COMPLEX EVENT PROCESSING OVER OUT-OF-ORDER EVENT STREAMS

Sinthuja Rajendran

(168259U)

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

February 2019

COMPLEX EVENT PROCESSING OVER OUT-OF-ORDER EVENT STREAMS

Sinthuja Rajendran

(168259U)

Thesis submitted in partial fulfillment of the requirements for the degree Master of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

February 2019

DECLARATION

I declare that this is my own work and this MSc project report does not incorporate without acknowledgement any material previously submitted for degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to the University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or another medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: .....
Date: .....
Name: Sinthuja Rajendran

We certify that the declaration above by the candidate is true to the best of our knowledge and that this report is acceptable for evaluation for the CS6997 MSc Research Project qualifying evaluation.

Supervisors

.....	.....
Dr. H. M. N. Dilum Bandara	Dr. Srinath Perera
.....	.....
Date	Date

ABSTRACT

Complex Event Processing (CEP) enables real-time inferring of events and patterns of interest. Aggregation on a time window of events and pattern matching are two of the core functionalities of CEP. Accuracy of these CEP operations depend on the order of the events received at the CEP engine. However, due to network delay, environmental differences in event producing sources, and distributed CEP systems, event arrival order at the CEP engine maybe different from the order of event generation at the source. Such out-of-order events may lead to incorrect output events by the CEP engine.

We propose a novel solution to handle the out-of-order events in three steps, namely (a) ordering events from the same source, (b) ordering events from multiple sources, and (c) optimizing query operator to further improve the accuracy after applying former steps. Sequence numbers are used to order events from a single source, whereas estimated time drift of each event source is used to order event from multiple event sources. Finally, the query operators are optimized to reduce the error of remaining out-of-order events. Performance of the proposed solution is evaluated using the DEBS 2013 Football dataset. The performance analysis shows that the proposed techniques result in 9600% to 21300% and 1200% to 2500% reduction in latency compared to MP-K-Slack and AQ-K-Slack techniques, respectively. Further, the proposed solution was able to order the events with 99.97% - 99.99% accuracy. While it is comparatively lower than MP-K-Slack which had an accuracy of 99.99% and better than AQ-K-Slack which had an accuracy of 99.02%. Therefore, the proposed solution provides a good balance between latency and accuracy. The additional optimizations carried out in aggregator and pattern matching operators further increased the accuracy of the results by 50% compared to the final results obtained without these query optimizations.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my deep sense of gratitude and profound feeling of admiration to my project supervisors. Many thanks go to all those who helped me in this work. My special thanks to University of Moratuwa for giving an opportunity to carry out this research project.

I would like to gratefully acknowledge to Dr. Dilum Bandara, the internal supervisor of the project, for sharing the experiences and expertise with the project matters. I would like to extend my heartfelt gratitude to Dr. Srinath Perera, the external project supervisor, for his continuous guidance and support throughout the whole duration of the project. Last but not least, I thank to Dr. Miyuru Dayarathna, for sharing the experience in the related work carried out with Siddhi engine, and all those who like to remain anonymous although the help they provided me was valuable.

Thank you.

TABLE OF CONTENTS

DECLARATION i

ABSTRACT ii

ACKNOWLEDGEMENTS iii

TABLE OF CONTENTS iv

LIST OF FIGURES vii

LIST OF TABLES ix

1. INTRODUCTION 1

1.1. Background 1

1.2. Motivation 2

1.3. Research Question 4

1.4. Objectives 5

1.5. Outline 5

2. LITERATURE REVIEW 6

2.1. Complex Event Processor Functionalities 6

2.2. Out-of-order Event Handling Approaches 7

2.2.1. Buffer-based Approach 7

2.2.2. Punctuation-based Approach 8

2.2.3. Speculation-based Approach 9

2.2.4. Approximation-based Approach 9

2.3. Buffer-based Techniques 9

2.3.1. K-Slack Approach 10

2.3.1.1. Out-of-order Event Processing in SASE 12

2.3.2. MP-K-Slack Approach 16

2.3.3. AQ-K-Slack Approach 18
2.3.4. Latency Distance and Purging Time Based out-of-order Event Processing (LDOP) 22
2.3.5. K-Slack Chain Approach 23
2.3.6. Summary 27
3. METHODOLOGY 29
3.1. Definitions 29
3.2. Proposed Solution 31
3.2.1. Handling Events Produced from Single Event Source 31
3.2.2. Handling Events Produced from Multiple Event Sources 36
3.2.3. Query Operators with Out-of-order Events 40
3.2.3.1. Time Batch Window and Aggregation Operators 40
3.2.3.2. Pattern Matching 42
3.4. Summary 43
4. IMPLEMENTATION 45
4.1. Summary 53
5. PERFORMANCE EVALUATION 55
5.1. Dataset 55
5.1.1. Dataset for Single Source 59
5.1.2. Dataset for Multiple Sources 59
5.1.3. Dataset with Time Drift for Event Source 60
5.1.4. Dataset for Pattern Matching 60
5.2. Experimental Setup 61
5.2.1. Prototype 61
5.2.2. Analysis of Out-of-order Events Handling Solution 61

5.2.2.1. Performance and Accuracy with Single Event Source 62
5.2.2.2. Performance and Accuracy with Multiple Event Sources 68
5.2.2.3. Accuracy with Time Drifted Event Sources 71
5.2.3. Analysis of Query Operators under Out-of-order Events 73
5.2.3.1 Aggregator Operator 73
5.2.3.2. Pattern Matching Operator 76
5.3. Summary 78
6. CONCLUSION 80
6.1. Summary 80
6.1. Research Limitations 82
6.2. Future Work 84
REFERENCES 86

LIST OF FIGURES

Figure 1.1: Out-of-order event arrival 2

Figure 2.1: Out-of-order event arrival – K- Slack 11

Figure 2.2: Sorting the events in the window with K-Slack 11

Figure 2.3: Event Query Plan 12

Figure 2.4: Query Evaluation of SASE 13

Figure 2.5: Problems exists in SSC and PSSC with out-of-order event arrival 14

Figure 2.6: Out-of-order event arrival handling in MP-Slack 18

Figure 2.7: Design of AQ-K-Slack 19

Figure 2.8: Adoption of α using a PD controller 21

Figure 2.9: A Global Query Graph 24

Figure 2.10: Shared disorder handling for a query graph G that has no filter operators: single K-slack versus K-slack chain 26

Figure 3.1: Event source S_1 sending event stream R_1 and event stream R_2 to CEP 32

Figure 3.2: Communication between the CEP receiver and event source to calculate the timestamp drift 37

Figure 3.3: Out-of-order event handling with multiple event sources with event stream R_1. 39

Figure 3.4: Multiple time batch windows to cover the events from ± (Ta/2) of original time batch window w. 41

Figure 4.1:Out-of-order event flow of an event stream within siddhi engine. 45

Figure 4.2: Sequence-based ordering in sequence-based reorder extension. 50

Figure 5.1: Playing field and its dimensions 55

Figure 5.2: Average events inter-arrival time with out-of-order event Dataset 1 and Dataset 2 57

Figure 5.3: Maximum events inter-arrival time in Dataset 1 and Dataset 2. 57

Figure 5.4:Average events inter-arrival time with out-of-order event Dataset 3 58

Figure 5.5: Maximum event inter-arrival time in seconds for Dataset 3. 58

Figure 5.6: Average and maximum latency of events for Dataset 1 with proposed sequence-based approach. 64

Figure 5.7: Average and max latency of events for Dataset1 with MP-K-Slack, AQ-K-Slack and sequence-based approach 64

Figure 5.8: Average latency of all three approaches with respective to the dataset. 65

Figure 5.9: Total number of out-of-order events for all three approaches and the respective datasets. 67

Figure 5.10: Variation of average latency in ms of all three approaches with number of event sources. 68

Figure 5.11: Average latency for sequenced based approach across all event sources. 69

Figure 5.12: Total out-of-order events with multiple event sources for all three approaches 70

Figure 5.13: The number of out order events with amount of time drifts for sequence based approach. 72

Figure 5.14: The difference between the of average values for normal time batch window, and reorder based time batch window. 75

LIST OF TABLES

Table 2.1: The comparison of the out-of-order event handling approaches 10

Table 2.2: Comparison of buffer-based approaches 28

Table 3.1: List of Symbols 33

Table 5.1: Description of event attributes 56

Table 5.2: Input Dataset and out-of-order events 56

Table 5.3: Specification of the machine that was used for the experiment and evaluation 61

Table 5.4: Overall Summary of Latency incurred for events in Dataset1 with all out-of-order handling approaches 65

Table 5.5: Total number of out-of-order events with Dataset1 with all out-of-order handling approaches 65

Table 5.6: Average Latency for all three approaches along with datasets. 66

Table 5.7: Accuracy of all three approaches with datasets. 66

Table 5.8: Average latency for all three approaches along with number of events sources 69

Table 5.9: Accuracy of all three approaches when publishing events with multiple sources. 70

Table 5.10: The total out-of-order events produced in sequence based approach with the time drift between the event sources, with and without time syncing of event source. 71

Table 5.11: The differences between the expected and actual average values. 74

Table 5.12: The average value of the velocity after using the reorder based time batch window. 75

Table 5.13: Results of the pattern matching dataset. 77

LIST OF ABBREVIATIONS

AIS	Active Instance Stack
ATM	Automated teller Machine
CEP	Complex Event Processor
DAG	Directed Acyclic Graph
IoT	Internet of Things
LDOP	Latency Distance and Purging Time
NFA	Nondeterministic finite automaton
NTP	Network Transfer Protocol
PD	Proportional Derivative
PSSC	Purged Sequence Scan Construction
QDDH	Quality Driven Disorder Handling
RFID	Radio-Frequency Identification
SC	Sequence Construction
SL	Selection
SS	Sequence Scan
TCP	Transmission Control Protocol
TF	Transformation
UDP	User Datagram Protocol
WD	Window

1. INTRODUCTION

Enterprises that monitor events in real-time from all their systems and swiftly respond to them have a greater competitive advantage over others. Complex Event Processor (CEP) [1] is the right solution for such real-time monitoring use cases, where it listens to events that are triggered from various sources and detect patterns in near real-time with minimum or no storage of events. CEP is being used in domains such as IoT, network and systems monitoring, banking, health care, etc. Therefore, it is important to have accurate and fast results. In CEP, most of the core operators such as pattern matching, time and batch window operations, aggregation operators, and join operators process the events based on the order they arrive at CEP node/engine. However, due to reasons such as network delay, environmental difference in event producing sources, network and machine failures, using the connection pool to publish events, and distributed processing in CEP the event arrival order at the CEP node may not be same as the actual occurrence of events at the event source. This is more prevalent in IoT environments due to the high rate of data transmissions, and involvement of multiple sensors and gateways. Consequently, the result of the intended analysis will not be accurate as events are processed out-of-order.

1.1. Background

The basic unit in the CEP engine is an *event* and it is a unit of data that usually contains a timestamp and a set of attribute values according to a defined schema. The infinite continuous sequence of events arriving on a particular type is called a *stream*, on which users can perform complex analytical processing. One event is associated with only one event stream, and all events of that stream have an identical set of attributes assigned with the same schema. This data in motion analytics can be provided to the system as query/rule that will be executed on each event as and when they arrive at the system in the continuous fashion.

CEP engine [1] is a real-time in-memory event processing engine which receives events from various sources, transports, and environments. CEP engine correlates

these events and performs more complex analysis on those events. As CEP engines are used in real-time analytics systems, CEP engines need to have high throughput while being able to process many queries with low latency and high accuracy. As the name implies, the CEP engines are not only used for simple filtering where the events could be filtered by certain attribute value, but also used for pattern matching of the event sequence for a given a sliding time or batch window, joining two or more streams, and performing aggregations on the window of events. These functions react based on the order of the events that is being received at the CEP engine than compared to the actual occurrence of the events.

1.2. Motivation

In CEP, the event processing accuracy is based on time and the order of events that were received at CEP engine with respect to the order that were originated in the source. Let us consider an example of tracking books in a book store where RFID [2] tags are attached to each book and RFID readers placed at key locations throughout the store, like bookshelves, checkout counters and the store exit. The RFID readers send data to CEP which check for potential theft of books and alert the staff. To detect and alert theft, a pattern matching query can be used. Typical checkout process can be modeled as a series of events where an event gets generated when a book is taken out from the bookshelf (e_1). Then there should be an event related to billing operation performed for the book (e_2) and followed by another event detecting the book passing the exit (e_3). If these three events do not occur in the said order, an alarm should be triggered to notify the staff. Therefore, in a genuine case if the events e_1, e_2, and e_3 have been received in order by the CEP engine no fraud pattern will be detected. In case if e_2 was delayed, and only e_2 and e_3 arrived at the CEP, then a false positive alarm will be triggered. This could even affect the business as the false alarm was associated with a legitimate customer.

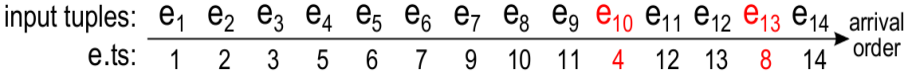


Figure 1.1: Out-of-order event arrival.

Figure 1.1 shows out-of-order event arrival in an arbitrary event stream in which the order of event arrival in the CEP is indicated as input events and the actual timestamp of the event occurrence as *e.ts*. While the events e_{10} and e_{13} were triggered at time 4 and 8, they are actually received after e_9 and e_{12} which were triggered at time 11 and 13 respectively. Also, the events which has been generated after e_{10} and e_{13}, e.g., e_4, e_5, e_6, have already reached the CEP before e_{10} and e_{13}. Such behavior could be due to reasons such as network jitters, environments of different event sources, and network and machine failures. Further it is difficult to preserve the order of events, if the incoming rate of the events are very high, and the event sources are located in different networks which may experience different delays and finally, result in out-of-order arrival in CEP.

Such out-of-order event arrival affects the accuracy of CEP queries such as pattern matching, aggregation queries within time window, and join queries. The order of the event arrival has different impact when we consider the aggregate and pattern matching scenarios, where the first case does not require the order of the events rather, when the window time elapses all the events for the given window should be arrived, but in the second case before proceeding the pattern matching operator the events needs to be sorted in the same order that have originated.

It is not trivial to handle out-of-order events in CEP due to the following reasons:

1. Impact of premature accuracy of the results, specially the output results that are triggered based on insufficient messages maybe irreversible.
2. Inability to predict delay accurately and allocate sufficient buffer time or capacity.
3. Buffering introduces delay and reduces throughput of CEP engine which are key features of CEP.
4. Increased memory and computing requirements of the CEP engine to store and handle out-of-order events.

Resolving the above challenges and building a solution which could satisfy all the use cases is not practical. Some use cases can compromise on the accuracy to gain low latency. For example, a query applying the average operator may produce a less precise result while achieving low latency, as no buffers are used to collect and correct out-of-order arrivals. This is acceptable in use cases where the out-of-order event arrival is rare or infrequent and inaccuracy is acceptable. Anyhow, this may not suit for use cases such as credit card fraud detection where the order of events matter. While false positive alerts are acceptable in this case false negatives are not acceptable. In certain use cases, once the alert has been triggered based on out-of-order arrival, it may be reversed by issuing a reverse alert. For example, let us consider a use case of actuating a fire alarm based on the pattern of temperature increase, once the alarm has been switched on, we cannot do much other than switching it off, even after detecting that is a false alarm.

Accuracy is a major challenge in this problem, where unless CEP engine receives an event that is having older timestamp than the latest event that the engine has seen so far, it does not know whether there is an event yet to be received for that time window or not. Therefore, we cannot be 100% certain about the produced results but rather can only have a confidence level associated with the final results. However, if the confidence level needs to be increased, it will increase the latency of the final result.

1.3. Research Question

The main operators provided by CEP engine such as pattern matching, aggregations operators, and join operators in time and batch window depend on the order of the events that have been received at the CEP engine. This phenomenon is more prevalent when multiple event sources produce events that are collectively processed and analyzed by the same CEP query. Therefore, the research question can be formulated as:

How to detect and overcome out-of-order event arrivals in complex event processing while increasing accuracy and minimizing time and space complexity?

1.4. Objectives

The main objectives of this research can be stated as follows:

1. Develop a suitable set of out-of-order event handling mechanism(s) that can provide increased accuracy and low latency which can work with multiple event sources.
2. For more complicated out-of-order arrival scenarios develop a technique to detect and estimate the possibility of the out-of-order event arrival with high confidence while having minimum overhead for pattern matching and aggregation operations.
3. Formulate a solution that could order the out-of-order events with a delay less than a few minutes and does not consider the very late event arrivals.
4. The solution may involve modifications in different components and stages of event flow including event sources, CEP event receiver and query operators.
5. Evaluate the accuracy and performance of the proposed techniques by simulating the out-of-order event arrival with varying network delays and multiple nodes.

1.5. Outline

Related work is presented in Chapter 2. Furthermore, it provides the functionality supported by CEP queries, critically evaluates the work carried out in each approach and suitability of those approaches for the objectives mentioned in Section 1.4. Chapter 3 presents the proposed methodology to handle the out-of-order events with single and multiple event sources, and with aggregation and pattern matching query operators. Chapter 4 explains the implementation details of the proposed methodology based on the Siddhi [3] CEP engine. Chapter 5 discusses the experiment carried out with different approaches, test results, and evaluation. Chapter 6 discusses the conclusions of the results obtained, research limitations, and future work.

2. LITERATURE REVIEW

This Chapter discusses in detail about the related work with regard to out-of-order event arrival problem in Complex Event Processing (CEP). In Section 2.1, CEP and several important functions supported by CEP are discussed. Methodologies that the researchers have followed to solve the problem of out-of-order event arrival are presented in Section 2.2. Section 2.3 provides a detailed analysis of buffer-based approaches that had been already researched.

2.1. Complex Event Processor Functionalities

CEP supports a set of common functionalities such as time batch window, pattern matching, and aggregations operations. The time window is one of the most important functions provided by the CEP engines, where a set of events within a given time window is considered and the user provided CEP query is executed on top of that. A time window quantifies the window based on time such as events collected in 5 minutes, 15 minutes, and 1 hour durations.

Query 2.1 shows an example query related to the time batch window. *TempStream* is a stream of temperature events from temperature sensors in rooms. It calculates the average temperature once a minute. Then the average value is used to create a new stream called *AvgRoomTempStream*.

```
from TempStream#window.timebatch(1 min)
select roomNo,avg(temp) as avgTemp
group by roomNo
insert all events into AvgRoomTempStream ;
```

Query 2.1: Time batch window.

Similarly, the pattern matching also can be performed in the window of streams, where a sequence of the expected events can be provided, the CEP engine will be listening to the pattern in the events streams and once it is matched it will be triggering an alert. Query 2.2 can be used to detect fraud in Automated Teller Machine (ATM) card transactions, where if there is an event a_1 which has *amountWithdrawed* less than 100

and following that event, if there is another event b_1 which has *amountWithdrawed* greater than 10,000 within a day interval for the same ATM card number, then insert the matching details to another event stream *possibleFraudStream* for alerting.

```
from
      every a1 = atmStatsStream[amountWithdrawed < 100]
      -> b1 = atmStatsStream[amountWithdrawed > 10000
      and a1.cardNo == b1.cardNo]
within 1 day
select
      a1.cardNo as cardNo,
      a1.cardHolderName as cardHolderName,
      b1.amountWithdrawed as amountWithdrawed,
      b1.location as location,
      b1.cardHolderMobile as cardHolderMobile
insert into possibleFraudStream;
```

Query 2.2: Pattern Matching.

Fraud detection systems, network monitoring and throttling systems, banking transaction systems, IoT network, and stock exchange systems are some of the example systems that uses CEP extensively. Time window aggregate queries are at the heart of many such continuous analytics applications.

2.2. Out-of-order Event Handling Approaches

Several prior works focus on handling out-of-order events in a single CEP event stream. These techniques can be broadly classified as buffer based [3], [4], punctuation based [5], [6], speculation based [7], [8], and approximation based [9] approach. In all approaches handing the disordered events consists of a trade-off between result accuracy and latency. Nevertheless, each of these techniques is discussed next to understand their design philosophy and the pros and cons.

2.2.1. Buffer-based Approach

Buffer-based approach [3], [4], [10] handles the disordered event reception by having a buffer to store and sort the events from an input stream based on the timestamps before presenting them to the query execution. While this is a simple and common

approach, buffering and sorting increase event processing latency. Several buffer-based solutions are discussed in Section 2.3 in detail.

2.2.2. Punctuation-based Approach

Punctuation-based disorder handling [5] depend on special events (punctuations) within data streams which indicate that no future events with timestamps smaller than the timestamp of punctuation are expected. When punctuation is received, the query operator determines windows for which no future late arrivals are expected and produces query results for these windows. Heartbeats and partial-order guarantees are the types of punctuations used [6]. The punctuations can be produced externally with data sources or generated within the system [11].

