

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.

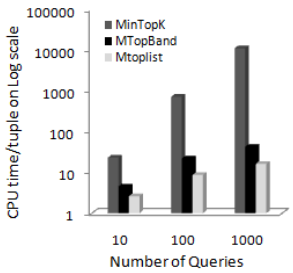


Figure 9.3: CPU time used by three algorithms with different $slide$ values

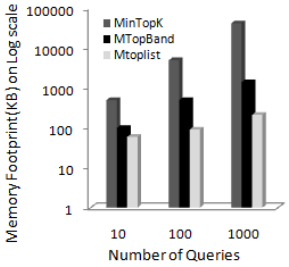


Figure 9.4: Memory space used by three algorithms with different $slide$ values

As shown in Figures 13 and 14, both the CPU and memory usage of MTopList is still significantly less than those utilized by state-of-art algorithm MinTopK [16] in all test cases. 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 independently, thus its resource utilization is expected to increase linearly with the number of future windows maintained, which is equal to $win/slide$.

Thus, MTopBand's increase in resource utilization is expected because *slide* is randomly picked in this case, 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 by the change in the win/slide ration. This is because MTopList only maintains distinct top-k objects which are not dependent on number of predicted views.

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

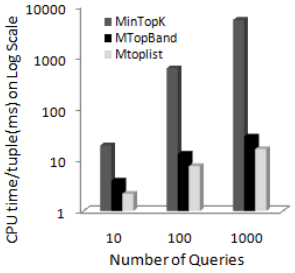


Figure 9.5: CPU time used by three algorithms with different $win$ values

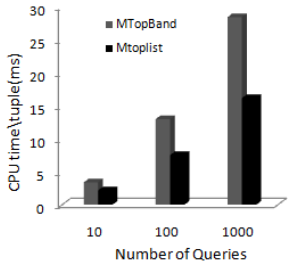


Figure 9.6: Zoomed in version of figure 15 - Comparison of CPU time used by only MTopBand and MTopList

Both the CPU and memory usage of the state-of-art algorithm MinTopK increases dramatically as the number of queries increases while the usage of proposed algorithms MTopList and MTopBand are still significantly less than MinTopK. More specifically, for processing 1000 queries with varying $win$ values, MinTopK takes a total of 3.112 seconds, MTopBand takes.038 seconds and MTopList takes only .016 seconds. Thus our proposed

algorithm MTopList takes around 1500 time less CPU time than the state-of-art algorithm MinTopK.

An important observation made in the scenes of varying slide sizes are valid here as well. More specifically, in this case as well the performance of MTopBand is affected by the $win/slide$ value of the queries. We note that, MTopList consumes at least 100 percent less CPU time than MTopBand for processing 1000 queries.

**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. Second, 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.

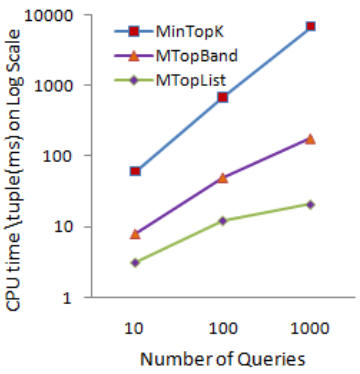


Figure 9.7: CPU time used by three algorithms with different $win$ and $slide$ values

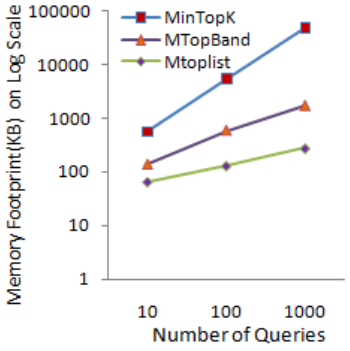


Figure 9.8: Memory space consumed by three algorithms with different $win$ and $slide$ values

As shown in Figures 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 and around 4.5 times from 100 to 1000 queries. 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.

Although 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.

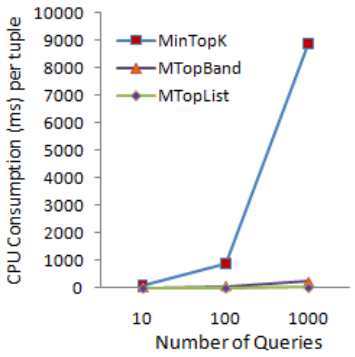


Figure 9.9: CPU time used by three algorithms with arbitrary $win$ and $slide$ parameters

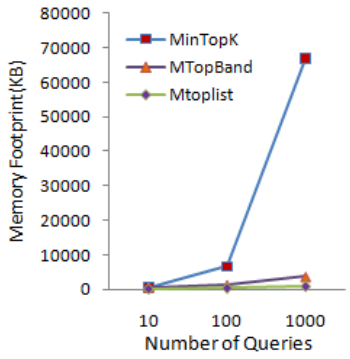


Figure 9.10: Memory space consumed by three algorithms with arbitrary $win$ an

Figure 19 and 20 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.

