CPR: Complex Pattern Ranking for Evaluating Top-k Pattern Queries over Event Streams

by

Xinxin Wang

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2011 by the
Graduate Supervisory Committee:

Kasim Selçuk Candan, Chair
Yi Chen
Hasan Davulcu

ARIZONA STATE UNIVERSITY

May 2011

ABSTRACT

Most existing approaches to complex event processing over streaming data rely on the assumption that the matches to the queries are rare and that the goal of the system is to identify these few matches within the incoming deluge of data. In many applications, such as stock market analysis and user credit card purchase pattern monitoring, however the matches to the user queries are in fact plentiful and the system has to efficiently sift through these many matches to locate only the few *most preferable* matches. In this work, we propose a complex pattern ranking (CPR) framework for specifying top-k pattern queries over streaming data, present new algorithms to support top-k pattern queries in data streaming environments, and verify the effectiveness and efficiency of the proposed algorithms. The developed algorithms identify top-k matching results satisfying both patterns as well as additional criteria. To support real-time processing of the data streams, instead of computing top-k results from scratch for each time window, we maintain top-k results dynamically as new events come and old ones expire. We also develop new top-k join execution strategies that are able to adapt to the changing situations (e.g., sorted and random access costs, join rates) without having to assume *a priori* presence of data statistics. Experiments show significant improvements over existing approaches.

DEDICATION

To my family for their love and support

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank all the important people who encouraged me and contributed towards an extremely satisfying and invaluable educational as well as research experience at Arizona State University over the past three years. I would like to express my sincere gratitude to my advisor, Dr. K. Selcuk Candan, the chairman of my advisory committee, for his continuous and invaluable guidance throughout my Master's program. He has been a great source of motivation and support throughout my research at ASU. This thesis could not have been completed without his continuous direction and support. I am also thankful to Dr. Hasan Davulcu and Dr. Yi Chen for agreeing to serve on my graduate committee. I would also like to thank my friends and colleagues at the Emit Lab: Renwei, Wei, Songling, Shruti, Mithila, Yan, Kim, Parth, Huiping, JungHyun and Mijung for helping me out through all the difficulties encountered during my research. Especially to Yan and Kim, as the senior in the lab when I first came, they gave me so much help so that I can adjust to my research role smoothly. I also want to mention my friends Xufei, Yichuan, Peng, Chuan, Lei, Jin and Ke for creating such an enjoyable atmosphere that has made the entire duration of my graduate study a memorable experience. I would like to thank my parents and my wife who have always been there with me no matter what decisions I make.

TABLE OF CONTENTS

Page

TABLE OF CONTENTS iv

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER 1

1 INTRODUCTION 1

1.1 Contributions 2

2 EXISTING WORK 4

2.1 Complex Event Processing 4

2.2 Top-k Join Processing 5

2.3 K Shortest Paths 5

3 PROPOSED FRAMEWORK 7

3.1 Overview of the Complex Pattern Ranking Framework (CPR) 7

Notation and Query Language 7

Architectural Overview 9

3.2 Sequence Ranking 11

Stratified Stream Graph 11

Preference Function and Vertex Weights 12

Stratified K Shortest Paths Algorithm 13

Finding a Shortest Path on a Stratified Graph 14

K Shortest Paths on a Stratified Graph 17

K Shortest Paths on Dynamic Graphs 20

Nonlinear Preference Sequence Ranking 23

3.3 Top-k Merge Module (TMM) 24

Thresholds, Boundaries, and Scheduling 24

Waste-Avoiding Boundary Selection 26

3.4 Dynamic Maintenance of Top-k Complex Query Matches 28

Dynamic Linear Preference Sequence Pattern Match Production 30

Dynamic Nonlinear Preference Sequence Pattern Match Production 32

Chapter Page

Dynamic Top-k Complex Query Matching Results Maintenance 34

4 EXPERIMENT EVALUATION 36

4.1 Data Set Specification 36

4.2 Setup 37

4.3 Evaluation of Top-k Sequence Enumeration on Dynamic Stratified Graphs . 38

4.4 Evaluation of Waste-Avoiding Top-k Joins 40

Waste-Avoiding Boundary Selection Strategy on Sensor Data 41

Waste-Avoiding Boundary Selection Strategy on Synthetic Data 43

4.5 Evaluation of Dynamic Top-k Merge Module (dTMM) 46

5 CONCLUSION 49

REFERENCES 50

LIST OF TABLES

Table Page

4.1 Value ranges represented by each event class 37

4.2 Experiment parameters 37

4.3 Sequence patterns used in the experiments 38

4.4 Experiment parameters 43

LIST OF FIGURES

Figure Page

1.1 An example shopping pattern query on credit card purchasing data 2

3.1 Top-k query pattern 8

3.2 Architectural overview of CPR framework 10

3.3 Stratified graph example 12

3.4 Stratified shortest path algorithm (*SSP*) 15

3.5 Next stratified shortest path algorithm (NSSP) 18

3.6 k stratified shortest path algorithm (k-SSP) 18

3.7 Sharing of top k results between moving windows 20

3.8 k dynamic stratified shortest paths algorithm (k-DSSP) 21

3.9 Dynamic top-k sequence match maintenance for nonlinear preference sequence queries 23

3.10 Two alternative termination *boundaries* that provide the same threshold (0.5) needed to commit top-3 candidates 25

3.11 Execution process of waste-avoiding boundary selection (WABS) 26

3.12 Waste-avoiding boundary selection strategy (WABS) 27

3.13 Revision of the $(i-1)^{th}$ boundary 29

3.14 Overall Process for Dynamic Complex Query Maintenance 29

3.15 Dynamic PQ-Array update for linear preference sequence queries (dUpdate-LP) 30

3.16 Incremental next stratified shortest path(iNSSP) 31

3.17 Dynamic update for nonlinear preference sequence queries(dUpdate-NLP) . . . 32

3.18 Incremental next nonlinear preference sequence (iNNLPS) 33

3.19 Dynamic top-k complex pattern query match maintenance in TMM (dTMM) . . 34

4.1 Performance of the original Yen's algorithm vs. D-Yen over dynamically evolving graphs 39

4.2 Comparison of top-k sequence matching approaches (k=10, sl=3, ws=50, step=20, ref=89) 39

4.3 Comparisons of *D-Yen* and k-*DSSP* for different parameter settings (Default: k=10, sl=3, ws=50, step=20, ref = 89X) 40

Figure Page
4.4 Different top-k join strategies (mf=avg, rss=1, q=2, ws=step=100, qp=[S7;S8]) . 41
4.5 Performance evaluation on two-sequence join queries on sensor data (Default: k=20, mf=avg, rss=1, q=2, ws=step=100, qp=[S7;S8], sl = 3 for both queues) . 41
4.6 Performance evaluation on three-sequence join queries on sensor data (k=20, rss=0.01, mf=avg, q=3, sl=3 , ws=step=100) 43
4.7 Comparison for varying access cost scenarios (Default: k=50, mf=min, rjs=1, uniform) 44
4.8 Performance comparison for varying parameter setting on synthetic data (Default: k=50, q =2 , mf=min, rjs=1, dist=uniform, cost ratio=[10:1, 1:1]) 45
4.9 Comparison of Dynamic *TMM* (*dTMM*) and non-dynamic *TMM*(*nTMM*) (Default: k=20, rss=1, mf=avg, q=2, q=[S7;S8], sl=3, ws=100, step=50) 46
4.10 Effect of nonlinear preference function in sequence query on Dynamic *TMM* (*dTMM*) (Default: rss=1, mf=avg, q=2, q=[S7;S8], sl=3, ws=100, step=50, pref(S7)=sum, pref(S8)=square root of sum of square) 47

Chapter 1

INTRODUCTION

Complex event processing systems usually deal with the examination of interesting event patterns that occur within data arriving in the form of a stream. This problem is also known as the *pattern matching* problem: given a pattern of interest (represented in different forms under different models), the system needs to discover all matching event instances of the query within the stream.

Most existing approaches to complex event processing (e.g., [26, 7, 17, 9]) rely on the assumption that the matches to the pattern queries are rare and that the goal of the system is to identify these rare matches within the incoming deluge of data. In many applications, such as credit card purchase pattern monitoring, however the matches to the user queries are in fact plentiful and the system has to efficiently sift through these many matches to locate only the few *most preferable* matches.

Figure 1.1 presents a sample query for an application tracking shoppers for expensive brand goods. In this example, the input is a stream of credit card purchase transactions. The query aims to identify the heavy brand shoppers: more specifically, the query seeks individuals who purchase three goods from brand-A, brand-B, and brand-C in sequence within 10 hours and meanwhile two other goods from brand-D and brand-E within 8 hours; among all the matches, we are interested in identifying 10 matches with the highest overall shopping expenditure and the smallest shopping span between purchase of brand-A and brand-C.

Despite extensive work in complex event processing (see Chapter 2), there has been very limited previous work that focuses on challenges that involve processing such top-k pattern queries over data streams. Many works, such as [6, 13, 14], assume that the tuples to be ranked or aggregated arrive in the stream that are already materialized in the form of sensor readings or documents. We on the other hand focus on *pattern matches*: the patterns to be ranked need to be discovered on the fly and we want to avoid the cost of enumerating all pattern matches.

```
SEQ       S1 = A; B; C
WITH      A=(purchases in Brand-A),
          B=(purchases in Brand-B),
          C=(purchases in Brand-C)
WITHIN    10 hours
WHERE     A.CCID = B.CCID AND B.CCID = C.CCID
PREF      MAX[S1.exp = A.exp + B.exp + C.exp] AND
          MIN[S1.t = C.t − A.t]
RETURN    10

SEQ       S2 = D; E
WITH      D=(purchases in Brand-D),
          E=(purchases in Brand-E)
WITHIN    8 hours
WHERE     D.CCID = E.CCID
PREF      MAX[S2.exp = D.exp + E.exp]

SEQ       P1 & P2
WITH      P1 = (sequence pattern S1),
          P2 = (sequence pattern S2)
WITHIN    10 hours
UPDATE    5 hours
WHERE     S1.A.CCID = S2.D.CCID
PREF      MAX[S1.exp+S2.exp] AND MIN[S1.t]
```

Figure 1.1: An example shopping pattern query on credit card purchasing data

1.1 Contributions

In this thesis we propose a *complex pattern ranking (CPR)* framework for specifying top-k pattern queries over streaming data, present new algorithms to support top-k pattern queries in data streaming environments and verify the effectiveness and efficiency of the proposed algorithms. The proposed framework lets the user specify pattern queries including user specified preference functions on event attributes. The algorithms we develop identify top-k matching results satisfying both the pattern query as well as additional criteria. The following is the list of our salient contributions:

- We first present a language for top-k complex event processing and pattern ranking over data streams.
- We show that top-k sequence pattern matching problem can be posed in the form of a shortest path problem on so called *stratified stream graphs*. We propose a novel stratified-graph based k-shortest path algorithm (*k-SSP*) that leverages the stratified nature of the data graphs to locate best sequence match results quickly and incrementally. We then extend the algorithm to handle dynamically evolving data

graphs without wasting resources for redundant re-computations as data graph evolves over time (k-*DSSP*).

- We propose an adaptive join scheduling strategy for TA and NRA top-k join processing algorithms [11] to combine sequence pattern match results. The proposed strategy is designed to tackle run-time variations in the data streams (e.g., variations in sorted and random access costs, join rates) without having to assume *a priori* presence of reliable data statistics (which are often impossible to assume in data streams). In particular, when dealing with a top-k join query with varying costs for sorted- and random-accesses for different data queues, the proposed *waste avoiding boundary selection (WABS)* approach adapts and re-schedules access orders to minimize the overall join cost.

- We develop a system with a dynamic mechanism to maintain top-k joined results for complex queries, avoiding the costs of re-computing the previous results.

- We experimentally evaluate the algorithms and compare against existing solutions under various scenarios.

Chapter 2

EXISTING WORK

2.1 Complex Event Processing

Early works on complex event processing were motivated by the limitations of DBMSs to scale to the performance requirements of publish/subscribe applications [9]. One of the earliest complex event processing works is SASE [26], which combines a series of optimization strategies to deal with classic sequence pattern queries on large scale of stream data. SASE includes plan based query monitoring and takes advantage of an extended NFA scheme to handle multiple queries at the same time. Later improvements on this work include [2, 27]. [7] presents Cayuga, another general purpose event monitoring system. Similar to the SASE, Cayuga focuses on pattern monitoring over large data sets and provides an automata-based pattern matching engine. The query language syntax used in Cayuga is similar to relational database query languages and, compared to SASE, is able to express more general queries.

These two automata-based works have been followed by many other approaches to large scale pattern matching. For instance, in [17], a petri-net based query model is proposed. Recent work in complex event processing also focused on out-of-order and uncertain events. proposed a K-Slack algorithm with K-delayed purging; i.e,.an out-of-order event is discarded if it is beyond a K boundary. [21] also deals with out-of-order events but proposes a stack-based mechanism to manage the events. Instead of focusing on patterns, [6, 13, 14] focus on ranking and aggregation of tuples in the input stream. [24] considers uncertainties that exist in timestamps and event attributes and focus on efficient evaluation of aggregates and joins under accuracy requirements.

In this work, we note that most existing mechanisms (including the automata based approaches) are suitable for scenarios where one needs to retrieve all matches to the user's query. For top-k queries, where only a select few matches are required, these approaches would be highly inefficient due to their need to first retrieving all matches. Therefore, we present k shortest paths and top-k join based scheme which can handle top-k CEP queries efficiently, without enumerating all matches.

2.2 Top-k Join Processing

Ranked join algorithms, including [10, 11] and others, rely on weight-sorted input streams for pruning unpromising matches when identifying top matches to the users query. In one of the earlier works, Fagin [10] proposed an algorithm, known commonly as the FA, which assumes the availability of both the sorted and random accesses to the data. FA considers data from the sources in (progressively) descending order of desirability and, with the help of random accesses, enumerates top-k join results without having to access all the data from these sources. Later improvements to FA include the threshold algorithm (TA), which pro-actively schedules random accesses when they are cheap, and the no-random access algorithm (NRA), which is applicable when random-accesses are either unavailable or extremely costly [11].

Among these and other variants to the top-k join processing problem, a key challenge is to identify an appropriate *schedule* which states in which order the incoming data queues will be considered and how sorted and random accesses will be alternated. [12] attempts to leverage a greedy gradient based heuristic for reaching the threshold condition earlier. [4] assumes a priori knowledge of the distribution for each join queue and predicts the potential costs and rewards for expanding each queue. [25] addresses scenarios where different predicates have different costs. The algorithm collects statistics and picks an appropriate schedule in the runtime. In contrast, in this thesis, we focus on situations (common in distributes scenarios) where reliable statistics about incoming streams are hard to obtain and that access costs and join rates can change over time.