Punctuations explicitly inform the query operator when to return results for windows, and until the punctuation is received the events will be buffered. Therefore, the accuracy of the punctuations determines the accuracy of the final results. Furthermore, in the environment with the network jitters and connectivity problems, the punctuation event stream itself can get affected when it is sent from the external data source and may not be received by CEP in the expected time. This leads to the poor accuracy of the final results produced. Also, the trade-off between the latency and the accuracy of the query results is a limitation of this approach. As the punctuation is providing the confirmation on not receiving any delayed events after the punctuation timestamp, it should not be generated before all late arrivals of the window are observed. Therefore, this approach shares the same latency issue as the buffer-based approach, where it cannot determine the buffer size until all the late arrivals are received. Another approach would be to treat late arrivals appearing after the corresponding punctuations as separate data partitions and to process the partitions independently [12]. Partial results of these partitions can be merged with previously produced inaccurate query results, which were caused by the early trigger of punctuation. However, this approach requires keeping the entire history of query results.

2.2.3. Speculation-based Approach

Speculation-based disorder handling [7], [8] can be considered as an aggressive approach because it assumes in-order arrival of events and produces the results of a window immediately when the window is elapsed. The other approaches such as buffer based and punctuation based can be considered as conservative. When a late arrival event *e* is detected, previously produced results which are affected by *e* are invalidated. New revisions of these results are produced by taking *e* into account. For data streams with frequent out-of-order events, one query result may be revised many times before the final exact revision is produced and this exhausts CPU and may cause high result latency.

2.2.4. Approximation-based Approach

Approximation-based disorder handling [9] computes approximate aggregates over data streams. This technique summarizes the raw data stream with a special data structure (e.g., histograms and *q*-digests) and produces approximate aggregate results based on these summaries. Approximation-based approaches follow an aggressive strategy. The difference from speculation-based technique to this approach is that when a late arrival is received, the approximation-based approach only ensures that this late arrival is accounted for in future aggregate computations, and it does not correct previously emitted results. For queries with small windows, this strategy leads to many incorrect output events.

Table 2.1 shows a comparison of the approaches discussed in the above Sections. To satisfy various objectives provided in Section 1.5, in this research work we will be focusing on the buffer-based approach which could provide the balanced behavior between the accuracy and latency of the produced results.

2.3. Buffer-based Techniques

Solutions based on the buffer-based approach can be broadly classified as K-Slack, MP-K-Slack, and AQ-K-Slack approach. K-Slack approach delays the incoming event by *k* time units, where *k* is defined as a priori. MP-K-Slack delays the incoming events

by k time units, which is the largest delay that has been seen so far at CEP engine. Furthermore, AQ-K-Slack delays the events by αk time units where k is the largest delay that has been seen so far and α is the dynamic parameter that is determined from user-specified result error threshold value. A detailed explanation of each of these methods is explained in the following Sections.

Table 2.1: The comparison of the out-of-order event handling approaches.

Out-of-order Event Handling Approaches	**Advantages**	**Disadvantages**
Buffer-based Approach	Accuracy of results can be increased by having large buffer size. Latency can be reduced by having small buffer length. Ability to adjust the buffer size and obtain balanced accuracy and latency.	Dilemma between latency and accuracy. Increased memory requirement to buffer the events.
Punctuation-based Approach	Accuracy of results can be increased by having large punctuation interval. Latency can be reduced by having small punctuation interval.	Punctuation event stream itself can get affected, which affects the accuracy of results produced. Dilemma between latency and accuracy. Increased memory requirement to buffer the events.
Speculation-based Approach	Less latency.	Increased traffic of correction events. Cannot control accuracy of the result produced.
Approximation-based Approach	Less latency.	Inaccurate results are left without correcting them. Cannot control accuracy of the result produced.

2.3.1. K-Slack Approach

K-slack transparently buffers and reorders the events before the actual event processing happens. It uses a buffer to delay each incoming event (e_i) by at most a predetermined k time unit from the current timestamp t_{curr} where $e_i.ts + k \leq t_{curr}$ [10].

In this method, *k* is assigned based on a priori knowledge about the incoming events stream. However, it is not trivial to estimate a suitable value for *k* in realistic environments with fluctuating network properties such as network traffic and delay.

Figure 2.1 depicts the out-of-order event arrival problem, where *id* indicates the event types received, *ts* is the actual event originated timestamp, and *clk* is the latest timestamp of the event that has been seen by the system. As shown in Figure 2.1, event C (with *ts* = 1) which was generated before event A (with *ts* = 2) but received later than event A at CEP engine. Similarly, event B (with *ts* = 3) was generated before event A (with *ts* = 4) also have been received at the CEP engine in reverse order. At the same time event E (with *ts* = 20) was received before event C (with *ts* = 12) and A (with *ts* = 15). As shown in Figure 2.2, with a re-ordering unit these out-of-order events can be sorted and then passed to the CEP engine for execution. For example, in Figure 2.2, the buffer size of five will be suitable to sort the out-of-order events, where the events seen so far have been delayed by at most five time units.

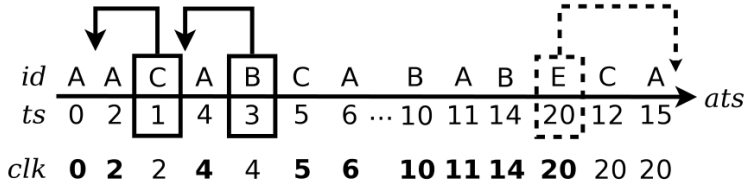


Figure 2.1: Out-of-order event arrival - K-Slack.

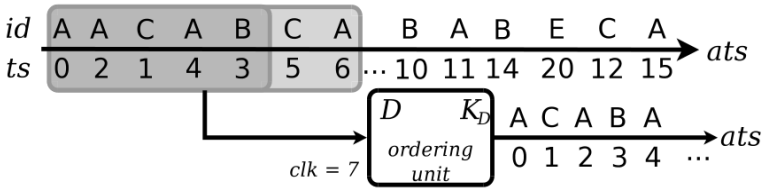


Figure 2.2: Sorting the events in the window with K-Slack [7].

Ming Li et al. [10] in SASE (see Section 2.3.1.1) proves that K-Slack can also be applied in distributed stream applications. Authors used local clock *clk* was used, where an event *e* is buffered at least as long as $ei.ts + k \leq clk$. As there is no global

clock in a distributed reactive system, each node synchronizes its local clock according to the largest timestamp seen so far on any incoming event.

The key issue with K-Slack is having a single fixed *k*, which cannot be easily derived and cannot adapt to network changes. Hence, use of conservative and overly large *k* values result in large buffers and high latency which are not usually preferred.

2.3.1.1. Out-of-order Event Processing in SASE

Ming Li et al. [10] studied the out-of-order event processing in a Stream-based and Shared Event Processing Engine (SASE) based on K-Slack. Authors translated the query plan, based on the operators such as Sequence Scan (SS), Sequence Construction (SC), Window (WD), Selection (SL) and Transformation (TF) as explained in Figure 2.3.

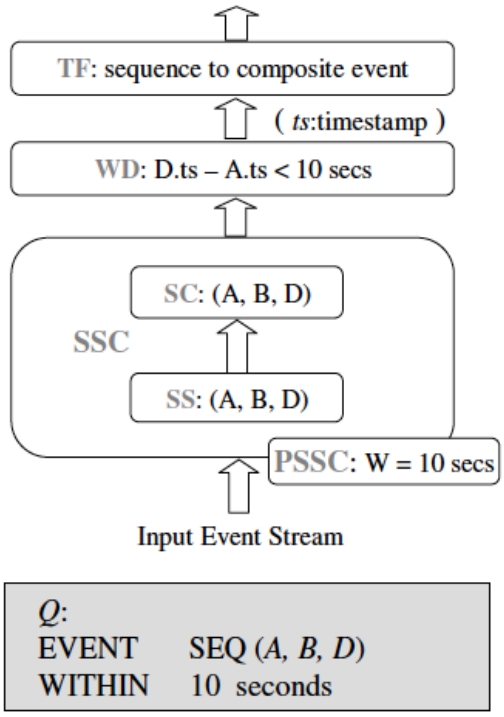
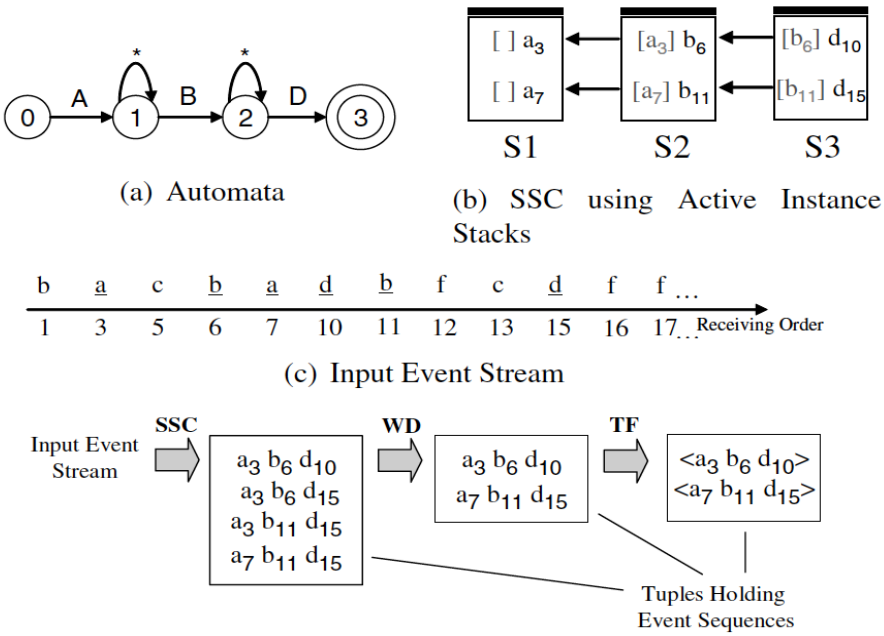


Figure 2.3: Event query plan [10].

The SS operator employs an NFA (Non-Deterministic Finite Automata) to detect matches to the event pattern specified in the query. As depicted in Figure 2.3, the SC

operator constructs the expected event sequences based on events retrieved by SS. SS and SC together create the Sequence Scan Construction (SSC) component. The SL operator filters event sequences by applying all the predicates specified in the query. The WD operator checks whether events in the input event sequence occur within a sliding window. The TF operator converts each input event sequence into a composite event.



(a) Automata

(b) SSC using Active Instance Stacks

(c) Input Event Stream

(d) Producing Result Tuples

Figure 2.4: Query Evaluation of SASE [10].

A data structure named Active Instance Stack (AIS) was proposed for the execution of SSC. Instead of using a single stack for the NFA, as illustrated in Figure 2.4(a), AIS associates a stack with each state of the NFA storing the events that triggered the NFA transition to that state. The events stored in each stack are called the active instances of the stack. Besides that, for each active instance *e* in the stack, an extra field is created to record the most recent instance in the stack of the previous state (RIP). Figure 2.4(c) shows a partial input event stream, and the events marked with an underscore were extracted during the sequence scan. All the retrieved events of type *A*, *B*, and *D* are kept by AIS. Figure 2.4(b) shows the content of the three AIS stacks after the arrival

of the event stream depicted in Figure 2.4(c). Each active instance of the accepting state initiates the sequence construction, and according to the provided event stream in Figure 2.4(c) d_{10} and d_{15} events will initiate the sequence construction.

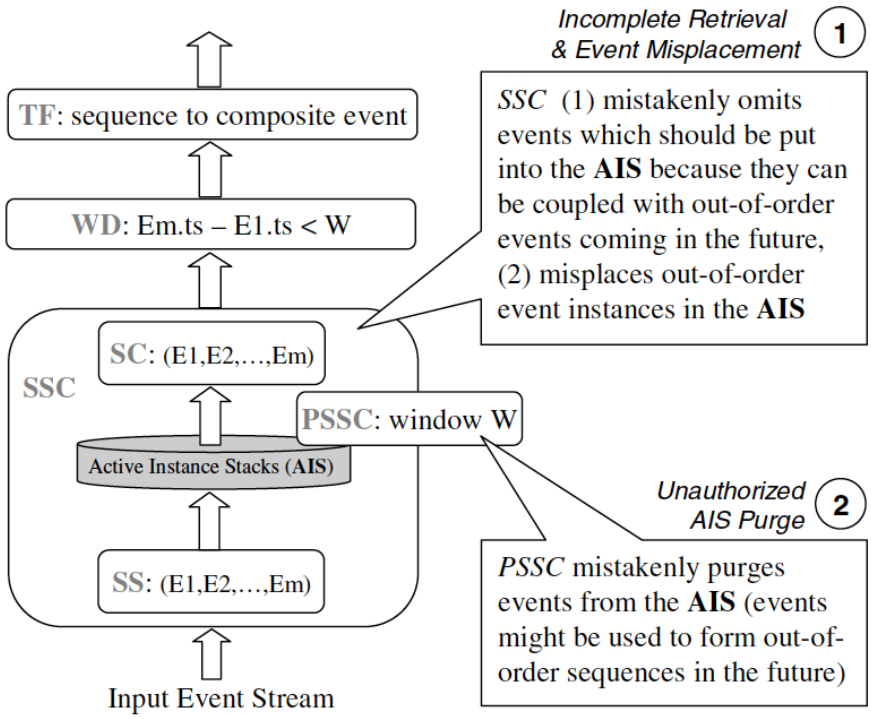


Figure 2.5: Problems exists in SSC and PSSC with out-of-order event arrival [10].

With AIS, the construction is simply done by a depth-first search in the DAG (Directed Acyclic Graph) that is rooted in this instance and contains all the RIP edges reachable from the root. Each root to leaf path in the DAG corresponds to one matched event sequence to be returned by the SSC operator. State purge on SSC is conducted based on window constraints for removing outdated events from AIS dynamically, and this function is called a Purged Sequence Scan Construction (PSSC). In this design, the out-of-order event arrival is causing problems in SSC such as incomplete event retrieval and event misplacement. Incomplete event retrieval problem involves discarding some events by the sequence scan which should have been kept. Event misplacement involves inserting into the wrong location of the stack. In AIS when an

event needs to be inserted it could have been just inserted into the top of the stack due to total order assumption, but that cannot be done when the total order assumption is wrong with out-of-order event arrival. Hence, the events need to be scanned and inserted into the correct location. In PSSC, with out-of-order data arrivals, above window constraint-based, AIS purge is no longer safe and referred to as unauthorized AIS purge as described in Figure 2.5. Therefore, any data removal of AIS is unauthorized unless the total order on the data arrival holds for the input stream.

The incomplete event retrieval problem is handled by setting all states of NFA to be active before the retrieval over the event stream. The event misplacement problem is dealt with applying Sort Semantics which involves in searching for a proper insertion place in the corresponding stack. Therefore, the sort semantics guarantees that the event instances in the same stack are in chronological order from bottom to top for each event instance that triggers a transition in NFA. Also, the context pointer (RIP) of the inserted event E_i needs to be correctly set. Besides that, if E_i is not the rightmost event type in the sequence query, RIP of the event instances in the right-adjacent stack might need to be updated as well.

To avoid such overhead caused by treating every event as a *potential* out-of-order event, the SSC operator maintains an *AIS-CLOCK* value, which is equal to the largest timestamp of events at AIS. Once a newly retrieved event with a timestamp greater than the current *AIS-CLOCK*, the *AIS-CLOCK* will be updated to the event's timestamp value. Therefore, the new event is handled with Append Semantics as in order events. Whenever a newly retrieved event with a timestamp smaller than the *AIS-CLOCK*, Sort Semantics is applied and corresponding out-of-order specific steps are executed.

To avoid these errors, no data purge can ever be applied on AIS. However, that is not a realistic solution due to the unbounded memory requirement. Thus, for unblocking PSSC functionality, authors have proposed to use the K-Slack based approach. Window purge using K-Slack compares the distance between the checked event and the latest event received at the system. A *CLOCK* value which equals to the largest

timestamp seen so far for the received events is maintained. Each time the *CLOCK* value is updated, PSSC will be notified. According to the sliding window semantics, for any event instance e_i kept in AIS, it can be purged from the stack if $(e_i.ts + w) < CLOCK$ where w is the size of the window. Thus, under the out-of-order assumption, the above condition will be $(e_i.ts + w + k) < CLOCK$. Anyhow holding the outdated event sequences in the AIS structure increases the workload of the SSC operator for event sequence construction. An event sequence produced by such construction can never be a result sequence because they would be removed later by window-based filtering (functionality of the WD operator). Thus, it also brings a burden to the window-based filtering computation. Many of the outdated event instances maybe kept in the AIS stacks, if the k value is significant. Thus, the overhead on sequence construction and AIS filtering should be considered.

As an optimization for the overhead problem in selecting large k value, each stack in AIS is divided into two parts such as outdated event instances and up-to-date event instances. A divider is set for each stack, as instances on or above that are outdated instances and instances below that are the up-to-date ones. For a stack without outdated events, the divider is set to *NULL* and when in-order event triggers the sequence construction in SSC, it considers the events under the divider in each of the stacks.

2.3.2. MP-K-Slack Approach

MP-K-slack is an extension of K-Slack approach, which was introduced by Mutschler and Philippsen [3] where the buffer size k used for event ordering can be dynamically adjusted. In this method, k is initialized to zero and a variable t_{curr} is used to track the greatest timestamp of events seen so far in the stream history. At query runtime, newly arrived events are first inserted into the buffer, and their timestamps are compared with the current t_{curr}. Each time when t_{curr} is updated, the below mentioned two actions are performed. Also, in the case of delayed event arrival where t_{curr} will not be updated. Therefore, the following actions will not be performed, and those will be performed when an event arrives that has timestamp > t_{curr}:

(1) Update k to $\max_i \{k, D(e_i)\}$, where $D(e_i) = (t_{curr} - e_i.ts)$ where $D(e_i)$ denotes the delay of the event e_i and it is evaluated for each event e_i that has arrived since the last update of t_{curr}.

(2) Emit any event e_i that satisfies the following condition from the buffer:

$$e_i.ts + k \leq t_{curr} \tag{2.1}$$

Figure 2.10 shows how MP-K-Slack approach works where the buffer input denotes the input sequence of events arrived at CEP, $e_i.ts$ denotes the timestamp of the event e_i, t_{curr} denotes the current timestamp at CEP, and k denotes the buffer size. As shown in Figure 2.6, in the beginning, k is zero, and event e_1 with $ts = 1$ (e_1^1) and event e_2^4 arrives to the system in order and updates t_{curr} value and then they are emitted from the buffer immediately. Thereafter, when event e_3^3 arrives it will not update t_{curr} as it is a late arrival. When e_4^5 arrives, it has a timestamp greater than t_{curr}; therefore, it updated t_{curr}. As t_{curr} was updated, the above mentioned two actions will be performed, where it calculates delay D for each delayed event (in this case e_3^3 where the $D(e_3^3) = 5 - 3 = 2$) and update the k value. Here as the $D(e_3^3) = 2$ is larger than the current $k = 0$, k is updated to 2 according to action (1) as mentioned above and e_3^3 is emitted from the buffer. Similarly, events e_4^5 and e_5^6 are kept in the buffer until e_6^9 arrives, and though events e_7^7 and e_8^8 are also kept in the buffer at the arrival of e_9^{10} events e_7^7 is emitted while updating k to 3. This is because $\max\{D(e_7^7), D(e_8^8)\} = \max\{3, 2\} = 3$, larger than the current $k = 2$. Then events e_8^8, e_6^9, and e_9^{10} are emitted in increasing timestamp order at the arrival of e_{10}^{13}.

Note that e_3^3 is not handled successfully by the buffer and it still results out-of-order event in the buffer output. For the sliding window aggregate operator that consumes the buffer output, at the arrival of e_3^3, the aggregate result of the window ending at timestamp four has already been produced. Namely, e_3^3 fails to contribute to the result of that window.

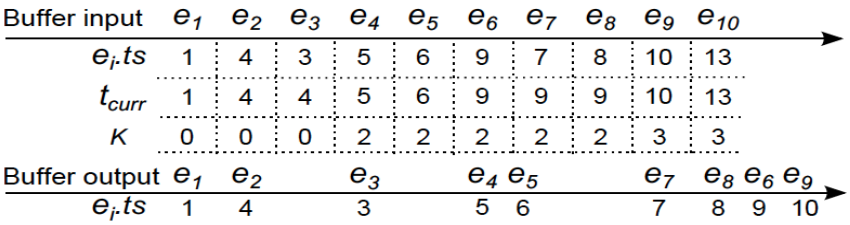


Figure 2.6: Out-of-order event arrival handling in MP-K-Slack [3].

Despite the unsuccessfully handled late arrivals such as event e_3^3, MP-K-slack in general aims to adjust the buffer size to a big-enough value to accommodate all late arrivals. While such dynamic buffer size re-configuration provides increased it comes at the expense of high result latency.

