High Performance Analytics in Complex Event Processing

by

Yingmei Qi

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

Jan 2013

APPROVED:

________________________________________
Professor Elke A. Rundensteiner, Thesis Advisor

________________________________________
Professor Mohamed Eltabakh, Thesis Reader

________________________________________
Professor Craig Wills, Department Head
Abstract

Complex Event Processing (CEP) is the technical choice for high performance analytics in time-critical decision-making applications. Although current CEP systems support sequence pattern detection on continuous event streams, they do not support the computation of aggregated values over the matched sequences of a query pattern. Instead, aggregation is typically applied as a post processing step after CEP pattern detection, leading to an extremely inefficient solution for sequence aggregation. Meanwhile, the state-of-the-art aggregation techniques over traditional stream data are not directly applicable in the context of the sequence-semantics of CEP. In this paper, we propose an approach, called A-Seq, that successfully pushes the aggregation computation into the sequence pattern detection process. A-Seq succeeds to compute aggregation online by dynamically recording compact partial sequence aggregation without ever constructing the to-be-aggregated matched sequences. Techniques are devised to tackle all the key CEP-specific challenges for aggregation, including sliding window semantics, event purging, as well as sequence negation. For scalability, we further introduce the Chop-Connect methodology, that enables sequence aggregation sharing among queries with arbitrary substring relationships. Lastly, our cost-driven optimizer selects a shared execution plan for effectively processing a workload of CEP aggregation queries. Our experimental study using real data sets demonstrates over four orders of magnitude efficiency improvement for a wide range of tested scenarios of our proposed A-Seq approach compared to the state-of-art solutions, thus achieving high-performance CEP aggregation analytics.
Acknowledgements

I would like to express my gratitude to my advisor professor Elke Rundensteiner. Thank for her continuous support for my research and thesis work. Thank for her time on revising my thesis again and again, to make it perfect. I really appreciate her patient guide, encouragement, as well as immense knowledge, which help me to continuously grow and improve during my master study.

My thanks also go to my thesis reader professor Mohamed Eltabakh, for his valuable advises on my thesis work, which helps me to improve the quality of this thesis.

I also thank the researchers from HP labs: Song Wang, Chetan Gupta, Ismail Ari, for all the inspirations and feedbacks on my research work.

I thank all my lab-mates in WPI DSRG: Lei Cao, Chuan Lei, Mo Liu, Di Yang, Medhabi Ray, Xika Lin, for all the feedbacks and discussion on my work; Cheng Cheng, Kenneth Loomis, for their hard work on the implementation. I also want to thank my best friend at WPI Yumeng Qiu, for all the fun we have during the last two and half years.

Lastly, I want to thank my whole family: my parents Wansong Qi and Jianhua Zhou; my fiance Jing Yang; and my sisters Yingli Qi and Yingjin Qi. Thank for giving me tremendous support for no matter my life or study.
Contents

1 Introduction 1
  1.1 Motivation .................................................. 1
    1.1.1 Running Example ....................................... 2
  1.2 State-of-art Limitations ................................. 3
  1.3 Challenges & Proposed Solution ......................... 4
  1.4 Contributions ............................................ 6

2 CEP Basics 8
  2.1 Basic Concepts ........................................... 8
  2.2 Language Specification .................................. 9
  2.3 Stack-Based Pattern Evaluation ......................... 10

3 A-Seq Single Query Strategy 12
  3.1 Basic Approach: Dynamic Prefix Counting ............ 12
  3.2 Sliding Window Support for A-Seq ..................... 17
  3.3 Negation Support for A-Seq ........................... 22
  3.4 Predicates Support for A-Seq ......................... 25

4 A-Seq Multi Query Strategy 27
  4.1 The Prefix Sharing Strategy .......................... 28
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Stack Based Pattern Evaluation</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Sequence Form Process</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Prefix Pattern Count Update Process</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>DPC Computation Process for Sequence Pattern (A,B,C,D)</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>Basic A-Seq Algorithm</td>
<td>17</td>
</tr>
<tr>
<td>3.5</td>
<td>Pushing Windows Down into A-Seq Using SEM</td>
<td>20</td>
</tr>
<tr>
<td>3.6</td>
<td>SEM A-Seq Algorithm</td>
<td>21</td>
</tr>
<tr>
<td>3.7</td>
<td>Pushing Negation Down to Sequence Detection Process</td>
<td>23</td>
</tr>
<tr>
<td>3.8</td>
<td>Hashed Prefix Counter (HPC)</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>PreTree Structure</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Connect Counts of Two Substrings</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>SnapShot Maintenance</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Snapshot Computation of Multi-connect</td>
<td>35</td>
</tr>
<tr>
<td>4.5</td>
<td>Mapping to Maximum Independent Set Problem</td>
<td>44</td>
</tr>
<tr>
<td>4.6</td>
<td>A-Seq Optimizer Greedy Search Algorithm</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>(a)(b) Single A-Seq Performance by Varying Pattern Length</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>(a)(b) Single A-Seq Performance by Varying Window Size</td>
<td>50</td>
</tr>
<tr>
<td>5.3</td>
<td>(a) Single A-Seq Stress Test (b) Negation Test for A-Seq</td>
<td>50</td>
</tr>
</tbody>
</table>
5.4 (a)(b) Chop-Connect Performance Evaluation by Varying Connect Event Frequency ................................. 53
5.5 (a)(b) Chop-Connect Performance Evaluation by Varying Shared Length and Shared Query Number ................................................. 53
5.6 (a)(b) Comparisons between Multi-query Optimizers in terms of Running Time and Effectiveness ....................... 55
List of Tables

4.1 Terminology Used in Cost Estimation . . . . . . . . . . . . . . . . . . 39
Chapter 1

Introduction

Complex Event Processing (CEP) systems detect complex sequence patterns over high speed event streams. As more applications from stock market analysis to seismic motion monitoring require time-critical decision-making support, enhancing the performance of CEP systems has drawn increased attention recently [15, 17, 25].

1.1 Motivation

Current research in CEP focuses on how to efficiently detect time-valued correlated sequence patterns [4, 17, 25]. Yet the processing of CEP aggregation, which is critical for high-performance analytics, has largely been overlooked. CEP aggregation queries focus on asking questions from an overall statistic view point, such as how frequently a certain sequence pattern happens over some period of time or what is the maximum value of event attributes in a pattern among all matched sequences. In fact, CEP aggregation queries are prevalent in every aspect of our digital daily life and business world alike. For example, in fraud detection, a susceptible credit card fraud might be defined as a particular online purchase pattern repeatedly arising with respect to the same credit card with
the total value over $10,000 in the last 60 minutes. In seismic monitoring, geologists define abnormal seismic motion patterns. Once the occurrence of those patterns surpassed a given threshold frequency, an alert must be issued.

As the above examples demonstrate, a timely response is of paramount importance when processing CEP aggregation queries. For applications, even a one-second delay may lead to a loss of huge amounts of money or even perhaps human lives. Thus, high-performance support for processing CEP aggregation queries is the central challenge tackled in this paper.

1.1.1 Running Example

Using a running example in E-Commerce, we now briefly introduce the state-of-art technique in CEP aggregation processing. In online shopping, customer shopping habits are crucial for optimally timing targeted promotions, product recommendations, and customized display layout on webpages. For example, consider the web click sequence pattern \((\text{ViewKindle}, \text{BuyKindle}, \text{ViewCase}, \text{BuyCase})\) (in short \(VK, BK, VC, BC\)) that corresponds to a purchase pattern revealing that a customer purchases a Kindle case shortly after they purchased a Kindle. If this pattern is extremely frequent, then it implies that customers prefer to buy a case to protect their E-Reader. Given this insight, merchants would want to adjust their market strategies such as displaying Kindles and cases on the same webpage when customers search for either product, recommending cases when customers view a Kindle webpage, as well as bundling a Kindle and a matching case together as a package with a promotion price.

For large E-Commerce companies, such as Amazon and eBay, a huge amount of web-click data is generated continuously every single second. Suppose the merchant asks for the total occurrence of the web-click pattern \((VK, BK, VC, BC)\), where the Kindle model is "touch" within the last 10 minutes. In current CEP systems [4, 17, 25], this query will
be processed using a two-step aggregation process, namely, first the **Sequence Detection and Construction (SDC)** and then the **Aggregation Computation Process (AGG)**. The SDC firstly finds all event instances that match the CEP pattern such as \((VK, BK, VC, BC)\) sequencing along with predicates such as "touch" model and constructs sequence results (see Chapter 2 for details). As the last AGG step, the aggregation function, such as COUNT, is applied to count the number of all matched sequences constructed in the SDC step.

### 1.2 State-of-art Limitations

This two-step aggregation process suffers from serious performance bottlenecks, including:

**Expensive CPU Costs.** CPU resources are spent on compute-intensive tasks, namely, first to maintain the ordering occurrence information among continuous incoming events when detecting each sequence match. Second, to construct each individual matched event sequence. However, these result sequences will be thrown away after they have been constructed, as what the query asks for is only the aggregated result: a simple number.

**Huge Memory Storage.** To detect and construct each full sequence match, memory resources must be dedicated to store all incoming event instances. In addition, the temporal sequence results must also be stored for a later negation check (if any) and subsequent aggregation computation (see step 2 above).

No research work to date has focused on this critical aggregation performance problem in the CEP context. In most current CEP systems [4, 17, 25], the above two-step aggregation process would thus be applied to process such queries. This points to the opportunity to greatly improve performance if aggregation could be obtained instantaneously as part of the sequence detection process itself, namely, without having to first
construct all matched sequences.

However, existing aggregation techniques in the literature do not tackle this challenge. On the one hand, most aggregation techniques in streaming environments are designed for independent data records [11, 13], i.e., they tend to have set-based semantics rather than sequence or pattern semantics. That is, they collect a set of independent data records irrespective of their time-order and then aggregate them. The time-order constraint between data items that are to be aggregated as in our context now clearly complicates the problem as we cannot accumulate the aggregation result with the arrival of event instances. On the other hand, techniques proposed for aggregation in traditional sequence databases assume data is static and given apriori [16, 19, 21]. A lot of techniques such as indexing is exploited to efficiently search for the data sequences that match the required pattern, however, for aggregation, they still apply the two-step post aggregation approach.

In addition, in the streaming context, the problems caused by sliding window semantics are non-negligible, as a high-performance re-computation and purging mechanism must be supported to correctly update the aggregation results due to data expiration. Other challenges specific to CEP include negation of a pattern are not tackled in static sequence databases. Therefore, a customized high-performance approach for processing sequence pattern-based aggregation over stream must be developed to fill this void.

### 1.3 Challenges & Proposed Solution

First, a light-weight technique must be designed to overcome the performance bottleneck of state-of-art methods both in terms of CPU resource consumption and memory utilization. To achieve this goal, we devise the $A$-Seq methodology that correctly pushes aggregation computation into the sequence detection process itself by dynamically calculating partial aggregation result without first having to detect each full sequence match.
As additional benefit, the actual sequence construction process is completely eliminated. Our preemptive Dynamic Prefix Counting (DPC) algorithm supports the computation of partial aggregation instantly upon the arrival of each event instance, while our compact data structure Prefix Counter maintains the relevant ordering semantics of event instances for a sequence match identification, but without storing individual event instances.

Second, achieving a light-weight aggregation solution is complicated by the fact that the aggregation result needs to be continuously updated with the expiration of event instances from the current window. The design of such an expiration mechanism is challenging, as one event expiration might cause an arbitrary number of sequence matches to become invalid. For this, we propose an effective expiration mechanism that is not only sequence-aware to detect invalid sequence matches, but also light-weight maintaining only aggregated results, instead of all raw event instances or worst yet all sequence matches. Furthermore, our algorithm also supports pattern queries with negation, namely, the occurrence of an event instance matching a negated expression within a sequence pattern may lead some detected sequence matches to be invalid. In addition, this mechanism is agile to efficiently function over high-speed streams with high expiration frequency.