2.3 K Shortest Paths

In this thesis, we rely on graph-based techniques to enumerate sequence patterns. The k-shortest paths problem dates back to early 1950s. Early solutions include Yen's algorithm [18], which is known to have a computational complexity of $O(kv^3)$, where v is the number of vertices. Later works by Lawler [20] and Katoh *et al.* [19] improved the basic Yen's algorithm, but the worse case complexity stays $O(kv^3)$. A recent improvement on

Yen's approach can be found in [15]. Another approach to the k-shortest paths, differing significantly from Yen's work, with worst running complexity of $O(m + vlog(v) + klog(k))$, where m is the number of edges, has been proposed by Eppstein in [8] and a lazy version of this algorithm has been presented in [16]. While approaches similar to Yen's work on undirected graphs, Eppstein's algorithm works on directed graphs. Secondly, while Yen's algorithm returns only simple paths, Eppstein's algorithm can also identify paths with repeated nodes. However, while Yen's algorithm can work incrementally, Eppstein's approach is unable to obtain k-shortest paths in an incremental manner. Other approaches to this problem includes [3], an A-* search based approach that is able to deal with situations where only partial graphs can be loaded into the main memory. This also has the same worst case complexity as Eppstein's algorithm and, similarly, is not incremental.

In this thesis, we propose k shortest paths algorithms that leverage the stratified nature of the data graphs that need to be considered during CEP processing for improved efficiency and that incrementally maintain the list of top-k shortest paths in the presence of dynamically evolving graphs.

Chapter 3

PROPOSED FRAMEWORK

3.1 Overview of the Complex Pattern Ranking Framework (CPR)

Notation and Query Language

As is common in complex event processing literature, we use **E** to represent event classes and **e** for event instances belonging to event class, E. Each event class, E, has certain associated attributes (**E.A**) and different event instances may have different values for these attributes. The timestamp, **t**, represents arrival time associated to each event instance. Then we define a sequence pattern (**SEQ**) as follows:

SEQ	S_i
WITH	Event classes
WITHIN	Window constraints
UPDATE	window update constraints
WHERE	Additional constraints
PREF	Preference specification
RETURN	Number of results to be returned

Where, a sequence pattern (S_i) consists of a series of event classes defined in the *WITH* clause. In *WITHIN* and *UPDATE* clauses, we define the time window and corresponding window update constraints for S_i. We only consider event instances within the time window as valid and shift the window at the time frequency defined by *UPDATE*. *UPDATE* is adopted here to reflect the freshness requirement for a given sequence query. In addition to these patterns, users can specify predicates on the attributes of involved events; we refer to these as additional constraints. The user can also provide a preference specification stating how the sequence matches should be ranked, where a sequence match is the combination of event instances that satisfy the sequence pattern. The *RETURN* is an optional clause, which specifies how many matches to return. In complex pattern queries shown later, each sub-query is one component and the *RETURN* clause is omitted, since user only needs to the number of top complex query matches and

PATTERN	Sequence/Conjunction/Disjunction
WITH	Sequence Patterns (SEQs)
WITHIN	Window constraints
UPDATE	Window update constraints
WHERE	Additional constraints
PREF	Preference specification
RETURN	Number of results to be returned

Figure 3.1: Top-k query pattern

sequence matches will be produced as needed.

For complex pattern queries, we consider three patterns over different sequence pattern: *Sequence, (S_i ; S_j)$_w$*, means all event instances of a sequence match of S_i occur before that of S_j within a time window w. *Disjunction, (S_i || S_j)$_w$*, means a sequence match of either S_i or S_j occurs within w. *Conjunction, (S_i & S_j)$_w$*, means sequence matches of both S_i and S_j must occur within w. More complex patterns can be constructed by replacing S_i and S_j above with other patterns. We do not support the kleene closure and negation patterns discussed in SASE [1].

The overall structure of a complex query statement is given in Figure 3.1. The *PATTERN* clause includes the pattern specification statements on sequences specified in the *WITH* clause. Similar to the sequence pattern, the *WITHIN* clause defines the *time window* within which we consider sequence matches as still being valid. Note that the window defined here provides a further constraint for each window defined by a sequence query. If one sequence query has a current window w_i beyond the overall window w, we only consider the part w_i overlapping with w as valid. The *UPDATE* clause specifies how often we update the overall match results. In the *WHERE* part, additional constraints on attributes of events cross different sequence patterns are provided. The preference specification is provided within a *PREF* clause, which decides how scores from different sequence patterns are merged. The *RETURN* clause specifies the number of target results. An example was provided in Figure 1.1.

Architectural Overview

We propose to tackle the problem of top-k pattern detection over data streams by splitting it into three sub-problems:

- *Top-k sequence detection and maintenance:* The first problem we tackle in this thesis work is to detect and maintain top-k sequence matches. Unlike the automata-based approaches to pattern detection, in this thesis, we propose a graph-based approach. In particular, we show that given a sequence pattern, related events in the current window can be modeled as a *stratified graph*. We then develop efficient top-k shortest path algorithms for the detection and dynamic maintenance of top-k sequences over stratified graphs.
- *Top-k complex pattern detection:* As we mentioned earlier, complex patterns can be seen as combinations of simpler sequence patterns. Therefore, top-k complex patterns can be enumerated by joining matches for sequence patterns. The challenge, of course, is to perform this combination efficiently and obtaining top-k complex pattern matches without having to enumerate too many matches for the constituting simple sequence patterns.
- *Dynamic Top-k complex pattern match maintenance:* In addition to dynamic maintenance of each individual top-k sequence matches across neighbor time windows, we also want to maintain the top-k complex pattern matches dynamically. The challenge here is how to utilize the previous computed results so that the re-computation costs can be avoided.

As can be seen in Figure 3.2, our CPR framework consists of three modules: (a) an *event processing module (EPM)*, (b) a *sequence ranking module (SRM)*, and (c) a *top-k merge module (TMM)*. *EPM* handles query registration and event dispatching. In *SRM*, we have one data structure for each sequence pattern in the query. *SRM* exposes access interfaces for sorted and random accesses for the next *TMM* module that combines partial results it pulls from its input queues into complex pattern rankings.

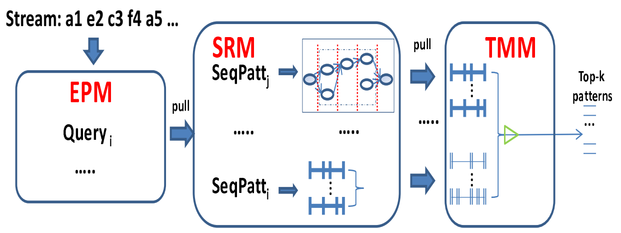


Figure 3.2: Architectural overview of CPR framework

Since the costs of sorted access and random accesses to different pattern queues can be drastically different (and may depend on the incoming data), it is the responsibility of *TMM* to decide in which order the partial results will be pulled from the input queues and when random accesses (as opposed to sorted accesses) will be scheduled. The *SRM* module operates on an *on-demand* basis and *incrementally* produces additional sequence matches (in decreasing order of preference) as requested by *TMM*.

The output of the system is the list of top-k matches to the CEP query within the current window, ordered by the preference function, and maintained current as the stream evolves. When the time window moves, the top-k results are updated dynamically: this involves dynamically updates to the sequence matches by *SRM* and dynamically updates to the top-k merged matches by the *TMM*.

Example: Let us reconsider the query in Figure 1.1, where we have two sequence queries S_1 and S_2. The first sequence query S_1 can be considered as two sub sequence queries $s_{1,1}$ and $s_{1,2}$, where $s_{1,1}$ queries “A;B;C” sequence with Max[A.exp+B.exp+C.exp] and “A.CCID” = “B.CCID” = “C.CCID”. $s_{1,2}$ ranks “A;C” sequence with “A.CCID” = “C.CCID” based on $[C.t - A.t]$. The “WINDOW” constraint for both of them stays the same as in S_1. Together in $s_{1,1}$ and $s_{1,2}$, we seek to find instances of the sequence "A;B;C" with the highest total purchase expense and the shortest time span between A and C. In S_2, we simply look for sequence matches of “D;E” with highest total purchase expense. Overall, we try to find the person that has highest expenditure sum over “A;B;C” and “D;E”, and the shortest time span over purchase of “A” and “C”. Suppose the events in current window are “a1 d2 b3 e4 a5 c6 e7 c8”. *EPM* picks events “a1 b3 a5 c6 c8” and forwards them to *SRM* in the form of a stream. Simultaneously, *EPM* also picks “a1 a5 c6 c8” and forwards them to another instance of *SRM* as a different stream. *SRM* enumerates “A;B;C” sequence matches in decreasing order of total expenditure and makes these available to *TMM*. The second instance of *SRM* enumerates the “A” and “C” pairs in decreasing order of span and makes these available to *TMM*. For S_2, similarly, *EPM* picks “d2 e4 e7” to another *SRM* instance, where matches are presented in decreasing order of total expenditure by *SRM*. Finally, *TMM* performs rank-joins on the “A;B;C” and “A;C” sequence matches based on the IDs of “A” and “C” events, simultaneously rank-joins on the “A;B;C” or ”A;C” with matches of “D;E” based on the IDs of “A” and ”D” to identify the top results best satisfying both the overall preference requirements. Since the overall preference specification in Figure 1.1 includes an “AND” operator, in this case *TMM* will use an appropriate score merge function (such as “min” [10]) that can represent fuzzy conjunction.

3.2 Sequence Ranking

In this section, we propose a graph-based approach to discover highly ranked sequences within an input stream. This algorithm forms the core of the sequence ranking module (SRM) and operates when the preference function is *linear*[1]. When the preference function is nonlinear, the enumeration process is handled directly by the top-k merge module.

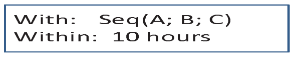
Stratified Stream Graph

Given a sequence pattern, S, and a finite stream window, str, we represent the input pattern and the data in the stream, in the form of a *stratified graph*.

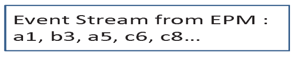
Definition 1 (Stratified Graph)**.** *An acyclic directed graph $G(V,E)$ is a p-*stratified *graph if the set of vertices V can be partitioned into p non-overlapping sets, V_1 through V_p, such that the vertices in partition (or* stratum*) V_i have incoming edges only from the vertices in V_{i-1} and have outgoing edges to only those vertices in V_{i+1}.*

The stratified graph is constructed as follows: Firstly, for each event type in the sequence pattern, we initialize a set (or stratum) of vertices. Then, for each event instance

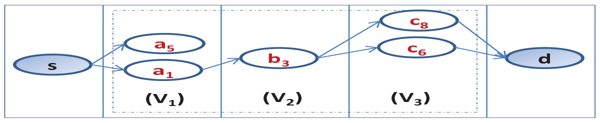
[1]In fact, any *monoid* function would be admissible. For simplicity, in this thesis, we only consider linear preference functions for sequence ranking by SRM.


With: Seq(A; B; C)
Within: 10 hours

(a) Sequence pattern


Event Stream from EPM :
a1, b3, a5, c6, c8...

(b) Input stream



(c) Stratified stream graph

Figure 3.3: Stratified graph example

in the input data stream, we create one new vertex for each corresponding event type in the sequence pattern and insert these vertices into the corresponding vertex strata. Then, for each pair of event instances in strata V_i and V_{i+1}, we check if they satisfy the same order in the data stream and if so, we add a directed edge from the vertex in stratum V_i to the vertex in V_{i+1}. We also create two dummy vertices, “start vertex” and “end vertex” denoted as s and d, such that s points to all vertices in strata V_1 and d is pointed by all vertices in strata V_p. Figure 3.3 shows an example.

Note our constructed stratified graph is similar to the stack based match buffer in existing automata based approach (e.g. SASE [1]), which enables our algorithms described below to be easily incorporated into existing approaches.

Theorem 1. *Let S be a sequence pattern and let str be a finite data stream (e.g., constrained by a window). There exists a stratified graph, $G_{S,str}$, that corresponds to the pattern S and the stream str.*

The proof follows trivially from the construction process.

Preference Function and Vertex Weights

In a given stratified stream graph, each vertex corresponds to an event instance and these event instances can have associated attribute values. Given a *linear* preference function, $f()$, on these attribute values, we can then encode the contribution of each event-attribute to the preference function in the form of vertex weights: Let e_i of type E_j be an event instance in vertex stratum V_j and let the contribution of E_j to the *linear preference* function, $f()$, be $(c \times E_j.a_k)$ for attribute, a_k. Then,

- if we seek to minimize $f()$, we annotate the vertex corresponding to e_i with weight $(c \times e_i.a_k)$;
- if, on the other hand, we seek to maximize $f()$, we annotate the vertex weight $(-c \times e_i.a_k)$;

For example, in the query given in Figure 1.1, we want to find a sequence with the *largest* total expenditure. Thus, each event instance, e_i, will be annotated with $-e_i.exp$.

Theorem 2 (Ranked Sequence Enumeration)**.** *Given a stratified graph, $G(V,E)$, whose vertices are weighted according to a* linear *preference function, $f()$, (a) each sequence match corresponds to a path from the special node $s \in V$ to the node $d \in V$; and (b) the smaller the overall weight of the path is, the better its rank with respect to the target preference function and the given maximization/minimization criterion.*

The proof of the theorem follows from the associativity of the linear preference functions and acyclicity of the stratified stream graphs. This theorem enables us to reformulate the ranked sequence enumeration problem in the form of the ranked shortest path enumeration (or k-shortest path, KSP) problem over the given weighted stratified stream graph. While there are a number of solutions to the KSP problem, including Yen's [18, 23] and Eppstein's [8, 16] algorithms, a straight-forward adoption of existing algorithms KSP algorithms is not appropriate for ranked sequence enumeration over data streams: (a) firstly, existing KSP algorithms are not designed to take advantage of the stratified and vertex-weighted nature of the stream graphs; (b) secondly (and more importantly) existing KSP algorithms are not able to deal with dynamically evolving graphs (due to shifting of the data window) efficiently. Therefore, we first develop a KSP algorithm that leverages the stratified nature of the stream graphs and then discuss how to extend this algorithm to dynamically evolving graphs.