2.3.3. AQ-K-Slack Approach

AQ-K-Slack is a buffer based, quality-driven disorder handling approach. This leverages techniques from the fields of sampling-based approximate query processing and control theory [4]. AQ-K-Slack is an extension of MP-K-Slack, where AQ-K-Slack dynamically increases the buffer size like MP-K-Slack. Furthermore, it can also decrease the buffer size while preserving the user-specified threshold for relative errors in produced query results. Due to its nature of adjusting the input buffer size dynamically to minimize the result latency AQ-K-slack requires no a priori knowledge of disorder characteristics of event streams and imposes no changes to the query operator implementation or the application logic.

In this approach, each sliding window aggregate query is annotated by the user with a result relative error threshold (ε_{thr}, δ) (i.e., Probability ($\varepsilon < \varepsilon_{thr}$) $\leq \delta$). To adhere to this threshold AQ-K-Slack introduces a new parameter α ($\alpha \in [0, 1]$) on top of the buffer size k (effective buffer size becomes αk) to determine the appropriate buffer size. Here α is initialized to one and dynamically adapted at query runtime.

In this approach when late event arrives, if it still falls under the prevailing window-slide range then the event will be considered, else it will be discarded. Consequently, the condition for releasing an event e_i from the buffer changes as follows:

$$e_i.ts + \alpha k \leq t_{curr} \tag{2.2}$$

Figure 2.7 depicts the overall design of AQ-K-slack, which consists of several components. The Buffer Manager maintains the buffer for sorting the input events. The Proportional-Derivative (PD) Controller determines the value of α. Window Coverage Threshold Calculator, translates a user-specified result relative error threshold $(\varepsilon_{thr}, \delta)$ to a window coverage threshold θ_{thr}. Window Coverage Runtime Calculator measures the window coverage of produced query results. Statistics Manager collects the required statistics for Window Coverage Threshold Calculator and Window Coverage Runtime Calculator.

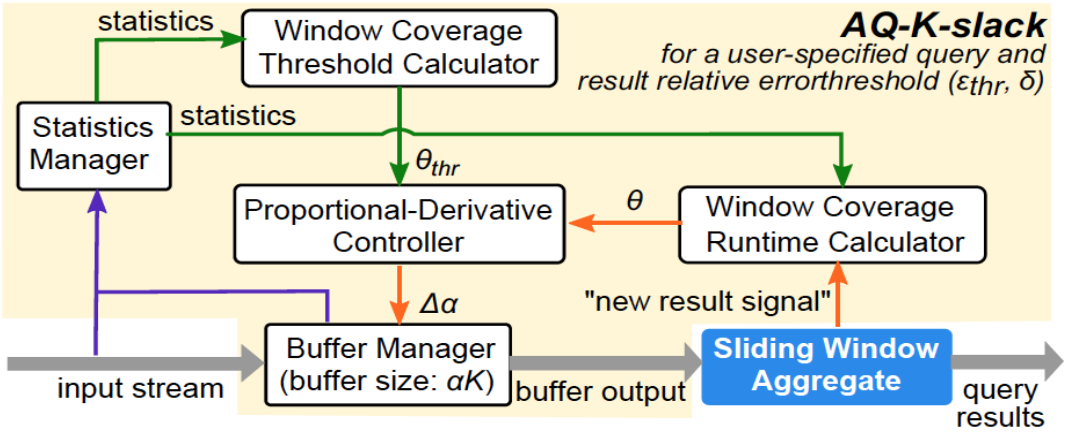


Figure 2.7: Architecture of AQ-K-Slack [4].

The quality of the produced query results is measured using an intermediate metric called window coverage θ. The θ of a query result is defined as the ratio between the number of events that contribute to the result and the total number of events that fall into the corresponding window of the result. The window coverage threshold (θ_{thr}) is derived from user-specified result relative error threshold (ε_{thr}, δ), which determines the expected quality of the output. Based on the deviation of the θ and θ_{thr}, the value of α of determined.

However, adjusting the buffer size directly based on relative errors of query results is expensive; because computing the relative error of a result requires the corresponding exact result, which cannot be obtained before receiving all late arrivals falling into the corresponding window of the result. Therefore, sampling-based approximate query processing is used to compute the query result of a window before receiving all late arrivals falling into the window and handle the out-of-order events. Further, by leveraging statistical inequalities and the central limit theorem, an error model is determined for the results of an aggregate query under sampling. This error model relates the sampling rate p to the relative error ε in a query result. Therefore, given a relative error threshold, the minimum sampling rate required to meet the threshold is derived. As the applied sampling rate determines the proportion of events within each window to be retained for processing, it also influences the window coverage. Therefore, with a minimum sampling rate, the minimum window coverage threshold (θ_{thr}) is also derived.

In sampling-based approximate query processing, the events are excluded from the window; therefore, the window coverage is determined only by the applied sampling rate. Hence, that approach cannot be used to calculate the runtime window coverage (θ) where it depends on both current applied buffer size (αk) and the out-of-order event arrival characteristics. Furthermore, the measured window coverage of a result gets more and more accurate over time, since the probability to receive all late arrival is increased. Therefore, the old window coverages are more accurate than the latest calculated window coverage, hence the old window coverage is considered to adapt the buffer size in AQ-K-Slack. The result produced before the maximum delay that has been seen so far (i.e., value of k) are assumed to be stable but this will lead to delayed adaptation of the buffer. Therefore, to reduce the delay and increase the accuracy in adaptation and remove the effect of abnormal spikes in the delay, the window coverage of the result that is produced L time units earlier than t_{curr}, where L equals to the q-quantile ($0 < q < 1$) [13] of delays of previously received late arrivals.

Figure 2.8 illustrates α adaptation solution which provides the adaptivity to this methodology. The output of the controller is the adjustment which is the increase or the decrease of α, denoted by $\Delta\alpha$. For the i-th measured window coverage $\theta(i)$ sent to the controller. The controller determines $\Delta\alpha(i)$ based on the control error, *err(i)* which is defined as the deviation between the measured window coverage $\theta(i)$ and the calculated window coverage threshold θ_{thr}, i.e., $err(i) = \theta_{thr} - \theta(i)$. The value of $\Delta\alpha(i)$ is determined by two components of the control error such as a proportional component (P), which corresponds to the present error, and a derivative component (D) which corresponds to a prediction of future errors. The value of $\Delta\alpha(i)$ is a weighted sum of the two components, where the weights are configured via parameters K_p and K_d as shown in Equation 2.3.

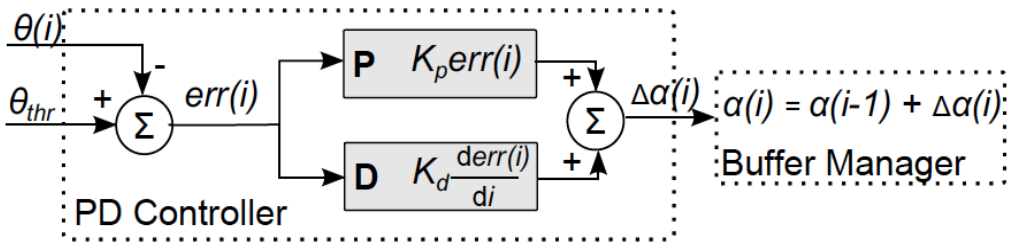


Figure 2.8: Adoption of α using a PD controller [4].

$$\Delta\alpha(i) = K_p\, err(i) + K_d\,(\mathrm{d}\ err(i) / \mathrm{d}i) \qquad (2.3)$$

Manual parameter tuning was used for PID controllers on top of the well-known Ziegler-Nichols method [14] to configure parameters K_p and K_d, so that they can minimize the resulting buffer size while respecting the user-specified result accuracy requirement.

However, this approach has the following disadvantages:

1. The performance of AQ-K-Slack is limited by the parameter setting for K_p and K_d of the PD controller, and it is currently performed manually.
2. This approach is well suited for aggregation operators but it is not well suitable for the functions such as pattern matching or join. Because the user-defined

error threshold can be only provided for the aggregate operations and not for the other operations mentioned above.

3. Delayed adaptation of the buffer size, as it considers the old window coverage of the result into consideration to determine the α.

2.3.4. Latency Distance and Purging Time Based out-of-order Event Processing (LDOP)

In this approach, the latency distance calculation and then purging time is used to handle out-of-order event arrival [15]. Consider a mixed event stream S <S_1, S_2, …, S_m>. The latency distance of S in time window w, denoted by LD (S, w), is defined as follows:

$$LD(S,w) = \frac{1}{N}\sum_{S_i \in S}\sum_{e_{ij} \in S_i}(e_{ij}.ats - e_{ij}.ts) \tag{2.4}$$

where e_{ij} denotes an out-of-order event in event stream S_i in time window w and N represents the number of out-of-order events from S in w.

Latency distance reflects the average delay between occurrence timestamps and arrival timestamps of all out-of-order events from S in time window w. The purging time of an event e_{ij}, denoted by $PT(e_{ij})$, is specified in Equation 2.5, where $|w|$ denotes the size of time window w and SF represents the slack factor which is a random variable uniformly chosen from a slack range. The value of SF can be adjusted according to the size of the available memory.

$PT(e_{ij})$ reflects the time when e_{ij} must be purged from the memory keeping event streams:

$$PT(e_{ij}) = e_{ij}.ts + |w| + SF \times LD\ (S,w) \tag{2.5}$$

Suppose that a mixed event stream S <S_1, S_2, …, S_m> is generated from heterogeneous networks and then transmitted to the event processing system where out-of-order arrivals of events may exist. out-of-order arrivals of events together with the limited memory space make the event processing system have to quickly purge those

unnecessary events for pattern matching by calculating the latency distance and purging time (called LDOP). This method first filters out those event instances whose event types do not appear in the pattern matching plan. Then, it calculates the latency distance based on Equation 2.4, and it calculates the purging time for each event instance based on Equation 2.5. Finally, it decides on whether an event instance must be purged from the buffer if the event was kept in the buffer more than the calculated purging time.

The disadvantage of this approach is that it is well suited for pattern matching query but not for the other query operators such as aggregate operators. Because with pattern matching operator, only the incoming events that are specified in the pattern sequence will be buffered and the rest of the events will be discarded. This filtering cannot be done with aggregate operators as all the events are required for the actual event processing. Therefore, with aggregate and join operators, the memory usage will be high compared to the pattern matching query. This results in a low value of the slack factor for non-pattern matching query operators, and hence the purging time will be less. This indeed results in the increase of unsuccessfully handled out-of-order events.

2.3.5. K-Slack Chain Approach

Streaming systems often run multiple queries concurrently and may exploit sharing opportunities across the concurrent queries. Under such shared query execution, stream-sorting buffers can be shared across queries as well, which can potentially reduce the overall memory cost incurred by the sorting buffers. Quality Driven Disorder Handling (QDDH) approach handles out-of-order events for concurrent queries which analyzes the same incoming event stream and share the stream-filtering operators [16]. In a join function which operates on a window of events, may produce an incomplete result set, if the window on one input stream has moved forward while there are delayed events from other event streams which are yet to arrive. Waiting for those delayed events can improve the final result quality (on accuracy or completeness), but at the same time increases the result latency. In this approach, the focus is on join and aggregate queries with sliding time windows. Also, it assumes that

the selection predicates that are common in different queries are evaluated by shared filters - a commonly used technique for sharing computations across concurrent queries in both database systems and streaming systems. The query operator network, after exploiting all sharing opportunities among concurrent queries' selection predicates, can be modelled by a Directed Acyclic Graph *G*, which is called as a global query graph. An example global query graph is shown in Figure 2.9.

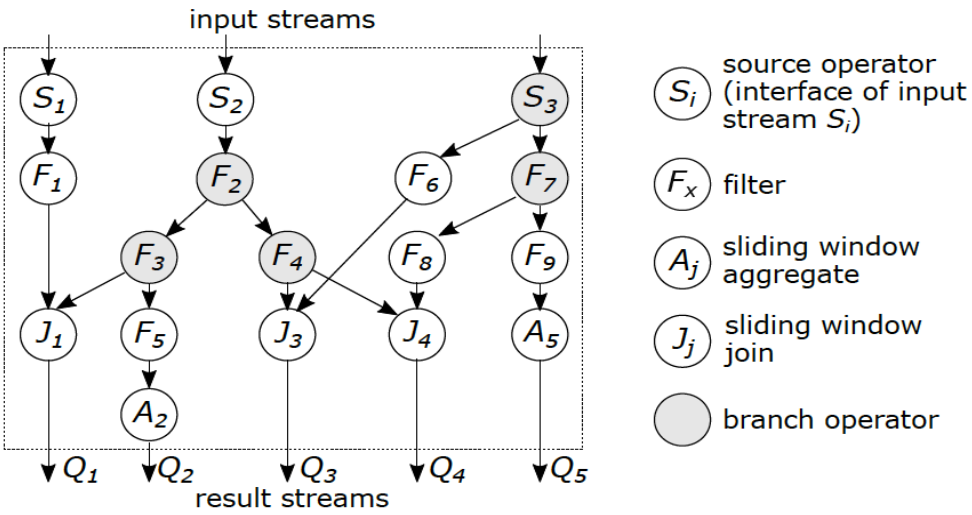


Figure 2.9: A global query graph [16].

Given a global query graph G and the user-specified quality requirements of the result for each query covered by G, the goal of QDDH is to minimize the result latency of each query, while subjecting the result quality of this query to its user-specified quality requirement.

A natural solution to achieving this goal is to have unshared event stream buffers. Namely, for each query *Q*, stream-sorting components K-slack buffers are placed right before the windowed join or aggregate operator in *Q* (i.e., right above leaf nodes in *G*); and sizes of these buffers are dynamically adjusted in a quality-driven way, independent of any other queries. However, such unshared QDDH may result in duplicate storing and sorting of the same data, wasting memory resources. To illustrate this, suppose that selectivity of *F8* and *F9* in Figure 2.13 are both 0.9 (i.e., 90% of the

events are filtered). Then, with unshared QDDH, at least 80% ($2 \times 0.9 - 1 = 0.8$) of the output events of F7 would be handled by both the buffer for query *Q4* and the buffer for query *Q5*. In the case that *F7* has a high output rate or a large fan-out degree, the memory waste could be enormous. One natural solution to avoid such duplicate disorder handling and saving memory resources are shared QDDH, namely, sharing K-Slack buffers across multiple queries. For the above example, it means placing a buffer below *F7*, rather than below *F8* and *F9* redundantly. As a result, the output of *F7* is sorted only once, and the sorting effort is shared between *Q4* and *Q5*. In general, shared disorder handling can happen right below any branch operator (i.e., an operator that has more than one child) in *G*, and even concurrently right below multiple branch operators. However, for a branch operator, sharing the disorder handling of its output does not always lead to lower memory consumption. For instance, when the selectivity of *F8* and *F9* in Figure 2.9 are both 0.1, then shared disorder handling right below *F7* has higher memory consumption than unshared disorder handling right below each of *F8* and *F9*. Moreover, naive sharing of K-Slack buffers may unnecessarily increase the result latency of queries that have low result-quality requirements, whereas smart sharing has a higher overhead of runtime adaptation when switching to a new configuration of the buffers. Hence, when doing QDDH for concurrent queries that have shared query operators, it is not obvious how to place K- Slack buffers within the global query graph, so that the goal of QDDH is achieved with minimum memory consumption. This is prevalent when the global query graph contains a large number of branch operators.

K-Slack chain allows performing shared disorder handling at a branch operator in *G*, without violating the goal of latency minimization for each query that shares the operator. For ease of illustration, let us consider a simple sub-graph *G(S1)* in Figure 2.10(a), which contains only one branch operator *S1* and does not contain filters. Assume that the optimal QDDH buffer sizes of queries in *G(S1)* satisfy $k3 < k1 < k2$. Let *v* denote a query operator in general, *Q(v)* denote the set of queries that share *v*, and *r* denote the average data rate of S_1.

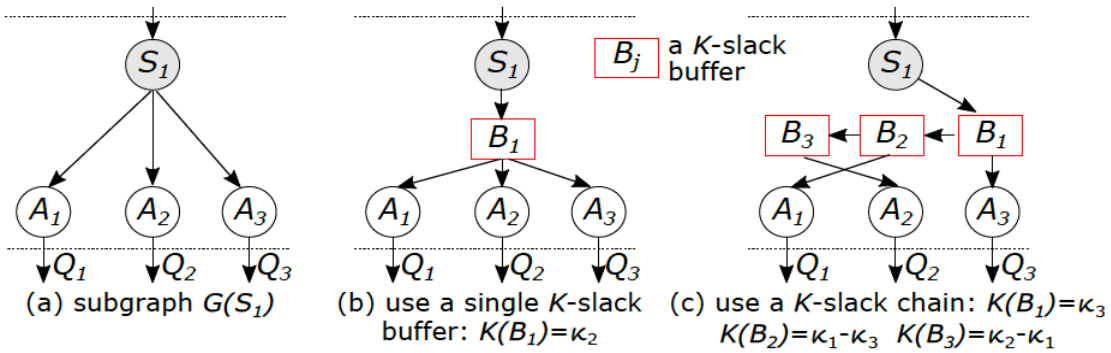


Figure 2.10: Shared disorder handling for a query graph G that has no filter operators: single K-Slack versus K-slack chain. (Assume that $k_3 < k_1 < k_2$) [16].

For the branch operator *S1* in *G(S1)*, unshared disorder handling of its output stream has a memory cost of $r(k_1 + k_2 + k_3)$, and shared disorder handling can reduce this memory cost. A naive solution of shared disorder handling is to place a single K-Slack buffer below the source operator *S1* (see Figure 2.10(b)). To meet the user-specified result quality requirement of each query, the size of this buffer (i.e., *K*(*B1*) in Figure 2.10(b)) must not be smaller than the greatest value among k_1, k_2, and k_3, which is k_2 in this example. However, this means that a larger than necessary buffer is applied for queries *Q1* and *Q3*, which violates the latency minimization condition.

As shown in Figure 2.10(c), to overcome the above drawback of a single K-Slack buffer, a chain of K-Slack buffers can be used. This approach is based on the property of the K-Slack algorithm: the disorder-handling effect of a single K-Slack buffer of size k is equivalent to that of a chain of K-Slack buffers whose accumulated size is k. In Figure 2.10(c), the size of the first buffer in the chain is set based on the smallest value among queries in $Q(S_i)$, i.e., $K(B_1) = k_3$. Output events of B_1 are forwarded to both operator A_3 and the second K-slack buffer B_2, so that they can be processed by A_3 without being delayed further, and at the same time, it can be handled further by B_2 to meet the result-quality requirements of the other two queries. The size of B_2 is $K(B_2) = k_1 - k_3$, i.e., the difference between the second smallest value and the smallest value among $\{k_i | i = 1, 2, 3\}$. The output of B_2 is forwarded to both operators A_1 and buffer B_3. The last buffer B_3 has $K(B_3) = k_2 - k_1$, i.e., the difference between the greatest value

and the second greatest value among $\{k_i | i = 1, 2, 3\}$; and its output is forwarded to A_2 only. The total size of buffers in the K-Slack chain in Figure 2.10(c) is $(k_2 - k_2) + (k_1 - k_3) + k_3 = k_2$. Hence, it has the same, which is also the optimal, memory cost as the single K-Slack buffer in Figure 2.10(b). However, it overcomes the drawback of the single buffer and satisfies the latency-minimization condition.

As this approach uses AQ-K-Slack to determine each buffer size, the disadvantages mentioned in the AQ-K-Slack also applies here as well. Also, in this case, as the buffers are shared among operators, the operator level optimizations such as filtering and only buffering the events that are specified in pattern matching, cannot be applied.

2.3.6. Summary

Table 2.2 provides a summary of each buffer-based approaches discussed, where each of them has its own advantages, as well as disadvantages. K-Slack approach adds a fixed amount of delay to all events to handle out-of-order event events. MP-K-Slack improves the K-Slack approach by finding the buffer size in runtime without requiring a priori fixed delay. However, the MP-K-Slack approach could only increase the buffer size; therefore, any burst event delays will affect the latency for all the following events. This limitation was overcome by the AQ-K-Slack approach with the ability to reduce and increase the buffer size based on the window coverage, but this approach is only applicable to aggregation operators.

Table 2.2: Comparison of buffer-based approaches.

Buffer-based Approach	**Advantages**	**Disadvantages**
K-Slack	Provides a fixed amount of delay to handle delayed event arrival.	Hard to find a suitable buffer size *k* for the system. Inability to adapt to the changes in the operating environment. Added overhead to all events though the events are received in order. Events are released from the buffer only when an event with largest timestamp is arrived.
MP-K-Slack	No requirement of finding a priori buffer size *k*. Ability to increase the buffer size *k* based on delay in event arrival.	Overly large buffer size *k* assigned can incur unbearable latency. Inability to reduce the buffer size *k,* when the network properties improve. Less frequent event delays could dominate the latency for all events and hence the impact is high. Events are released from the buffer only when an event with largest timestamp is arrived.
AQ-K-Slack	Ability to reduce and increase the buffer size *k*. Provides the guarantee on the user-defined error threshold for the final result produced.	Manual parameter tuning of K_p and K_d which determines the performance. Not suitable for operations such as join and pattern matching. Delayed buffer size adaptation. Events are released from the buffer only when an event with largest timestamp is arrived.
Latency Distance and Purging Time based out-of-order Event Processing	Improved memory utilization. Well suited for pattern matching operations.	Not suitable for aggregate operations where all events are required for processing. Out-of-order event processing cannot be successfully handled with limited memory.
K- Slack Chain	Ability to handle out-of-order event arrival from multiple event streams. Less memory consumption for multiple streams.	Operator-level optimizations are not considered. Uses AQ-K-Slack to find the buffer size; therefore, disadvantages mentioned in AQ-K-Slack applies here as well.