Third, applications often process many similar queries over the same popular data stream [5, 11, 15]. The sharing of computations among queries in a workload has been shown to be vital for scalability in the literature [16, 28], yet these work focus either on non-sequence based queries [11, 24] or on the computation sharing of sequence construction process [15]. Given that A-Seq approach avoids the construction of sequences, we now set out to obtain the aggregation result of a full query pattern by stitching together aggregations of sub-patterns. However, this leads to the paradox that keeping the time implicit order semantics among sub-patterns to ensure correctness would prevent us from exploiting the key merit of our given A-Seq approach. We overcome this challenge and design a computation method that leverage the aggregate meta information maintained by
our A-Seq technology to successfully stitch aggregation results of different sub-patterns together. This method makes the selectively sharing aggregation computations over arbitrarily sub-patterns within multi queries possible.

Fourth, we show that the selection of a good sharing plan for a given query workload is an NP-Complete problem (Section 4.3.3). We devise a cost-based optimizer that is practical in spite of the complexity of this search space. The optimizer applies a hill-climbing search strategy, which is shown to be efficient to produce a good sharing plan close to the optimal solution in terms of its quality by our experimental studies in Section 5.3.

### 1.4 Contributions

In our A-Seq approach, we successfully tackle all challenges outlined above. Contributions of our work on solving this real time CEP query aggregation problem include:

- **The A-Seq Methodology.** We design a dynamic computation algorithm and a compact data structure for processing the aggregation of sequence pattern queries, A-Seq succeeds to conduct sequence detection and aggregates computation simultaneously in an extreme efficient manner, which overcomes the performance shortcoming of the state-of-art two-step aggregation method, greatly enhancing both CPU costs and memory utilization by several orders of magnitude (See Section 5.2).

- **CEP Featured Problems Solution.** We propose a set of techniques to cope with the core CEP specific problems. For instance, an agile mechanism is proposed to update the aggregation result timely with the event expiration that caused by sliding window constraint. For negated patterns, an recounting principle is plugged into the sequence detection process which enables the on-the-fly negation check.
• **Multi-query Sharing Strategy.** To support effective aggregation computation sharing of arbitrary sub-patterns among a given set of queries. Our *multi A-Seq* strategy chops queries into sub-pattern and reuses the aggregation results of sub-patterns among multi queries that share them. Effective data structures and sophisticated computation algorithm are devised to support the arbitrary sub-pattern sharing.

• **Cost-based Multi A-Seq Optimizer.** We design a cost-driven algorithm (*A-Seq* optimizer) to constructs the a good sharing plan for a given query workload. *A-Seq* optimizer employs several well-known algorithm to simplify the sharing plan search problem from NP-hard complexity to guaranteed polynomial running time. Complete complexity analysis of both exhaustive and optimized search algorithm is given. The practicability and accuracy of *A-Seq* optimizer is demonstrated in our experimental study (Section 5.3).

• **Experimental Evaluation.** We conducted extensive experiments over real world data streams validating the effectiveness of our proposed methods. The experimental study compares the performances with state-of-art approaches, demonstrating over four orders of magnitude efficiency improvement for a wide range of tested scenarios of our proposed *A-Seq* approach.

The rest of the paper is organized as follows. Chpater 2 gives a brief introduction of the preliminary knowledge about CEP. The *A-Seq* approach for single query is optimization introduced in Chapter 3, the multi-query solution is presented in Chapter 4. Experimental results are analyzed in Chapter 5. Chapter 6 covers related work, while Chapter 7 concludes the whole paper.
Chapter 2

CEP Basics

2.1 Basic Concepts

An event instance is an occurrence of interest denoted by lower-case letters (e.g., ‘e’). We use $e_i.ts$ to denote the time-stamp when $e_i$ arrives to system. For compactness, the time of occurrence of an event instance is denoted by its subscript $i$.

An event type $E$ of an instance $e_i$ describes the essential features associated with the event instance $e_i$, and is denoted by $e_i.type$. For the example in Section 1.1.1, clicks on a page to view a Kindle are Clicks on a page to view a Kindle, denoted as $vk_i$, are instances of event type ViewKindle, that is, $vk_i.type = ViewKindle$.

Sequence Pattern Query. We focus on sequential pattern queries denoted by SEQ operator below, a core feature of most event processing systems [4, 25].

Definition 1 A SEQ operator specifies an order on the time-stamps in which the instances of specific event types must occur to match the sequence pattern.
SEQ(E_1, E_2, ..., E_n) = \{ < e_1, e_2, ..., e_n > | e_1.ts < e_2.ts < ... < e_n.ts \land (e_1.type = E_1) \land (e_2.type = E_2) \land ... \land (e_n.type = E_n) \}.

(2.1)

The symbol "!" before an event type E_i indicates that the instance of type E_i is not allowed to appear within the specified position in the stream [25]. We call such E_i a negative event type. Consequently, when an event type E_i is used in a SEQ construct without “!”, we call it a positive event type.

SEQ(E_1, !E_i, E_n) = \{ < e_j, e_k > | (e_j.ts \leq e_j.te < e_k.ts \leq e_k.te) \land (e_j.type = E_1) \land (e_k.type = E_n) \land (\neg \exists e_f where (e_f.type = E_i) \land (e_j.te < e_f.ts \leq e_f.te < e_k.ts)) \}.

(2.2)

Beyond Equation 2.2, negative event types can also exist in the beginning or the end in a SEQ operator. For details see [25]. Other pattern operators such as conjunction (AND) and disjunction (OR) can be defined in a similar manner [17]. But henceforth, we focus on ordered sequence patterns.

2.2 Language Specification

We adopt a CEP query language commonly used in the literature [14, 15, 25], which has the following structure:

PATTERN <event pattern>
[WHERE <qualification>]
The PATTERN clause composed of the SEQ operator specifies sequence and negation constraints has been explained above. The WHERE clause contains predicates on attributes of the events, such as the E-Reader’s model, etc. The AGG clause specifies an aggregation function such as COUNT the number of matched sequence result. The WITHIN clause ensures that the time difference between the first to the last event instances matched by a pattern query falls within the window constraint.

The query example in Section 1.1.1 can thus be represented as:

```
Pattern SEQ <VK, BK, VC, BC>
Where VK.model=BK.model=
    VC.model=BC.model="touch"
Agg COUNT
Within 10 mins
```

### 2.3 Stack-Based Pattern Evaluation

First, each pattern query $q_i$ is compiled into a query plan. A window sequence operator, denoted by WinSeq($E_1, \ldots, E_n$, window) extracts all matches of instances within the sliding window as specified in query $q_i$. Queries with negative event types, denoted by WinSeq($E_1, \ldots, ! E_i, \ldots, E_n$, window), verify that no event instances of negative components such as $E_i$ exist in the indicated location among the positive instance matches.

An indexing data structure named $SeqState$ associates a stack with each event type $E$ in the query. Each new event instance of $e_{type} = E$ is appended to the end of its corresponding stack $E$. Event instances are augmented with pointers $ptr_i$ to adjacent events to
facilitate locating related events in other stacks during result construction. When an event instance \( e_n \) of the last event type \( E_n \) of a query \( q_i \) arrives, the compute function of \( q_i \) is initiated\(^1\). The result construction proceeds in a depth-first search along instance pointers \( ptr_i \) rooted at that last arrived instance \( e_n \). Each path composed of edges “reachable” from this root \( e_n \) to a leaf corresponds to one matched event sequence returned for \( q_i \).

**Example 1** Figure 2.1 depicts the event instance stacks for the pattern \( Q = SEQ(VK, BK, VC, BC) \). In each stack, its instances are sorted from top to bottom by their timestamps. When \( bc_2 \) of type \( BuyCase \) arrives, the most recent instance \( vc_2 \) is the last event event in \( ViewCase \) stack. The pointer of \( bc_2 \) thus points to \( vc_2 \), as shown in the parenthesis preceding \( bc_2 \). As \( BuyCase \) is the last event type in \( Q \), \( bc_2 \) triggers result construction. Two results \(<vk_1, bk_1, vc_2, bc_2>\) and \(<vk_1, bk_2, vc_2, bc_2>\) are constructed involving \( bc_2 \).

![Stack Based Pattern Evaluation](image)

Figure 2.1: Stack Based Pattern Evaluation

Example 1 illustrates the **Sequence Detection and Construction (SDC)** step discussed in Section 1.1.1. For aggregation computation, the aggregation function will be applied to the sequence results once they have all been constructed. The performance bottleneck of this two-step aggregation process has been discussed in Section 1.2.

\(^1\)If the last event type \( E_n \) in query \( q_i \) is a negative event type, postponed sequence evaluation is applied. We omit the details here.
Chapter 3

A-Seq Single Query Strategy

We now introduce our A-Seq methodology that computes aggregation on-the-fly. Here we first focuses on COUNT computation, which is one of the most popular aggregation operation in the context of sequence analysis (see Sections 1.1 and 1.1.1). In fact, most research related to sequence aggregation focuses on the COUNT operation from data mining, frequent sequence mining, to statistics [16, 20, 23]. Henceforth, we thus explain our strategies using the COUNT aggregation function in the rest of the paper.

3.1 Basic Approach: Dynamic Prefix Counting

Next, we introduce the key idea of pushing the aggregation computation into the sequence detection process, named the Dynamic Prefix Counting (DPC) method. To better understand how DPC works, let’s start by examining the sequence match formation process using the concrete example of pattern \((A, B, C)\) as in Figure 3.1. At time \(t_i\), one match \((a_1, b_1, c_1)\) has been found, while \(a_2\) is waiting for further event instances that can participate in the formation of new matches.

When \(b_2\) arrives at time \(t_{i+1}\), together with \(a_1\) and \(a_2\), we form two new sub-sequences
Figure 3.1: Sequence Form Process

(a_1, b_2) and (a_2, b_2). When c_2, the instance of last event type to form sequence (A, B, C), arrives at time t_{i+2}, we append it to subsequences (a_1, b_1), (a_1, b_2) and (a_2, b_2) to form 3 new (A, B, C) sequences. Thus, the total count of sequences constructed that match pattern (A, B, C) at t_{i+2} is 4, including the 3 newly formed sequences and one formed earlier. From this process, we observe that when each c_i arrives, we can obtain the count of pattern (A, B, C) by adding counts of two sub-patterns, namely, 1) the count of sub-pattern (A, B), to which c_i will be able to append to form new (A, B, C) matches, and 2) the count of the previously detected pattern (A, B, C). Generalizing this observation, when event E_n arrives at t_i, the count of a sequence pattern (E_1, E_2, ..., E_n) can be computed from the count of its Longest Prefix Pattern (LPP) (E_1, E_2, ..., E_{n-1}) at t_{i-1}, in the following manner:

\[
\text{Count}(E_1, E_2, ..., E_n)_{t_i} = \text{Count}(E_1, E_2, ..., E_{n-1})_{t_{i-1}} + \text{Count}(E_1, E_2, ..., E_n)_{t_{i-1}}
\]

(3.1)

Thus, for pattern (A, B, C), we recursively compute the count from the singleton prefix (A), until we get count of the full pattern (A,B,C). Given this recursive structure,
our problem can be solved through dynamic programming. That is, to get the count of the whole pattern (the final problem), we count all its prefix patterns (sub-problems) incrementally and stored those counts. Moreover, in the streaming context, the counts of these respective prefix patterns will be updated correspondingly upon the arrival of new events to continuously reflect real-time changes.