Stratified K Shortest Paths Algorithm

In this subsection, we develop an efficient k-shortest path algorithm for *static* stratified graphs. We note that stratified stream graphs are (a) acyclic and directed, (b) edges exist only between neighboring strata, (c) (assuming that events are indexed such that the earlier an event, the lower its in-stratum index) if there exists an edge between vertex pair from $v_{i,j}$, (i.e. vertex v_i in stratum V_j), to vertex $v_{i',j+1}$, (i.e. vertex v'_i in stratum V_{j+1}), there exist edges between all pairs $v_{m,j}$, and $v_{i',j+1}$, $1 \leq m \leq i$, and (d) the graph is vertex-weighted (as opposed to edge weighted).

We leverage the stratified nature of the input graph to construct a **p**attern **q**uery **array** (PQ-Array) data structure[2]: For each stratum, V_j, we create an array A_j; then, for each vertex, $v_{i,j} \in V_j$ (i.e. the i^{th} vertex of V_j), we insert a triple $\langle minWeight, hPtr, vPtr \rangle$ into A_j:

- *minWeight* stores the best known total vertex weight (i.e. shortest path weight) from s to any $v_{i',j}$, $i' \leq i$;
- *hPtr* points (*horizontally*) to the latest vertex in the previous stratum that has an edge to $v_{i,j}$; and
- *vPtr* points (*vertically*) to the vertex that is *before or identical to* $v_{i,j}$ in stratum V_j and has the smallest *minWeight*. If two events before or identical to $v_{i,j}$ have the same smallest *minWeight*, then *vPtr* points to the later event (i.e. event with larger in-stratum index).

Initially, for each vertex, *minWeight* is set to ∞ and *vPtr* is set to *null* ($\perp$). The start tuple, s, has both its *hPtr* and *vPtr* as $\perp$ and its *minWeight* is set to 0.

Finding a Shortest Path on a Stratified Graph

We first develop an algorithm for finding the *shortest* path from a given source vertex v_s to d (the dummy end vertex) in a stratified graph (Figure 3.4), which is later used as a sub-module in the k-shortest paths algorithm. The stratified shortest path algorithm

[2]Note that PQ-Array data structure is reminiscent of the *pathstack* [5] used in XML path matching, though it is structured, constructed, and used differently.

```
Algorithm: Stratified-Shortest-Path
Input:
  PQ-Array; $v_s$ /* $v_s$ is the vertex where the shortest path starts */;
Output:
  the shortest-path $p$ from $v_s$ to end vertex d
Procedure:
  $V_1 = \{v_s\}$ ;
  $minWeight(v_s) = weight(v_s)$
  $vPtr(v_s) = v_s$;
  $hPtr(v_s) = s$ /*$s$ is the dummy start vertex*/
  for each stratum $V_j$ (j= 2,3,...,$|V|$) do
      for each vertex $v_{i,j}$ in stratum $V_j$ do
          $v_p = hPtr(v_{i,j})$;
          $w_p = weight(v_{i,j}) + minWeight(v_p)$; /*smallest sum path
          with $v_{i,j}$ */
          $w_c = minWeight(v_{i-1,j})$; /*smallest sum path without
          $v_{i,j}$*/
          if $w_p \leq w_c$ then
              $minWeight(v_{i,j}) = w_p$;
              $vPtr(v_{i,j}) = v_{i,j}$;
          else
              $minWeight(v_{i,j}) = w_c$;
              $v_c = vPtr(v_{i-1,j})$;
              $vPtr(v_{i,j}) = v_c$;
          end if
      end for
  end for
  follow $hPtr$ and $vPtr$ from the end vertex $d$ to start vertex $s$ and
  best path $p$ consists of vertices from each stratum pointed by $vPtr$ ;
  return p;
```

Figure 3.4: Stratified shortest path algorithm (*SSP*)

uses dynamic programming on the *PQ-Array* data structure. The algorithm works by finding the shortest sub-path from the given source vertex v_s to the current stratum at each iteration and increasing the stratum in the following iteration. Once the algorithm completes, we can enumerate the shortest path by moving backwards among the strata from d: this is achieved by following *hPtr* from one stratum to the previous one and then following the *vPtr* within the current stratum (to locate the vertex on the shortest path in the current stratum). Note that the use of two pointers to track backwards (as opposed to only one) leverages the properties of the stratified stream graphs to enable the algorithm to compute in $O(|v|)$ time, and, as we will see later, enables efficient dynamic updates on the graph.

Correctness proof of *SSP*: We first present two theorems and based on the two theorems, we show the correctness of *SSP*.

Theorem 3. *By following SSP, we ensure that the $minWeight$ of each vertex $v_{i,j}$ in* *stratum V_j stores the smallest weight sum reachable from v_s (including v_s) to any vertex $v_{i',j}$ in V_j, where $i' \leq i$.*

Proof: To prove the correctness of Theorem 3, we use mathematical induction. Since there is only one vertex v_s in the first stratum V_1, the weight of v_s is the smallest weight sum from v_s to any vertex with a smaller or equal index to v_s in stratum V_1. Hence, the theorem is correct for case $j = 1$. Assume for case $j = k$, Theorem 3 holds. Then when $j = k + 1$, for each vertex $V_{i,j}$ from bottom up in stratum V_j, we consider two values w_p and w_c. Since by assigning w_p as $weight(v_{i,j}) + minWeight(v_p)$, we actually make w_p store the total weight of the shortest sub-path from v_s to $v_{i,j}$. In addition, since the algorithm enforces to process vertex in stratum V_j from the bottom up, when dealing with $v_{i,j}$, we already set the $minWeight$ values and corresponding pointer in $v_{i-1,j}$. By assigning w_c with $minWeight$ of $v_{i-1,j}$, we ensure that w_c stores the total weight of shortest sub-path from v_s to any $v_{i',j}$ where $i' < i$. Then when we set the $minWeight$ with the smallest of w_p and w_c, we guarantee that $minWeight$ stores the smallest weight sum reachable from s to any vertex $v_{i',j}$, $i' \leq i$. Hence for $j = k + 1$, Theorem 3 also holds. Therefore, we prove that Theorem 3 is correct.

Lemma 1. *By following SSP, we also ensure that $vPtr$ of each vertex $v_{i,j}$ correctly points to the vertex as its definition.*

Proof: First, by definition we know that the $vPtr$ of a vertex $v_{i,j}$ only points to either itself or a vertex below it in the same stratum V_j. In the procedure of *SSP*, we ensure that $vPtr$ always points to the smaller one. Or in the case that two vertices are equal in terms of $minWeight$, we assign $vPtr$ to the vertex $v_{i,j}$ itself, which is later event than all events below $v_{i,j}$. Therefore, we prove the correctness of Lemma 1.

Theorem 4. *In SSP, the found path by following hPtr and vPtr backwards from d until it reaches s is indeed the shortest path from v_s to d.*

Proof: First, from Lemma 1 and the initial stratified graph construction process, it can be seen that *vPtr* and *hPtr* indeed store the value as defined. Then by definition, we know *vPtr* points to the vertex that is *before or identical to* $v_{i,j}$ in stratum V_j and has the smallest *minWeight*. For any given vertex $v_{i,j}$, by following its *vPtr*, we can find the vertex in stratum V_j that makes up the shortest path until $v_{i,j}$. Also, we know *hPtr* points to the latest vertex v in the previous stratum that has an edge to $v_{i,j}$. By following *vPtr* of v, again we can find the vertex in stratum V_{j-1} that the shortest path goes through. Therefore, when we start to follow *hPtr* and *vPtr* from end vertex d to v_s, it guarantees the vertex pointed by each *vPtr* form the shortest path from v_s to d

Relying on Theorem 3 and 4, we can see that the shortest path can indeed be found by following $hPtr$ and $vPtr$ backwards from end vertex d until v_s based on *SSP*.

Complexity of SSP: The SSP algorithm works in $O(|v|)$ time, where $|v|$ represents number of vertices, because each vertex is considered only once and for each vertex only one incoming edge is investigated

K Shortest Paths on a Stratified Graph

Once the shortest path in the stratified stream graph is found, we locate the k shortest paths incrementally (Figure 3.5) : to find the i^{th} shortest path, we use the previously discovered $(i-1)$ shortest paths stored in $\mathcal{L}$. We create a min-heap to store potential candidates for the i^{th} shortest path. For each vertex v of the $(i-1)$th shortest path (denoted as p_{last}), we check if there is a path $p \in \mathcal{L}$ sharing the sub-path from the start event, s, to current vertex, v. If so, we set the value of the vertex following v on p to ∞. After this resetting of the vertex value, we re-apply the shortest path algorithm described above to find the shortest path p_{sub} from v in this revised stratified graph, combine p_{sub} with the sub path from s to v in p_{last}, and store it in the candidate heap. After this, we pick the path with the smallest value from the candidate heap and mark it as the i^{th} shortest path. Then in Figure 3.6, we continue this process until all required k shortest paths are found; the algorithm completes in $O((|s|k)^2 + |v|sk)$, where $|v|$ is the number of vertices (i.e., events in the current window), $|s|$ is the number of strata (i.e., the sequence pattern length), and k is the number of shortest paths required. Note that the complexity of the algorithm is significantly lower than that of conventional KSP algorithms (see

```
Algorithm: Next-Stratified-Shortest-Path
Input:
  PQ-Array;
Output:
  next shortest path in current PQ-Array
Procedure:
  $\mathcal{L}$ = PQ-Array.getCurSPList(); /*get the list of previous found shortest
  paths stored in PQ-Array*/
  if $\mathcal{L}.size() = 0$ then
    $p$ = Stratified-Shortest-Path(PQ-Array, s) ; /*shortest path from s*/
    $\mathcal{L}$.add($p$);
    return $p$
  end if
  get the last added path $p_{last}$ from the $\mathcal{L}$
  for the $j^{th}$ vertex $v_j$ in $p_{last}$, $j = 0, 1, 2, \ldots, |p_{last}|$ /*$v_0 = s$*/ do
    restore the original PQ-Array;
    weight($v_{j+1}$) = $\infty$;
    for each path $p_i$, $i = 1, 2, ..., |\mathcal{L}| - 1$ in $l$ do
      if $p_i$ shares the same sub-path $p_{sub,1}$ from start vertex s to
      $v_j$ with $p_{last}$ then
        set the weight of next vertex of $v_j$ in $p_i$ as $\infty$;
      end if
    end for
    $p_{sub,2}$ = Stratified-Shortest-Path(PQ-Array', $v_j$); /* shortest path
    from $v_j$ to the end */
    combine $p_{sub,1}$ and $p_{sub,2}$ to obtain $p'$
    insert the combined path $p'$ into candidate heap $\mathcal{H}_{cand}$;
  end for
  remove top $p_{top}$ from $\mathcal{H}_{cand}$ and insert it to $\mathcal{L}$
  PQ-Array.setCurSPList($\mathcal{L}$);
  return $p_{top}$
```

Figure 3.5: Next stratified shortest path algorithm (NSSP)

```
Algorithm: $k$-Stratified-Shortest-Path
Input:
  PQ-Array; $k$;
Output:
  $k$ shortest paths
Procedure:
  $\mathcal{L} = \phi$;
  $i = 1$;
  while $i \leq k$ do
    p = Next-Stratified-Shortest-Path(PQ-Array);
    $\mathcal{L}.add(p)$;
  end while
  return $\mathcal{L}$;
```

Figure 3.6: k stratified shortest path algorithm (k-SSP)

Chapter 2) for small k and s.

Correctness proof of k-*SSP*: First, we show that algorithm *NSSP* is correct. To show the correctness of *NSSP*, we first present the following proposition.

Proposition 1. *In a given stratified stream graph, the i^{th} shortest path p_i must coincide with the $(i-1)^{th}$ shortest path p_{i-1} until some j^{th} vertex, $j = 1, 2, ...|s|$, where $|s|$ is the* *number of strata.*

Proof of Proposition 1: Since we synthetically create the start vertex s, any two paths at least will share the same vertex s, hence the proof.

Then, based on the Proposition, suppose we have previously found the first $i-1$ shortest paths and we are looking for the i^{th} shortest path p_i by *NSSP*. When $i=1$, i.e. we are looking for the first shortest path, we can simply call our previous *SSP*, which returns the shortest paths in the current PQ-Array. When $i>1$, we assume p_i and p_{i-1} coincide until the j^{th} vertex. The algorithm sets the weight of the next vertex of all top $(i-1)$ paths that share first j^{th} vertex with p_m as ∞. This ensures that no new discovered shortest path will be the same as the previous $(i-1)$ shortest paths from the $(j+1)^{th}$ vertex. Note that If the next shortest path is a deviation of previous found shortest paths, we would have found that in previous searching and stored it in the candidate heap. The algorithm considers all vertices in p_{i-1} (except the last dummy end vertex) and, for each case, it identifies a candidate shortest path (maintaining at most k candidates). The Proposition 1 ensures that the i^{th} shortest path will be among the candidates that have been enumerated.

Finally, to search for k shortest paths, we simply call the *NSSP* k times and return the results.

Complexity of k-SSP: When we compute the i^{th} shortest path p_i, we need to investigate the graph for each vertex v in the $(i-1)^{th}$ path p_{i-1} and find the shortest path from v in it. Since this includes setting values for each of the previous shortest paths in their first i vertices coinciding with p_{i-1} and performing a shortest path search, the worst case time complexity of this step is $O(|s|i+|v|)$, which is $O(|s|k+|v|)$. Since each vertex in each of the $|s|$ strata of the $(i-1)^{th}$ shortest path needs to be considered, this step takes $O((|s|k+|v|)|s|)$ time. Then, computing k results takes in the worst case $O((|s|k+|v|)sk)$ or $O((|s|k)^2+|v||s|k)$ time.