3. METHODOLOGY

Out-of-order event arrival can be experienced within the event stream from both a single event source and across multiple event sources. This is more prevalent in cases where multiple event sources are publishing within a Local Area Network (LAN) or Wide Area Network (WAN) with different time drifts among them. Furthermore, given out-of-order arrival, there could be different requirements for accuracy and latency based on the use cases. As summarized in Table 2.2, all current approaches add the latency to not only to the delayed events but also to in-order events after a late arrival, and those do not address the problem of events produced from multiple sources where the drift in timestamps are common and not synced. Also, when event sources produce events at a high rate (e.g., in IoT domains), larger buffers are required to handle out-of-order event arrival.

In this chapter, we discuss the proposed approach where the latency is introduced to the event only when an event is delayed. In Section 3.1 relevant definitions are described. The details of the proposed solution are provided in Section 3.2. Further, Section 3.3 discusses the implementation details of the system. Finally, Section 3.4 provides the overall summary of the proposed methodology.

3.1. Definitions

This Section defines the elements in the event stream and the attributes associated with the out-of-order event handling problem. First, we define the elements of the problem, such as the event sources, event, event stream, and timestamps associated with event processing. We then explain the proposed solution for the out-of-order event handling problem which includes handling out-of-order events from the single event source, multiple events sources, and effect in query operators due to out-of-order event arrival.

The event sources generate the data that are published to CEP as events to be analyzed. The event sources could be edge devices such as IoT sensors, mobile phones, smart TVs, and wearable devices, to high-end computer systems and network systems. Let

us denote the set of event sources that publish events to an event stream as **I**, which consists of *n* event sources **I** = {S_1, S_2, S_3, ..., S_n} where each of them independently collects the events and transmits to the CEP engine.

Event Streams are the grouping of the same type of continuous events that are received by the CEP engine. One event source can produce one or more event streams. Let **R** denote the set of event streams and there can be *m* event streams such that **R** = {R_1, R_2, R_3, ..., R_m}, for which the events can be generated from any number of event sources in **I**.

Events are the collection of attributes such as actual payload of the data, the name of the data stream it belongs to, and the generated timestamp. Let the set of events be denoted as **E** = {E_1, E_2, E_3, ..., E_k} which belong to one of the streams in **R**.

We consider the following four main types of timestamps:

1. Source time – The timestamp in which the event was generated at the source. This indicates the actual event occurring time. Let source time of event *e* of event source *s* be denoted as t_e^s.
2. Reference time – The timestamp at CEP engine when the event is originated at the event source. The difference between this timestamp and source time is that the reference timestamp is time at CEP engine, and source time is the time at event source when the event is generated. Let us denote this timestamp as t_e^{ref}. The time difference between the t_e^{ref} and t_e^s is the time drift between the CEP server and event sources.
3. CEP arrival time – The time which the event is received by the CEP engine and excludes CEP processing time. Let us denote this as t_e^r. This timestamp will include the any delays added to the event from to t_e^s.
4. CEP start time – The timestamp of the event in which the event has been admitted into the CEP engine for further processing. Let us denote this timestamp as t_e^p, i.e., $t_e^p > t_e^r > t_e^s$.

3.2. Proposed Solution

Next, we discuss the proposed solution to solve the out-of-order handling problem in CEP. We propose to address this problem for the events generated from both single and multiple event sources. The former problem can be handled by adding a sequence number to each event which is used by the CEP engine to put events back in order. This is discussed in Section 3.2.1. For the latter problem of ordering the events from multiple event sources, we use the timestamp drift of each event sources to find the estimated global order of events. In Section 3.2.3, we discuss how the query operators can operate on the event stream that is received from multiple event sources while reducing any anomalies that are left untreated after the approach proposed in Section 3.2.2.

3.2.1. Handling Events Produced from Single Event Source

Out-of-order events within a single event source can be handled to a certain extent in the transport level if the events that are sent with TCP or another reliable protocol, where the order of the events is preserved. However, if the event source is producing a very high number of events, then the event source could use multiple parallel connections with the connection pool to send the events. Hence, in-order delivery is not guaranteed by the transport protocol. Further, if we consider distributed processing at CEP, then the events will be processed separately and will be collected at a node which expects the events to be in order, to produce the final results. To solve this problem, we can add a sequence number for the events produced by the event source. This sequence number is local to the event source and the event stream. Figure 3.1 shows how the events are buffered and passed to the CEP engine for further processing by the query operators.

The events will be ordered based on the sequence number of the events (labeled by the source) at the CEP event receiver component which consumes events just after the transport socket and before passing the events to the query operators. Once an event is received, CEP event receiver will check the sequence number, and an event is released to CEP engine only if it matches the next expected sequence number. Any event with

a higher sequence number will be added to a buffer and will be released only when the expected sequence number is received. For example, let us assume the events from event source D_1 are received in the order $E_1, E_2, E_4, E_5, E_6, E_3, E_7$. In that case, up to E_2 the events are received in order; hence, those will be passed to the next level in the CEP. The next expected event is E_3, but the received event is E_4. Therefore, events E_4, E_5, E_6 will be buffered without passing to the next stage. Once expected event E_3 arrives, it will be moved immediately to the next stage for further processing, and subsequently the next expected events E_4, E_5, E_6 will be released in order immediately following E_3.

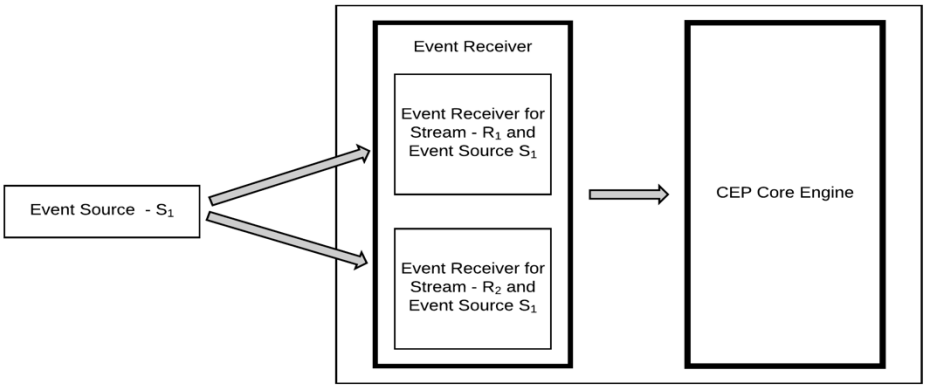


Figure 3.1: Event source S_1 sending event stream R_1 and event stream R_2 to CEP.

To handle the event loss in the network, there is also a wait timeout, so that the CEP engine can continue without waiting indefinitely. The timeout value $t_{timeout}$ can be calculated based on the following parameters (list of symbols is given on Table 3.1).

1. Inter-arrival time between in-order events (t_{inter})
2. Time spent on the buffer (t_{buffer})
3. User configured maximum latency threshold ($t_{latency-th}$)

The first parameter t_{inter} can be calculated by combining average t_{avg} and mean deviation t_{dev} of the inter-arrival time between in-order events as shown in Equation 3.1. We use mean deviation (*mdev*) than standard deviation (*sdev*) because it is a good approximation to standard deviation and easier to calculate [17]. Also, for most of the

common distributions the factor goes closer to 1 ($mdev = \sqrt{\frac{\pi}{2}}\ sdev$, $\sqrt{\frac{\pi}{2}} \approx 1.25$) as discussed in [17] for calculating TCP timeout for retransmission.

Table 3.1: List of symbols.

Symbol	Description
t_e^s	Timestamp in which event *e* was generated at event source *s*
t_e^{ref}	Timestamp at CEP engine when event *e* is originated at event source *s*
t_e^r	Time which event *e* is received by CEP engine and excludes CEP processing time
t_e^p	Timestamp of event *e* in which events has been admitted into CEP engine for further processing
t_{inter}	Inter- arrival time between in-order events
t_{avg}	Average inter-arrival time between in-order events
t_{dev}	Mean deviation of inter-arrival time between in-order events
$t_{current}$	Current timestamp of the event
$\propto$	Smoothing factor used to find t_{avg} and t_{dev}
t_{buffer}	Time spent on the buffer
$t_{buffer-avg}$	Average time spent on the buffer
$t_{buffer-dev}$	Mean deviation of time spent on the buffer
$t_{current-buffer-delay}$	Duration that the current event is held in the buffer
β	Smoothing factor used to find $t_{buffer-avg}$ and $t_{buffer-dev}$
$t_{latency-th}$	User configured maximum latency threshold
T_i	Round trip time of an event source in *i*-th time sync iteration
t_i^d	Time drift of an event source in *i*-th time sync iteration
T_s	Round trip time of event source *s*
t_s^d	Time drift of event source *s*

$$t_{inter} = t_{avg} + 2t_{dev} \quad (3.1)$$

The inter-arrival time of events are measured and monitored for each event source at the CEP event receiver component. For the calculations of t_{avg} and t_{dev}, the last arrived in order event timestamp of that particular source (t_{last}), and the current in-order event timestamp $t_{current}$ is stored. Then t_{avg} and t_{dev} will be calculated based on Exponential smoothing as in Equation 3.2 and Equation 3.3 where $\propto$ is a smoothing factor. We opted to use the exponential smoothing because we will be using the calculated time values to adjust the timeout of the events arriving early. Therefore, we need to give more weightage to the recent time measurements, than older measurements which can be set via α.

$$t_{avg} = (\propto t_{avg}) + (t_{current} - t_{last})\ (1 - \propto)) \quad (3.2)$$

$$t_{dev} = (\propto t_{dev}) + \left|[\ (t_{current} - t_{last}) - t_{avg}]\ (1 - \propto)\right| \quad (3.3)$$

Initially t_{avg} and t_{dev} will be initialized to zero, and the hence the first t_{avg} will be calculated as shown in Equation 3.4.

$$t_{avg} = (t_{current} - t_{last}) \quad (3.4)$$

Once the first t_{avg} is calculated, the t_{dev} can be calculated with the next in-order event, and first t_{dev} will be calculated as shown in Equation 3.5.

$$t_{dev} = (t_{current} - t_{last}) - t_{avg} \quad (3.5)$$

Second parameter t_{buffer} , can be deduced by combining both average $t_{buffer-avg}$ and mean deviation $t_{buffer-dev}$ of the time interval that the early events was held in the buffer as shown in Equation 3.6.

$$t_{buffer} = t_{buffer-avg} + (2\ t_{buffer-dev}) \quad (3.6)$$

Initial value of t_{buffer} will be zero. Any events arriving early (i.e., the expect next sequence number x did not arrive but the event with sequence number $x + n$ ($n > 1$) has arrived) will be buffered. Once the event with the expected sequence number is

received, the buffered events will be released based on the sequence number order. Therefore, when releasing those buffered events, the average time $t_{buffer-avg}$ and mean deviation time $t_{buffer-dev}$ that the events was held in the buffer can be calculated using Equation 3.7 and Equation 3.8 respectively, where $t_{current-buffer-delay}$ is the time that the particular event was held in the buffer and β is buffered delay smoothing factor.

$$t_{buffer\text{-}avg} = (\beta\ t_{buffer-avg}) + (t_{current-buffer-delay}\ (1 - \beta)) \tag{3.7}$$

$$t_{buffer\text{-}dev} = (\beta\ t_{buffer-dev}) + \left|[(t_{current-buffer-delay} - t_{buffer-avg})\ (1-\beta)]\right| \tag{3.8}$$

In the initial stage of the system t_{buffer} will be initialized to zero, and the first t_{buffer} will be calculated as:

$$t_{buffer-avg} = t_{current-buffer-delay} \tag{3.9}$$

Once the first $t_{buffer-avg}$ is calculated, the $t_{buffer-dev}$ can be calculated when clearing the buffer next time. Hence, the first $t_{buffer-dev}$ will be be calculated as shown in Equation 3.10.

$$t_{buffer\text{-}dev} = \left|[\ (t_{current-buffer-delay}) - t_{buffer-avg}]\ \right| \tag{3.10}$$

The final timeout $t_{timeout}$ can be calculated as:

$$t_{timeout} = \text{Min}\ (t_{latency-th}, \text{Max}\ (t_{inter}, t_{buffer})) \tag{3.11}$$

Based on Equation 3.11, when the event inter-arrival time is large, the user-defined latency threshold dominates for the timeout calculation. However, it is fair to assume that out-of-order events are rare for longer event inter-arrival streams as they have sufficient time to tolerate network delays and distributed CEP processing. Therefore, using user-defined latency threshold for timeout in such cases will not bring any negative impact to the solution.

This technique is suitable for continuous stream of high traffic events, where the legitimate events will not be having any additional overhead and will be just passing

through the system. Only the events which arrive early will be held back to match with other event sources. In related work such as MP-K-Slack and AQ-K-Slack events will be held in the selected size of buffer even though the event stream is in order.

3.2.2. Handling Events Produced from Multiple Event Sources

Next, we focus on handling out-of-order event arrival among multiple event sources. It is reasonable to assume that the events are ordered within a single event source using the sequence-based approach proposed in Section 3.2.1. To handle this problem, we propose an approach where the CEP event receiver estimates the time drift between the CEP receiver and actual event sources and then ordering the events from multiple event sources based on this estimate.

In this approach, the time synchronization will be performed between the event receiver and the CEP engine. Event receiver will order events from each event source based on sequence number per event stream, and then pass the events to the Synchronizing Component. *Synchronizing Component* is responsible to order the events from multiple event sources per event stream and release global ordered event stream to the query operators. Each event streams will be handled by different threads in the Synchronizing Component. As event sources could be in different time zones without using standard time and the clocks may be out of sync, we cannot order the events based on the timestamp t_e^s by the event source. However, to find the global order of the event sequence, we need to map the timestamp of the events from different sources to a single base clock. As all the event sources send events to the CEP engine, events could be ordered by taking the CEP engine's clock as the reference. Thus, each event's timestamp can be mapped to t_e^{ref}, which is the timestamp at the CEP engine at the time of event generation at source. For this, we need to calculate the clock drift between the CEP receiver and each event source and store them within the CEP receiver.

Therefore, we use an approach similar to the Network Transfer Protocol (NTP) time synchronization [18] to calculate the round trip delay and the time drift between the

event source and CEP receiver. Here, the CEP server will act as an NTP server, and the event source will act as an NTP Client, and it uses UDP transport to calculate the time drift. We calculate the round trip delay and time drift for each event source and use that to correct the incoming event's timestamp in the CEP server, without changing the event source's local clock.

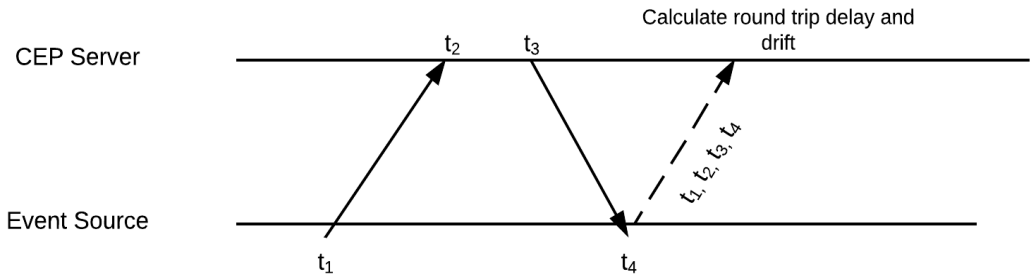


Figure 3.2: Communication between the CEP receiver and event source to calculate the timestamp drift.

The time sync process will be initiated first by the event source before transmitting the actual events. During the time synchronization process, CEP Server and Event source will record the timestamps at the arrival and departure of the time sync packets similar to NTP [18] as shown in Figure 3.2. Therefore, in the *i*-th time synchronization attempt, the round trip time T_i is calculated as per Equation 3.12, and time drift t_i^d is calculated as per Equation 3.13 [18].

$$T_i = (\mathrm{t}_4 - \mathrm{t}_1) - (\mathrm{t}_3 - \mathrm{t}_2) \tag{3.12}$$

$$t_i^d = [(t_2 - t_1) + (t_3 - t_4)]/2 \tag{3.13}$$

Equation 3.13 assumes that network delays are a stationary random process, but practically stochastic network delays are common, and the network queues can grow and shrink in a chaotic fashion. Therefore, the accuracy of the time drift can be impaired and hence the bounds of the actual time drift t_s^d can be as Equation 3.14 [18]:

$$t_i^d - \frac{T_i}{2} \leq t_s^d \leq t_i^d + \frac{T_i}{2} \tag{3.14}$$

Based on Equation 3.14, we can claim that the true time drift of the event source t_s^d, must lie in the interval of size equal to the measured delay and centered about the measured time drift.

Event source will send a series of time sync requests to increase the accuracy of the round trip delay and time drift calculation. And then we use NTP data filtering method - *minimum filter* [18] in CEP server, which selects the time sync request with lowest delay T_i to calculate the final round trip delay, and time drift (T_s, t_s^d) for among all the time synchronization requests.

Once the time sync process is complete, the event source will start to publish the actual events. And hence when a new event arrives, the calculated drift t_s^d for its event source, can be added to the event timestamp t_e^s to find the reference timestamp t_e^{ref} of that event. For example, assume there are three event sources, S_1, S_2, and S_3 which publishes for event stream as shown in Figure 3.3. The respective calculated time drifts are t_{s1}^d, t_{s2}^d, and t_{s3}^d. Therefore, the CEP's synchronizing component will be adding the timestamp drift of the source to each of its events and compare with all events from all sources and provide an estimated global order per event stream as shown in Figure 3.3 and Equation 3.15.

$$t_e^{ref} = t_e^s + t_s^d \tag{3.15}$$

There could be also local clock drift at event sources and CEP with time; hence, we also need to periodically re-calculate the drift for each event source. This could be achieved by repeating the same process which could update the record drift maintained at the event receiver. The period in which the calculation can occur can be set during the event source initialization.

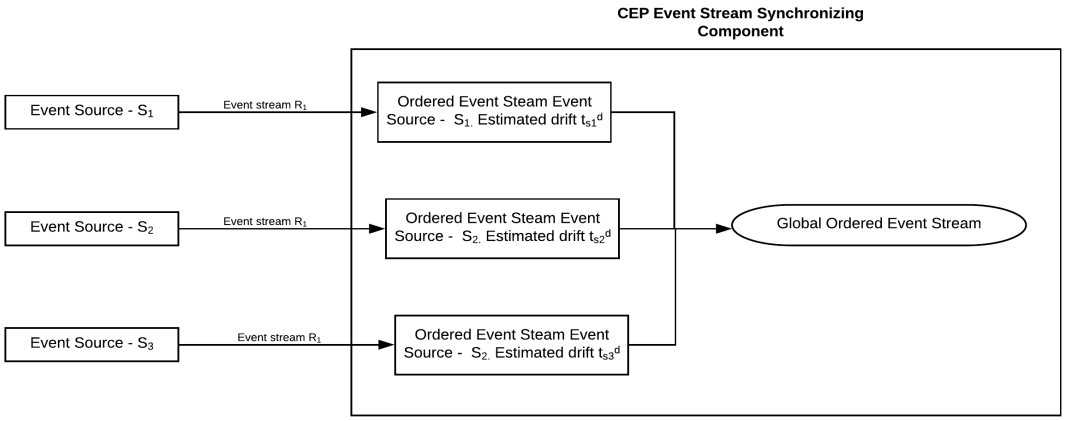


Figure 3.3: Out-of-order event handling with multiple event sources for event stream R_1.

As shown in Figure 3.3, the ordered events received from each source is globally ordered by the Synchronizing Component. Each event stream is handled by a separate thread in the Synchronization Component, which also corrects the time based on the estimated drift. As shown in Figure 3.3, the events of an event stream are already sorted based on the sequence number. Therefore, the synchronizing component will fetch the first event (i.e., the lowest sequence and timestamp event) of each event sources that publish to the particular event stream. The reference timestamp of the fetched events will be calculated by adding the estimated time drift as explained in Equation 3.15. The event which has the lowest reference timestamp among all fetched events will be released to the next process. Then, the component will fetch the next event from the same event source that the last released event belongs to. It then calculates the reference timestamp for the newly fetched event and continues the same comparison and releasing process.