**Example 2** Figure 3.2 depicts the prefix count update process based on the computation rule given above when a new instance of a particular type arrives. The number at the lower right corner of each circle represents the count of this prefix pattern at the respective moment in time indicated at top of each column. When event instance \( b \) arrives at time \( t_{i+1} \), new matches of the prefix pattern that end in \( B \) will be triggered, namely, \((A, B)\). For this, we simply add the existing counts of \((A) = 3\) and \((A, B) = 2\) to get the new count of \((A, B) = 5\). The counts of all other prefix patterns remain unchanged. Similarly, when the instance \( d \) arrives, the same update process is applied to compute the new count of its corresponding pattern \((A, B, C, D)\).

In general, the rules for a pattern \((E_1, E_2, ..., E_i, ..., E_n)\) when an event instance \( e_i \) arrives at time \( t_j \), can be formulated as below:

**Update Rules:**

\[
\begin{align*}
e_{i,\text{type}} & \in E_1 : \\
e_{i,\text{type}} & \in E_i, (1 < i \leq n) : \\
e_i & . \text{type} \\
i. & C(E_1)_{t_j} = C(E_1)_{t_{j-1}} + 1, \\
ii. & C(E_1, ..., E_i)_{t_j} = C(E_1, ..., E_{i-1})_{t_{j-1}} + C(E_1, ..., E_i)_{t_{j-1}}.
\end{align*}
\]

We classify event types of a sequence pattern \((E_1, E_2, ..., E_i, ..., E_n)\) into three categories based on the different operations that event instances of this type must trigger:

- **Start Event Type (START).** The first event type \( E_1 \) in a sequence pattern \((E_1, E_2, ..., E_n)\).
When a \textit{START} instance arrives, we increase the count of \((E_1)\) by 1 since there is no prefix pattern prior to it.

- \textbf{Update Event Type (UPD)}. All other event types except \textit{START} in a sequence pattern \((E_1, E_2, \ldots, E_n)\). When a \textit{UPD} instance arrives, the count of the prefix pattern it triggers (ending with this \textit{UPD} type) will be updated. For example, the count of \((A, B, C)\) should be updated when a \(C\) instance arrives.

- \textbf{Trigger Event Type (TRIG)}. The last event type \(E_n\) in a sequence pattern \((E_1, E_2, \ldots, E_n)\).

  It not only performs the \textit{UPD} update operation, but also represents the completion of the full pattern. Thus, the just generated aggregation result will be returned to the user.

An example of this \textit{DPC} method is illustrated in Figure 3.3.

\textbf{Example 3} In Figure 3.3, when \(a_1\) arrives, as it is an \textit{START} instance, the count of \((A)\) is updated to 1. When \(b_1\) arrives, the count of \((A, B)\) is calculated as \(1(A) + 0(A, B) = 1\).
Similarly, count of \((A, B, C)\) is updated to 1 when \(c_1\) arrives. Continuing whenever a new event instance arrives, the count of the prefix pattern that ending with this instance type will be updated. Finally, when \(d_1\) arrives, we output result 2 to the user.

As we can see from Figure 3.3, A-Seq is a lightweight approach. From the CPU view, only one "ADD" calculation is required for each new event instance. From the memory view, we do not store any event instances. Rather, instances are immediately discarded upon their arrival and instantaneous processing. Since the count update only depends on the counts at the previous time point, it is sufficient to only store one single time slice copy of the data. Namely, only the counts for all the prefix patterns in the most recent column, instead of the whole update table depicted in Figure 3.3 are kept. We call this one column data structure the Prefix Counter (PreCntr).

The pseudocode for our basic A-Seq algorithm for a single query is given in Figure 3.4.
**Basic A-Seq** (Input: pattern query $q_i$, stream $S$)

- $e_i$: an event instance
- $e_i.cat$: event category $e_i$ falls in $(START, UPD, TRIG)$
- $L_q$: the number of event types in $q_i$

1. initialize a $PreCntr$ of size $L_q$
2. for each arriving event $e_i$ in $S$
3.     if $e_i.cat = \text{START}$
4.         apply update rule i
5.     if $e_i.cat = \text{UPD}$
6.         apply update rule ii
7.     if $e_i.cat = \text{TRIG}$
8.         return the full sequence count

---

**Figure 3.4: Basic A-Seq Algorithm**

### 3.2 Sliding Window Support for A-Seq

**Sliding Window Problem.** In the sliding window scenario [3], a continuous CEP aggregation query $Q(S, \text{win}, \text{slide}, Agg)$ returns the result of sequence aggregation $Agg$ computed over the content in the query window $W_i$ on the data stream $S$. Here we assume the window semantics in CQL [3], namely, the query window has a fixed window size $Q.\text{win}$ and a $Q.\text{slide}$ that slides from $W_i$ to $W_{i+1}$ whenever a new event instance arrives.$^{1}$ When the window slides, the events which have fallen out of the window should no longer participate in any future CEP sequence construction. Thus, they can be purged. However, for

---

$^1$With the tuple-driven sliding window type becoming among the most commonly used window type for CEP pattern detection query [4, 15, 25], we similarly use these semantics here. Otherwise, our technique could be easily extended to support other sliding window types.
CEP sequence aggregation, purging only the expired events is not sufficient. With events get expired, any sequence match that contain expired events would also become invalid. Thus, the partial aggregation results maintained by our A-Seq method would at that point account for many potentially invalid sequence matches. Thus, we must design a solution to correct the erroneous counts when events expire.

**Event Marking Strategy.** The Window-ID (WID) approach [13] introduced for processing non-CEP aggregation CQL queries, provides insights into how to tackle this event purging and result updating challenge. Namely, for our tuple-driven sliding window type, it marks each tuple with its life span, which reflects how long this instance will stay active. Only currently active tuples will be considered in the aggregation when ever the window slides. However, this technique, being designed for set-based aggregation, would now require us to keep all instances in the stacks, and then force us to repeat the fully computation over and over for each new event. This is incompatible with the core principle of our A-Seq method of keeping compact counts instead of recounting aggregation from scratch for each window. We now consider how to adapt the key principles of this strategy to our CEP time-order semantics.

First, we compute and mark the life span, namely, the timestamp when each particular event instance expire. Second, we record the count for each event instance, which indicates how many sequence matches would contain this event instance. For example, in Figure 3.1, the count information for $a_1$ is 2 (two sequences contain $a_1$), while for $a_2$ it is 0. We notice that this count keeps changing. As future events arrive, more sequences will be formed based on this event. Lastly, when an event instance expires, we must purge it and the count attached to this event should not be considered as part of the final total result.

However, this modified WID approach seriously complicates our core A-Seq solution. As we can see, we would now have to 1) store information about all incoming event
instances including their life span and sequence count contribution, 2) continuously modify the sequence counts formed on each active event instance within the window. This maintenance process greatly degrades the overall performance, rendering our solution ineffective. A more in-depth examination reveals that it is not necessary to store the above information for all event instances. The earliest time a sequence match becomes invalid would exactly be the time when the first event in this sequence (START instance) expires. Thus, we obtain an important observation that:

**Lemma 1** For a pattern query \( q_i = (E_1, E_2, ..., E_n) \), once an instance \( e_i \) with \( e_i.type \in E_i \) expires, all sequence instances matches of \( q_i \) that contain this \( e_i \) should correspondingly be expired.

Therefore, we now propose that recording the life span and sequence count information on each START instance is sufficient. The expiration of all other event type instances are shown to make no impact on the result update process. Thus, we can continue to discard them immediately as in the basic A-Seq approach. Thus, our core idea here is to compute the counts built on each START instance separately to avoid undue impact of expired START instances on output results.

**Start Event Marking (SEM) Solution.** Based on this idea, we now propose a Start Event Marking (SEM) technique. SEM marks each START instance with its lifespan, namely, the expiration time-stamp. The general process is composed of the following steps:

1. When a START instance arrives, a PreCntr is created for it to record the number of sequence matches formed using this START instance.

2. When a UPD instance arrives, the same update process as in our basic A-Seq approach is applied. However, this update is now applied to the prefix counters of all active START instances.
3. When a TRIG instance arrives, counts on all active prefix counters are summed together and output as aggregation result. Expired counters are ignored.

4. When the window slides, any expired START event with its corresponding PreCntr are removed. If output result is required, then the count on this PreCntr will be simply subtracted from the total count result.

This SEM computation process is illustrated in Figure 3.5. Figure 3.6 gives the pseudocode of the revised A-Seq algorithm with SEM.

Q: Pattern SEQ <A, B, C, D>

Within 7s

![Stream Example](image)

<table>
<thead>
<tr>
<th>Stream</th>
<th>a₁</th>
<th>b₁</th>
<th>c₁</th>
<th>a₂</th>
<th>c₂</th>
<th>b₂</th>
<th>d₁</th>
<th>c₃</th>
<th>a₃</th>
<th>d₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>1s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PC
Exp:8s
AB 1 1 1 1 2 2
ABC 0 1 1 2 2 2
ABCD 0 0 0 0 0 2

PC
Exp:11s
AB 0 1 1 1 1 1
ABC 0 0 0 1 1 1
ABCD 0 0 0 0 0 1

PC
Exp:15s
Ab 0
ABC 0
ABCD 0
Output: 2
Output: 1

Figure 3.5: Pushing Windows Down into A-Seq Using SEM

Example 4 In Figure 3.5, when a₁ arrives at time t = 1s, we first calculate its expiration timestamp (Exp), where Exp = arrTime + Q.win. Thus, a₁.Exp = 1s + 7s = 8s, that is, a₁ and its aggregation result will become invalid at t = 8s. Then we create a PreCntr for a₁. There is no count for prefix (A) in each PreCntr, as we have separated each instance of A to maintain its own counts. Thus, the count for (A) is always 1.
**SEM A-Seq**

- $e_{i,ts}$: arrival timestamp of $e_i$
- $PreCntr_{exp}$: expiration timestamp of a $PreCntr$ (a START instance $e_i$)
- $ts$: current system timestamp
- $Win$: query window size (time based)

1. agg = 0
2. for each arriving event in stream $S$
3. //step 1: Create $PreCntr$
4. if $e_{i,cat} = START$
5. create a $PreCntr$ of size $L_q-1$
6. mark $PreCntr_{exp} = e_{i,ts} + Win$
7. //step 2: Update Count
8. else
9. for each $PreCntr$
10. if $PreCntr_{exp} < ts$
11. apply update rule ii
12. //step 3: Sum aggregation result
13. if $e_{i,cat} = TRIG$
14. for each $PreCntr$
15. if $PreCntr_{exp} < ts$
16. agg = agg + $PreCntr$.count
17. else
18. remove this expired $PreCntr$
19. return agg

---

Figure 3.6: SEM A-Seq Algorithm

The update process of the basic A-Seq method is utilized when $b_1$ and $c_1$ arrive. When $a_2$ arrives at $t = 4s$, we now in addition create a $PreCntr$ for $a_2$ and mark it with its $Exp$. Subsequently, when $c_2$ and $b_2$ arrive, the count update takes place on both $a_1$'s $PreCntr$ and $a_2$'s $PreCntr$, as both $a_1$ and $a_2$ are active.

When $d_1$ arrive at $t = 7s$, first we update the count on both $PreCntrs$. Then, as instances of event type $D$ trigger the completion of the full sequence query, we compute the final the aggregation result by suming of the counts of pattern $(A, B, C, D)$ from all active $PreCntrs$. Thus, the output result is $2 = 2(a_1 PreCntr) + 0(a_2 PreCntr)$.