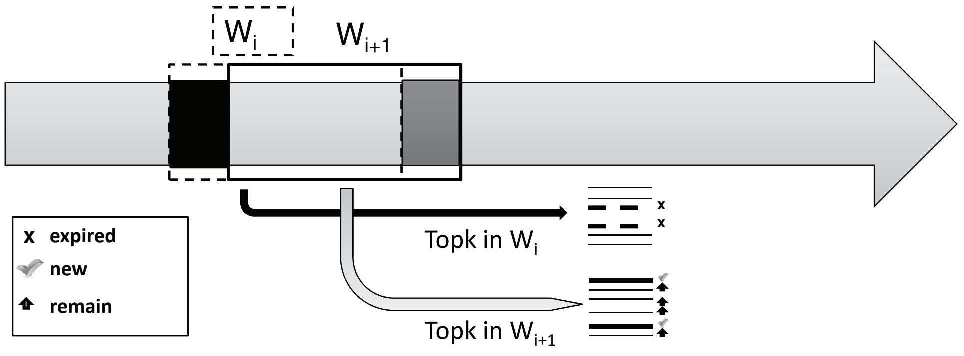


Figure 3.7: Sharing of top k results between moving windows

K Shortest Paths on Dynamic Graphs

The algorithm described above leverages the stratified nature of the stream graphs to significantly reduce the cost of shortest path enumeration. However, when used naively, it would need to recompute paths from scratch each time the window shifts over the data stream. In Figure 3.7, we illustrate the scenario when the time window shifts from W_i to W_{i+1}. Among all top-k matches corresponding to W_i, the dashed lines (the ones with cross marks next to it) represent expired matches. And among all top-k matches in W_{i+1}, the bold black lines (the ones with check marks next to it) represent new added matches. Potentially many matches are shared between W_i and W_{i+1}. Therefore, it is desirable to devise dynamic algorithms to reuse previous matches.

In Figure 3.8, we show the dynamic algorithm for keeping the top-k pattern matches up-to-date as the *PQ-Array* data structure changes over time. Suppose we already have k results for the current window. When the window shifts, some of the events, $\mathcal{E}_{old}$ will expire and some new events new events $\mathcal{E}_{new}$ will need to be considered.

We first consider the expired events. The first step of the algorithm removes out-dated events from the PQ-Array. In the second step, any path that contains any event in $\mathcal{E}_{old}$ is dropped from the set, $\mathcal{L}$, of current top-k paths. Before we consider newly arriving events, we copy the remaining events in the PQ-Array and denote the copied PQ-Array as PQ-Array$_{exist}$. Then we move remaining paths from $\mathcal{L}$ to $\mathcal{L}_{exist}$. We next consider the new

```
Algorithm:
$k$-Dynamic-Stratified-Shortest-Path
Input:
  PQ-Array, $k$, $\mathcal{L}$, $\mathcal{E}_{old}$, $\mathcal{E}_{new}$ /* $\mathcal{L}$ is the list of existing $k$ shortest paths,
  $\mathcal{E}_{old}$ denotes the set of expired events, and $\mathcal{E}_{new}$ denotes the set of
  new events */
Output:
  updated $k$ shortest paths $\mathcal{L}$
Procedure:
  1. update PQ-Array by removing out-dated events;
  2. remove paths with out-dated events from existing $k$ shortest paths
  list $\mathcal{L}$;
  3. move remaining shortest paths from $\mathcal{L}$ into a list $\mathcal{L}_{exist}$
  4. copy PQ-Array as PQ-Array$_{exist}$; /*used to produce further short-
  est paths */
  5. set weights of each non-expired vertex $v$ in the last stratum of
  PQ-Array as $\infty$;
  6.
  for each vertex $v_{i,j}$ corresponding to a new event ordered by time
  do
      insert $v_{i,j}$ into stratum $V_j$;
      $hPtr(v_{i,j}) = v_{l,j-1}$ /*$v_{l,j-1}$ is the latest vertex in stratum
      $V_{j-1}$*/
      minWeight($v_{i,j}$) $= \infty$; $vPtr(v_{i,j}) = \perp$;
  end for
  7. pull and put into $\mathcal{L}$ the shortest k paths from PQ-Array$_{exist}$ and
  PQ-Array by calling Next-Stratified-Shortest-Path() correspondingly:
  when pulling from PQ-Array$_{exist}$, we consider paths from $\mathcal{L}_{exist}$
  first.
  8. restore weights of vertices previously set as $\infty$ in the last stratum
  return $\mathcal{L}$
```

Figure 3.8: k dynamic stratified shortest paths algorithm (k-DSSP)

events in the window. First we update the PQ-Array with the new events. We then identify new candidate shortest paths by constraining the new paths to pass through at least one of the new events in the window (by setting remaining events in the last stratum as ∞).

The final results are obtained by pulling the shortest paths from PQ-Array$_{exist}$ (storing all old paths, i.e. the path that exists before window moves and is still valid now) and PQ-Array (storing all new paths, i.e. the path with at least one new event). From PQ-Array, we call *NSSP* each time to obtain the next shortest path (among all new paths). From PQ-Array$_{exist}$ we first use the remaining shortest paths from $\mathcal{L}_{exist}$ and if all paths in $\mathcal{L}_{exist}$ are part of final top-k shortest paths, to further obtain next path, we call NSSP on PQ-Array$_{exist}$. Each time we put the shorter path pulled from PQ-Array and PQ-Array$_{exist}$ into $\mathcal{L}$ and continue this process until k paths have been found.

The worst case complexity of the algorithm is $O((|s|k)^2 + |v||s|k)$ as before; however, as the experiments in Chapter 4 show, the worst case rarely happens.

Correctness proof of k-*DSSP*: First, in Steps 1 and 2 of k-*DSSP*, we simply remove vertices corresponding to outdated events from the PQ-Array and out-dated paths from the current k-shortest paths list $\mathcal{L}$. In Steps 3 and 4, we copy existing PQ-Array into a PQ-Array$_{exist}$ and move matches to $\mathcal{L}_{exist}$. So far, only removal and copy operations are performed. In Step 5, the algorithm temporarily sets the weights of all remaining vertices in the last stratum $V_{|s|}$ of PQ-Array to ∞. We observe that any remaining vertex v in $V_{|s|}$ cannot be part of a new path, since all new vertices are later than v in terms of their corresponding timestamps. Other remaining vertices, on the other hand, can be part of new paths and, thus, we need to further consider them. Step 5 ensures that each of the new discovered paths will go through at least one new event in the last stratum. Step 6 revises the PQ-Array. Since new coming events are ordered by time, we can insert event one by one into PQ-Array in their time order. Each time we insert one event $v_{i,j}$ (i.e. the i^{th} vertex in stratum V_j), we set its *hPtr* as the current last vertex $v_{l,j-1}$ in its previous stratum V_{j-1}, since events inserted later into V_{j-1} will have no edge to $v_{i,j}$. Once the k-SSP algorithm is executed on the revised PQ-Array data structure, due to the careful placement of ∞ weights, the newly discovered shortest paths are guaranteed to go through at least one new event, ensuring that no paths are redundantly discovered. In Step 8, we each time choose one shorter path from shortest paths obtained from PQ-Array and PQ-Array$_{exist}$. Since we have previously demonstrated the correctness of NSSP, we ensure that by calling NSSP on PQ-Array, each time we obtain the next shortest new path. Also, in PQ-Array$_{exist}$, we only have removal operation, the remaining paths in $\mathcal{L}_{exist}$ would still be the shortest paths in PQ-Array$_{exist}$. When further paths are required, by calling NSSP we can produce more shortest path incrementally. Finally in Step 8, we restore the weight of each vertex whose weight was temporarily set as ∞ in Step 5. This is because some of the previous paths containing such a vertex but not in the earlier top-k shortest paths list are now possible to be in top-k after the update of PQ-Array. By restoring the weights in Step 8, we ensure that those paths can still be considered.

Complexity of k-DSSP: Since the revision of the PQ-Array data structure takes only $O(|v|)$ time, the worst case complexity of k-DSSP is similar to that of k-SSP.

```
Algorithm:
Dynamic-top-$k$-nonlinear-pref-sequence-
maintenance
Input:
  $q_i$, $k$, $\mathcal{L}$, $\mathcal{E}_{old}$, $\mathcal{E}_{new}$ /* $q_i$ is the event queue corresponding to the $i^{th}$
  event in the sequence query, $\mathcal{L}$ is the list of existing top-$k$ sequence
  matches, $\mathcal{E}_{old}$ denotes the set of expired events, and $\mathcal{E}_{new}$ denotes
  the set of new events */;
Output:
  top-$k$ sequence matches in $\mathcal{L}$
Procedure:
  1. remove out-dated events from each $q_i$;
  2. insert each event $e$ from $\mathcal{E}_{new}$ into corresponding queue $q_i$; /*i.e.
  event $e$ has the same event class as the one $q_i$ corresponds to in
  the sequence query*/
  3. sort each $q_i$;
  4. build two index for each queue $q_i$, one for existing events and the
  other for new inserted events.
  5. remove out-dated sequence matches from $\mathcal{L}$;
  6. perform rank-join on each queue with any arbitrary scheduling
  strategy and update $\mathcal{L}$ until top-$k$ matches found.
  During the join process:
  if the current visiting event has been visited before then
      join it with only the new inserted events in other queues;
  else
      join it with all events in other queues;
  end if
  return $\mathcal{L}$;
```

Figure 3.9: Dynamic top-k sequence match maintenance for nonlinear preference sequence queries

Nonlinear Preference Sequence Ranking

For sequence queries with nonlinear preference functions, we process them with our top-k merge module directly. Since the preference function here works as the merge function during the rank join, as commonly assumed [10, 11], we require that the preference function is monotonic on the score of each joined event class. We consider event instances of each event class in the sequence query to form a queue and pull from each queue to obtain the sequence matches that have the top score value according to the given preference function. This found sequence matches can be used for further complex query matching the same way as previous introduced stratified-graph based approach. To efficiently maintain top-k sequence matches at runtime, we use the dynamic procedure shown in Figure 3.9. In the first four steps, for each queue, we update them by first removing out-dated events, and then inserting new events. After sorting each queue, we build two indices for the existing events and new inserted events respectively. In Step

5, we remove obsolete matches from existing top-k sequence match list $\mathcal{L}$. To complete the top-k matches, we do ranked-join in Step 6 on each updated queue by adopting any existing join algorithm (e.g. TA [22]). When performing the join, if the current accessed event instance has been visited previously, we only consider to join it with new event instances in other queues, otherwise, we join it as normal with both new inserted events and existing events. In this manner, we ensure that no previous computation is repeated.

Correctness proof of algorithm in Figure 3.9: In Steps 1 and 2, we remove expired events and insert new events. This ensures that events stored currently are only valid and all valid events are included. In Steps 3 and 4, we preprocess each queue for later rank join. In Step 5, we remove out-dated events, which still leaves the valid matches in the same order. Step 6 performs the join. In the process, since all event visited in previous window has been joined with all events from previous window, we don't need to join them again in the current window. Also, as having been approved in previous join algorithms [10, 11], the threshold based algorithms can correctly produce the top-k matches. Therefore, the proposed procedure in Figure 3.9 is indeed correct.

Complexity: Since operations except the join process require constant time, the worse case complexity only depends on the joining algorithm selected.

3.3 Top-k Merge Module (TMM)

When a complex query pattern includes combinations of multiple sequence patterns or when we need to deal with nonlinear preference functions, the top-k merge module (TMM) is used for producing and ranking combined results. Since disjunction can be handled by picking the highest scoring entries from the input queues, in this section, we focus on conjunctive patterns which require top-k joins.

Thresholds, Boundaries, and Scheduling

As discussed in Chapter 2, there are many top-k join algorithms, differing in the way the sorted and random accesses are scheduled and how the stopping condition is sought. The TA algorithm [22], for example, maintains the lower bound, lb, of the candidate results

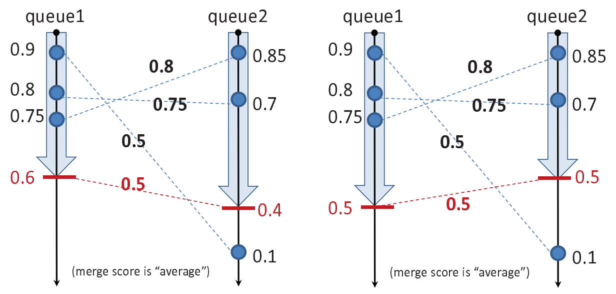


Figure 3.10: Two alternative termination *boundaries* that provide the same threshold (0.5) needed to commit top-3 candidates

discovered so far and also computes an upper bound, ub, on the best value of the undiscovered objects; i.e., a *threshold* that the value of lb has to pass to declare the candidate objects as the top ones. When $lb \geq ub$, TA stops and returns the results found so far. NRA algorithm [11] also works similarly, but unlike TA which aggressively schedules random accesses to keep the lower bound as high as possible, the NRA algorithm does not schedule any random accesses. Consequently, the lb and ub cross later than in TA. In other words, NRA trades-off random accesses for sorted accesses as a scheduling strategy.

Even when TA or NRA is selected a priori, there is still room for improving the number of accesses to the input queues. The difficulty in both cases is that there can be different ways to reach the termination *boundary* (where the lower bound of seen and upper bound of unseen objects cross – Figure 3.10). Thus, the goal of any scheduling strategy would be to access the queues in an order in which the stopping condition is reached as early as possible. This requires the ability, at any given point in time, to *predict* the distance –in terms of execution time– to the termination condition for each alternative access option. [11] relies on a round-robin based strategy that does not give preference to any of the input queues. [12] attempts to reach the termination condition by always accessing to the data queue that has the steepest current core gradient. This, however, fails to account for the differences in the sorted and random access costs for different input queues. [25] addresses situations where different queues have different costs by collecting statistics and picking an optimized schedule in the runtime. The algorithm, however, needs

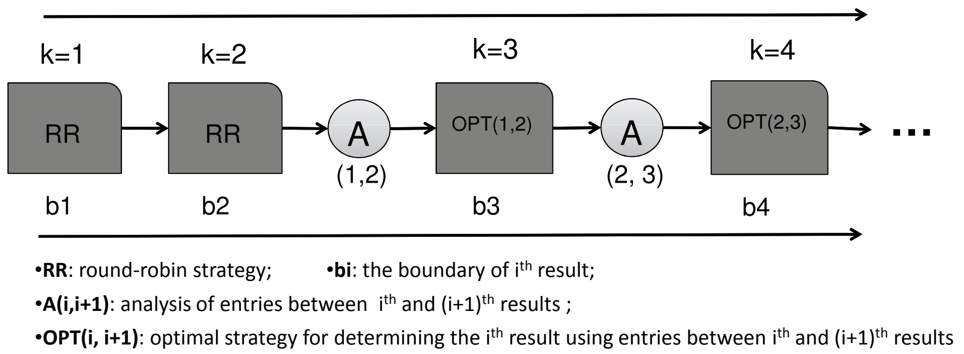

k=1
k=2
k=3
k=4
RR
RR
A
(1,2)
OPT(1,2)
A
(2, 3)
OPT(2,3)
…
b1
b2
b3
b4
•RR: round-robin strategy;
•bi: the boundary of ith result;
•A(i,i+1): analysis of entries between ith and (i+1)th results ;
•OPT(i, i+1): optimal strategy for determining the ith result using entries between ith and (i+1)th results

Figure 3.11: Execution process of waste-avoiding boundary selection (WABS)

statistics that can properly represent the whole data set (obtained through sampling of the whole data set), and as we will see in the experiments, when joining set of sequences that have not even been enumerated, effective samples are not always available.