However, not all event sources will have events during the synchronization process. Therefore, if one of the event sources does not have an event during the synchronization process, then the component will wait until the timeout defined for that event source as per Equation 3.11. If an event arrives for that particular event source before the timeout, then it will be fetched and used in the comparison and releasing process along with other event sources. But in case, if the event did not arrive

within the timeout, then the synchronization component will flag the event source as timed out. This flag will be down once a new event arrives for the source. When the flag is up for a particular source, the synchronization component can skip it without waiting, e.g., the event source may have gone offline or stopped publishing events.

However, overall with this approach, the event source with the highest out of order event distribution dominates the latency incurred in the events whereas, for larger event inter-arrival event sources, the user-defined latency threshold dominates.

3.2.3. Query Operators with Out-of-order Events

Once the out-of-order events are ordered as per the mechanisms discussed in Section 3.2.1 and 3.2.2, the ordered events are presented to the query operators to perform the actual query execution on those events. Due to timeout-based buffers, a few events may still be out-of-order; hence, need to be handled further. Next, we focus on two query operators such as aggregation operator and pattern operator, which depend on the order of the event arrivals when processing within a window of events.

3.2.3.1. Time Batch Window and Aggregation Operators

Based on the multiple source out-of-order approach, we calculate the time drift t_s^d for each event source, and event timestamps are adjusted accordingly. As mentioned in Equation 3.14, the calculated drift t_s^d can vary $\pm(T_s/2)$. In that case, there exists an ambiguous range of events in a window which may or may not actually correspond to that window. Therefore, we cannot have a window which has the size less than the T_s because all the events that are included on the window will be in the ambiguous range.

Let us consider T_a as the maximum round-trip time of all event sources.

$$T_a = Max(T_s) \tag{3.16}$$

Let *w* be the windows length in time. As seen on Figure 3.4, the events from t to $t - \left(\frac{T_a}{2}\right)$ and from $t - \left(\frac{T_a}{2}\right) + w$ to $t + w$ are in the ambiguous range. Similarly, the events

from $t - \left(\frac{T_a}{2}\right)$ to t and from $t + w$, and $t + \left(\frac{T_a}{2}\right) + w$ maybelong to the time window, though that is out of the window time. Therefore, when query operators are used along with the window of events, there are five different kinds of events produced at the end of the window, they are:

1) Events between the $t - \left(\frac{T_a}{2}\right)$ and t.
2) Events between t and $t + \left(\frac{T_a}{2}\right)$.
3) Events between $t + \left(\frac{T_a}{2}\right)$ and $t - \left(\frac{T_a}{2}\right) + w$.
4) Events between $t - \left(\frac{T_a}{2}\right) + w$ and $t + w$.
5) Events between $t + w$, and $t + \left(\frac{T_a}{2}\right) + w$.

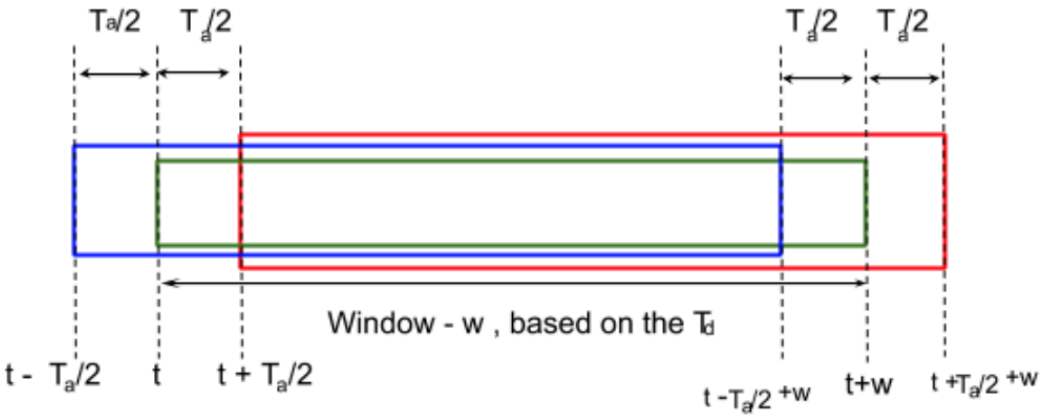


Figure 3.4: Multiple time batch windows to cover the events from ± $(T_a/2)$ of original time batch window *w*.

As per Figure 3.4, the events between time $t + \left(\frac{T_a}{2}\right)$ and $t - \left(\frac{T_a}{2}\right) + w$ corresponds to the window w. But the events from $t - \left(\frac{T_a}{2}\right)$ to $t + \left(\frac{T_a}{2}\right)$ and events from $t - \left(\frac{T_a}{2}\right) + w$ to $t + \left(\frac{T_a}{2}\right) + w$ may or may not belong to this window. Therefore, given the knowledge about different kind of events passed to the query operators, users can write their CEP queries to accommodate this behavior. Hence, we can have three time batch windows that range from $t - \left(\frac{T_a}{2}\right)$ to $t - \left(\frac{T_a}{2}\right) + w$, from t to $t + w$, and from $t + \left(\frac{T_a}{2}\right)$ to $t + \left(\frac{T_a}{2}\right) + w$. This multiple window can be used for the aggregation calculation to

reduce the inaccuracies that could have been introduced by the time drift calculation. Therefore, we can perform the aggregations for three windows of events $t - \left(\frac{T_a}{2}\right)$ to $t - \left(\frac{T_a}{2}\right) + w$, t to $t + w$, and $t + \left(\frac{T_a}{2}\right)$ to $t + \left(\frac{T_a}{2}\right) + w$ and emit the average of all three windows as the final aggregated results, which will be more accurate than considering the single estimated window t to $t + w$.

This method reduces impact due to any inaccuracies in the time drift calculation and increases the accuracy of the query result after the window and aggregation operation. However, the disadvantage of this method is that the users have to be aware of this condition and rewrite the query to make use of the three time windows, as this will not be handled automatically within the CEP engine.

3.2.3.2. Pattern Matching

The order of the events significantly influences the pattern matching decision. Therefore, once the events are ordered with the sequence-based approach, we can use the pattern matching operator directly to detect the pattern. However, as mentioned in Section 3.2.3.1, due to the uncertainty in calculating the exact drift there will be a certain ambiguous time range of the event in which it could have actually occurred. Therefore, we need to consider the time difference of the events which contributed to the pattern matching to evaluate its correctness of the general pattern matching decision. Let us consider an example, where e_1 and e_2 as two events and the actual pattern to detect is event e_1 followed by event e_2. The pseudocode to detect the pattern in this case that handles the inaccuracies in drift is shown in Pseudocode 3.1.

As shown in Pseudocode 3.1, we try to detect both combinations of pattern e_1 followed by e_2, and e_2 followed by e_1. In both cases, we evaluate the time difference of the event occurrences, and if the time difference is less than time T_a mentioned in Equation 3.16, then we can mark it as *Uncertain pattern detection* because it could be actually correct decision or false positive. If it is greater than T_a, then we can be certain about our final decision, and we can conclude as the pattern is detected or not.

```
if e1 -> e2,
      if(e2.timestamp - e1.timestamp <= T_a )
            then
                  PRINT 'Uncertain Pattern Detection'
            else
                  PRINT 'Confirmed Pattern Detection'

if e2 -> e1,
      if(e1.timestamp - e2.timestamp <= T_a)
            then
                  PRINT 'Uncertain Pattern Detection'
            else
                  PRINT 'Pattern Not Detected'
```

Pseudocode 3.1: Pattern detection code for pattern event e1 followed by e2.

The main advantage of this method is that it makes sure that there are no false negative pattern decisions; therefore, it is very suitable for domains such as finance, health care, and fraud detection, where false negative is not acceptable at all. Since the pattern matching result is provided with additional information about the uncertainty, the final decision can be taken by the end users based on the business context of the pattern matching query. The disadvantage of this method is that the users will have to rewrite the pattern matching query by utilizing the time drift calculation, and it is not handled within the CEP engine automatically. If the pattern has more events, then the modified pattern matching query will be more complex.

3.4. Summary

This Chapter discussed the terminology and proposed a solution to handle out-of-order events. Our proposed approach contains three primary steps, that includes ordering the events for single sources based on the sequence numbers, ordering the events with multiple sources with time drift calculations, and finally presenting the events to window and query operators. In our proposed approach, the incoming events are initially ordered based on the sequence numbers which are injected into the events by the event sources, and then the ordered events of a single source are added to the queue for multiple source synchronization. The time drift is calculated at the time of event source initialization by using the NTP like technique, and it is used to calculate the reference time of the event received based on its event source ID. Further, this

calculated reference timestamp is used to order the events among multiple sources. As there could be irregularities in the calculation of the reference time of a source, we have proposed multiple window approach where additional two windows are maintained in the difference of the max delay time of all sources for the event stream. This information needs to be utilized explicitly when writing the aggregator and pattern matching queries. The aggregator operators can be written in a way that it can utilize multiple windows and increase the accuracy of the final result. Similarly, pattern matching query can also be written to consider max delay time into its final pattern matching decision and increase its accuracy of the decision with an additional attribute to specify about the confidence level of the pattern matching result.

4. IMPLEMENTATION

To demonstrate the proposed solution, it is implemented on the WSO2 CEP Siddhi engine [19]. Siddhi is selected as it is open-source, has low latency, and capable of analyzing millions of events per second [19]. The main reasons for selecting Siddhi CEP are:

1. Known to consume minimum resources such as memory and CPU.
2. It has throughput high as 30 Million events/sec and latency ranges from 1 ms to 130 ms for most of the cases such as pass-through, filter query, and time windows [19], [20].
3. Have multiple extension points such as Stream Function Extension, Stream Processor Extension, and Window, Aggregate and Custom Function Extensions [21], [22].
4. Easy-to-use and Open Source CEP engine under Apache Software License v2.0

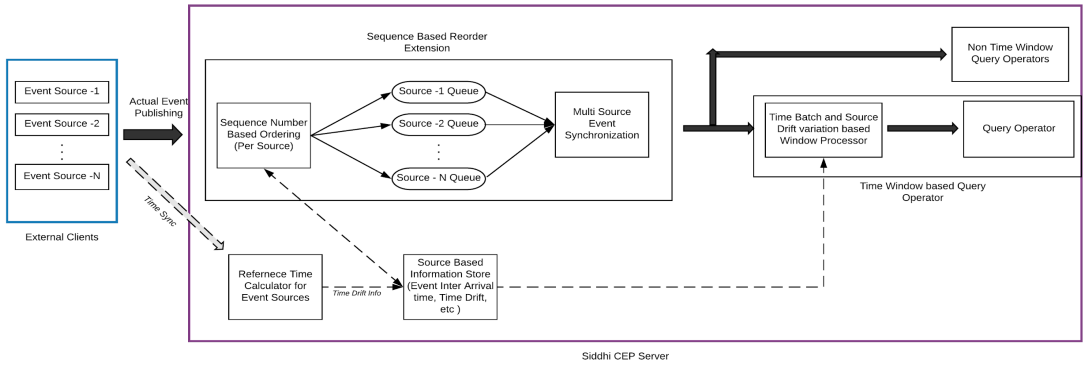


Figure 4.1: Out-of-order event flow of an event stream within siddhi engine.

Figure 4.1 shows the main components and the event flow to handle out-of-order events of an event stream in Siddhi engine. Note that the entire process is executed per event stream by a different thread, and there are no resources shared among multiple event streams including the event buffers and queues. *Sequence-Based Reorder Extension* is the heart of solution where out-of-order events are ordered by sequence

numbers for a single event source, and reference timestamps for multiple event sources for a particular event stream. *Source-based Information Store* holds the information about the time drift, inter-arrival time and time spent on buffer per event stream. And this information is used by *Sequence-based Reorder Extension* to find the global order of the events in the event stream from multiple sources, and *Source Drift Variation based Window Processor* to maintain multiple windows as explained in Section 3.2.3.1. If the user-defined CEP query has time batch window query, then the output events from *Sequence-based Reorder Extension* will be first passed to the *Source Drift Variation based Window Processor*, and then the events expired from the windows are presented to the query operator such as aggregator operators to produce the actual result of the query for events within the time window. If the user-defined CEP query does not involve any time batch window operators, then the output events from *Sequence-based Reorder Extension* will be directly fed into the query operators such as pattern matching, filter queries, and pass through queries.

As depicted in Figure 4.1, the following steps are followed when handling the out-of-order events in Siddhi CEP Engine:

1. As a first step, the event sources will send initialization *Time Sync* events to Siddhi CEP Server along with its source ID to calculate the source drift and reference timestamp t_e^{ref} as mentioned in Section 3.2.2. Once the initialization is successfully performed, the event source will start to publish its actual event stream.
2. The actual events are passed into the *Sequence-based Reorder Extension*, that orders the events based on the sequence number of each source ID and synchronizes the events across all the event sources based on the reference timestamp t_e^{ref} as mentioned in Equation 3.12.
3. If the user has used time batch window as the CEP query, then the output events from the *Sequence-based Reorder Extension* are fed into the *Source Drift Variation based Window Processor.* Else, the output events from the *Sequence-based Reorder Extension* are directly fed into the intended query operators (e.g., pattern matching) of the defined CEP query.

4. If the events are fed into the *Source Drift Variation based Window Processor,* then it maintains multiple windows to cover the events from ± $(T_a/2)$ of original time batch window *w* as mentioned in Section 3.2.3.2. Once the events are expired from time batch window, those are presented to the query operators (e.g., aggregator operator) to produce the final results as discussed in Section 3.2.3.1.

Next, we explain the detailed implementation on top of WSO2 Siddhi CEP engine. We use TCPNettyClient [23], [24] to publish events into an event stream in a remote CEP server. But this client does not have the time syncing process that we discussed in Section 3.2.2. Therefore, the new client was implemented by extending the existing TCPNettyClient [23] to include the time syncing process. The client can be initialized as shown in Sample Code 4.1.

```
TCPNettyClient tcpNettyClient = new TCPNettyClient();
      //Host: 10.100.1.138
      //Actual Event Receiver Port:9892
      //Time Sync Receiver UDP Port: 7452
      //Source ID: "Sensor1"
      //Number of Time Sync Attempts: 10
tcpNettyClient.connect("10.100.1.138", 9892, 7452, "Sensor1", 10);
```

Sample Code 4.1: Extended TCPClient initialization with time syncing.

A new *UDP (User Datagram Protocol) Time Sync Server* was implemented to receive the time sync events in the Siddhi CEP server. It calculates the time drift based on the event source and updates the *Source-based Information Store* as shown in Figure 4.1. This data is later used when reordering events in the *Sequence-based Reorder Extension* and *Source Drift Variation based Window Processor.* Once the initialization is successful, the event sources can send the actual events. The event stream defined to handle out-of-order events should consist of additional attributes to specify the source ID of the event source and sequence number of that event. For example, Query 4.1 and 4.2 introduce additional attributes *sourceId* and *seqNum* to handle the out order events. Because the stream definition consists of additional attributes *sourceId* and

seqNum, the events published to the event stream from event sources also should consist of the same data.

```
define stream inputStream (timestamp long, temperature
double);
```

Query 4.1: Stream definition without out-of-order handling.

```
define stream inputStream (timestamp long, temperature double,
sourceId string, seqNum long);
```

Query 4.2: Stream definition with out-of-order handling.

Once the actual events are received, those are passed to the *Sequence-Based Reorder Extension* which is the main component of the out-of-order event handling. This is developed by extending the Stream Processor extension without modifying the core Siddhi CEP code. The stream can be enabled with this reorder extension as shown in Query 4.3.

```
from inputStream#reorder:sequence(sourceId, seqNum, timestamp,
10L, 0.6, 0.6, false)
select timestamp, temperature
insert into inOrderStream;
```

Query 4.3: Query with sequence based reorder extension.

There are mainly seven configurable parameters for this extension as indicated in Query 4.3, namely:

1. Attribute name of the event stream definition that denotes the source ID field *sourceId*
2. Attribute name of the event stream definition that denotes the sequence number field *seqNum*.
3. Attribute name of the event stream definition that denotes the timestamp field. If this is not provided, then the event's default timestamp is taken
4. The user-defined timeout in ms and this will be used to control the maximum time the events are kept in the buffer.

5. The value for smoothing factor $\propto$which can be a decimal number between 0 to 1. This value is used to calculate event inter-arrival time as explained in Section 3.2.1.
6. The value for smoothing factor β which can be a decimal number between 0 to 1. This value is used to calculate the duration of early events spent in the buffer as explained in Section 3.2.1.
7. The Boolean parameter to configure whether to drop or pass late arrival events after the timeout.

Once the event is received by the *Sequence-based Reorder Extension*, the flow illustrated in Figure 4.2 is executed to order the events based on sequence numbers per event source. When an event is received, the event's sequence number (*seqNum*) is checked against next expected sequence number (*expSeqNum = last seen in-order sequence number + 1*) for the event source. If it is equal, the event is passed to multi-source synchronization, and subsequently, release all buffered events that follow the current event's sequence number. If the sequence number is greater than expected sequence number, then it will put the event into the buffer. If the sequence number is less than the expected sequence number, then based on the user-defined configuration of the *Sequence-based Reorder Extension,* the event will be dropped or put into multi-source synchronization immediately. The in-order event arrival time and buffered event time are calculated (Section 3.2.1) respectively when an event arrived in order (i.e., *seqNum == expSeqNum*), and when the buffered events are released after the expected sequence number has arrived. These values are used to calculate the timeout as explained in Section 3.2.1 when an event is added into the buffer. Once the events are ordered for each event sources, the events are added to the source specific event queues for multi-source event synchronization.

Once the events are added into the source specific queues, those are picked up by multi-source synchronization, which orders the events based on t_e^{ref}. Effectively it collects the smallest t_e^{ref} event from all event queues and releases it to the next process. It then polls the next event from the same queue which had the smallest t_e^{ref} and

continue the same iteration. In case, if the event queue is empty, then it will wait till the same timeout calculated by Equation 3.11 then continue the synchronization process.

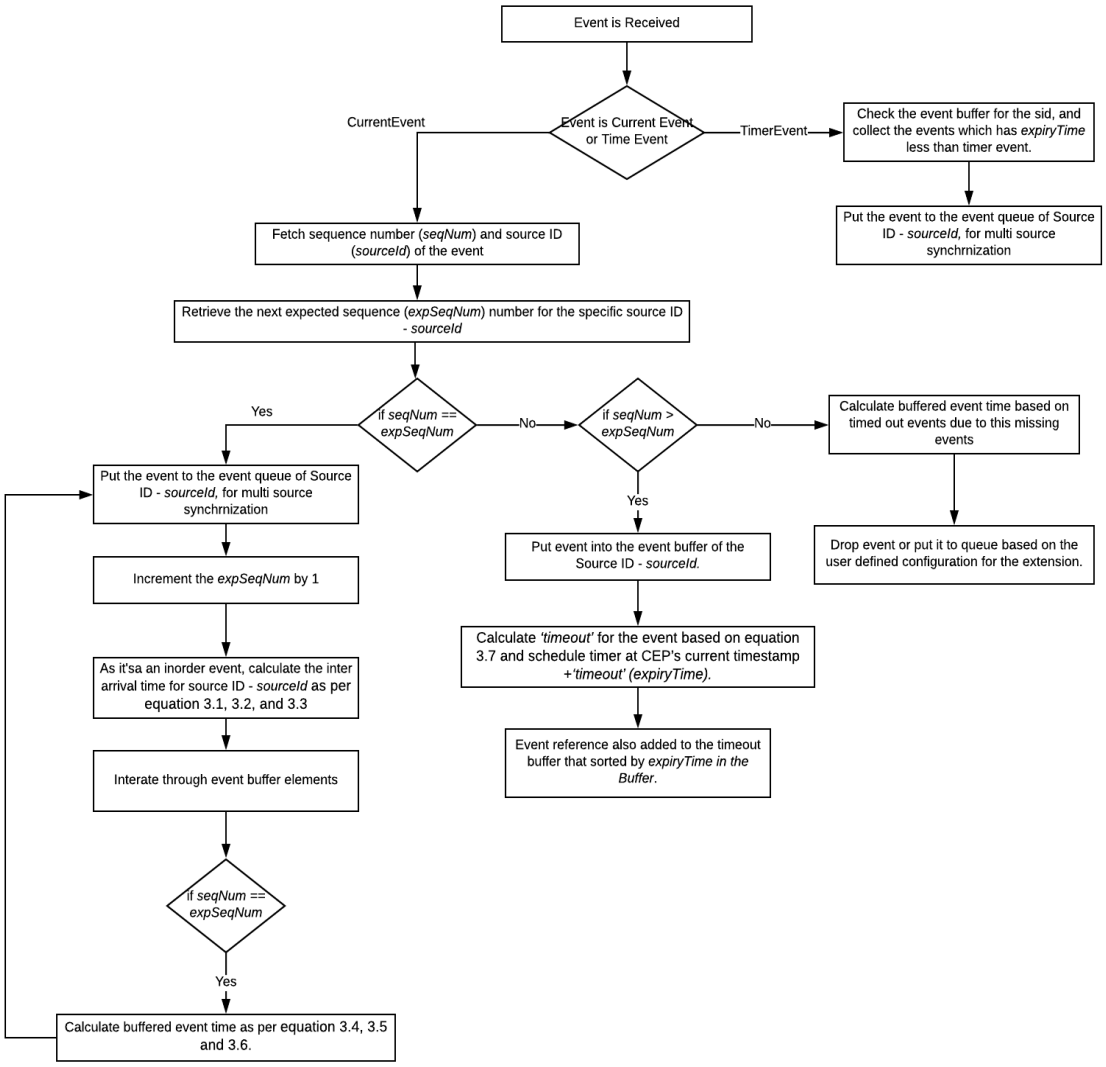


Figure 4.2: Sequence-based ordering in sequence-based reorder extension.