Next, $c_3$ arrives at time $t = 8s$. When we perform the update on each $PreCntr$, we
find that $a_1$ expires at 8s and $a_1$’s PreCntr should be purged. If users require a result at this moment, the output would be 0 instead of 2. Same steps as above are applied when $a_3$ and $d_2$ arrive. The output result becomes 1 after $d_2$ is processed.

3.3 Negation Support for A-Seq

Invalid Sequence Check Problem. Negation in CEP aggregation requires us to assert the non-occurrence of any instances of the negated event types at certain positions in a sequence pattern. For example, online retailers considering the web advertisement revenue might be interested in pattern (VK, BK, !REC, VC, BC) to track how many customers purchases a case after the purchase a kindle, yet without first clicking through the "Recommendation" link. The problem of negation is that, if negative event instances of "Rec" occur, they can cause previously detected prefix sequences ($vk_i, bk_i$) to become invalid. One way to solve this problem would be to add a negation filter on top of the query plan as typically done in the literature [4, 25] to discard all positive matched sequences ($vk_i, bk_i, vc_i, bc_i$) that have $rec_i$ between $bk_i$ and $vc_i$. An obvious problem with this later-filter-step solution is not only that it generates a potentially huge number of intermediate results, many of which may be filtered out eventually, but also that it is incompatible with our core A-Seq approach of time-sensitive aggregation. We now propose a solution that solves this problem by pushing this negation check into our A-Seq approach.

The Immediate Re-Counting Solution. We propose the Recounting Rule (RR) technique to tackle this challenge. We find an important observation here:

Lemma 2 For a pattern query with negation $q_i = (E_1, E_2, ..., !E_i, ..., E_n)$, when a negative event instance $e_i$ with $e_i.type \in E_i$ arrives, the count of the prefix pattern previous to $E_i$, namely, $(E_1, E_2, ..., E_{i-1})$ becomes invalid.

Since in our A-Seq solution, the count of the prefix pattern prior to this negative event
type is preserved, when a negative event instance arrives, we propose to simply reset the count of the previous prefix pattern to 0. This simple reset corresponds to an effective re-counting.

However, it should be noted that NOT all prefix patterns previous to the negative event type need to be reset. For example, for the negation sequence pattern \((A, B, !C, D)\), though prefix \((A, B)\) becomes invalid when a \(C\) instance arrives, matches of the prefix \((A)\) continue to remain valid and thus can continue to be connected with future \(B\) and \(D\) events to complete the full sequence detection. In general, as the counts of all prefix patterns are preserved using the \(PreCntr\) structure, when \(C\) instance arrives, only the count of \((A, B)\) must be cleared to 0, while the counts of all other prefixes, here \((A)\) and \((A, B, D)\) remain unchanged.

\[
\begin{array}{cccccc}
\text{Stream} & a_1 & b_1 & c_1 & a_2 & b_2 & d_1 \\
\hline
A & 1 & 1 & 1 & 2 & 2 & 2 \\
AB (!C) & 0 & 1 & 0 & 0 & 2 & 2 \\
ABD & 0 & 0 & 0 & 0 & 0 & 2 \\
\end{array}
\]

Output: 2

Q: 
Pattern SEQ (A,B,!D) 
Agg COUNT 
\(<a_1, b_1, d_1>\) 
\(<a_2, b_2, d_1>\) 
without constructing

Figure 3.7: Pushing Negation Down to Sequence Detection Process

**Example 5** Figure 3.7 illustrates how RR works through a sequence pattern \((A, B, !C, D)\). When \(c_1\) arrives at \(t = 3s\), the count of \((A, B)\) is cleared to 0 while the \((A)\) and \((A, B, D)\) counts are kept. The output result is 2 when \(d_1\) arrives, because the sequence \(<a_1, b_1, d_1>\) is not counted as \(c_1\) is between \(b_1\) and \(d_1\).
Several cases of special negation locations in a pattern, namely, at the start, at the second and at the end position of a pattern, require customized processing, as discussed next.

**Negation Start.** For query pattern \((!A, B, C, D)\), \(RR\) cannot be applied as there is no prefix pattern previous to \(A\). The challenge caused by this special negation start case is that, initially made invalid \((B, C, D)\) sequences following an \(A\) instance will eventually become valid again, once the \(A\) instance is expired. Thus, simply applying SEMA-Seq to detect the pattern \((B, C, D)\) is insufficient. To capture the negation start impact, a separate list recording the expiration timestamp of each \(A\) instance is maintained. When summing the counts on each \(PreCntr\) to output a result, a check of the \(A\) list is performed to ensure there is no \(A\) instances before this \(PreCntr\) that are still valid. For this, we compare the expiration timestamp with this \(PreCntr\). If not expired, we add 0 to the aggregation result for this \(PreCntr\) without modifying the \(PreCntr\) count.

**Negation End.** For query pattern \((A, B, C,!D)\), if aggregation result is output when each \(D\) instance arrives, the output is reset to 0 as the count of \((A, B, C)\) is reset to 0 according to the \(RR\). To avoid this ”always 0 output”, one method is when a \(D\) instance arrives, we output the \((A, B, C)\) count first and then reset it to 0. Another solution would be to change the output style. For example, the aggregation result is output when required by the user instead by the arrival of each triggering event instance. In this way, the \(RR\) would effectively handle this Negation End case.

**Negation Second.** For query pattern \((A, !B, C, D)\), the count of \((A)\) should be cleared to 0 when \(B\) instance arrives. However, since counting is separately processed for each \(A\) instance due to handle sliding window semantics. Since no counts are recorded for the \(A\) instances, thus, the existing \(PreCntr\) structure for each \(A\) should now be completely purged, instead of first cleared to 0.
3.4 Predicates Support for A-Seq

Below, we present solutions of how to push the predicate evaluation into our aggregation process.

Local Predicates. The most common predicates over event data are local predicates, which impose constraints on the attribute values of an event instance (e.g. Kindle.model = ”touch”). Predicates are evaluated on the relevant event instances before these instances are involved in the aggregation process. Event instances that do not satisfy the predicates are immediately discarded.

Equivalence Predicate Test. It is well-known that CEP queries often using equivalence predicates correlate events in a sequence pattern [25]. For example, in our online shopping habit tracking scenario, the clicks should be from the same customer. Similarly, in the stock market example, the pattern of price increases and decreases should be of the same stock. An equivalence test essentially partitions an event stream into several sub-streams, where events in the same partition have the same value for the attribute used in the equivalence test (i.e. equivalence attribute). Here, we propose a technique to dynamically partition the event stream during the sequence aggregation, henceforth called Hashed Prefix Counter (HPC).

The basic idea of HPC is that the aggregation process is applied separately to each equivalent partition. Prefix counters for a START instance are created upon their arrival and hashed into the corresponding partition based on the equivalence attribute value of this instance. Other event instances are similarly hashed to their corresponding partition. Aggregation results are computed based on each partition.

Pattern SEQ <A, B, C, D>

Where A.id = B.id = C.id = D.id

Within 7s
**Example 6** Figure 3.8 illustrates the HPC data structure to cope with the equivalence predicates processing in sample query $Q_2$. The equivalence test for $Q_2$ is on attribute $id$. The $id$ value of each incoming event will be evaluated. Assuming three distinct $id$ values (1, 2, 3), then we create three hash partitions as depicted in Figure 3.8 with $id$ values as key and the prefix counters as the value. For instance, the $id$ values of instances $a_1$ and $a_4$ are 1. Thus their prefix counters are created in the $ID_1$ partition. For instances of type $B, C, D$, they will update the prefix counters in the partition determined by their respective $id$ values.
Chapter 4

A-Seq Multi Query Strategy

For rich data streams from web-clicks to stock ticker streams, large workloads of similar queries may be processed. Since executing each query separately could lead to scalability and performance problems. Thus, we explore the opportunity to share the computation across a workload of CEP aggregation queries [15]. Consider the running example in Section 1.1.1, where merchants are interested in various purchase patterns to determine customer shopping habits, including 1:

\[
\begin{align*}
Q_1 &= (V\text{Kindle}, B\text{Kindle}, V\text{Case}, B\text{Case}) \\
Q_2 &= (V\text{Kindle}, B\text{Kindle}, V\text{KindleFire}) \\
Q_3 &= (V\text{Kindle}, B\text{Kindle}, V\text{Case}, B\text{Case}, V\text{eBook}, B\text{eBook}) \\
Q_4 &= (V\text{Kindle}, B\text{Kindle}, V\text{Case}, B\text{Case}, V\text{Light}, B\text{Light}) \\
Q_5 &= (V\text{iPad}, V\text{KindleFire}, V\text{Kindle}, B\text{Kindle})
\end{align*}
\]

Several common sub-queries (substrings) arise across these 5 pattern queries, such as \((V\text{Kindle}, B\text{Kindle})\). The intuition is that if we were to compute intermediate aggregates for the \((V\text{Kindle}, B\text{Kindle})\) substring, then would we be able to share this result among the other queries that contain this substring. This way, redundant aggregation

\(^1\)capital letter \(V\) represents \(View\), and \(B\) represents \(Buy\)
computation for common substrings would be avoided. In principle, the more queries share common substrings, the more computational resources could potentially be saved. This promise of scalability leads us to our proposed techniques below.

4.1 The Prefix Sharing Strategy

We observe that once the aggregation of a full sequence pattern is computed by $A$-Seq, then the aggregation of all its prefix patterns would also have been obtained as side-effect. Thus, the most straightforward way to achieve multi-query aggregation sharing is to share the prefix computations among queries with the same prefixes. Based on this observation, we now propose a *Prefix Tree (PreTree)* data structure in support of the *Prefix Sharing (PreShare)* strategy in the *Multi A-Seq* approach.

![PreTree Structure](image)

**Example 7** Figure 4.1 shows an example of the PreTree structure for $Q_1 \sim Q_4$, namely, these four prefix counters are organized into a PreTree structure. For computation of these four patterns, the result of substring $(v$Kindle, bKindle) is pipelined into both $Q_1$ and $Q_2$, while the $Q_1$ result is pipelined to both $Q_3$ and $Q_4$.

Compared to computing each query independently without sharing also using $A$-Seq solution, this process using PreTree is now slightly adapted as below to account for the
arrival of instances of each event category:

1. When a START instance arrives, a **Prefix Tree (PreTree)** will be created instead of a previous **PreCntr**. For example, in Figure 4.1, one single PreTreeCntr structure would be initialized, compared to the four normal PreCntrs that would have been created in the non-share approach.

2. When a UDP instance arrives, the count update occurs on the corresponding location in each PreTree only once. For example, when a bKindle instance arrives, one update on PreTree is needed in place of four updates on each of the four PreCntrs.

3. When a TRIG instance arrives, results for the respective query can be directly accessed from the appropriate locations within the PreTree.

This way, the redundant computation of common prefix patterns is avoided. As is apparent, very little overhead is introduced to maintain such a tree structure or to output results for this multi-query case. In fact, the more queries share the same prefix patterns, the more computation costs should be saved.

### 4.2 The Arbitrary Sub-pattern Sharing Mechanism: Chop-Connect

**Overall Idea of Chop-Connect Method.** In general, queries may feature common sub-expressions at random locations rather than only at their prefix positions. In the above example, $P_5$ shares $(V\text{Kindle}, B\text{Kindle})$ with the other four query patterns. Since we can obtain the count of $(V\text{Kindle}, B\text{Kindle})$ from the prefix tree that has been set up for $Q_1 \sim Q_4$, then if we could potentially compute $(V\text{iPad}, V\text{KindleFire})$ separately, then we could potentially connect the counts of these two substrings together. This way, the
computation of \((v\text{Kindle}, b\text{Kindle})\) would be shared by all five queries. This intuition leads us to propose the Chop-Connect (CC) idea for tackling this general problem of sharing the computation arbitrarily of common sub-expressions. The idea is to Chop a query into substrings, with each substring being computed separately. Finally, the results of all substrings are later Connected together to get the count of the whole sequence pattern.