Waste-Avoiding Boundary Selection

When searching for the k^{th} best result after the $(k-1)^{th}$ is located, the ideal strategy would be the one that requires the least amount of time to locate the next termination boundary. We refer to this as the *waste-avoiding* strategy and the corresponding boundary as the *waste-avoiding boundary*. Prior work tries to predict the score gradient, data/score distribution, join selectivity, and/or access costs to seek the waste-avoiding boundary. These are neither trivial tasks, nor (as the experiments in Chapter 4.4 show) work very well for combining dynamically generated sequences. In this work, we propose to tackle this difficulty by a *continuously adaptive boundary targeting* strategy which relies on the following simple assumption: "*the best scheduling strategy to locate the termination boundary for the $(i+1)^{th}$ result will have changed less drastically since the $(i-1)^{th}$ result than since the $(i-j)^{th}$ result, for $j > 1$*". This implies that we can reduce the waste in the search for the $(i+1)^{th}$ boundary if we could find the best strategy for the $(i-1)^{th}$ boundary *in the light of the most recent inputs* (possibly revising any boundary found earlier) and use this to guide the search.

The overall execution process of *waste-avoiding boundary selection* is shown in

```
Algorithm:
Waste-Avoiding-Boundary-Selection
Input:
  data queues, dq; $k$;
Output:
  Top-$k$ results $\mathcal{R}$;
Procedure:
  apply existing approach (e.g. round-robin) to find the top-2 results.
  put top-2 results into $\mathcal{R}$
  if $k > 2$ then
    while $i <= k - 1$ do
      for each alternative termination boundary, $\beta^*_{i-1}$ among the
      newly discovered data entries between the discovery of $(i-1)^{th}$ and the $i^{th}$ results do
        /*$lb_{i-1}$ is the score of $\beta_{i-1}$*/
        if score($\beta^*_{i-1}$) $\leq lb_{i-1}$ && cost($\beta^*_{i-1}$) $<$ minCost then
          minCost = cost($\beta^*_{i-1}$); /*initially, minCost = $\infty$*/
          for $h = 1, 2, \ldots, |q|$ do
            $a_h$ = access-ratio($\beta^*_{i-1}$,$h$); /*access-ratio($\beta^*_{i-1}$,$h$)
            is the overall ratio of accesses to queue $q_h$ accord-
            ing to $\beta^*_{i-1}$*/
          end for
        end if
      end for
      while TRUE do
        $\forall\ h = 1, 2, \ldots, |q|$, schedule $a_h$ accesses to $q_h$
        /*let $u$ be the number of results found*/
        put the new $u$ results into $\mathcal{R}$;
        i = i+$u$;
        if $u > 0$ then
          break;
        end if
      end while
    end while
  end if
  return $\mathcal{R}$
```

Figure 3.12: Waste-avoiding boundary selection strategy (WABS)

Figure 3.11. As can be seen in this figure, when searching for the best boundary for the first two results, we adopt basic round-robin access scheduling strategy due to the lack of priori knowledge. Then for the $(i+1)^{th}$ result, where $2 \leq i \leq k-1$, we analyze the objects discovered between the recent two results (i.e $(i-1)^{th}$ and i^{th} results) and locate the best accessing strategy for the $(i-1)^{th}$ result and use this as the accessing strategy for the searching of the $(i+1)^{th}$ result. This process continues until we successfully determine the top k matches.

We provide the pseudo-code for the *waste-avoiding boundary selection (WABS)* strategy in Figure 3.12. In a given window, the strategy proceeds stepwise seeking the stopping conditions for 1^{st}, 2^{nd}, ..., and k^{th} results incrementally. The strategy is revised between each consecutive pair of results. Let σ_i denote the access strategy (i.e., access ratios for each queue) used for identifying the boundary β_i, for the i^{th} result, with combined score, lb_i. The boundaries for the first two results are sought using an existing technique, such as round-robin or gradient-based approaches. Let us assume that we already have reached the i^{th} $(i \geq 2)$ boundary and we are seeking the boundary for the $(i+1)^{th}$ result (Figure 3.13(a)). In order to revise the access strategy for this boundary, we search for alternative ways to achieve the boundary β_{i-1} (the boundary for the $(i-1)^{th}$ result) and also take into account the data entries accessed between the enumeration of $(i-1)^{th}$ and i^{th} results.

We achieve this by considering newly discovered data entries (in increasing order of score) and see if they would lead to a boundary combination with score $\leq lb_{i-1}$ with all the data entries discovered so far. For each alternative combination whose overall score is $\leq lb_{i-1}$, we assess (based on the actual access times that have been observed) how long it would have taken the algorithm to reach to that particular combination of entries and locate the *cheapest* boundary among all alternatives (Figure 3.13(b)). Once located, we use the relative ratios of entries in each queue for this boundary combination as the revised access strategy, σ'_{i-1}.

Since it represents the most locally-informed and least wasteful boundary decision available, WABS then uses strategy σ'_{i-1} to seek the threshold for the next result (i.e., as the strategy σ_{i+1}). Let us assume that there are m queues and between the $(i-1)^{th}$ and i^{th} results, Δ_j data entries are accessed on the j^{th} queue. The cost of strategy adaptation depends on the sizes of Δ_j and how far one has to go back to discover a combination with threshold τ_{i-1}. As we will see in the experiments Chapter, the cost of this process is smaller than other techniques and, especially within the context of complex event processing, negligible with respect to the gains in access time.

3.4 Dynamic Maintenance of Top-k Complex Query Matches

Since event streams evolve in the real-time, it is desirable to reuse previous results and dynamically maintain results instead of computing the top-k results from scratch each time. For top-k sequence pattern match maintenance, we have described the algorithms in

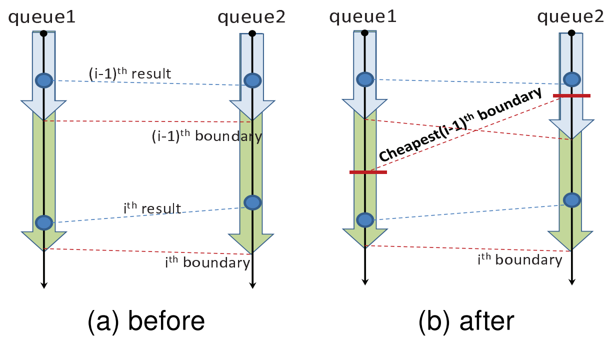


Figure 3.13: Revision of the $(i-1)^{th}$ boundary

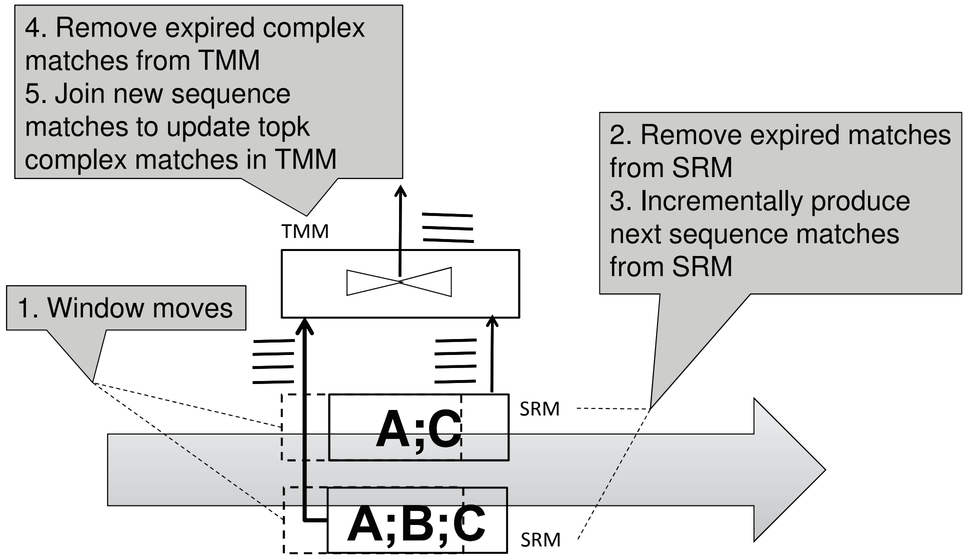

4. Remove expired complex matches from TMM
5. Join new sequence matches to update topk complex matches in TMM
2. Remove expired matches from SRM
3. Incrementally produce next sequence matches from SRM
TMM
1. Window moves
A;C
SRM
A;B;C
SRM

Figure 3.14: Overall Process for Dynamic Complex Query Maintenance

previous Chapter 3.2. In this section, we focus on top-k complex pattern query matches maintenance. The whole dynamic process can be separated into two phases: 1. dynamic maintenance of previous found sequence matches from each sequence pattern query component; 2. dynamic maintenance of top-k complex pattern query matching results. By the first phase, we ensure that when time windows move forward, we do not repeat enumerating the same sequence match (which has been computed and buffered in TMM previously). By the second phase, we make sure that previously joined complex matches are not processed again, which reduces the re-computation costs of join process. As can be seen in the experimental evaluation in Chapter 4, when windows shift in relatively small steps, the gain of dynamic maintenance is quite significant, as many existing matches

```
Algorithm:
Dynamic-PQ-Array-Update
Input:
  PQ-Array, $\mathcal{E}_{old}$, $\mathcal{E}_{new}$ /* $\mathcal{E}_{old}$ denotes the set of expired events, and
  $\mathcal{E}_{new}$ denotes the set of new events */
Output:
  PQ-Array$^{exist}$ and PQ-Array$^{new}$
Procedure:
  1. update PQ-Array by removing out-dated events and denote it as
  PQ-Array$^{exist}$;
  2. remove paths with out-dated event from existing $k$ shortest paths
  list $\mathcal{L}$;
  3. create a new PQ-Array$^{new}$ and copy all events except those in
  the last stratum from PQ-Array$^{exist}$.
  4.
  for each vertex $v_{i,j}$ corresponding to a new event ordered by time
  do
      insert $v_{i,j}$ into stratum $V_j$;
      $hPtr(v_{i,j}) = v_{l,j-1}$ /*$v_{l,j-1}$ is the latest vertex in stratum
      $V_{j-1}$*/
      minWeight$(v_{i,j}) = \infty$;
      $vPtr(v_{i,j}) = \perp$;
  end for
  5. return PQ-Array$^{exist}$ and PQ-Array$^{new}$
```

Figure 3.15: Dynamic PQ-Array update for linear preference sequence queries (dUpdate-LP)

remain in the top-k lists.

In Figure 3.14, we illustrate the dynamic maintenance process for top-k complex matches. After windows move, we remove expired sequence matches from each SRM module. Then by using the dynamic mechanism in SRM, we only produce top sequence matches that are not previously found. Finally, in TMM, we first remove expired matches and then each new found sequence match is joined with previous discovered matches to update the existing top-k results.

Dynamic Linear Preference Sequence Pattern Match Production

Potentially, a certain number of top-k sequence pattern matches discovered in the previous window will remain valid in the current window. We don't want to re-pay this costs by enumerating them again. Therefore, we design the following mechanism in Figures 3.15 and 3.16 to avoid the enumeration of previous found matches, while still enable the sorted access for the TMM. Each time the time window moves, we first call the procedure of *the dynamically update for linear preference sequence pattern query* (*dUpdate-LP*) in Figure 3.15 once to update the current PQ-Array. Afterwards, each time

```
Algorithm:
Incremental-Next-Stratified-Shortest-Path
Input:
  PQ-Array$^{exist}$, PQ-Array$^{new}$
Output:
  next shortest path $p_{next}$
Procedure:
  $\mathcal{L}_1$ = PQ-Array$^{exist}$.getCurSPList(); /*return current shortest paths*/
  $\mathcal{L}_2$ = PQ-Array$^{new}$.getCurSPList();
  if $\mathcal{L}_1$.numNewPaths() = 0 then
      $p_1$ =Next-Stratified-Shortest-path(PQ-Array$^{exist}$);
  end if
  if $\mathcal{L}_2$.numPaths() = 0 then
      $p_2$ =Next-Stratified-Shortest-path(PQ-Array$^{new}$);
  end if
  $p_1 = \mathcal{L}_1.getLast()$; /*get the last inserted shortest path from $\mathcal{L}_1$*/
  $p_2 = \mathcal{L}_2.getLast()$; /*get the last inserted shortest path from $\mathcal{L}_2$*/
  if weight($p_1$) $<$ weight($p_2$) then
      $p_{next}$ = $p_1$;
      Next-Stratified-Shortest-path(PQ-Array$^{exist}$);
  else
      $p_{next}$ = $p_2$;
      Next-Stratified-Shortest-path(PQ-Array$^{new}$);
  end if
  return $p_{next}$;
```

Figure 3.16: Incremental next stratified shortest path(iNSSP)

the sorted access is requested, we call the *incremental next stratified shortest path algorithm* (*iNSSP*) in Figure 3.16 to incrementally produce the next top sequence match.

To be specific, in Figure 3.15, when stream window moves forwards, we remove all out-dated events from the existing PQ-Array(denoted as PQ-Array^{exist}). Meanwhile, given all previous visited sequence matches stored in list $\mathcal{L}$, we update $\mathcal{L}$ by removing paths with out-dated events. In addition, we copy all events from PQ-Array^{exist} except the ones from last stratum into a new PQ-Array^{new}. Then according to the time order of new events, we insert each of them into PQ-Array^{new} and update its $hPtr$ to point to the last event in the previous stratum, minWeight as ∞, and $vPtr$ as $\perp$. After these updates, now we can provide sorted-access by choosing the smallest path from PQ-Array^{exist} and PQ-Array^{new} as described in Figure 3.16, where PQ-Array^{exist} stores all potential paths with only existing events, and PQ-Array^{new} stores all new paths (paths with at least one new event). Since in both of the PQ-Arrays, we incremental fetch the next shortest paths, this saves us the costs of enumerating previous matches again.