If the user-defined CEP query consists of time batch window, then in the ordered event stream is passed to the *Source Drift Variation based Window Processor*. This custom window processor [22] implementation will retrieve T_a as per Equation 3.16 from the *Source-based Information Store*, and will maintain three different windows as follows:

1. Low window – The window events from event from $t - \left(\frac{T_a}{2}\right)$ to $t - \left(\frac{T_a}{2}\right) + w$.
2. Middle window – The window events from event from t to $t + w$.
3. High window – The window events from event from $t + \left(\frac{T_a}{2}\right)$ to $t + \left(\frac{T_a}{2}\right) + w$.

The window can be defined as in Query 4.4 along with the *Sequence-based Reorder Extension*.

```
from inputStream#reorder:sequence(sourceId, seqNum, timestamp,
10L, 0.6, 0.6, false)#reorder:externalTimeBatch(timestamp, 1
sec, false)
select timestamp, temperature
insert into inOrderWindowedStream;
```

Query 4.4: Reorder time batch window.

The window can have three different configurations as follows:

1. The timestamp field name in which the external time batch window should operate, i.e., *timestamp* field.
2. Time window Size (e.g., 1 sec).
3. Boolean parameter to configure whether immediately emit the windowed events as the time elapsed. The Low window will be elapsing first and the High window will be elapsing last. And this configuration controls whether the events of the window needs to be released immediately when that specific window is elapsed, or to hold those are release together at time $t + \left(\frac{T_a}{2}\right) + w$. According the Query 4.4, this parameter set to false.

The expired events from the time window will have an additional attribute as *windowType* to specify which type of time window that the event belongs to.

Now we will focus on using the query operators after the time window. Let us consider the sum operator as a sample query operator, and with the above window of events that can be used as shown in Query 4.5. The individual sum is calculated for each window of events, and the average of those are considered into the final output, which reduces the effect of irregularities in calculating the event source's time drifts.

```
from inputStream#reorder:sequence(sourceId, seqNum, timestamp,
10L, 0.6, 0.6, false)#reorder:timeBatch(1 sec, false, false)
select
(sum(ifThenElse(windowType=='LOW',price, 0.0f))+
sum(ifThenElse(windowType=='MIDDLE',price,0.0f))+
sum(ifThenElse(windowType=='HIGH',price, 0.0f)))/3 as
totalPrice
insert all events into outputStream;
```

Query 4.5: Sample siddhi query to calculate average of aggregated results among different windows.

If the user-defined CEP query does not consist of time batch window, the output events from the *Sequence-based Reorder Extension* will be directly fed into the query operators without passing through the *Source Drift Variation based Window Processor*. Pattern matching is one type of query operator which does not require to be used along with time batch window. Now let us focus on using the pattern matching query operator with out-of-order event arrival with Siddhi CEP Engine.

For pattern matching, we have to consider the actual intended pattern and the other combinations of the event sequence of the pattern and incorporate all those into the pattern matching query. Once the pattern is detected we will have to compare the time difference of the events that participated in detecting the pattern, with the max delay ($T_a/2$) of the event sources, and decide whether the pattern detection is confirmed, or uncertain as explained in Section 3.2.3.2. Therefore, we have implemented a new function *maxDelay* to retrieve the max transport delay of the event sources that publish events to the intended original event stream. Let us consider an example of detecting the pattern where an event e_1 which has temperature greater 30 is followed by an event e_2 which has temperature greater than 35 in the event stream *inOrderStream*. Query 4.6 shows how the pattern-matching query operator can utilize the *maxDelay* function and produce the result with additional field *confidence* as explained in Section 3.2.3.2.

```
from every e1=inOrderStream [temperature > 30]
-> e2=inOrderWindowedStream [temperature > 35]
select ifThenElse(e2.relativeTimestamp -
e1.relativeTimestamp<=reorder:MaxDelay('inputStream')
```

```
,'UncertainPatternDetected', 'Confirmed') as confidence,
e2.temperature, e2.timestamp
insert into patternStream;

from every e2=inOrderStream [temperature > 35]
-> e1=inOrderWindowedStream [temperature > 30]
select ifThenElse(e1.relativeTimestamp -
e2.relativeTimestamp<=reorder:MaxDelay('inputStream')
,'UncertainPatternDetected', 'Not Matched') as confidence,
e2.temperature, e2.timestamp
having confidence == 'UncertainPatternDetected'
insert into patternStream;
```

Query 4.6: Pattern matching query with *maxDelay* function.

Though we are interested in single pattern (e_1 followed by e_2), Query 4.6 consists of two pattern matching queries. It is because, as we discussed in Section 3.2.3.2, the relative order of the events cannot be accurately decided if the time difference between them is less than *maxDelay*. Therefore, we write the pattern matching query for both event sequence, e_1 followed by e_2 (first part of Query 4.6) and e_2 followed by e_1 (second part of Query 4.6). In both pattern matching cases, we check the time difference of the events in the pattern with *maxDelay* and add the *confidence* attribute to express the level of confidence about the pattern detection. As the first part of Query 4.6 consists the actual pattern, if the event time difference is less than or equal *maxDelay,* then the *confidence* attribute is set to *UncertainPatternDetected,* else it is set to *Confirmed*. Since the second part of the Query 4.6 consists of the alternate event sequence of the actual pattern, if event time difference is less than or equal *maxDelay,* then the *confidence* attribute is set to *UncertainPatternDetected,* else it is set to *NotMatched*. Finally, with the second part of the Query 4.6, we only trigger the pattern detection if the *confidence* attribute is equal to *UncertainPatternDetected*, and ignore pattern detected with *NotMatched.*

4.1. Summary

This Chapter discussed the implementation details of the proposed methodology. We have implemented many custom extensions for WSO2 Siddhi CEP Engine, to implement the proposed methodology. A new UDP based *Time Sync Server* is developed to calculate the time drift and round trip time as explained in Section 3.2.2.

Also, a new *TCP-Netty-Client* is implemented to include the time syncing process which transmits a series of UDP time sync messages before transmitting the actual events stream. The event sources should use the new implementation of the *TCP-Netty-Client* and add two additional attributes to specify the event sequence number and the source ID when publishing each event. The *Sequence-based Reorder Extension* is the custom extension developed to handle the out-of-order event arrival, and it is the core part of the implementation. The events received for an event stream are passed into this extension as the first step before it is processed by any query operators. This extension orders events from both single and multiple event sources. The *Source Drift Variation based Window Processor* is custom window extension implemented to handle the multiple windows as explained in Section 3.2.3.1, and it is used when the user had defined the time batch window in the CEP query. We also discussed how Siddhi queries for aggregation and pattern matching operators can be modified as discussed in Section 3.2.3.1, and 3.2.3.2.

5. PERFORMANCE EVALUATION

In this Chapter, we will evaluate the performance of the proposed sequence based reordering approach. Section 5.1 presents the dataset and how it is used for the evaluation. Section 5.2 discusses the experimental setup and hardware configurations. Section 5.3 compares the performance of the proposed solution with other reordering techniques such as AQ K-Slack and MP-K-Slack. Section 5.4. presents in-depth information on accuracy and latency.

5.1. Dataset

The datasets used for this evaluation were derived from the football dataset used in DEBS (Distributed Event-based Systems) 2013 Grand Challenge [25]. The data is collected by the real-time locating system for ball and players, that is deployed on a football field of the Nuremberg Stadium in Germany and it contains 47 Million rows of events. Data originates from sensors located near the players' shoes (single sensor per leg) and in the ball (single sensor). The goalkeeper is equipped with two additional sensors, one at each hand. The sensors in the players' shoes and hands produce data with 200 Hz frequency, while the sensor in the ball produces data with 2,000 Hz frequency.

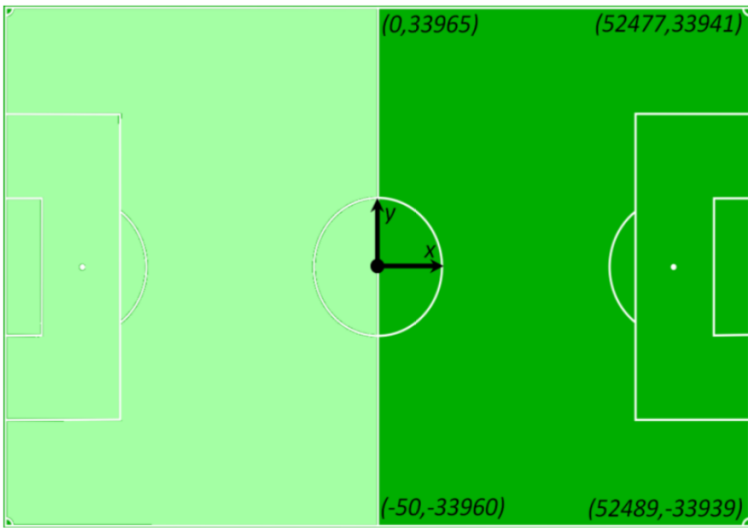


Figure 5.1: Playing field and its dimensions in mm [25].

The total data rate reaches roughly 15,000 position events per second. Every event describes the position of a given sensor in a 3D coordinate system. See Figure 5.1 for the dimensions of the playground and the coordinates of the kickoff. The event schema is given below, and the description of each attributes are provided in Table 5.1.

sid, ts, x, y, z, |v|, |a|, vx, vy, vz, ax, ay, az

Table 5.1: Description of event attributes.

Symbols	Description
sid	Sensor Id- Produced the position event
ts	Timestamp- Defined in picoseconds e.g.: 10753295594424116 (with the value of 10753295594424116 designating the start and 14879639146403495 the end of the game)
x, y and z	Position of the sensor in mm
|v|	Velocity of the ball in µm/s
|a|	Absolute acceleration of the ball in µm/s&sup2
vx, vy and vz	Direction by a vector with size of 10,000 (in m/s)
ax, ay and az	Constituents of absolute acceleration in three dimensions

The original dataset contains the events that are generated from the sensors and ordered by the timestamp. We extracted a smaller dataset of roughly 500,000 events from this original dataset and prepared three different datasets by introducing completely different out-of-order event distributions into them. Table 5.2 shows the total number of events and the number of events that were out-of-order from those.

Table 5.2: Input Dataset and out-of-order events.

Dataset Name	Total Events	No of Out-of-order Events
Dataset 1	499982	62473
Dataset 2	499499	62409
Dataset 3	999499	83212

The distribution of latencies introduced in out-of-order event dataset 1, 2, and 3 are shown in Figure 5.2 to 5.5.

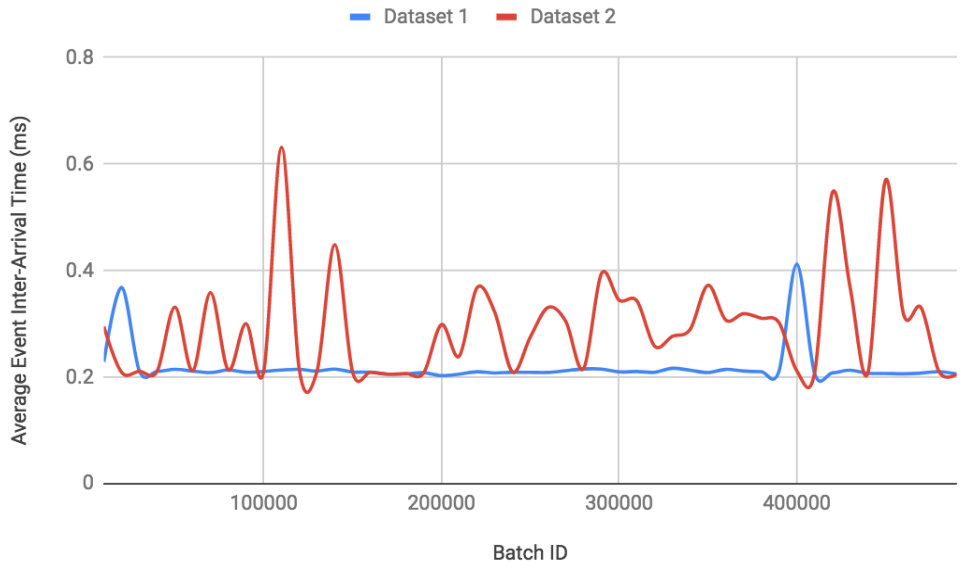


Figure 5.2: Average events inter-arrival time with out-of-order event Dataset 1 and Dataset 2.

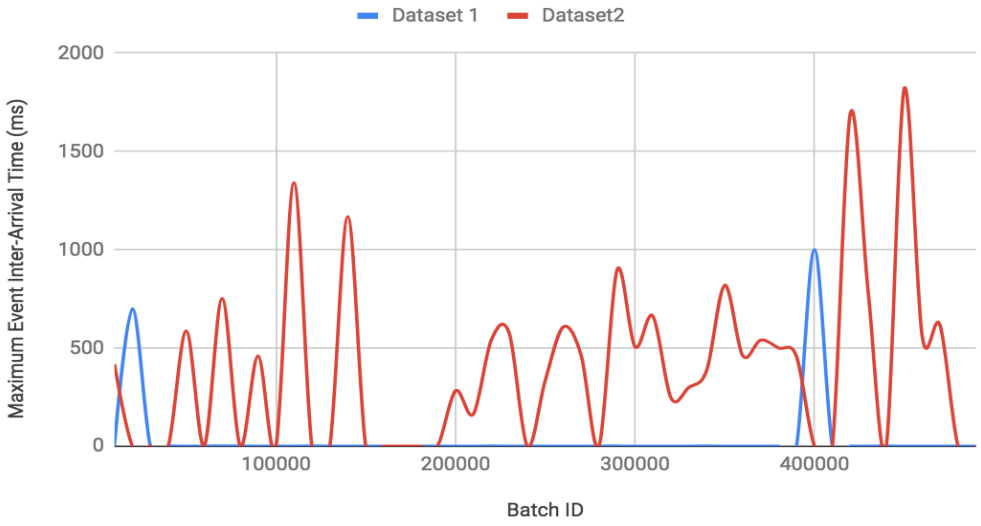


Figure 5.3: Maximum events inter-arrival time in Dataset 1 and Dataset 2.

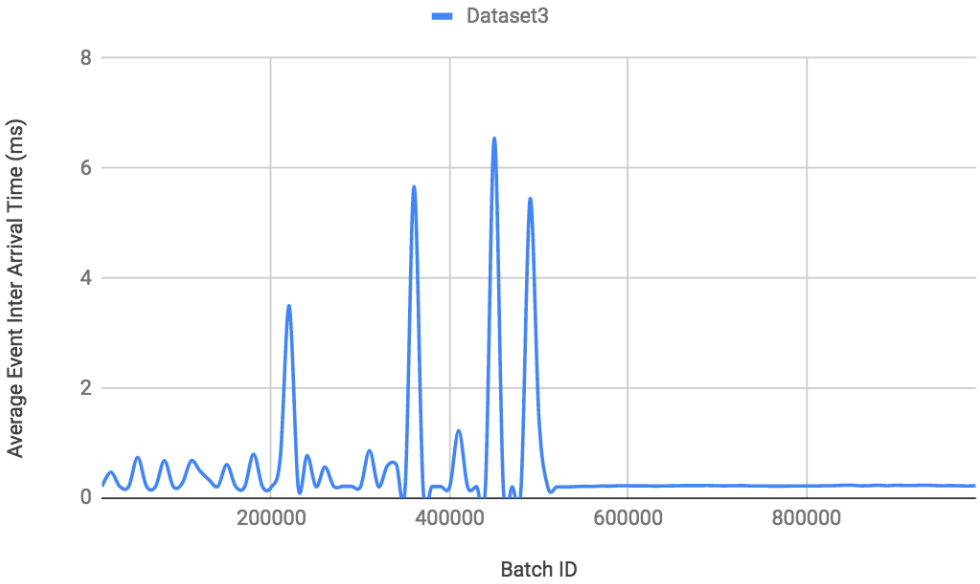


Figure 5.4:Average events inter-arrival time with out-of-order event Dataset 3

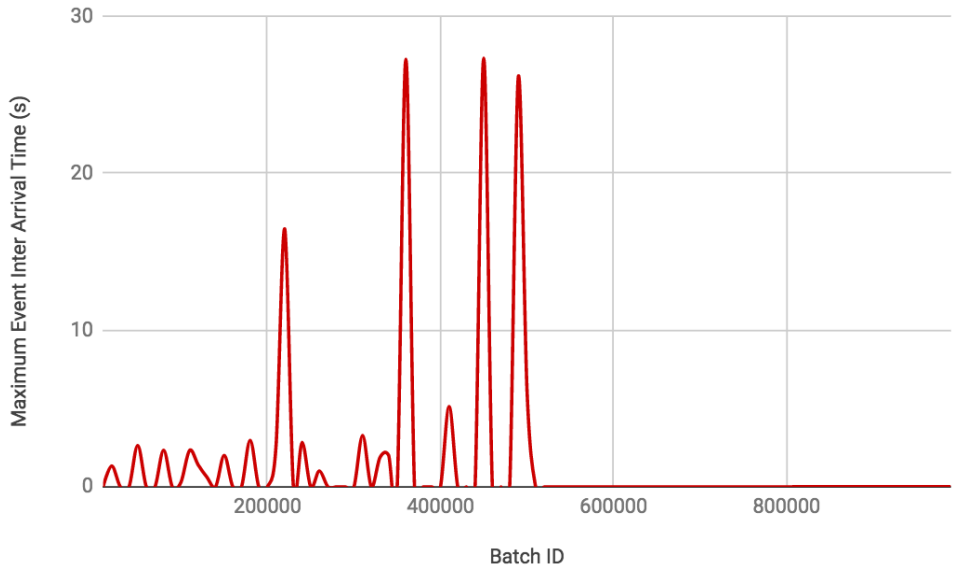


Figure 5.5: Maximum event inter-arrival time in seconds for Dataset 3.

Dataset 1 has mainly two significant late arrivals and average late event inter-arrival time is between 0.1 to 0.3 ms. Also, the significant late arrival events in the Dataset 1 were having a max delay from 750 to 1000 ms. Whereas Dataset 2 had a much wider

distribution of out-of-order events and average late inter-arrival time is less than 1 ms. The maximum delay of the out-of-order events were less than 2 seconds. Dataset 3 was created with a larger average late inter-arrival time ranging from 1 ms to 7 ms. Also, the maximum delay of out-of-order events ranges in seconds and goes up to 30 seconds.

5.1.1. Dataset for Single Source

All three datasets were tagged with additional attribute *sourceId* and *seqNum* to denote the identifier of the event source that the event belongs to and the sequence number of the event. For the single source performance evaluation purpose, all three dataset events were added with the same source ID and the sequence number is added in the increasing order of the event's timestamp. The stream definition defined in Query 5.1 was used on these datasets.

```
@source(type='tcp', @map(type='binary'))
define stream inputStream (sid int, ts long, x int, y int, z
int, v_abs int, a_abs int, vx int, vy int, vz int, ax int, ay
int, az int, sourceId string, seqNum long);
```

Query 5.1: Stream definition for datasets.

5.1.2. Dataset for Multiple Sources

The evaluation of the proposed methodology is performed for 2, 5, 10, and 20 event sources. Dataset 3 was used for this evaluation, as it has comparatively large average late inter-arrival time compared to the other two datasets and has relatively wider out-of-order event distribution. Also, we used multiple clients to simulate multiple event sources. Additionally, the actual timestamp of the event was increased by 3 ms with increasing source ID. For example, let us assume an event was published at timestamp t (in milliseconds) with single source, then with multiple sources the source ID 0 will be publishing the same event at timestamp t, source ID 1 will be publishing the same event with timestamp $t + 3$, source ID 2 will be publishing the event at timestamp $t + 6$, and so on. This modification was made to the dataset to make more realistic test

with multi sources and to evaluate the effect of multi-source event synchronization. The stream definition defined in Query 5.1 was used for this evaluation as well.