**Analyzing the Connect Problem.** Analyzing the Connect problem, we notice that we cannot simply construct a simple Cartesian Product of two previously computed sub-counts to get the count of the longer connected sequence. Rather, the time order between instances of the two connect event types \(V\text{KindleFire}\) and \(V\text{Kindle}\) matters. Since the \(A\text{-Seq}\) approach does not maintain any time information (except START for expiration), additional ordering knowledge between connect events must be recorded correctly to enable the full sequence count, namely, TRIG from the first substring and START from the second substring.

One straightforward solution to solve this problem is to maintain the timestamps of all connect event instances during the computation process, and join the counts of two substrings by comparing the timestamps of all connect event instances. For ease of exposition, let us now use \(\text{sub}_1 = (A, B, C)\) and \(\text{sub}_2 = (D, E)\) to denote the two substrings that need to be connected, then \(C\) and \(D\) are the connect event types. As in the single \(A\text{-Seq}\) approach, the expiration timestamps of all START \((D)\) instances in \(\text{sub}_2\) are recorded, we now also need to maintain the timestamps of each \(C\) instance in \(\text{sub}_1\).

**Proposed Solution.** However, after careful examination, we find that it is not necessary to store the timestamps of all \(C\) instances. Instead, it is sufficient to only record the \(C\) instance that is the most recent before the arrival of a \(D\) instance.

**Lemma 3** Any other instance that arrives before this most recent \(C\) instance is guaranteed to be valid to connect with this \(D\) instance.

30
For example, if the arrival order of $C$ and $D$ is: $< c_1, c_2, c_3, d_1, ... >$, then $c_3$ is the most recent $C$ instance for $d_1$. Clearly, $C$ instances before $c_3$ can safely be connected with $d_1$ while those after $c_3$ cannot. To conclude, the key principle in our aggregation context is to obtain meta-data about how many $sub_1$ matches have been detected before each START in $sub_2$ arrives. Example in Figure 4.2 illustrates how our above strategy works.

**Example 8** Figure 4.2 illustrates how to connect the counts of $sub_1 = (A, B, C)$ and $sub_2 = (D, E)$ to get the count of $(A, B, C, D, E)$. First, we compute two substrings using our single A-Seq separately. However, to prepare for a later “connect” computation, when $d_1$ arrives at time $t = 6s$, besides creating the $PreCntr$ for $(D, E)$, we also extract the count of $sub_1$ at that moment ($\#(A, B, C) = 1$) and attach this count to $d_1$’s $PreCntr$ for later use. Similarly, when $d_2$ arrives, we attach $\#(A, B, C) = 4$ to $d_2$’s $PreCntr$.

When TRIG instances arrive and trigger the output of the aggregate result for the
full sequence pattern, a multiplication of (1) the \#(sub\_2) and (2) the attached \#(sub\_1) on each PreCntr of D instance is performed. For example, when e\_1 arrives, there is 1 sequence match of (D, E) formed on d\_1 (\#(D, E) = 1), and 1 match of (A, B, C) formed before this d\_1 (attached \#(A, B, C) = 1). Thus, \#(A, B, C, d\_1, E) = 1 * 1 = 1. Similarly, for d\_2’s PreCntr, \#(A, B, C, d\_2, E) = 4 * 1 = 4.

Lastly, we sum the results of all active PreCntr to derive the final result, that is, when e\_1 arrives, the total count \#(A, B, C, D, E) = 1 + 4 = 5.

**Challenge Caused by Expiration.** However, the above connect solution assumes that events never expire. Suppose \#(sub\_1) is attached to a START instance of sub\_2 when that START instance arrives at time t\_i. However, when we use this \#(sub\_1) in the multiplication as a TRIG instance of sub\_2 arrives at time t\_{i+1}, this attached \#(sub\_1) might already have become invalid at that time. As \#(sub\_1) is calculated separately from sub\_2, \#(sub\_1) might become invalid during at the t\_{i+1} time, due to the START instance expiration of sub\_1. (See Section 3.2 for the event expiration details). To illustrate this problem, consider example 8. When e\_1 arrives at time t = 9s, \#(A, B, C) should no longer be 4 if a\_1 had expired before t = 9s. Thus, the attached count would have intermittently become incorrect, no longer accurately reflecting the most recent count upon the arrival of TRIG instances. This causes erroneous aggregation.

**SnapShot Solution.** We now propose a solution to this connect expiration problem. When a START instance of sub\_2 arrives, instead of attaching the current sub\_1 count, we record sub\_1 counts on each sub\_1’s PreCntr separately with their expiration timestamps. Therefore, when a TRIG instance arrives, the respective counts expired by its arrival can be safely discarded.

Lastly, we design the (SnapShot(SS)) data structure to represent counts on all sub\_1’s PreCntr whenever a sub\_2 START instance arrives. The snapshot is organized as a table. Each row represents the count snapshot of a sub\_1’s PreCntr. The columns contain meta-
data: 1) PreCntr tag, indicating the START instance to which this PreCntr belongs to, 2) the corresponding expiration time of this PreCntr, and 3) the count of this PreCntr.

Figure 4.3 illustrates the snapshot maintenance of sub₁ (A, B, C) for sub₂ (D, E). Compared to the stream in Figure 4.2, we now add the extra instance a₃ before d₂ to show the change in the snapshots.

![Diagram of snapshot maintenance](image)

**Example 9** In Figure 4.3, when d₁ arrives, a snapshot (SS) of d₁ with counts and expiration timestamps of all sub₁ prefix counters is created, and attached to d₁. When d₂ arrives, similarly, another snapshot is created to record the prefix counter status of sub₁ at that moment and attached to d₂. When e₁ arrives at t = 10s, after the update aggregation process applied to the (D, E) PreCntrs, the SS expiration timestamp check is
applied to each table row of sub2 PreCntr. Expired rows are discarded, and thus won’t be involved in any future aggregations. For example, when e1 arrives at t = 10s, the SS check finds that a1 expires at that time. Thus, only counts of PreCntrs of a2 and a3 will be used. The total count #(A,B,C,D,E) at t = 10s thus is calculated as:

\[
\begin{align*}
\text{Count}_1 &= \#(d_1 \text{ PreCntr}) = #(A,B,C,d_1,E) = 1 \times 0 = 0, \\
\text{Count}_2 &= \#(d_2 \text{ PreCntr}) = #(A,B,C,d_2,E) = 1 \times 0 + 1 \times 1 = 1 \\
#(A,B,C,D,E) &= \text{Count}_1 + \text{Count}_2 = 0 + 1 = 1
\end{align*}
\]

For above we now introduce a new event category, called **Connect Event Type (CNET)** (besides START, UPD and CNET). CNET is the START event type in sub2 when connecting sub1 and sub2. The arrival of a CNET instance will trigger the process of checking counts on sub1’s PreCntrs and creating the sub1 snapshot for sub2. For example, event type D falls into the category CNET. Meaning when connecting (A, B, C) and (D, E), a snapshot of (A, B, C) count is created for (D, E) whenever a new D d_i instance arrives.

**Multi-Connect Process.** A query might be chopped into multi pieces rather than just only 2. For example, pattern query (A, B, C, D, E, F, G) might be chopped into: sub1 = (A, B, C), sub2 = (D, E), and sub3 = (F, G), as sub2 and sub3 are shared by other queries, respectively. In this case, when the CNET instance F in sub3 arrives, it triggers the creation of a snapshot of (A, B, C, D, E) and then attaches to this snapshot to sub3 PreCntr. Due to the expiration problem, we postpone summing the counts on each (D, E). Instead, we calculate the snapshot counts of (A, B, C, D, E) for each A instance, that is, counts of (a_i, B, C, D, E) for all active a_i. When F instance arrives, these counts can be calculated in the following two steps:

1. Calculate count of \((a_i, B, C, D, E)\) on each \((D, E)\) PreCntr. For each \((D, E)\) PreCntr, to which the snapshot of \((A, B, C)\) is attached to, we multiply the count.
in the snapshot with the count on \((D, E)\) on this PreCntr one by one. That is, we calculate the counts for \((a_i, B, C, d_j, E)\).

2. Then we calculate count of \((a_i, B, C, D, E)\) on all \((D, E)\) PreCntrs. Given the counts calculated by step 1, we apply a sum over those snapshot counts with the same \(a_i\) tag across all \((D, E)\) PreCntrs.

**Example 10** Figure 4.4 illustrates the multi-connects calculation process of the above example. When \(f_1\) arrives at \(t = 12s\), we first multiply the count on each \((D, E)\) PreCntr with its corresponding snapshot counts of \((A, B, C)\). Then, we plus the count 1 with tag \(a_2\) on \(d_1\)'s PreCntr to the count 2 with tag \(a_2\) on \(d_2\)'s PreCntr. Similarly, we add the counts with same tag across all \((D, E)\) PreCntrs. Then we attach these counts to the \(f_1\) PreCntr representing the snapshot of \((A, B, C, D, E)\).
4.3 Sharing Plan Selection

In the above, we have introduced a solution for tackling the Connect challenge assuming we are given the substrings into which a given set of pattern queries is to be decomposed. However, since some computation and storage overhead is introduced by each connect operation, we now need an effective strategy for chopping the workload into sharable sub-strings. Thus, we design a global sharing plan and call it "Chop Strategy". Given this "Chop Strategy", we chop each query in the workload into smaller substrings, such that the computation of those common substrings is shared across queries in this workload. Our goal is to find a solution with the "minimal" overall computation costs.

**Multi A-Seq Optimization Problem:** Given a query workload $WL$, find a shared execution plan for $WL$ that indicates which substrings are shared by which queries (if any), with the lowest overall execution costs.

One straightforward optimal solution for this problem is exhaustive search, which is composed of 3 steps: 1) find all potential sharing substring candidates, which satisfy two conditions: substring length $l \geq 2$, and shared by $k \geq 2$ queries; 2) for each candidate (sharing substring), calculate its Benefit Value ($BValue$), namely, the execution costs gained when sharing this candidate compared with not sharing. 3) compose all candidate combinations (subsets of all candidates), and find the combination with the largest $BValue$. This maximal benefit candidate combination is our final shared execution plan.

**Complexity Analysis.** Several practical considerations make this solution challenging. Suppose there are $n$ queries in this $WL$, and the length of each query is $m$. For step 1), we can simply run a search over all queries to find all substrings of length from 2 to $m - 1$, and then filter those not shared by at least two queries. However, too many unqualified substrings ($k < 2$) will be generated, especially for larger $n$ and $m$. Thus,
a smarter solution is required. For step 2), a cost model that can correctly measure the computation gain (sharing versus non-sharing) of employing a particular substring candidate must be developed. For step 3), the time complexity of generating all subsets of the given candidates set is exponential. An exhaustive search for the optimal combination will thus need to traverse too huge of a search space for a large query workload. Thus it is in general not practical to find the optimal solution. Thus, we now propose the following solutions to cope with above three challenges.

4.3.1 Apriori for finding all potential sharing candidates

Given the similarity between this sharing candidate generation problem and the frequent itemset identification problem in association rule mining, we now deploy the Apriori algorithm [2] to solve our problem. Apriori is a candidate generation and test approach. The key concept of Apriori is its pruning principle: if there is any itemset which is infrequent, then its superset is guaranteed to also be infrequent. When applying the Apriori algorithm here, the pruning principle for our problem can be stated as: if a substring \((E_1, E_2, ..., E_{i-1})\) of length \(i - 1\) is infrequent (not shared by at least 2 queries), then any substring \((E_1, E_2, ..., E_{i-1}, E_i)\) of length \(i\) would not be frequent.