Correctness Proof of Algorithms *dUpdate-LP* and *iNSSP* in Figures 3.15 and 3.16: By the procedure shown in Figure 3.15, we maintain two PQ-Arrays: one holding all valid

```
Algorithm:
Dynamic-Nonlinear-Pref-TMM-Update
Input:
  event instance queues $q_i$, $\mathcal{E}_{old}$, $\mathcal{E}_{new}$ /* $\mathcal{E}_{old}$ denotes the set of
  expired events, and $\mathcal{E}_{new}$ denotes the set of new events */
Procedure:
  1. remove out-dated events from each $q_i$;
  2. insert each event $e$ from $\mathcal{E}_{new}$ into corresponding queue $q_i$; /*i.e.
  event $e$ has the same event class as the one $q_i$ corresponds to in
  the sequence query*/
  3. sort each $q_i$;
  4. build two index for each queue $q_i$, one for existing events and the
  other for new inserted events.
  5. remove out-dated sequence matches from $\mathcal{L}$;
```

Figure 3.17: Dynamic update for nonlinear preference sequence queries(dUpdate-NLP)

paths before new events inserted, and the other holding all new ones. The idea is similar to our previous proof for the correctness of k*-DSSP*. By not copying any event from the last stratum of PQ-Array^{exist}, we make sure each path in PQ-Array^{new} goes through at least one new inserted event in the last stratum. In addition, it can be seen that PQ-Array^{exsit} and PQ-Array^{new} together contain all potential paths in the current stream window. Then in algorithm *iNSSP* (Figure 3.16), we pick the shortest path from PQ-Array^{exsit} and PQ-Array^{new} by calling *Next-Stratified-Shortest-Path (NSSP)* on the corresponding PQ-Arrays. Since any potential path only comes from either PQ-Array^{exsit} or PQ-Array^{new}, by choosing the smallest one among the two PQ-Arrays, we guarantee that the found shortest path is indeed the next shortest path.

Complexity: It can be seen that the updating of event in PQ-Array is only related to the number of events stored, which means the time complexity would be $O(|v|)$, where $|v|$ is the number of events (i.e. vertex in the corresponding stratified graph) stored. In addition, we have shown previously that the *Next-Stratified-Shortest-Path* has a complexity of $O((sk + |v|)s)$ for a PQ-Array with v events, therefore, the complexity for algorithm in Figure 3.16 would be max{$O((sk + |v_1|)s), O((sk + |v_2|)s)$}.

Dynamic Nonlinear Preference Sequence Pattern Match Production

When dealing with incremental production of nonlinear preference sequence pattern match, we rely on the procedures of *dUpdate-NLP* shown in Figure 3.17 and

```
Algorithm:
Incremental-Next-Nonlinear-Pref-Sequence
Input:
  event instance queues $q_i$; list of existing sequence matches $\mathcal{L}$;
Output:
  next best nonlinear preference sequence query match
Procedure:
  1.  k = getNumTopMatches(); /* return the number of current top
  match results determined*/
  2. k = k+1; /*the next top match to look for*/
  3.
  while True do
      $\forall h = 1, 2, \ldots, |q|$,
      curValueEachQueue[h] = $q_h$.getCurrentAccessedValue();
      if computerMergeScore(curValueEachQueue) $\leq$ getCurTop-
      MatchScore(k) then
          return the $k^{th}$ matching sequence;
      else
          perform rank-join and update $\mathcal{L}$.
          During the join process:
          if the current visiting event has been visited before then
              join it with only the new inserted events in other queues;
          else
              join it with all events in other queues;
          end if
      end if
  end while
```

Figure 3.18: Incremental next nonlinear preference sequence (iNNLPS)

iNNLPS in Figure 3.18. In particular, each time the time window moves, we follow *dUpdate-NLP* to update each queue q_i and the existing matched sequence results. Then, for event instances in each queue, we build two indices, which are used later for incremental production of next top sequence match. To produce the next top sequence match, we call module in Figure 3.18, which first queries its current top match list (i.e. $getNumTopMatches()$) to determine what is the next top match to target for. Note that, each time after time window is updated, the $getNumTopMatches()$ method will return 0, since the existing top matches do not necessarily remain in the top list. After that, the current determined number of top matches will be recorded and returned by the $getNumTopMatches()$ method. If we have already determined the requested k^{th} match, we return it immediately. Otherwise, we perform rank-join on the each q_i to find the next top joined match and return it. Same as before, we can pick any arbitrary join scheduling algorithm to determine how sorted access and random access are scheduled in each round. If the current accessed event e has been used previously, since we maintain all valid previous results, we only need to consider the joint results of e and the newly inserted

```
Algorithm:
Dynamic-Top-$k$-Complex-Query-Match
Input:
  sequence matching queues $sq_i$, $\mathcal{E}_{old}$ /*$sq_i$ stores previous found se-
  quence matches for each component sequence query, $\mathcal{E}_{old}$ denotes
  the set of expired events*/
Output:
  Top-$k$ complex matching list $\mathcal{L}$
Procedure:
  1. remove out-dated sequence matches from each $sq_i$;
  2. remove out-dated complex matches from $\mathcal{L}_{visited}$;
  while top-$k$ not determined do
    3. schedule sorted access according to WABS on each sequence
    pattern query component by calling incremental sequence pro-
    duction algorithm (i.e. iNSSP and iNNLPS)
    4. perform rank-join on found sequence matches.
    5. put all matched results in $\mathcal{L}_{visited}$.
  end while
  copy the top-$k$ matches from $\mathcal{L}_{visited}$ to $\mathcal{L}$.
  return $\mathcal{L}$
```

Figure 3.19: Dynamic top-k complex pattern query match maintenance in TMM (dTMM)

events. For other events, we join it with all possible events from other queues as normal.

Correctness proof of algorithms iUpdte-NLP and iNNLPS in Figures 3.17 and 3.18: In dUpdate-NLP, we perform necessary preprocess each time the time window moves. This preprocess includes removing expired events and matches, as well as inserting new events. This process ensures that our new top-k query processing only considers valid events and all valid events are included. Clearly, this process won't affect the correctness of top-k results produced. Then in iNNLPS, we adopt a similar process as in the previous top-k nonlinear preference sequence query match maintenance in Figure 3.9. Again the correctness relies on the underlying join algorithms used. As long as we use a correct top-k join algorithm, the correctness of dUpdate-NLP is guaranteed.

Complexity: Similar to complexity analysis in Chapter 3.2, the complexity depends on the underlying top-k join algorithms.

Dynamic Top-k Complex Query Matching Results Maintenance

In Figure 3.19, to maintain top-k complex query matches as windows evolve, we first update the list of previously found sequence matches from each sequence queue. Then we delete all previous found complex query matches that are obsolete from the visited list $\mathcal{L}_{visited}$. After all these update process, we do as before to access each candidate queue with our proposed *WABS* strategy. The sorted access to each component sequence pattern is provided by our incremental production algorithm (i.e. iNSSP and iNNLPS for linear and nonlinear preference sequence pattern respectively). All newly found matches will also be added into $\mathcal{L}_{visited}$. Since by using our incremental production approach in assessing each sequence matching queue, we make sure that previous found sequence pattern matches would not be accessed again. In this way, we manage to reuse the previous results and dynamically maintain the top-k results.

Chapter 4

EXPERIMENT EVALUATION

In this Chapter, we evaluate the proposed top-k complex pattern ranking framework and the underlying algorithms. We describe three sets of experiments: first, we evaluate the stratified graph based k shortest algorithms underlying the sequence ranking module (SRM). Second, we assess the effectiveness of the waste-avoiding boundary selection strategy underlying the top-k merge module (TMM). Finally, we show the performance of the proposed mechanism for dynamic top-k maintenance on top of TMM.

4.1 Data Set Specification

For the experiments in this Chapter, we use the "Intel Berkeley Research lab[1]" data set containing $\sim$ 2.3 million sensor readings with the following schema:

date:yyyy-mm-dd	time:hh:mm:ss.xxx	epoch:int	moteid:int
temperature:real	humidity:real	light:real	voltage:real

Here, *epoch* is an integer corresponding to a time stamp obtained approximately every 30 seconds and is monotonically increasing. There are readings with the same *epoch*, but with different *moteid*; the *epoch* attribute uniquely determines the *epoch* and *time*. *Temperature* is in degrees Celsius. *Humidity* is temperature corrected relative humidity, ranging between 0-100. *Light* is in Lux and *Voltage* in volts. Also, we have the 2D location information corresponding to each *moteid*. In particular, we use each sensor's distance from the point, $(0, 0)$, as an additional attribute value.

In our evaluation scenarios, we focus on the following three attributes: *epoch*, *moteid*, and *temperature*. We also consider the location information associated to the sensor. We convert *epoch*s to unique timestamps by randomly perturbing the *epoch* values. For each temperature reading, we compute the temperature change from the

[1] Available at: http://db.csail.mit.edu/labdata/labdata.html

EA: [-160, -140)	EB: [-140,-120)	EC: [-120, -100)
ED: [-100, -80)	EE: [-80,-60)	EF: [-60, -40)
EG: [-40, -20)	EH: [-20, 0)	EI: [0]
EJ: (0, 20]	EK: (20,40]	EL: (40, 60]
EM: (60, 80]	EN: (80,100]	EO: (100, 120]
EP: (120, 140]	EQ: (140,160]	

Table 4.1: Value ranges represented by each event class

Parameters for the top-k sequence match experiments	
k	number of results to be returned
ref	relative event frequency
sl	sequence length
ws	window size
step	window shift length
Additional parameters for top-k joins on sensor data	
mf	merge function
rss	relative sensor selectivity
q	number of input queues
qp	query pattern

Table 4.2: Experiment parameters

previous reading on the same *moteid* and split the overall range (from -150 to 160) evenly into following subranges, each corresponding to a different event class as shown in 4.1:

4.2 Setup

Table 4.2 lists the key experiment parameters. While most parameters are self-explanatory, the relative event frequency (ref) and relative sensor selectivity (rss) require explanations: (a) ref is proportional to the frequency of the last event class in the sequence pattern (we experimented with three event class with frequencies $1 \times f_{base}$, $10 \times f_{base}$, and $89 \times f_{base}$, where f_{base} is $\sim 3 \times 10^{-3}$); (b) rss is proportional to the join likelihood of the sequences in the input queues to the top-k join module (rss = 1 corresponds to the case where sequences match can be joined if the sensors that generate them are within id range +/- 1; in rss=0.1 and rss=0.01, the join conditions are 10 times and 100 times more strict, respectively).

In addition, we introduce two synthetic event classes: *ESTART* and *EEND* to represent the start and end event.

Sequence patterns used in the experiments are listed in Table 4.3. For sequence

Sequence queries used for the top-k sequence match experiments	
S1: sequence with ref=1X	EG EH EN
S2: sequence with ref=10X	EG EH EO
S3: sequence with ref=89X	EG EH EJ
S4: sequence with length 2	EG EH
S5: sequence with length 3	EG EH EJ
S6: sequence with length 4	EG EH EJ EI
Sequence queries used for top-k joins on sensor data	
S7	EG EH EJ
S8	EK EL EM
S9	EI EN EO
S10	EG EH
S11	EJ EK EL EM

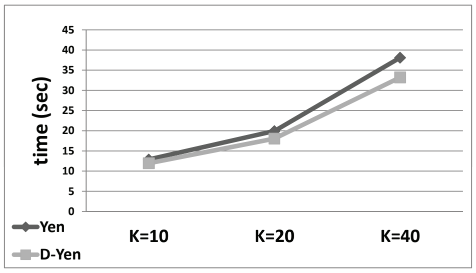
Table 4.3: Sequence patterns used in the experiments

matching experiments, we pick sequences with varying relative occurrences in the last event class (i.e. S1, S2 and S3). To evaluate the effects of sequence length, we choose three sequences (i.e. S4, S5, S6) with different lengths varying only the last event classes. For sequences used in top-k joins, we choose S7, S8, S9 shown in Table 4.3. For the experiments in which we evaluate the effects of divergent sequence lengths in the join, we used S10 (lenght=2) and S11 (length=4), also shown in Table 4.3

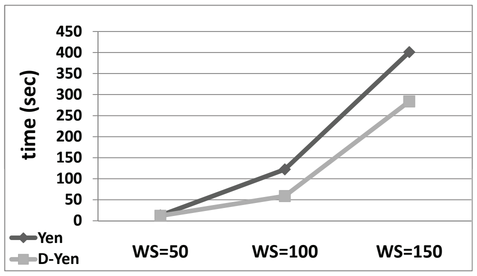
Experiments were run on a Windows XP box with Intel dual-core CPU @2.33GHz and 1.95GB RAM.

4.3 Evaluation of Top-k Sequence Enumeration on Dynamic Stratified Graphs

We compare our k stratified shortest path algorithm against the following schemes: (a) *Enum-All*, an exhaustive algorithm which enumerates all matches before picking the top-k; (b)*D-Yen*, an dynamic version of Yen's algorithm [23] based on the code available at *http://code.google.com/p/k-shortest-paths/* modified to eliminate redundant work in dynamically evolving graphs. To implement D-Yen, we follow an approach similar to the k dynamic stratified shortest path algorithm shown in Figure 3.8; we essentially replace the calls to the k stratified shortest paths algorithm with calls to Yen's k shortest paths algorithm. Figure 4.1 verifies under various parameter settings that our modification of the Yen's algorithm is indeed working more efficiently in the case of dynamically evolving graphs than a naive application of the Yen's algorithm. Here we do not consider Eppstein's



(a) k (ws=50, step=20)



(b) ws(k=10, step=20)

Figure 4.1: Performance of the original Yen's algorithm vs. D-Yen over dynamically evolving graphs

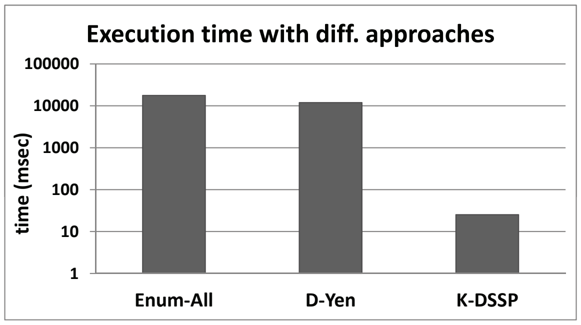
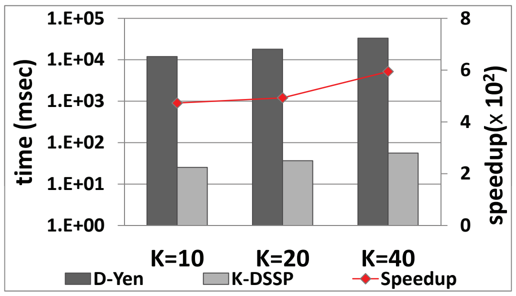


Figure 4.2: Comparison of top-k sequence matching approaches (k=10, sl=3, ws=50, step=20, ref=89)

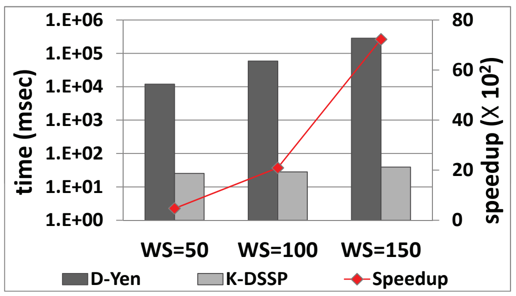
non-incremental algorithm [8] which cannot be used to supply *TMM* with on-demand sequence matches.