## 9.2 Overhead Evaluation

Next, we compare the performance of the proposed algorithms MTopList and MTopBand in terms of both CPU and memory utilization with the state-of-the-art single query solution for small workloads of size 1, 2 and 5 queries.

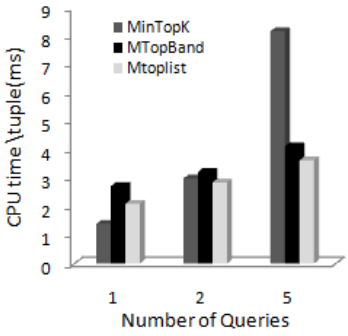


Figure 9.11: CPU time used by three algorithms for processing 1, 2, and 5 queries

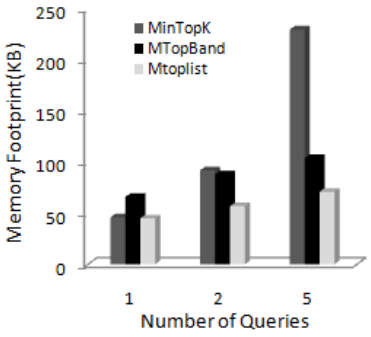


Figure 9.12: Memory space consumed by three algorithms for processing 1, 2, and 5 queries

As shown in figure, we evaluate the overhead incurred by our proposed algorithms due to simultaneous processing of multiple top-k queries. As discussed, in section IV before actual top-k query processing, we first analyze the parameters of each of the input queries in the workload so as to integrate all the queries into a single meta query.

MTopList takes maximum CPU time amongst the three algorithms for processing single query. More specifically, MTopList consumes 2.8 ms/tuple , while MTopBand and MinTopK consume 2.1 and 1.3 ms/tuple respectively. This is expected as MinTopK algorithm does not analyze the parameters to identify the sharing opportunity. MTopList takes only 3.1 and 3.7 ms, while MinTopK requires 3.8 and 8.5 ms to process a workload of cardinality 2 and 5 respectively. Thus we conclude that the CPU overhead incurred by our proposed algorithms becomes negligible as we increase the cardinality of workload. This is because the savings achieved in the actual top-k query processing supersedes the extra analysis time consumed by our algorithms tremendously.

Memory wise, MTopList always maintains minimum amount of tuples and incurs no overhead at all. Thus, for processing one query both MinTopK and MTopList require equivalent memory space. While, our basic algorithm, MTopBand, consumes more memory space as it maintains multiple references for the same object if the object participates in more than one window. However as the cardinality of workload is increase from 1 query to 2 queries, the memory space consumption of MinTopK becomes almost equal to MTopBand and it increases to 1.5 times more than MTopList. For a workload of 5 queries MinTopK consumes memory space 2.5 times and 5 times more than MTopBand and MTopList respectively.

# Chapter 10

# 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 also confirm that MTopS's processing overhead attributed to query parameter analysis is minimal and it wins over state-of-art solutions even for workloads of very small size. MTopS also exhibits excellent scalability in terms of being able to handle thousands of queries under high speed input streams in our experiments.

An important area of improvement is using this framework to scale-up considering multiple machines and grouping of workloads into sub-workloads to be assigned to different

machines. We believe that the techniques proposed in this work, can be extended for such parallel processing of multiple top-k queries.

Another major research direction is to study other data mining queries utilizing this framework such as outliers, associations etc. The component based design of our framework can be reused for shared processing of other types of multiple queries.

# Chapter 11

# Bibliography

[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 $SIGMODconference$, pages 391-402, 2000.

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

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

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

[6] P. Haghani, S. Michel, and K. Aberer. Evaluating top-k queries over incomplete data streams. In CIKM, pages 877886, 2009.

[7] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In $SIGMOD$, pages 346357, 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 Proceedings of ICDE (1999).

[10] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In ICDE, pages 189200, 2003.

[11] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. TODS,27(2):153187, 2002.

[12] C.-M. Chen and Y. Ling. A sampling-based estimator for top-k query. In ICDE, pages 617627, 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, pages 23-28.

[15] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In $SIGMOD$, pages 635646, 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 Proceedings of EDBT, pages 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):301312, 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):748762, 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, pages 189200, 2003.

[21] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In VLDB, pages 648659, 2004.

[22] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. In ICDE, pages 277, 2003.

[23] A. Arasu, Jennifer Widom. Resource Sharing in Continuous Sliding-Window Aggregates. In proceedings of the 30th VLDB Conference, Toronto, Canada, 2004.

[24] M. Theobald. TopX Efficient and Versatile Top-k Query Processing for Text, Structured, and Semistructured Data. Dissertation, Max-Planck-Institut fr Informatik, April 2006.