Thus, we now design the following potential candidates generation algorithm:

1. Initially, scan all queries once to get all frequent substring candidates \((k \geq 2)\) with \(l = 2\).

2. Repeatedly generate candidates with length \(l = i\) from frequent candidates with \(l = i - 1\) \((i\) starts from 2\).

3. Terminate when no new frequent candidates can be generated any more.

This way, a large number of unqualified substring candidates would be pruned right
away upon their generation. This greatly reduces the candidates size and thus saves both memory and CPU processing resources.

### 4.3.2 Cost-Based Benefit Model

Next, we establish a model to calculate the CPU costs of both sharing a substring candidate $X$ (CC method), versus not sharing it (NonShare method). Then the BValue of $X$ is obtained by: $X_{BValue} = X_{NonShare} - X_{CC}$. Now we analyze the estimated costs of processing a common substring $X$ by either of two computation approaches. To make the cost model easy to understand, we assume our stream is uniformly distributed, with the data frequency of each event type in this stream the same. Namely, each event type has the same number of instances. We estimate the cost of processing of all instances in a sample stream. We divide event types into four categories $START, UPD, CNET, TRIG$ based on the different operations they trigger upon their arrival (See section 3.1 and 4.2). Generally, we estimate the cost of all instances in each category, and then sum them together as the total cost. The total cost of a single query can thus be formulated as:

$$Cost_{total} = \sum_{i=1}^{4} N_{cat_i} \times Cost_{cat_i} \times Num_{cat_i}$$

(4.1)

Table 4.1 represents the parameters and terminology used in the cost estimation. Take the computation cost for pattern query $(A, B, C, D)$ as an example. There is one START, two UPD and one TRIG event types in this query. Then based on the Equation 4.1, the total cost of $(A, B, C, D)$ in a uniformly distributed stream is: $Cost_{(A,B,C,D)} = C_{START} \times N_{START} \times Num_{START} + C_{UPD} \times N_{UPD} \times Num_{UPD} + C_{TRIG} \times N_{TRIG} \times Num_{TRIG} = C_{START} \times N_0 \times 1 + C_{UPD} \times N_0 \times 2 + C_{TRIG} \times N_0 \times 1$. 

38
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_0$</td>
<td>The instances number of each event type in a uniformly distributed stream (except the CNET instances number)</td>
</tr>
<tr>
<td>$f$</td>
<td>the occurrence frequency of CNET instances, that is, $(# \text{ of CNET})/N_0$</td>
</tr>
<tr>
<td>$fN_0$</td>
<td>the number of CNET instances in stream</td>
</tr>
<tr>
<td>$l$</td>
<td>the length of a substring $X$</td>
</tr>
<tr>
<td>$k$</td>
<td>the number of queries that share $X$</td>
</tr>
<tr>
<td>$N_{cat_i}$</td>
<td>The number of instances of event category $i$</td>
</tr>
<tr>
<td>$Num_{cat_i}$</td>
<td>The number of event types of category $i$ in a pattern</td>
</tr>
<tr>
<td>$C_{START}$</td>
<td>The cost of a START instance: create a $PreCntr$, time complexity $O(C)$</td>
</tr>
<tr>
<td>$C_{UPD}$</td>
<td>The cost of a UPD instance: update count of all active $PreCntr$s, at most $N_0$ in Naive and $fN_0$ in CC. Time complexity $O(n)$</td>
</tr>
<tr>
<td>$C_{TRIG}$</td>
<td>The cost of a TRIG instance: update and sum count of all active $PreCntr$s, worst case $N_0$, time complexity $O(n)$</td>
</tr>
<tr>
<td>$C_{CNET}$</td>
<td>The cost of a CNET instance: compute and create snapshot, worst case $N_0^2$, time complexity $O(n^2)$</td>
</tr>
</tbody>
</table>

**NonShare Method.** The cost of computing a common substring $X$ without sharing (duplicate in $k$ queries). Then, the total cost is simply compute $X$ $k$ times. We can utilize Equation 4.1 to get the costs of computing $X$ once. To simplify the cost model but without underestimation, no matter what event category $X$ has, we assume that $C_{START} = C_{TRIG} = C_{CNET} = C_{UPD} = N_0$ operation cost. Thus, for computing $X$ once, the cost is: $X_{once} = N_0 * fN_0 * 1 + N_0 * N_0 * (l - 1)$. The cost is composed of two parts: (1) cost of all CNET instances, and (2) cost of all other instances (as the number of CNET is different with other events based on our assumption). The first $N_0$ in both parts represents the operation costs of an individual event instance (no matter what event category). We simplify $(l - 1)$ as $l$, then, the total cost of computing $X$ in $k$ queries is:
\[ X_{Naive} = (N_0 \ast fN_0 \ast 1 + N_0 \ast N_0 \ast l) \ast k \]
\[ = N_0^2 \ast k \ast (l + f) \]  

**ChopConnect Method** As we can see from the operation analysis for CC in Section 4.2, the overhead costs of CC mainly come from the connect operation, namely, the snapshot computation and creation operations when the CNET arrives. This snapshot computation cost for a multi-connect operation includes two parts: 1) the multiplication of count of each \( sub_2 \text{PreCntr} \) with its attached \( sub_1 \) snapshot count, and 2) the summing of the counts across all \( sub_2 \text{PreCntr}s \). There are at most \( N_0 \) records in the \( sub_1 \) snapshot, also at most \( N_0 \) \( sub_2 \text{PreCntr}s \). Thus, the worst case cost for a multi-connect snapshot computation is \( N_0^2 \).

Based on the above cost analysis, we conclude that the data frequency of CNET instances dominantly determine the CC costs, as the costs of other event categories is at most \( N_0 \). Besides, the CNET corresponds to the START in \( X \). Thus, the fewer CNET instances, the fewer \( PreContrs \) the UPD instances need to update. Namely, at most \( fN_0 \) \( PreContrs \) will be created, and thus the \( C_{UPD} \) should be at most \( fN_0 \) in this case. To measure the effect of the data frequency of CNET, we set a parameter \( f \) in our Benefit Model. Theoretically, the lower \( f \), the lower the CC costs.

In the CC method, we only need to compute substring \( X \) once. However, we need to connect it thereafter with other substrings in the \( k \) queries that share it. To simplify the cost model but without underestimating the costs, here we make two assumptions: 1) share \( X \) in one query will cause one *chop* before this substring, thus there are total \( k \) connects, and 2) we use the multi-connect costs (\( N_0^2 \)) for all connects, no matter it is single or multi connect. Therefore, except the CNET costs that need to be calculated \( k \) times, the costs of other event categories (only \( UPD \) if share \( X \)) would be calculated only
once. Given the above two assumptions, we conclude the computation costs of processing $X$ by the CC method as in Equation 4.3:

$$X_{CC} = C_{CNET} \times k \times fN_0 + C_{UPD} \times (l - 1) \times N_0$$

$$\approx N_0^2 \times k \times fN_0 + fN_0 \times l \times N_0$$

$$= N_0^3 \times f \times k + N_0^2 \times f \times l$$

**Benefit Model.** Now we calculate the Benefit Model together with using the $X_{CC}$ and $X_{Naive}$ costs from Equation 4.3 and 4.2 respectively.

$$X_{BV} = X_{Naive} - X_{CC}$$

$$= N_0^2 \times k \times (l + f) - N_0^3 \times f \times k - N_0^2 \times f \times l$$

From Equation 4.4, three parameters together impact the computation gains of sharing a candidate substring. To observe the effect of each parameter independently, we transform the Benefit Model by treating one parameter as variable while the other two are being set to some constant. This way we obtain the following three linear equations:

$$X_{BV} = \begin{cases} (N_0^2k - N_0^3k - N_0^2l) \times f + N_0^2kl, & f \text{ as variable} \quad (a) \\ (N_0^2k - N_0^2f) \times l + (N_0^2 - N_0^3)kf, & l \text{ as variable} \quad (b) \\ (N_0^2l + N_0^2f - N_0^3f) \times k - N_0^2f l, & k \text{ as variable} \quad (c) \end{cases}$$

In the Equation 4.5(a), we see that $(N_0^2k - N_0^3k - N_0^2l) < 0$ as $N_0^2 < N_0^3$ for $N_0 > 1$. Thus, the smaller $f$, the larger the $BValue$. Moreover, since $N_0^2$ is usually a large number, only a light change in $f$ will cause significant change in $BValue$. This cost
analysis corresponds to our intuition that $f$ is the major factor that affect the sharing cost.

In the Equation 4.5(b), we can see that $(N_0^2k - N_0^2f) > 0$ as $f < 1$ while $k >= 2$. Thus, the larger $l$, the larger the $BValue$. This indicates that the longer a substring we share, the more we can gain.

Lastly, in the Equation 4.5(c), we see that $N_0^2l + N_0^2f - N_0^3f$ could be either greater or smaller than 0. This corresponds to our intuition that it is hard to decide whether we can benefit more if more queries share a common substring. As the larger $k$, the more chops and connects are introduced. Though we can eliminate more duplicate computation, but higher $CONNECT$ costs overhead will also arise.

4.3.3 Mapping to the Maximum Independent Set Problem

Given the set of potential sharing candidates and the Benefit model for assessing the quality of a chosen candidate substring, we are now ready to calculate the $BValue$ of all candidates. The next step is to generate all possible combinations (subset) of the potential candidates, calculate the total $BValue$ of each combination, until we select the subset with the largest $BValue$ as the final sharing plan.

**Pruning using Heuristic: Sharing Compatibility.** Next, we introduce some criteria for pruning those impossible candidates combinations with the sharing incompatibility problem. For example, suppose there two candidates: $(A, B, C)$ is shared by $q_1, q_2,$ and $q_4$, $(B, C, D)$ is shared by $q_1, q_2,$ and $q_5$, in the same combination. We note that these two candidates have overlapped substring $(B, C)$. Once we decide to share $(A, B, C)$ in $q_1, q_2,$ and $q_4$, we cannot share $(B, C, D)$ in $q_1, q_2$ any more, as it will introduce another chop between $A$ and $B$, which make the substring sharing impossible. Thus, we cannot choose this combination as our sharing plan. Instead, if $(B, C, D)$ were to be only shared by $q_3$ and $q_5$, namely, these two candidates are not shared by the same queries, then they could be shared at the same time. The notion of **Sharing Conflict** thus is defined as: given
two substring candidates \( can_1 \) and \( can_2 \), if there is a pattern overlap among them, then we check whether there is an overlap in their respective shared query group. If yes, \( can_1 \) and \( can_2 \) are said to be conflict.

Therefore, combinations that have this sharing conflict issue are discarded. Since \textit{Apriori} approach generates candidate substring from shorter length to longer length within the same query pattern, the sharing conflict can be prevalent among candidates. Consequently, generating all combinations of all candidates produces too many combinations with conflict and that subsequently would be discarded, wasting CPU and memory resources. Given the above notion of the sharing conflicts exist in members in a combination, and the Benefit Model of each candidate, we now illustrate that our “optimal sharing plan search” problem can be mapped to a well known graph problem, namely, finding the Maximum Independent Set.

\textbf{Definition 2 Mapping Theorem.} Given the set of all potential sharing candidates, we define an undirected graph \( G = (V, E) \), where \( v_i \) denotes a candidate, and an edge \( e(v_i, v_j) \) denotes sharing conflict between \( v_i \) and \( v_j \) exists. Now, our aim is to find an \textbf{maximum independent vertex set} \( V_i \), among all possible \( V_i \subset V \), where no vertices in \( V_i \) are connected (no sharing conflict), with the largest overall \textit{BValue} \( \max_{V_i}(\sum_{v_i \in V_i} B\textit{Value}(v_i)) \).