In Figure 4.2, we compare the time costs of processing top-k sequence matches (for patterns of length 3 – the scoring of the events is based on the distance of the sensors from a given point in space). As can be seen here, the proposed approach outperforms both exhaustive and dynamic Yen approaches by almost 3 orders of magnitude.

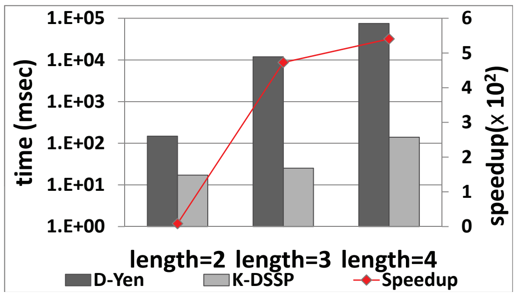
Figure 4.3(a) shows that the proposed algorithm scales well as k increases from 10 to 40 and the difference between the dynamic Yen's and the proposed approach gets larger when k increases. Figure 4.3(b) shows that, for larger window sizes, the event graph tends to be denser which results in more time to enumerate the top-k results. The *speedup* provided by the proposed approach also increases as the amount of needed work increases. This is also confirmed in Figure 4.3(c) which varies the target sequence length. Figure 4.3(d) evaluates the impact of the window shift size: A larger step implies a bigger



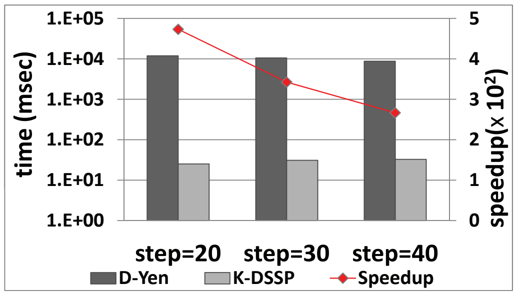
(a) k



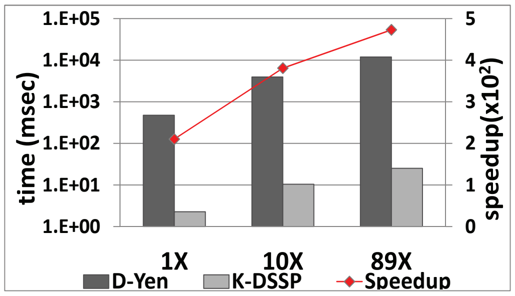
(b) ws



(c) sl (sequence length)



(d) step



(e) ref

Figure 4.3: Comparisons of *D-Yen* and k-*DSSP* for different parameter settings (Default: k=10, sl=3, ws=50, step=20, ref = 89X)

change in the stratified graph and less of the top-k results remain in the next window. As expected, the speedup lowers, but the proposed algorithm remains $> 100\times$ faster. Finally, Figure 4.3(e) compares three sequence queries which differ only in the frequencies of the last event class. As before, the speedup increases with the graph complexity.

4.4 Evaluation of Waste-Avoiding Top-k Joins

In this section, we evaluate the proposed waste-avoiding boundary selection strategy (WABS) underlying the top-k merge module with both sensor data and synthetic data. In the experiment with sensor data, when joining sequences on sensor IDs, random accesses (to fetch the best matching sequence with the given sensor ID in a given queue)

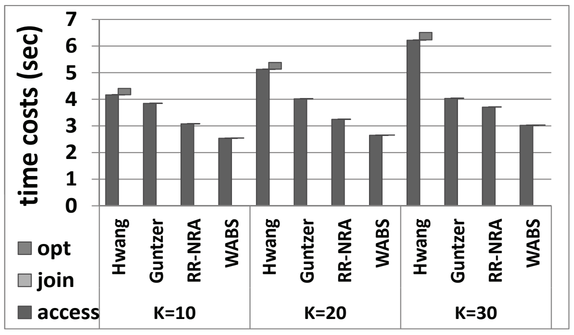


Figure 4.4: Different top-k join strategies (mf=avg, rss=1, q=2, ws=step=100, qp=[S7;S8])

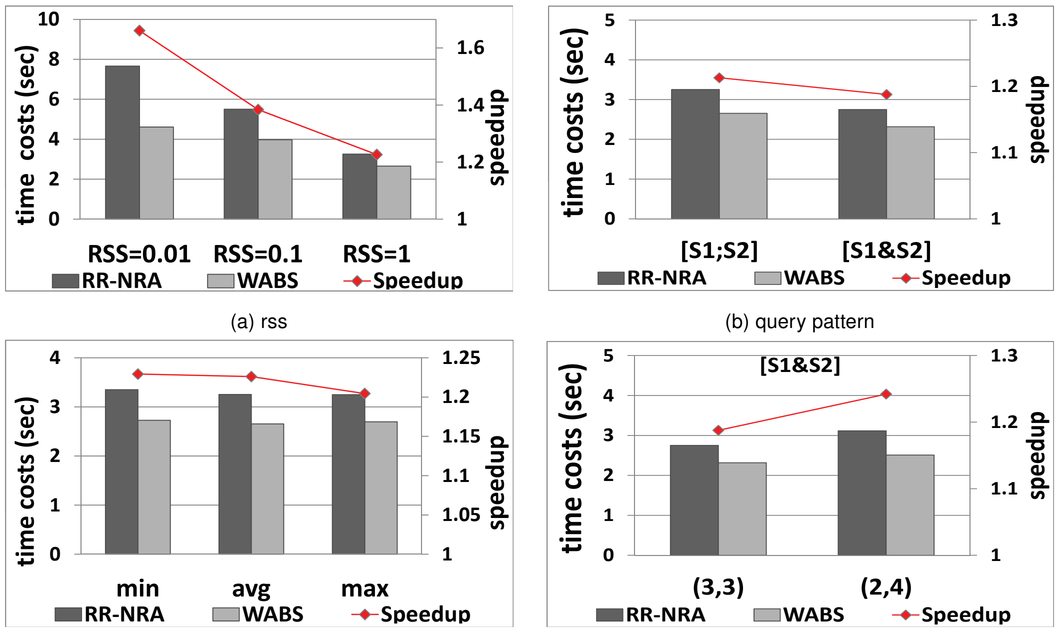


Figure 4.5: Performance evaluation on two-sequence join queries on sensor data (Default: k=20, mf=avg, rss=1, q=2, ws=step=100, qp=[S7;S8], sl = 3 for both queues)

are very expensive. Thus we adopt the WABS strategy with NRA. Performance of WABS on TA based algorithms are shown in experiments on synthetic data, together with more diverse parameter settings.

Waste-Avoiding Boundary Selection Strategy on Sensor Data

To evaluate WABS, we compare it to round-robin, gradient-based (Güntzer [12]), and statistics-based (Hwang [25]) strategies. For the latter, statistics are collected based on data that have already been seen.

The key experiment parameters are listed in Table 4.2. Queries involve sequence patterns and their combinations. For example, the pattern qp = {[S7&S8];S9} would state that (a) sequences S7, S8 and S9 (all generated by the same sensor mote) should occur in the same window and (b) both S7 and S8 must occur before S9. Unless specified otherwise, we consider sequence patterns each with three event classes and linear sum of each event score as the overall sequence score.

Also, in this section, we consider the impact of the waste-avoiding strategies assuming that the *TMM* recomputes matches from scratch at each iteration. Later in Chapter 4.5, we investigate the impact of the fully dynamic operation of *TMM*.

Figure 4.4 shows that the proposed WABS strategy performs the best among the alternatives for different k. Curiously, the gradient-based and statistics-based scheduling strategies perform worse than round-robin, indicating that they in fact lead to misleading or out-of-date plans. Therefore, in the rest of the experiments, we compare WABS directly to the round-robin strategy.

Figure 4.5 compares the WABS strategy against the round-robin strategy for two-queue top-k joins under different settings. Figure 4.5(a) shows that the more restrictive the join condition is the higher the speedup provided by WABS: this is because more work needs to be done to identify the k results and WABS helps prune redundant work. This is confirmed in Figure 4.5(b), where a more restrictive temporal order constraint ([S7;S8]), and in Figure 4.5(c), where a harder to commit merge function (*min*), provide larger speedups. Figure 4.5(d) shows that when there is a marked difference between the complexities (thus enumeration costs) of the sequence patterns being combined, WABS is able to identify and leverage this difference to boost the speedup it provides.

Figure 4.6 compares the results for more complex queries combining three sequence patterns. This figure also confirms the observation that the speedups provided by WABS increases as the pattern becomes more restrictive and, consequently, as it becomes harder for the NRA to identify the target k matches to the query.

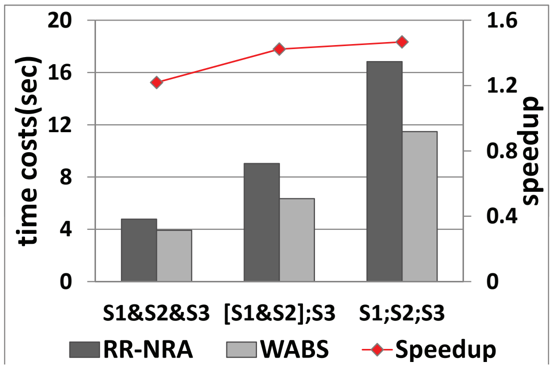


Figure 4.6: Performance evaluation on three-sequence join queries on sensor data (k=20, rss=0.01, mf=avg, q=3, sl=3 , ws=step=100)

Parameters for top-k join experiments on synthetic data	
k	top results to be returned
mf	merge function
rjs	relative join selectiviy
q	number of queues to be joined
dist.	uniform, normal or zipfian
cost ratio	$(cs_1 : \ldots : cs_i, cr_1 : \ldots : cs_i)$, for the i^{th} queue

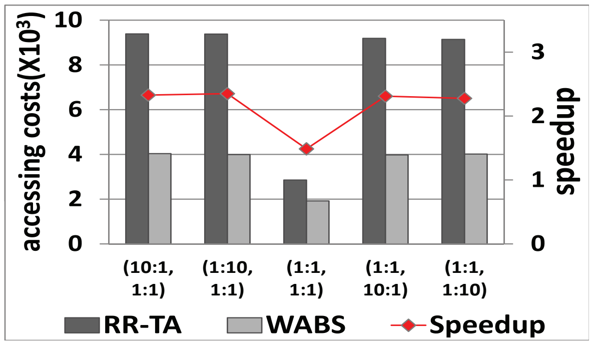
Table 4.4: Experiment parameters

Waste-Avoiding Boundary Selection Strategy on Synthetic Data

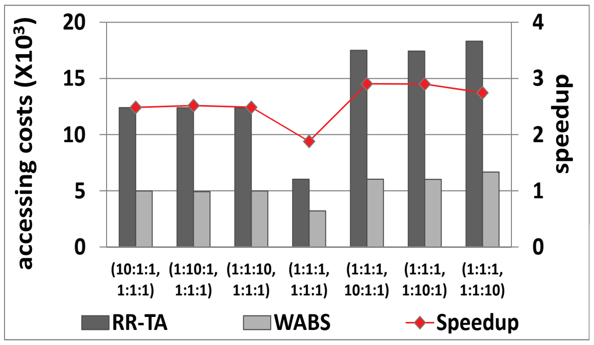
Due to the important limitation of joining sequences (as opposed to events) that random accesses to the different queues are expensive to enumerate, in the previous evaluation with sensor data, we applied WABS and NRA. In this section, we evaluate WABS using synthetic data; this provides more diverse parameter settings, including different value distributions as well as random accesses. Thus, this also enables us to evaluate WABS strategy in the case of TA-based top-k joins.

For each experiment reported in this section, we generate five sets of data and report the average performance. Each queue has 10K inputs and each input has a unique ID occurring once in one queue. Table 4.4 lists the experiment parameters used in this section. As before, k is the target number of results, mf denotes the merge function, and q denotes the number of queues. The other parameters are described below:

- **rjs** denotes the *relative join selectivity* of each queue. To be specific, when rjs=1,



(a) two-queue join



(b) three-queue join

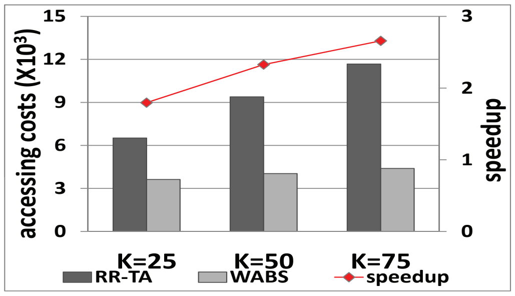
Figure 4.7: Comparison for varying access cost scenarios (Default: k=50, mf=min, rjs=1, uniform)

each entry in a given queue joins with exactly one other entry (the one with the same ID) on the other queues. When rjs=0.1, however, one of ten times entries with the same ID can be joined. Similarly, when rjs=0.01, one of one hundred times entries with the same ID can be joined. The later two cases provide 10 and 100 times more strict relative join selectivity than the case rjs=1.

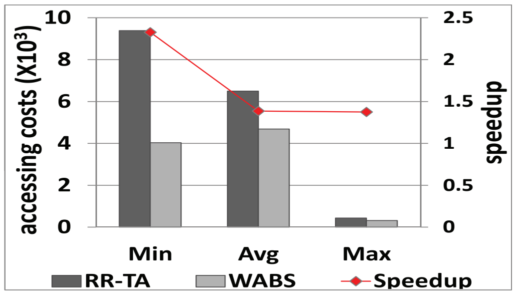
- **dist** indicates the distribution of the scores in each queue. Queues are assumed to be independent.

- **cost ratio** indicates the relative access costs for different queues. For example, $(\alpha_1 : \alpha_2, \beta_1 : \beta_2)$ means the sorted access to the first queue has a cost of α_1 units and random access to it has a cost of β_1 unit; in contrast for the second queue, sorted and random accesses have α_2 and β_2 unit costs, respectively.