5.1.3. Dataset with Time Drift for Event Source

The same Dataset 3 was improved to evaluate the effect of time drift. A new field was injected into the dataset as *driftedTs* which will have the (actual timestamp + time drift). Therefore, when an event source is initiated, based on the time drift provided for the evaluation purpose, it will add the new field *driftedTs* to the event, and publish. For example, let us assume *t* is reference timestamp of an event and *d* is the drift for the event source. Therefore, the client will send two timestamp attributes, where *ts* attribute will be having the value *t* and *driftedTs* will be having the value *t* + *d*. Further, the CEP server will be using the *driftedTs* to actually order events between the event sources, and *ts* will be used to evaluate the accuracy of the re-ordering approach. Since we modified the attributes in the original dataset, the stream definition is modified as shown in Query 5.2.

```
@source(type='tcp', @map(type='binary'))
define stream inputStream (sid int, ts long, x int,
y int, z int, v_abs int, a_abs int, vx int, vy int,
vz int, ax int, ay int, az int, sourceId string,
seqNum long, driftedTs long);
```

Query 5.2: Stream definition for Dataset with time drift.

5.1.4. Dataset for Pattern Matching

A new dataset, namely Dataset 4, was generated for pattern matching which has the events in the pattern in out-of-order fashion. The dataset consists of a pattern of goal score event, which is triggered based on the events from sensors located on the ball and on the shoes of the football players. The two events which participate in this pattern matching is having ~44 ms time difference. To simulate the multiple event sources condition with drift, the Dataset 4 is also published similar to the manner explained in Section 5.1.3.

5.2. Experimental Setup

The entire experiment was primarily performed on a single machine with specification listed in Table 5.3. The machine was configured with Oracle JDK 1.8.0_101. All the tests were carried out without having any restriction on memory

Table 5.3: Specification of the machine that was used for the valuation.

Processor Name	Intel(R) Core(TM) i7-4770HQ CPU
Processor Speed	2.2 GHz
Number of Processors	1
Total Number of Cores	4
L2 Cache (per Core)	256 KB
L3 Cache	6 MB
Memory	16 GB
Hyper Threading Enabled	Yes

5.2.1. Prototype

Tests were carried out using the implementation described in Chapter 4. We used Siddhi version 4.2.33 to implement the proposed solution. The Siddhi server is started with extensions and queries, that are required for each experiment. Further, we have created a Java based application to simulate the event sources, that reads from the datasets discussed in Section 5.1. As we required to evaluate the proposed solution with multiple event sources, the simulated client program will spawn different threads to act as different event sources for the Siddhi engine. To simulate real-world condition, the in order events were published with actual intervals that were present in the dataset.

5.2.2. Analysis of Out-of-order Events Handling Solution

Recall that our primary objective is to have high accuracy and low latency for the incoming events while solving the out-of-order events and having a minimal impact on the in-order events. As mentioned in Chapter 4, the implementation was done to

handle the main cases of the out-of-order events with single and multiple event sources. In this Section, we will be focusing on latency and accuracy of the proposed solution, compared to other buffer based techniques such as MP-K-Slack and AQ-K-Slack that was discussed under Section 2.3.2 and 2.3.3, respectively.

As we will be focusing mainly on the latency and accuracy, let us first define how those are calculated.

1. Latency – The latency was calculated as the total time duration an event took to reach the pass-through output stream from the time that event was actually published from the client.
2. Accuracy – A current event is considered as out-of-order, if the last received event is having the timestamp greater than the current event. Therefore, the accuracy is the percentage of the total corrected events out of the total out-of-order events that were injected into the stream.

5.2.2.1. Performance and Accuracy with Single Event Source

In this Section, we analyze the performance and accuracy of the proposed out-of-order handling approach for a single source. The Siddhi query that was executed during this evaluation is shown in Query 5.3, 5.4, and 5.5 for sequence-based approach, MP-K-Slack approach and AQ-K-Slack approach respectively for input stream definition mentioned in Query 5.1.

```
from inputStream#reorder:sequence(sourceId, seqNum, driftedTs,
500L,0.6, 0.6, false)
select sourceId, seqNum, eventTimestamp() as
relativeTimestamp,ts
insert into outputStream;
```

Query 5.3: Siddhi query with sequence based reorder extension

```
from inputStream#reorder:kslack(ts)
select sourceId, seqNum, eventTimestamp() as
relativeTimestamp,ts
insert into outputStream;
```

Query 5.4: Siddhi query with MP-K-Slack extension

```
from inputStream#reorder:reorder:akslack(ts, v_abs)
select sourceId, seqNum, eventTimestamp() as
relativeTimestamp,ts
insert into outputStream;
```

Query 5.5: Siddhi query with AQ-K-Slack extension.

As shown in Query 5.3, we have used 0.6 as the smoothing factor to calculate event inter-arrival time and duration of early events spent in the buffer as defined in Section 3.2.1. This value was producing less latency and improved accuracy for the datasets.

The average and max latencies incurred with different approaches are shown in Figure 5.6 and 5.7. Among all three approaches, the highest latency was experienced when there was a high delay on the out-of-order event as per dataset distribution described in Figure 5.2 and 5.3. As per Table 5.4, we can see that the sequence-based approach has an overall average latency of 4 ms, whereas MP-K-Slack and AQ-K-Slack approaches are having an overall average latency of 400 ms and 79 ms, respectively. Similarly, sequence-based approach has an overall average maximum latency of 10 ms. Whereas an overall maximum latency for MP-K-Slack is 465.37 ms, and for AQ-K-Slack is 133.27. Therefore, we can clearly see that the sequence-based approach has much lower latency compared to MP-K-Slack and AQ-K-Slack with respect to both the overall average and maximum latencies introduced to the events. As per the summary listed in Table 5.4, the sequence-based approach is 9600% faster than MP-K-Slack and 1800% faster than AQ-K-Slack. Whereas the maximum latency is 4100% and 1100% lower than MP-K-Slack and AQ-K-Slack, respectively.

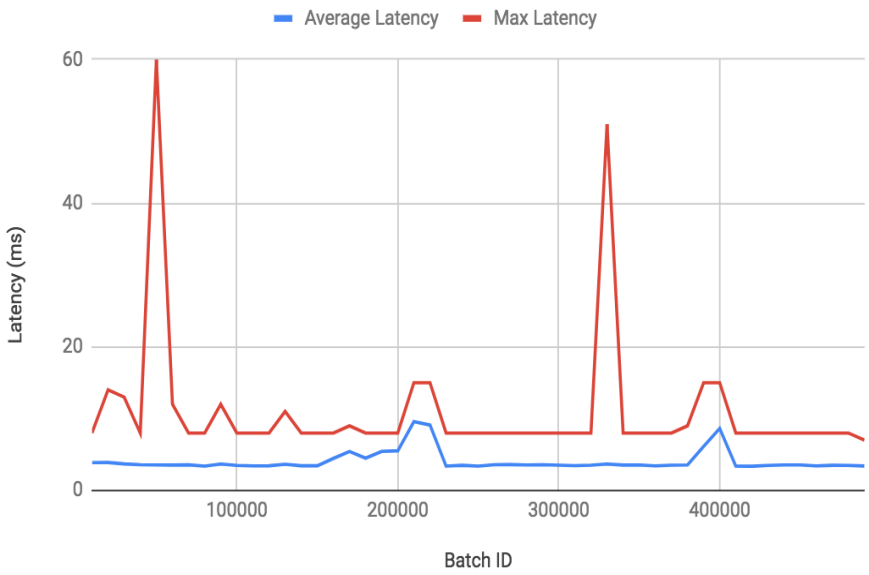


Figure 5.6: Average and maximum latency of events for Dataset 1 with proposed sequence-based approach.

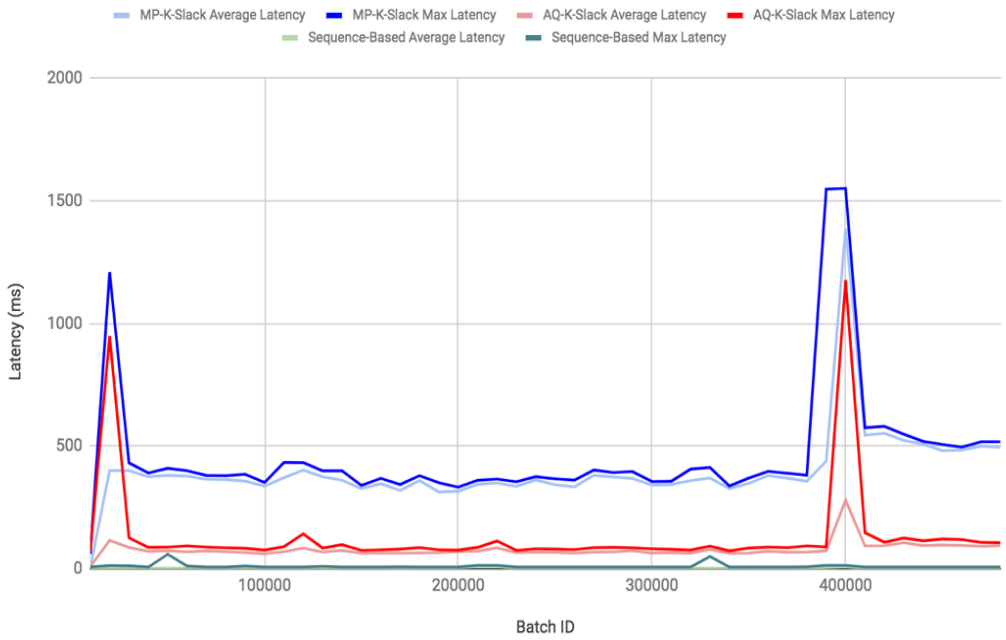


Figure 5.7: Average and max latency of events for Dataset1 with MP-K-Slack, AQ-K-Slack and sequence-based approach

Table 5.4: Overall summary of latency incurred for events in Dataset1 with all out-of-order handling approaches

Out-of-order Handling Approach	Overall Average Latency (ms)	Overall Average Max Latency (ms)
Sequence Based	4.09	10.98
MP-K-Slack	399.56	465.37
AQ-K-Slack	79.04	133.27

Table 5.5: Total number of out-of-order events with Dataset 1 with all out-of-order handling approaches.

Out-of-order Handling Approach	Total Out-of-order Events	Accuracy
Sequence Based	5	99.99%
MP-K-Slack	5	99.99%
AQ-K-Slack	5	99.99%

As shown in Table 5.5, all three approaches resulted in the same number of out-of-order events. Therefore, with respect to the accuracy, all three approaches perform at the same level at 99.99% (compared to all events being in order). But when comparing latency and accuracy, the sequence-based approach provides greater performance.

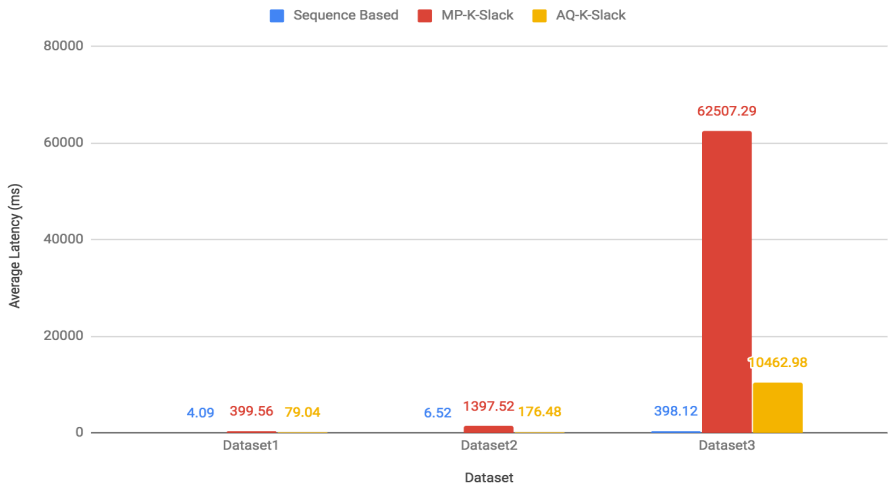


Figure 5.8: Average latency of all three approaches with respective to the dataset.

Table 5.6: Average latency for all three approaches along with datasets.

Dataset	Sequence Based (ms)	MP-K-Slack (ms)	AQ-K-Slack (ms)
Dataset 1	4.09	399.56	79.04
Dataset 2	6.52	1397.52	176.48
Dataset 3	398.12	62507.29	10462.98

As shown in Figure 5.8 and Table 5.6, we can see the latency had increased from Dataset 1 to 2 and then from 2 to 3. This is because, as mentioned in Section 5.1, the Dataset 1 has less variant distribution of out-of-order events compared to Dataset 2 and Dataset 3. Also, for Dataset 2 the maximum event inter-arrival time can be as high as 1800 ms with very high variations, whereas Dataset 1 has only two peaks of up to 1000 ms. Similarly, Dataset 3 has more high inter-arrival times between the out-of-order events, and maximum is 30 seconds. Therefore, it is expected that the latency will increase from Dataset 1 to Dataset 3.

Further, based on Figure 5.8 we can see, that MP-K-Slack has high latency while it is relatively less in AQ-K-Slack. As AQ-K-Slack is having an adaptive buffer (see Section 2.3.3), latency is less compared to MP-K-Slack. Whereas sequence-based approach has drastically less latency compared to the other two approaches. Therefore, based on these results, we can see that the sequence-based approaches latency is lower than MP-K-Slack by 21300% and AQ-K-Slack by 2600% for Dataset2. Similarly, sequence-based approach performs faster than MP-K-Slack as 15600% and AQ-K-Slack as 2500% for Dataset 3.

Table 5.7: Accuracy of all three approaches with datasets.

Datasets	Sequence-Based (%)	MP-K-Slack (%)	AQ-K-Slack(%)
Dataset 1	99.99	99.99	99.99
Dataset 2	99.97	99.98	99.96
Dataset 3	99.98	99.99	99.98

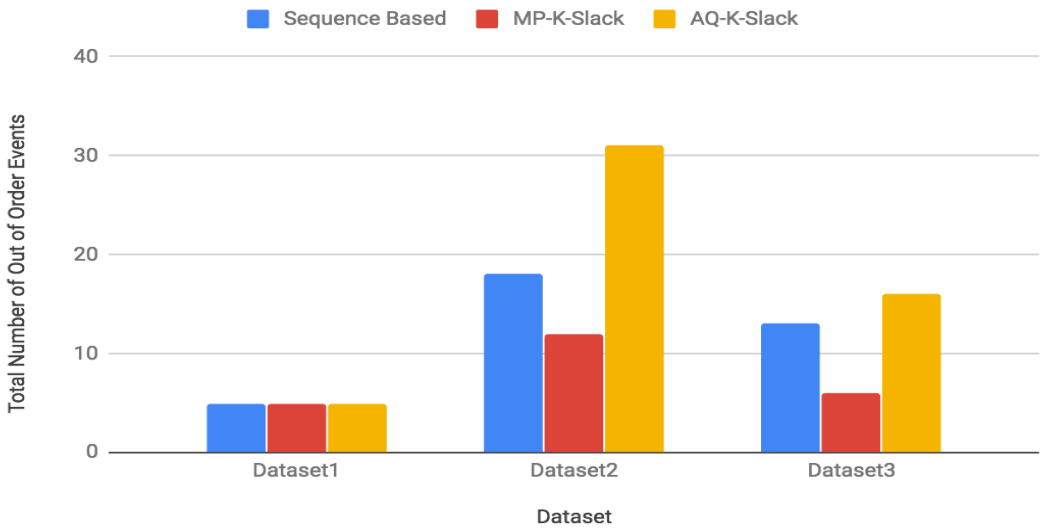


Figure 5.9: Total number of out-of-order events for all three approaches and the respective datasets.

Next, we evaluate the accuracy of these all three methods under all three datasets. The total number of out-of-order events encountered for these datasets are shown in Figure 5.9 and Table 5.7. As shown in Figure 5.9, overall MP-K-Slack produces the lowest number of out-of-order events and AQ-K-Slack approach produces the highest number of out-of-order events in all three datasets. Also, we can see the MP-K-Slack has high accuracy compared to the other two methods and has accuracy high as 99.99% for all datasets as shown in Table 5.7. This is because the K-Slack set the buffer size to be high as the maximum late arrival time interval (see Section 2.3.2). Hence, it results in high accuracy and also high latency. Because the AQ-K-Slack can adjust its buffer size based on the sampled statistics, it is resulting in less latency compared to MP-K-Slack but has lower accuracy. The accuracy of the AQ-K-Slack was 99.99% for Dataset1, but then reduced to 99.96% with Dataset 2, and 99.98% with Dataset 3. Further, the accuracy of the sequence-based approach was 99.99% with Dataset1 but reduced to 99.97% in Dataset2, and 99.98% with Dataset 3. The proposed sequence-based approach has the lowest latency compared to all three approaches and has acceptable accuracy which is in between of MP-K-Slack and AQ-K-Slack.

5.2.2.2. Performance and Accuracy with Multiple Event Sources

Next, we consider the performance when publishing with multiple event sources. To evaluate the behavior of multiple sources we used Dataset 3 as explained in Section 5.1.2. As per Figure 5.10 and Table 5.8, the sequence-based approach has the lowest latency compared to the other two approaches. MP-K-Slack is having a higher latency compared to AQ-K-Slack, as AQ-K-Slack dynamically changes the buffer size based on the runtime window coverage. The average latency of MP-K-Slack does not increase with the increasing number of sources because its buffer size had been already increased to the maximum latency of out-of-order events. As we are using the same out-of-order event distribution for multiple event sources, the maximum time interval for out-of-order event arrival does not have an impact. However, when we increase the event sources, the latency of the AQ-K-Slack increases. This is because, with multiple event sources, AQ-K-Slack buffer minimization logic did not reduce the buffer size drastically since the runtime window coverage was low due to the increase of the out-of-order events after passing through AQ-K-Slack as shown in Table 5.9.

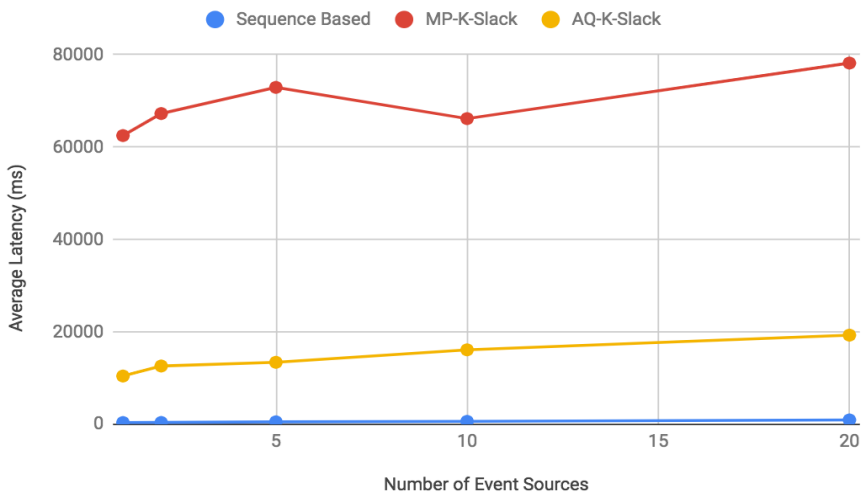


Figure 5.10: Variation of average latency in ms of all three approaches with number of event sources

Table 5.8: Average latency with varying number of events sources.

No of Event Sources	Sequence Based (ms)	MP-K-Slack (ms)	AQ-K-Slack (ms)
1	398.12	62507.29	10462.98
2	412.23	67274.41	12634.21
5	552.72	72937.56	13432.69
10	652.34	66178.05	16134.3
20	963.54	78200.31	19313.43

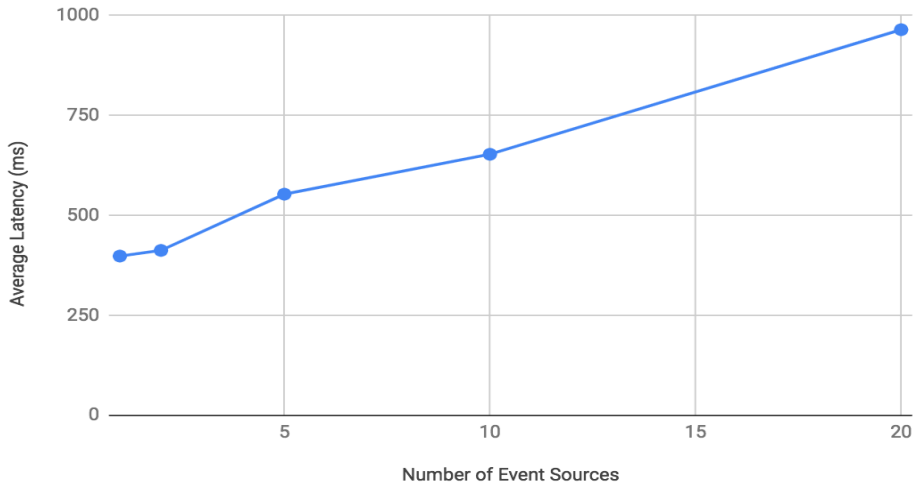


Figure 5.11: Average latency for sequenced-based approach across all event sources.