\textbf{Example 11} Figure 4.5 illustrates an graph example of a maximum independent set problem. There are 5 sharing candidates and the table represents the queries that share them and their corresponding \textit{BValues} (Figure 4.5(a)). In Figure 4.5(b), we construct the graph based on the Mapping Theorem, where candidates with sharing conflict are connected. The \textbf{maximum independent set of this candidate set is} \( \{(A, B, C), (C, D, Q), (E, F)\} \) (vertices in gray).

The maximum independent set is known to be a N-P hard problem [10], meaning that the optimal solution to our problem an thus not be achieved in polynomial time com-
<table>
<thead>
<tr>
<th>Candidate</th>
<th>Queries</th>
<th>BValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B,C,D)</td>
<td>2,5</td>
<td>0.57</td>
</tr>
<tr>
<td>(C,D,Q)</td>
<td>2,5,6</td>
<td>0.121</td>
</tr>
<tr>
<td>(C,D,E,F)</td>
<td>3,4</td>
<td>0.4</td>
</tr>
<tr>
<td>(A,B,C)</td>
<td>1,3,4</td>
<td>0.9</td>
</tr>
<tr>
<td>(E,F)</td>
<td>3,4</td>
<td>0.33</td>
</tr>
</tbody>
</table>

Figure 4.5: Mapping to Maximum Independent Set Problem

plexity. Usually, for a large query group with many similar queries, a huge number of potential candidates could be generated, which leads to a huge search space. Even if we simply remove the candidates with negative BValue\(^2\) to reduce the space, the number of positive candidates might still be significant. Given this scalability consideration, we now design an efficient algorithm to tackle this problem using greedy search strategy. The well-known hill-climbing search approach is applied here to find the local optimal solution by incrementally adding candidate with larger BValue each time. The pseudo code of our greedy search algorithm is illustrated in Figure 4.6.

**Complexity Analysis.** This algorithm firstly sort all the candidates by their BValue. Suppose there are \(n\) candidates, any comparison-based sorting algorithm can finish in

\(^2\) would not affect the optimality, as the total BValue is a linear sum of each single BValue, negative BValues will only make the total BValue smaller.
\(X_i\): an candidate substring  
\(S_i\): an candidate substring in final solution set  
\(X_i_{\text{BV}}\): the BValue of candidate \(X_i\)

1. remove candidates with \(X_i_{\text{BV}} < 0\)  
2. sort all positive candidates by \(X_i_{\text{BV}}\)  
3. from largest to smallest into a CheckList  
4. create a empty final solution set FinalSet  
5. for (\(X_i\): CheckList)  
6. for (\(S_i\): FinalSet)  
7. check ifConflict \((X_i, S_i)\);  
8. if ifConflict is false  
9. put \(X_i\) to FinalSet;  
10. else  
11. discard \(X_i\);

Figure 4.6: A-Seq Optimizer Greedy Search Algorithm

\(O(n\log n)\) time. The time complexity of compatibility check with the candidates in FinalSet depends on the size of the FinalSet, which is at most \(n\). Thus, the upper bound complexity of compatibility check for \(n\) candidates is \(O(n^2)\), which however is an upper bound that never would be achieved. Besides, the FinalSet would not be too large in practical due to the sharing conflict pruning method as discussed above. Thus the worst case of our greedy algorithm would not exceed \(O(n^2)\). The performance and accuracy of this greedy algorithm will be evaluated in our experimental study in Section 5.3.
Chapter 5

Performance Evaluation

5.1 Experimental Setup

We describe our methodology for evaluating the performance of both single and multi A-Seq approach. Single A-Seq Evaluation. We compare our proposed A-Seq approach that pushes aggregation into sequence detection (see Chapter 3), with the state-of-art post aggregation approach (using the popular stack-based join method for sequence detection and construction) [15, 25]. We examine the performance of both method by varying query key parameters, namely, the pattern length \( (l) \) and the window sizes \( (w) \). Besides, a stress test for our single A-Seq solution is conducted to show the scalability. We also evaluate the A-Seq performance for processing negation queries.

Multi A-Seq Evaluation. First, we evaluate the performance and sensitivity of our multi A-Seq technique (Chop-Connect, CC) by independently varying each of the three key parameters identified by our cost analysis (see Section 4.3.2) in the Benefit Model, namely, connect event frequency \( (f) \), shared substring length \( (l) \), and sharing cardinality queries \( (k) \). We study the changing trend of CC performance through comparing the average processing time (ms/event) with non-share computation method, namely, computing
each query separately without sharing.

Second, we compare our Multi A-Seq optimizer with the optimal search method (Exhaustive) \(^1\) on 12 different multi-query groups. Each method returns a sharing plan for a given query group. We verify how often our efficient optimizer returns the same sharing plan as the optimal method (evaluate the effectiveness of A-Seq). We also compare the optimization search time of these two methods, illustrating the efficiency of our Multi A-Seq optimizer.

**Experiment Environment.** We implement our proposed A-Seq inside the \(X^5\) stream management system (cite) using Java. We run the experiments on Intel Core 2 Duo CPU 2.4 GHz WITH 4 GB RAM.

**Evaluation Matrix.** Performance is measured by the average processing time for each incoming event instance, namely, \(\text{Processing Time} = \frac{T_{\text{elapsed}}}{|\text{Events}|}\), where \(|\text{Events}|\) represents the number of tuples that arrive to our system and \(T_{\text{elapsed}}\) represents the total elapsed time to process all tuples in our sample stream so far. Peak memory usage is calculated by measuring the number of created Java objects. Namely, for the state-of-art approach, we count the sum of the three types of objects: active events inserted in the stacks, pointers, as well as intermediate sequence matches for later aggregation. For our A-Seq approach, we count the number of active prefix counters (\(\text{PreCntr}\)) created, in which all the information for aggregation computation is stored. The maximum object counts for these two approaches are reported as peak memory usage.

**Data Sets.** We evaluate our single query techniques using real stock trades data from [8], which contained stock ticker, timestamp and price information. The portion of the trace we use contains 10,000 event instances. For evaluating the multi-query techniques, to make longer queries and larger query workload, we also generate synthetic stock

\(^1\) Since the query group is large, the exhaustive search causes seems infinite execution time, we apply a heuristic to the optimal solution, yet won’t affect the optimal result. That is, remove candidates with negative \(BV\) value before generating all combinations.
streams with more event types and more instances introduced. There are 20 stock types and around 10,000 event instances in our synthetic streams respectively. Each stream has different distributions as explained in later experiment.

5.2 Single Query Evaluation

Experiment 1 - Varying pattern length $l$. In this experiment, we evaluate each approach’s sensitivity to the pattern length $l$ by varying $l$ from 2 to 5, while the window size is fixed as $100\text{ms}$. The average processing time per event and peak memory usage for these two approaches are reported in Figures 5.1(a) and 5.1(b). The pattern queries are as below, with event types the ticker name of each stock.

$q_1 = (\text{DELL, IPIX})$
$q_2 = (\text{DELL, IPIX, QQQ})$
$q_3 = (\text{DELL, IPIX, QQQ, AMAT})$
$q_4 = (\text{DELL, IPIX, QQQ, AMAT, MSFT})$

Figure 5.1(a) shows the average processing time of both approaches using logarithmic scale on Y-axis. As can be seen, A-Seq outperforms state-of-art by several orders of magnitude at all lengths. When $l = 2$, the processing time of A-Seq is as low as $0.0012\text{ms/event}$. As the pattern length increases, the performance of state-of-art approach seriously degrades due to the exponentially time complexity of sequence construction; while the performance of A-Seq only slightly decreases. At length 5, A-Seq is almost 16736-fold faster than the state-of-art method. This can be explained by the fact that A-Seq avoids the expensive sequence construction process, resulting in remarkably scalability.
Figure 5.1(b) shows the peak memory usage of the two approaches for varying pattern lengths. Y-axis is used again in a logarithmic scale. As can be seen, A-Seq again wins significantly in terms of memory utilization, because the only objects created by the A-Seq solution are prefix counters, whose cardinality is equivalent to all START instances in the stream. On the other hand, the state-of-art approach must store all relevant event instances along with the pointers indicating events order, and the intermediate sequence result matches. As the pattern length increases, the PreCntr number remain stable for a given input stream, because the maximal number of active START instances is fixed. Meanwhile, the peak memory of the state-of-art increases dramatically, as more events are inserted into stacks as the pattern length grows. More importantly, a huge number of intermediate results are being generated and must be stored for the later aggregation computation.

![Figure 5.1](image.png)

Figure 5.1: (a)(b) Single A-Seq Performance by Varying Pattern Length

**Experiment 2 - Varying window sizes** $w$. In this experiment, we evaluate each approach’s sensitivity to the window size $w$ by varying $w$ from 100ms to 500ms. We fix the query length at 3 (as $q_2$ in Experiment 1). We again measure both the average processing time and peak memory of each approach. As depicted in Figure 5.2(a), the performance
of the state-of-art approach degrades much faster than that of the A-Seq approach. This is because the sequence construction cost in the state-of-art solution is much more complicated than the operation costs in A-Seq, including $PreCntr$ initialization, count updating and summation. Thus, as more events are involved in the sequence construction process due to the window size increasing, the state-of-art loses more. Similarly as in Experiment 1, the A-Seq approach also shows high superiority in memory usage with increasing
window sizes (see Figure 5.2(b)).

**Experiment 3 - Scalability/Stress Test for A-Seq.** As demonstrated above, the performance of the state-of-art approach degrades dramatically with the growth in either the pattern length or the window size. Now we examine the scalability of A-Seq under stress tests in which the traditional stack-based join method fails in our system (i.e. memory overflow). Namely, we set the pattern length from 6 to 10, and the window size is fixed at 2000ms instead of 500ms earlier. The pattern queries are as below:

- \( q_5 = (DELL, IPIX, QQQ, AMAT, MSFT, CSCO) \)
- \( q_6 = (DELL, IPIX, QQQ, AMAT, MSFT, CSCO, INTC, ORCL) \)
- \( q_7 = (DELL, IPIX, QQQ, AMAT, MSFT, CSCO, INTC, ORCL, YHOO, RIMM) \)

Figure 5.3(a) illustrates the stress test result of A-Seq in terms of the average processing time. No significant performance degradation of A-Seq, even at the most extreme case (at length=10 and window=2000). As can be observed that the performance (0.0219 ms/event) is almost the same as that of state-of-art performance at length=2 and window=100 (0.02 ms/event).

**Experiment 4 - Negation performance for A-Seq.** In this experiment, we study A-Seq’s performance for processing queries with negation. We compare the performance of A-Seq (negation pushed down) approach with the state-of-art (post-filtering negation check) approach, by processing the queries \( q_8 \) and \( q_9 \) below:

- \( q_8 = (DELL, IPIX, AMAT) \)
- \( q_9 = (DELL, IPIX, !QQQ, AMAT) \)

\( q_8 \) and \( q_9 \) have the same positive pattern \((DELL, IPIX, AMAT)\) while \( q_9 \) has the additional negative event type \( QQQ \) filter inserted. When processing \( q_9 \), the state-of-art approach will first collect all matches of the positive pattern \((DELL, IPIX, AMAT)\), and then filter out those matches that contain \( QQQ \) instances between \( IPIX \) and \( AMAT \) instances.
For A-Seq, as the negation checked is pushed down, the sequence detection will terminate the matching of an instance once a negative event instance arrives. Figure 5.3(b) depicts the average processing time results for these two methods. We can observe that the A-Seq approach, experiences almost no overhead for processing the negation query, while the state-of-art approach suffers from significant overhead introduced by the post-filtering negation check.