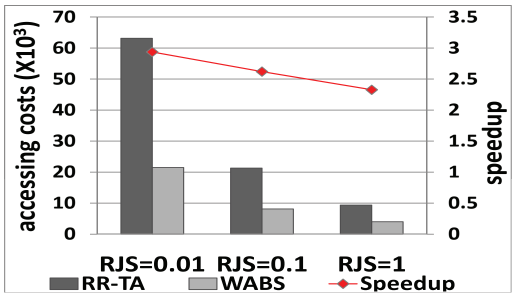
Figures 4.7(a) and (b) compare the top-k evaluation performance, under different cost ratio scenarios, for two and three queues, respectively. As can be seen here, the



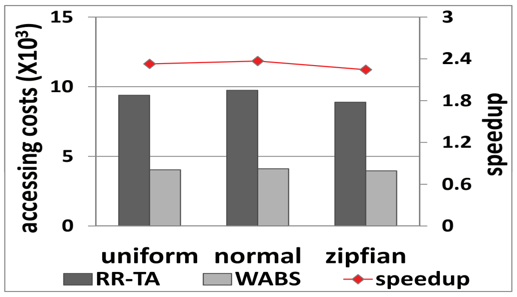
(a) k



(b) merge function



(c) relative join selectivity

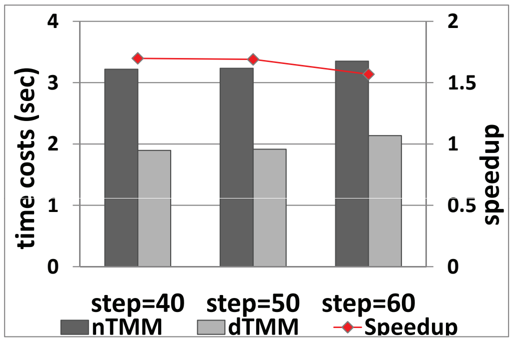


(d) distribution

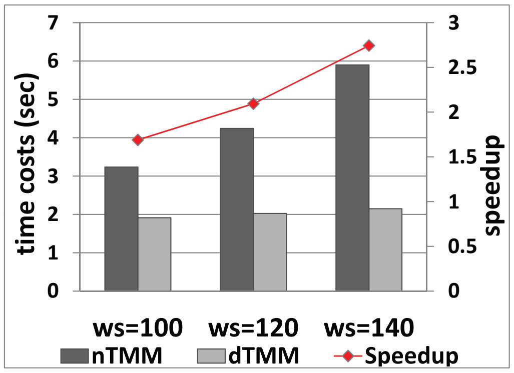
Figure 4.8: Performance comparison for varying parameter setting on synthetic data (Default: k=50, q =2 , mf=min, rjs=1, dist=uniform, cost ratio=[10:1, 1:1])

WABS strategy is able to adjust to the differences in access cost ratios and provides the best speedups ($\sim 3\times$) as the complexity of the top-k join increases (e.g., when the number of input queues increase or when the cost asymmetry among the queues is high). Interestingly, even in the cases where the access costs are symmetric, WABS is able to provide up to $2\times$ gains.

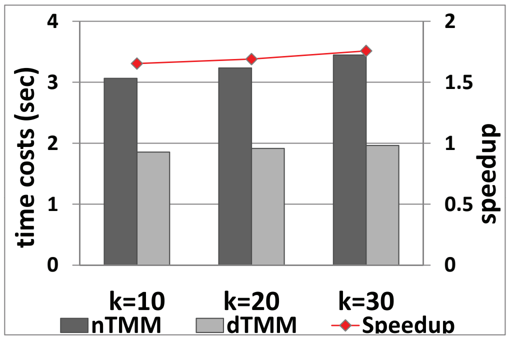
Figure 4.8 studies the speedup provided by WABS strategy as for different parameters. Figure 4.8(a) through (c) show that, as was the case for the experiments on sensor data, the WABS provides ever increasing speedups as the top-k problem gets harder: in Figure 4.8(a) higher speedup is observed for larger k, in Figure 4.8(b) higher speedup is observed for the *min* merge function (which in general requires more accesses than other merge functions), and in Figure 4.8(c) higher speedup is observed for $rjs = 0.01$, which is the case where it is hardest to locate inputs that satisfy the join condition. Finally, Figure 4.8(d) studies the speedup provided by WABS strategy as for different score distributions. While, the speedups are similar, there are some noticeable differences. As expected, Zipfian distribution, in which there are less high scoring inputs is



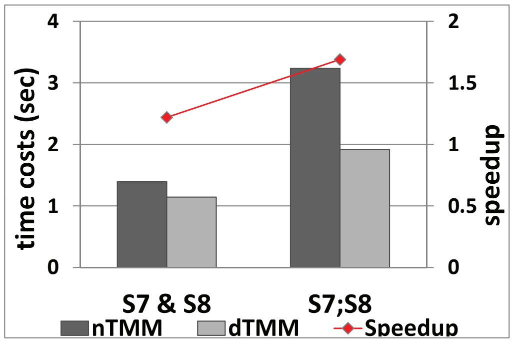
(a) *step*



(b) *ws*



(c) k



(d) pattern

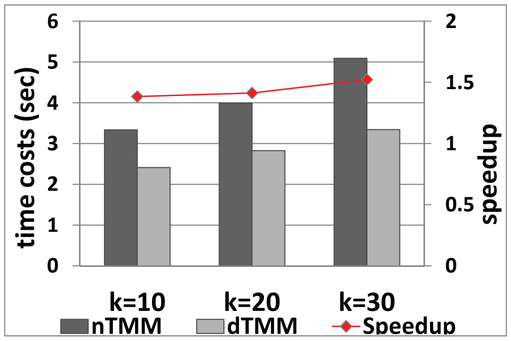
Figure 4.9: Comparison of Dynamic *TMM* (*dTMM*) and non-dynamic *TMM*(*nTMM*) (Default: k=20, rss=1, mf=avg, q=2, q=[S7;S8], sl=3, ws=100, step=50)

able to complete with least number of accesses, therefore the speedup is the lowest ($\sim 2.25\times$); in contrast, the normal distribution where there are a large number of mediocre (close to the mean) scores takes the most number of accesses and, consequently, the speedup is the largest ($\sim 2.35\times$) among the three distributions.

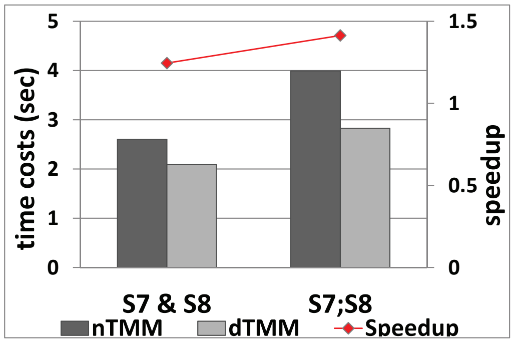
4.5 Evaluation of Dynamic Top-k Merge Module (dTMM)

In this section, we assess the effectiveness of proposed dynamic top-k complex match maintenance mechanism with the sensor data. Figure 4.9 provides results showing the performance gain of the dynamic version of *TMM* (*dTMM*), which reuses prior join computations when the window moves, over the non-dynamic version (*nTMM*), which recomputes top-k matches from scratch for each window[2]. Both versions of TMM adopt WABS as the join scheduling strategy. As can be seen in this Figure 4.9(a), dynamic

[2]Note that, both dTMM and nTMM leverage dynamic computation of the shortest paths on the stratified graph; their difference is in the computation of the join process.



(a) k



(b) pattern

Figure 4.10: Effect of nonlinear preference function in sequence query on Dynamic *TMM* (*dTMM*) (Default: rss=1, mf=avg, q=2, q=[S7;S8], sl=3, ws=100, step=50, pref(S7)=sum, pref(S8)=square root of sum of square)

operation of *TMM* provides significant amounts of savings (>1.5$\times$ speedups) under different window update frequencies (i.e. step size). Since a smaller $step$ enables more of the previous matches to remain in the window, as expected the highest speedup in the figure is achieved when $step = 40$. In Figure 4.9(b), we show the effects of different window sizes (ws). As ws increases, the underlying stratified graphs get more complex, which causes the increase of cost in accessing each sequence match. Consequently, the join process gets more expensive. This explains why in Figure 4.9(b) time costs of both approaches increase as ws gets larger. However, since dTMM benefits from the dynamic maintenance of previously accessed results, it leads to a much smaller increase. Thus the "speedup" of dTMM over nTMM gets higher with bigger window size. Figure 4.9(c) evaluates the effects of varying number of results to return. When we seek more results (i.e. higher k), as time window shifts, more previous results are likely to remain valid, leading to less recurring accessing costs in dTMM. Therefore, larger k produces more gain of dTMM over nTMM as shown by the "speedups". We also evaluate the effects of varying pattern relation between sequence queries in Figure 4.9(d). Since pattern $[S7; S8]$ enforces a stricter join condition than $[S7\&S8]$, as expected, the time costs of both nTMM and dTMM are higher in pattern $[S7; S8]$. Besides, a stricter join condition implies more costs in discovering the top matches, which enables the dynamic maintenance of top-k approach (dTMM) to provide more gain. Hence the "speedup" is higher in pattern $[S7; S8]$ than $[S7\&S8]$.

So far, we have considered the sum of each event weight as preference function of a sequence query. Figure 4.10 shows the result of dTMM versus nTMM when one of the sequence query has a nonlinear preference function. In this experiment, we still consider a complex query pattern with two sequence queries. However, instead of using linear sum as the preference score function for both sequences, we use linear sum on each event weight for S7 and square root of sum of square on each event weight for S8. Note that as described before in Chapter 3.4, when sequence queries have nonlinear preference function, we handle it with an instance of TMM module. In Figure 4.10(a), we see that with a nonlinear preference function, the proposed dTMM still maintains $\sim 1.4\times$ speedup compared to nTMM. Figure 4.10(b) again shows that in a query pattern of more strict join condition (i.e. $[S7; S8]$), more accessing efforts are required and thus make dTMM more rewarding in terms of “speedups”.

Chapter 5

CONCLUSION

In this work, we introduced the problem of top-k pattern evaluation over event streams and introduced a complex pattern ranking (CPR) framework. In order to efficiently identify the top sequence matches, we proposed a graph-based sequence enumeration strategy. We have first shown that events in a given window can be organized into what we refer to a *stratified stream graph* and that the best matches can be identified within this graph using a novel k-shortest path algorithm. We have also shown that within a streaming environment data changes in shifting windows can be captured as dynamic evolutions of stratified graphs and that shortest paths can be maintained without having to re-enumerate all shortest paths for each window shift. Experiments show that the proposed algorithms provide $> 100\times$ gains in execution times over alternatives. We next showed that in streaming environments top-k joins for combining sequence matches into complex patterns require new strategies that can adapt effectively (and cheaply) the changes. We proposed a waste-avoiding boundary selection strategy (WABS) for scheduling accesses to the input queues. Experiment results on the Intel-Berkeley sensor data showed that the proposed strategy can reduce the amount of work needed to enumerate top-k complex patterns using an NRA-based strategy up to 60%. TA-based experiments on synthetic data showed up to $3\times$ speedups when the access costs are highly asymmetric. In addition, by adopting a new proposed dynamic top-k complex query match maintenance mechanism, we avoid the re-computation of previous results, enabling the query answering process to be much more efficient.

REFERENCES

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. pages 147–160, 2008.

[2] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *SIGMOD*, 2008.

[3] Husain Aljazzar and Stefan Leue. K * : A directed on-the-fly algorithm for finding the k shortest paths. Technical report, Univ. of Konstanz, 2008.

[4] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.

[5] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD '02*, pages 310–321. ACM, 2002.

[6] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Nikos Sarkas. Ad-hoc top-k query answering for data streams. In *VLDB*, pages 183–194, 2007.

[7] Alan Demers, Johannes Gehrke, and P. Biswanath. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[8] David Eppstein. Finding the k shortest paths. In *Proc. 35th Symp. Foundations of Computer Science*, pages 154–165. IEEE, 1994.

[9] F. Fabret, H. A. Jacobsen, F. Llirbat, Jo ao Pereira, Jo A. Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, pages 115–126, 2001.

[10] Ronald Fagin. Combining fuzzy information from multiple systems . In *J. Comput. System Sci*, pages 216–226, 1996.

[11] Ronald Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31:2002, 2002.

[12] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießing. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, pages 622–628, 2001.

[13] Parisa Haghani, Sebastian Michel, and Karl Aberer. Evaluating top-k queries over incomplete data streams. In *CIKM*, pages 877–886, 2009.

[14] Parisa Haghani, Sebastian Michel, and Karl Aberer. The gist of everything new: personalized top-k processing over web 2.0 streams. In *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010*. ACM, 2010.

[15] John Hershberger, Matthew Maxel, and Subhash Suri. Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Trans. Algorithms*, November 2007.

[16] Víctor M. Jiménez and Andrés Marzal. A lazy version of eppstein's k shortest paths algorithm. In *WEA*, pages 179–190, 2003.

[17] Xingyi Jin, Xiaodong Lee, Ning Kong, and Baoping Yan. Efficient complex event processing over rfid data stream. In *ICIS*, 2008.

[18] J.Y.Yen. Finding the k shortest loopless paths in a network. *Managemen Science*, 17:712–716, 1971.

[19] Mine H Katoh N, Lbaraki T. An efficient algorithm for k shortest simple paths. *Networks*, page 411, 1982.

[20] E.L. Lawler. A procedure for computing the k best solutions to discrete optimisation problems and its application to the shortest path problem. *Management science*, page 401, 1972.

[21] Ming Li, Mo Liu, Luping Ding, Elke A. Rundensteiner, and Murali Mani. Event stream processing with out-of-order data arrival. *Distributed Computing Systems Workshops*, 0:67, 2007.

[22] Surya Nepal and M.V. Ramakrishna. Query processing issues in image(multimedia) databases. In *ICDE*, pages 22–29, 1999.

[23] M. Pascoal and E. Martins. A new implementation of yen's ranking loopless paths algorithm. *4OR: A Quarterly Journal of Operations Research*, 2000.

[24] Thanh Tran, Liping Peng, Boduo Li, Yanlei Diao, and Anna Liu. Pods: a new model and processing algorithms for uncertain data streams. In *SIGMOD*, 2010.

[25] Seung won Hwang and Kevin Chen chuan Chang. Optimizing top-k queries for middleware access: A unified cost-based approach. *ACM Trans. Database Syst.*, 32, March 2007.

[26] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.

[27] Haopeng Zhang, Yanlei Diao, and Neil Immerman. Recognizing patterns in streams with imprecise timestamps. *PVLDB*, 3(1):244–255, 2010.