5.3 Multi Query Evaluation

Experiment 1 - Evaluation of Chop-Connect Performance. In this experiment, we evaluate the sensitivity of CC method with respect to its cost model key parameters. This experiment is run on a given query group with the fixed sharing plan below (with window fixed at 1000 ms for all queries):

WorkLoad1 (WL1):

q1=(DELL, AMAT, YHOO, AMAZ, MSFT, ORCL, RIMM, CSCO, INTC)
q2=(IPX, FB, YHOO, AMAZ, MSFT, ORCL)
q3=(QQQ, IBM, MSFT, ORCL, RIMM, CSCO, INT)
q4=(VMW, GOOG, YHOO, AMAZ, MSFT, ORCL, RIMM, CSCO, INTC)
q5=(LNKD, NTAP, YHOO, AMAZ, MSFT, ORCL, RIMM, CSCO, INTC)

There are 5 queries in WL1. We fix the sharing plan as: q1, q4 and q5 share sub1 = (YHOO, AMAZ, MSFT, ORCL, RIMM, CSCO, INTC). To observe the effect of each parameter independently, each time we vary only one parameter of sub1 while the other two are fixed.

Varying the Connect Event Frequency \( f \). The connect event type in sub1 is YHOO. We first vary the instance occurrence frequency of YHOO in the stock stream at 0.1%,
2.5%, 5%, 7.5% and 10%, while the other is uniformly distributed event types. The length of \( sub_1 \) is 7 \( (l = 7) \) and it is shared by 3 queries \( (k = 3) \). The results of CC and NonShare methods are illustrated in Figure 5.4(a). We observe that CC performs better than NonShare when \( f \) is smaller than some value between 0.1% and 2.5%. The performance degrades quickly with \( f \) increases, indicating that the CC performance is...
quite sensitive to $f$. This result is consistent with our cost analysis (see Section 4.3.2), which demonstrates that the connection cost of connect events contribute significantly to the CC overhead cost. Thus, the more connect event instances exist (larger $f$), the worse CC performs. To observe the changing point of the CC performance more clearly, we conduct this experiment again by varying $f$ at much smaller values (see Figure 5.4(b)). We see that CC performance begins to degrade at 1.3%. Now, we plug in the fixed $l$ and $k$ of the two shared substrings into our Benefit model and determine that: to make their $BV_{value} > 0$, the $f$ of $sub_1$ should be smaller than 1%, to ensure that we can benefit from using the CC solution.

Varying the Connect Event Frequency $l$. We vary the $sub_1$ length from 4, 6, 8, 10 and 12 in the following way:

- $l=4$: (YHOO, AMAZ, MSFT, ORCL)
- $l=6$: (YHOO, AMAZ, MSFT, ORCL, RIMM, CSCO)
- $l=8$: (YHOO, AMAZ, MSFT, ORCL, RIMM, CSCO, INTC, IBM)
- $l=10$: (YHOO, AMAZ, MSFT, ORCL, RIMM, CSCO, INTC, IBM, GOOG, LNKD)
- $l=12$: (YHOO, AMAZ, MSFT, ORCL, RIMM, CSCO, INTC, IBM, GOOG, LNKD, FB, NTAP)

The frequency of YHOO is set to 0.5% to ensure that CC outperforms NonShare. $k$ is 3 means that $sub_1$ is still shared by 3 queries. The results of CC and NonShare methods are illustrated in Figure 5.5(a). We observe that, compared to the NonShare method, the performance of CC is increasingly better as $l$ grows, as the processing time difference of these two methods increases. This indicates that the superiority of sharing is more pronounced when the shared length is longer.

Varying the Number of Shared Queries $k$. In this experiment, we add more queries into the $WL_1$ to compose a larger query group with 8 queries. To vary $k$ at 2, 3, 4, 5 and 6, only 2 of the 8 queries share $sub_1$, while the other 6 queries do not contain $sub_1$. 54
Then, each time we change the pattern of one more query to contain \( \text{sub}_1 \). Due to space limitations, we omit the specific queries here. \( f \) is still set to 0.5% and \( l \) is 7. The results of \( CC \) and \( \text{NonShare} \) methods are illustrated in Figure 5.5(b). We observe that the processing time difference of these two methods is increasing, indicating that compared to \( \text{NonShare} \) method, the performance of \( CC \) improves with the growth of \( k \). Thus, more queries share \( \text{sub}_1 \), the better \( CC \) performs.

To sum up, the \( CC \) performance is most sensitive to \( f \) compared with \( l \) and \( k \). For a substring, if its \( f \) is low, the longer this substring and the more queries share it, the better \( CC \) will perform. This result is consistent with our Benefit Model analysis (see Section 4.3.2).

![Figure 5.6: (a) Running Time of Multi-query Optimizers](image1)

![Figure 5.6: (b) Effectiveness of Multi-query Optimizers](image2)

(a) Running Time of Multi-query Optimizers  
(b) Effectiveness of Multi-query Optimizers

**Figure 5.6:** (a)(b) Comparisons between Multi-query Optimizers in terms of Running Time and Effectiveness

**Experiment 2 - Evaluation of Multi A-Seq Optimizer.** In this experiment, we compare the two optimization algorithms: our optimizer (Multi A-Seq) and the optimal search (Exhaustive), by applying them to the same query groups. We design 12 query groups (QG), where \( QG_1 \) to \( QG_3 \) contain 8 queries, \( QG_4 \) to \( QG_6 \) contain 12 queries, \( QG_7 \) to \( QG_9 \) contain 16 queries, and \( QG_{10} \) to \( QG_{12} \) contain 20 queries. To ensure that queries have common substrings to share within each group, we first design 8 substrings of length 2 to 3. Then we compose queries by randomly forming combinations of these 8 sub-
strings. Thereafter, we adjust the event stream by controlling the frequency of the connect event types. This provides the opportunity for sharing plan to have positive benefit values, otherwise our optimizers would quickly suggest to the NonShare solution. Due to space limitations, we do not list these 12 query groups here. We run both approaches on these 12 query groups, record their optimization running time, and compare whether our optimizer returns the same sharing plan as the optimal algorithm. The results are illustrated in Figure 5.6(a).

As depicted in Figure 5.6(a), the running time of the Exhaustive optimizer is significantly longer than that of our Multi A-Seq optimizer on average, with Y-axis in a logarithmic scale. The larger the query group is, the more sharing chances arise. Consequently, the longer time the Exhaustive optimizer takes. While the performance of Multi A-Seq optimizer is efficient and fairly stable. For the quality of the optimizers found, Multi A-Seq optimizer returns optimal results in 11 of 12 groups. For $QG_7$, the result slightly differs, however, below we show that it does not make big difference in performance. This confirms that our optimizer achieves high accuracy in practice..

In addition, we process these 12 query groups with sharing plans suggested by the optimizers, and record the average processing time result to test the effectiveness of our Multi A-Seq optimizer (Figure 5.6(b)). Generally speaking, if the Multi optimizer suggests to share, the performance of share (gray bar) is always better than non-share (white bar). For $QG_3, QG_5$ and $QG_6$, both optimizers suggest computation each query separately without sharing. Thus no result is shown for these 3 groups. For $QG_7$, where the Multi A-Seq optimizer returns a different sharing plan compared to the Exhaustive optimizer, we thus process $QG_7$ with 3 plans: NonShare, our Greedy optimizer, and Exhaustive optimizer (black bar). We can see that the latter two plans perform indeed both better than NonShare, with the Exhaustive optimizer is only slightly better than our Greedy optimizer. To conclude, our Multi A-Seq optimizer is highly accurate yet extremely efficient.
Chapter 6

Related Work

Complex Event Processing. CEP systems have been developed for scalable pattern detection over high-speed data streams. SASE [9, 25] employs an NFA-based matching model for stack-based sequence construction. Cayuga [4] employs a more general NFA system for processing complex events. These two systems inherit the limitations of the NFA-based model including the late negation-filter processing. ZStream [17] optimized this CEP sequence matching process by selecting a flexible tree-based query execution plan using costing in place of the fixed-order NFA evaluation. However, no technique has been proposed to address the aggregation computation performance over sequence patterns. Rather, in these CEP systems, aggregations would be applied as a post-pattern-detection step, resulting in inefficient solution as demonstrated by our experiments (Section 5.2).

Aggregations over Stream Data. Research on traditional aggregation over data streams has also been studied. [12, 13] propose incremental techniques that avoid recomputation among overlapping sliding windows. [26] provides an optimal solution to top-k aggregation. Unlike the work mentioned above, [27] maintains aggregates using multiple levels of temporal granularities: older data is aggregated using coarser granu-
larities while more recent data is aggregated with fine detail. However, these state-of-art aggregation techniques are all set-based (like aggregation in a relational database) instead of sequence-based. That is, the aggregates are computed over individual data events of the stream rather than over detected complex sequences.

**Aggregation in Static Sequence Databases.** For static sequence databases, SQL-style languages support order join operation among data records and also aggregation functions [1, 16]. However, these works assume that the data is still statically stored apriori to processing. Also, sequence is defined by time-based predicates instead of continuously arriving event streams. Thus, customized query processing strategies are designed to implement operators with such time-based predicates that effectively utilize disk-and-buffer resources. In contrast, A-Seq targets dynamic stream data where results are produced instantaneously and continuously upon the arrival of data. [1, 22] support range-based aggregation, where independent data records within a certain time range are aggregated. A-Seq instead works at a higher level, where aggregates are over multi-sequences rather than within a single sequence. Moreover, in [19, 21], aggregations are specified for patterns with recursion, however, again on independent data records.

**Data Mining of Sequential Patterns.** Sequential pattern mining aims to discover all subsequences that frequently arise over sequential data. Typically, they use a Prefix Tree (PF-Tree) structure and the Apriori Principle to find all frequent subsequences [6, 7, 18, 20]. Clearly, the problem introduced by those works is distinct from ours. First, in our context, the pattern query is pre-specified by users and thus our task is to search for occurrences of that particular sequence pattern rather than to discover all possible frequent patterns. Second, the notion of sliding window semantics is not adopted in sequential pattern mining. Thus, solutions to tackle efficient data purging or result updating are not required. Third, existing sequential pattern mining does not handle CEP specific problems including negation.
Multi-query Sharing. A lot of research has also been done on multi-query sharing topics. [28] presents a method to detect potential sub-expression sharing opportunities among multi expressions. [16] supports OLAP based aggregation operations for sequence by incorporating hierarchy for pattern sharing. However, these two works are for traditional static databases, which cannot solve the issues caused by continuous streaming data. [11] extends aggregation to multi-query workloads by sharing streaming aggregate queries with differing periodic windows and arbitrary selection predicates. Yet, these techniques focus on non CEP pattern queries, where sequence constraint is not taken into account. citeMOEcube proposes concept and pattern hierarchy to explore the sharing opportunities among multi CEP pattern queries. But those sharing strategies are designed for stack-based sequence construction computation, thus can only be applicable to the post two-step aggregation.
Chapter 7

Conclusion

In this thesis, we present the A-Seq approach for high-performance processing of CEP aggregation queries in streaming environments. A-Seq pushes the aggregation computation into the pattern detection process. It gracefully tackles the CEP-specific challenges including window constraints, negation and predicates. Compared to the state-of-art two-step solution, A-Seq is a lightweight computation method, which achieves several orders of magnitude savings in terms of both CPU costs and memory utilization. Techniques for A-Seq aggregation computation sharing among multi-query workloads is also developed for multi-query optimization. First, the problem is shown to be NP-hard, then an efficient cost-driven Multi A-Seq optimizer is designed to search and form a sharing plan for a given multi-query workload, which is shown to be practical while achieving high-quality (near optimal) sharing plans.
Bibliography