Figure 5.11 shows the average latency variation for sequence based approach from Figure 5.10. Based on Table 5.8 and Figure 5.11, we can claim that the average latency is linearly increasing when increasing the number of event sources. The sequence based approach, have to synchronize the ordered events coming from multiple sources based on the event's timestamp. As discussed in Section 3.2.2, during this process, the synchronization component will wait for an event to be received for each event sources to make sure the lowest timestamp event is received and considered for the synchronization process. Therefore, when the number of event sources is increased, more time will be spent on waiting for events from all event sources. Therefore, the latency has linearly increased when we increase the number of event sources.

Figure 5.12 and Table 5.9 shows the variation of the out-of-order events, and the accuracy of each approaches. We can see that AQ-K-Slack is having less accuracy compared to the other two methods, as the dynamic buffer size was not big enough to get all late arrival events, and it drops to 99.02%. Whereas MP-K-Slack continues to have high accuracy for most of the cases. This is because MP-K-Slack adjusts the buffer size based on the maximum delay seen up to that point in time. While this increases accuracy, it also highly increases the average latency of events. The accuracy of the sequence-based approach has between AQ-K-Slack and MP-K-Slack. More importantly, the accuracy does not change with the number of event sources.

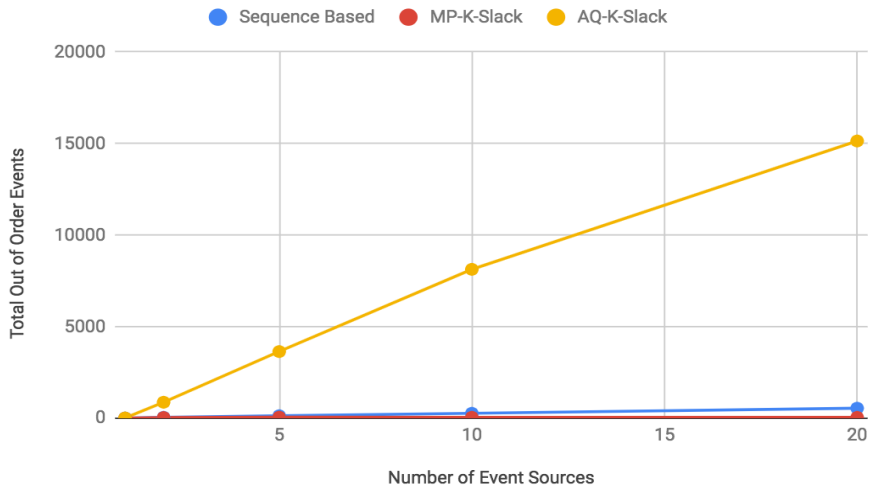


Figure 5.12: Total out-of-order events with multiple event sources for all three approaches.

Table 5.9: Accuracy of all three approaches when publishing events with multiple sources.

No of Event Sources	Sequence Based (%)	MP-K-Slack (%)	AQ-K-Slack (%)
1	99.98	99.99	99.98
2	99.97	99.98	99.47
5	99.97	99.99	99.12
10	99.97	99.99	99.02
20	99.97	99.99	99.08

Therefore, based on the results of average latency incurred with the multiple sources, we could see that the sequence-based approach is 15600% faster than K-Slack, and 2500% faster than AQ-K-Slack. Also, the accuracy remained higher as 99.97% with the sequence-based approach, AQ-K-Slack dropped down to 99.02%, and MP-K-Slack remained high as 99.98%. Therefore, we can conclude that the sequence-based approach performed well with respect to the latency and accuracy when comparing the other two methods for both single and multiple sources scenarios.

5.2.2.3. Accuracy with Time Drifted Event Sources

We further focus on the effect of the time drift in the event sources. For this evaluation, we do not consider MP-K-Slack and AQ-K-Slack as those approaches do not consider this problem, and we noticed once we introduced the time drift the accuracy reduced significantly as these approaches only consider event timestamp to order the events. We used the dataset explained in Section 5.1.3 for this evaluation. We evaluated the effect of time drift between two event sources, by introducing a time drift for one event source and having another event source without any time drift.

Table 5.10: The total number out-of-order events produced and accuracy in sequence based approach with and without time syncing of event sources.

Time Drift	Without Initial Time Sync		With Initial Time Sync	
	No of Out-of-order events	Accuracy (%)	No of Out-of-order events	Accuracy (%)
0ms	52	99.97	52	99.97
1ms	502560	-201.98	78	99.95
5ms	505235	-203.58	75	99.95
10ms	539849	-224.38	72	99.96
1min	441979	-165.57	74	99.96
1hour	18001	89.18	73	99.96

As seen in Table 5.10, once we introduced the time drift between the sources, a very high number of out-of-order events was produced without having initial time sync operation. Actually, the number of out-of-order events produced in this case was higher than the original number of out-of-orders presented in the dataset. The original

number of out-of-order events with two event sources is 166424 (83212 x 2) based on Table 5.2. But based on Table 5.10, we can see that once we introduced the time drift, the total number of out-of-order events is more than 3 times higher than the original number of out-of-order events, which resulted in accuracy to be in negative value and low as -224.38%. This is because the drifted timestamp is used without any correction. Therefore, the events from a source with less time drift will be getting the precedence over events produced by sources with higher drift. This is the same reason for having a significant drop in the total out-of-order events to 18000 for 1 hour time drift. With time drift high as 1 hour, the most of the events from lower time drift sources are first released until the event timestamps from lower time drift event source meets the first event's timestamp of higher drifted event sources. Once we introduce the time syncing, the total number of out-of-order events is reduced, and it stabilized around 75 out-of-order events regardless of the change in the time drift.

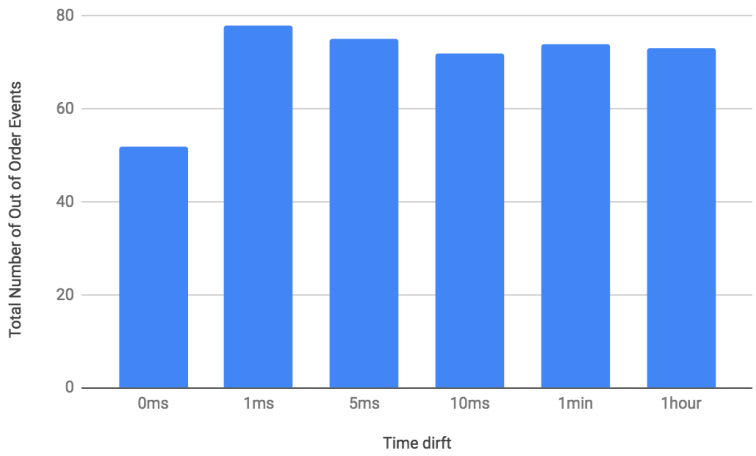


Figure 5.13: The number of out-of-order events with amount of time drifts for Sequence based approach.

As shown in Figure 5.13, the sequence-based approach has the same number of out order events while having different time drifts. But there is a small increase in the number of out-of-order events when there are no time drifts and having the time drift between the event sources. The reason for this behavior is the inaccuracies in calculating the time drifts as mentioned in Section 3.2.2.

5.2.3. Analysis of Query Operators under Out-of-order Events

Based on the above evaluations, we noticed that inaccuracies in the transport delay calculation had increased the number of out-of-order events with multiple event sources. Therefore, in this Section, we will focus on the effect of those with query operators and the final results.

5.2.3.1 Aggregator Operator

Let us focus on using an aggregation operator with out-of-order events from multiple event sources. To evaluate this, we enhanced the setup mentioned in Section 5.2.2.3 by adding further transport delay of 0.125 ms. This value was selected as this was able to introduce a considerable difference in the final result of the aggregation with the time window and practically feasible transport delay. We used Query 5.6 to calculate the average velocity of 10 seconds time batch window.

```
from inputStream#reorder:sequence(sourceId, seqNum, driftedTs,
500L, 0.6, 0.6, false) select sourceId, seqNum,
eventTimestamp() as relativeTimestamp, ts as occuredTime,
v_abs
insert into outputStream;

from outputStream#window.externalTimeBatch(occuredTime, 10
sec)
select avg(v_abs) as AvgVelocity, occuredTime
insert into aggregateOutputStream;
```

Query 5.6: Calculating average velocity.

This evaluation was carried forward with two event sources, and the outputs were obtained for the following two cases:

1) Event sources do not have any drift and no inaccuracies in the transport delay. The dataset used for this was mentioned under Section 5.1.2.
2) Event sources have a drift of 5ms and 0.125ms is the calculated max delay for these event sources.

The average output values from the time batch windows for these cases are listed in Table 5.11, where we could see small irregularities between the expected and actual averaged values.

Table 5.11: The differences between the expected and actual average values.

Time Batch Window ID	Average velocity without drift & transport delay (µm/s)	Average velocity with drift & transport delay (µm/s)	Difference of Expected vs Actual Average
1	144123.87	144123.87	0
2	133834.17	133834.29	-0.12
3	138424.12	138422.58	1.54
4	157036.12	157037.65	-1.53
5	154401.79	154401.73	0.06
6	138580.31	138580.56	-0.25
7	443901.59	443900.16	1.43

As this is attributed to the time drift and transport delay calculations, we further evaluated the method proposed in Section 3.2.3. Query 5.7 was used to evaluate the effect of this method. As shown in Query 5.7, the average was calculated for each LOW, MIDDLE, HIGH window independently, and the average of those was taken as a final average value.

```
from inputStream#reorder:sequence(sourceId, seqNum, driftedTs,
500L,0.6, 0.6, false) select sourceId, seqNum,
eventTimestamp() as relativeTimestamp, ts as occuredTime,
v_abs
insert into outputStream;

from outputStream#reorder:externalTimeBatch(10 sec,
occuredTime, false)
select (avg(ifThenElse(windowType=='LOW',v_abs, 0))+
avg(ifThenElse(windowType=='MIDDLE',v_abs, 0))
+avg(ifThenElse(windowType=='HIGH',v_abs, 0)))/3 as
avgVelocity, occuredTime
insert into aggregateOutputStream;
```

Query 5.7: Average velocity calculation by using reorder based external time batch window.

Table 5.12 shows the difference between the average velocity calculated in an ideal environment with no time drift and transport delay and the average velocity calculated with drift and transport delay by using Query 5.7. We can see the difference between the actual and expected values is reduced after using the reorder based time batch window as mentioned in Query 5.7.

Table 5.12: The average value of the velocity after using the reorder based time batch window.

Time Batch Window ID	Average velocity without drift & transport delay (µm/s)	Average velocity with drift & transport delay (µm/s)	Difference of Expected vs Actual Average
1	144123.87	144123.87	0
2	133834.17	133833.95	0.22
3	138424.12	138423.91	0.21
4	157036.12	157036.48	-0.36
5	154401.79	154401.22	0.57
6	138580.31	138580.91	-0.60
7	443901.59	443901.55	0.04

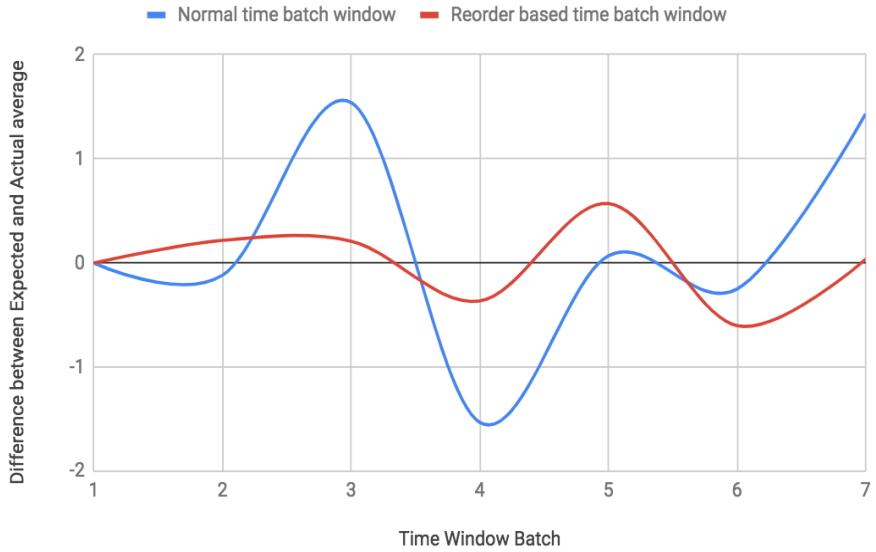


Figure 5.14: Difference between the of average values for normal time batch window, and reorder based time batch window.

Figure 5.14 shows the actual variation of the difference in the final output when using the general time batch window and the reorder-based time batch window. We can see that the reorder-based time batch window is closer to the x-axis, in other words, closer

to the expected value. Also, it reduces the deviation of the results by ~ 50% compared to the value obtained by using the normal window.

5.2.3.2. Pattern Matching Operator

For the evaluation of the pattern matching, we use the Dataset 4 described in Section 5.1.4. In this case, the goal score event needs to be detected based on the sensor events from the ball and the shoes of the football players as defined by the pattern matching query in Query 5.8. The query involves two events *e*1 and *e*2 which should occur one after the other with some preconditions being satisfied such as the position and acceleration of the ball. The *x* and *y* values in query indicate the boundary points of the goal region. The attribute *sid* corresponds to the sensor ID which is a unique identifier of the sensor which produced a signal used for the calculation of the position event as explained in Section 5.1. Further, as we are using two event sources to match this pattern where we consider the *e1* to be matched form Source ID 0, and *e2* to be matched in Source ID 1.

```
from inputStream#reorder:sequence(sourceId, seqNum, driftedTs,
500L, 0.6, 0.6, false)
select sid, ts, x, y, z, v_abs, a_abs, sourceId, seqNum,
eventTimestamp() as eventTime, driftedTs, relativeTimestamp
insert into outputStream;

from every e1=outputStream [sourceId == '0' and (x>29880 or
x<22560) and y> (-33968) and y <33965 and (sid==4 or sid ==12
or sid==10 or sid==8)]
-> e2=outputStream [sourceId == '1' and (x<=29898 and x>22579)
and y<= (-33968) and z<2440 and a_abs>=55000 and (sid==4 or sid
==12 or sid==10 or sid==8)]
select e2.sourceId, e2.ts, e2.driftedTs, e1.ts as e1Ts,
e1.driftedTs as e1drift
insert into patternStream;
```

Query 5.8: Pattern matching query to match the goal events.

The query was run against multiple cases and the pattern matching results are listed in Table 5.13. The pattern is correctly detected until the transport delay increased up to 50 ms. This is because the timestamp of events *e1* and *e2* have a difference of 44 ms. Therefore, the drift correction is effective only up to 50 ms. Therefore, to handle this

situation, the pattern matching query can be rewritten as shown in Query 5.9 (as described in Section 3.2.3.3).

Table 5.13: Results of the pattern matching dataset.

Test Case	Pattern Matching Detected or Not
Two event sources without any Drift	Pattern Detected
Two event sources with 1 second Drift	Pattern Detected
Two event sources with 1 Seconds Drift and Transport delay of 0.125 ms	Pattern Detected
Two event sources with 1 Seconds Drift and Transport delay of 1 ms	Pattern Detected
Two event sources with 1 Seconds Drift and Transport delay of 10 ms	Pattern Detected
Two event sources with 1 Seconds Drift and Transport delay of 50 ms	Pattern Not Detected

The modified query captures two possibilities of the pattern, which are *e1* followed by *e2* and *e2* followed by *e1*. It then compares the time difference of the events that matched the pattern and verify whether it is less than the maximum delay calculated for the event sources that publish to the input event stream. In the pattern *e1* followed *e2* case, we trigger a pattern matches event, with the confidence attribute set to indicate the confidence level of the pattern matching decision based on the time difference and the max delay of the event sources. If the time difference between *e1* and *e2* is greater than *maxDelay*, then we trigger pattern matched event with confidence level set to *Confirmed*, else with confidence level set to *UncertainPatternDetection*. However, for the *e2* followed by *e1* pattern, we trigger the pattern matching event only if the differences between the event is less than the maximum delay of the event sources with confidence level set to *UncertainPatternDetection*. Therefore, we reduce unnecessary false positives by following the above approach. With this approach, the pattern was able to be detected for the last test case in Table 5.13 with the confidence level set to *UncertainPatternDetection*.

```
from inputStream#reorder:sequence(sourceId, seqNum, driftedTs,
500L, 0.6, 0.6, false)
select sid, ts, x, y, z, v_abs, a_abs, sourceId, seqNum,
eventTimestamp() as eventTime, driftedTs, relativeTimestamp
insert into outputStream;

from every e1=outputStream [sourceId == '0' and (x>29880 or
x<22560) and y> (-33968) and y <33965 and (sid==4 or sid ==12
or sid==10 or sid==8)]
 -> e2=outputStream [sourceId == '1' and (x<=29898 and
x>22579) and y<= (-33968) and z<2440 and a_abs>=55000 and
(sid==4 or sid ==12 or sid==10 or sid==8)]
select ifThenElse(e2.relativeTimestamp -
e1.relativeTimestamp<=reorder:getMaxDelay('inputStream')
,'UncertainPatternDetection', 'Confirmed') as confidence,
e2.sourceId, e2.ts, e2.driftedTs, e1.ts as e1Ts, e1.driftedTs
as e1drift
insert into patternStream;

from every e2=outputStream [sourceId == '1' and (x<=29898 and
x>22579) and y<= (-33968) and z<2440 and a_abs>=55000 and
(sid==4 or sid ==12 or sid==10 or sid==8)]
 -> e1=outputStream [sourceId == '0' and (x>29880 or x<22560)
and y> (-33968) and y <33965 and (sid==4 or sid ==12 or
sid==10 or sid==8)]
select ifThenElse(e1.relativeTimestamp -
e2.relativeTimestamp<=reorder:getMaxDelay('inputStream')
,'UncertainPatternDetection', 'NotMatch') as confidence,
e2.sourceId, e2.ts, e2.driftedTs, e1.ts as e1Ts, e1.driftedTs
as e1drift
having confidence == 'UncertainPatternDetection'
insert into patternStream;
```

Query 5.9: Pattern matching query to remove the effect of the time drift calculation inaccuracies.

5.3. Summary

This Chapter discussed the details related to the performance evaluation and the results obtained through tests. We used the dataset from the DEBS 2013 football game to evaluate different cases of the proposed methodology. We focused on the latency and accuracy of different out-of-order event handling approaches, with single and multiple event sources without and with different time drifts. In all cases, we observed that the proposed sequence-based approach provided significantly low latency where it was 9600%-15600% lower compared to MP-K-Slack and 1200% -2500% lower compared to AQ-K-Slack. We also observed that accuracy of the proposed method was between 99.97% - 99.99% for different cases and this is relatively less than MP-K-Slack which

always gives 99.99% accuracy but greater than AQ-K-Slack which has accuracy as low as 99.02% in several cases. Therefore, when we compare both accuracy and latency, the proposed sequence-based approach performed well compared to the other two approaches. We also looked into the possible errors that can be caused by drift calculation and the effect of those in time batch window based aggregation and pattern matching operators. We were able to bring down the difference of the aggregated time batch window and expected results, by 50% after using the multiple window average method. We also observed that the pattern matching query was able to provide correct results after including the max delay time into the consideration, regardless of the increased error in the drift calculation.

6. CONCLUSION

6.1. Summary

Complex Event Processing Engines (CEP) is heavily used in many domains such as IoT, Banking Systems, Telecommunication and Networking Systems, and Health care to analyze data produced in these systems and produce the results accurately in real time. We often use query operators such as aggregators on time windows and pattern matching to analyze the data in CEP. But the accuracy of these query operators highly dependent on the order of the events received. The event sources used in some domains (e.g., IoT) produce a very high rate of events, and those data could be in different networks and time zones without using the standard time. Hence, the events can be bundled together and transmitted via connection pool and could be transmitted with UDP protocols. Therefore, we cannot expect the events will be received in the same order as they were produced at the event sources. Such out-of-order events could lead to inaccurate decisions depending on the types of CEP queries that process those events. Also, this problem exists in distributed CEP processing, where the data will be analyzed by multiple CEP nodes. When those events are transmitted from peer CEP nodes to a single node for the final aggregation, the order of the events received by that node will not be following the actual order in which the events were produced in the event source.

We analyzed several approaches to solving this problem, but those approaches either increase the latency or reduce accuracy. Also, those approaches do not solve the problem with multiple event sources which will have time drifts among them. Therefore, we proposed an approach based on the event sequence numbers to achieve a good balance between latency and accuracy. Moreover, the proposed technique works with multiple event sources that may have time drifts. The sequence-based approach requires adding two attributes to the event data namely, the source identifier and sequence number. Here the source identifier is the specific value assigned to the event source, and sequence number is an incremental number that is assigned to the event based on the time it was generated at the event source per event stream. In the

multi-source case, the event source will first send initialization time sync requests to CEP in order to calculate the event source's time drift. This solves the problem of having event sources with different time zones. Once the initialization is completed, the actual events are ready to be transmitted to the CEP engine with the two additional attributes mentioned above. Once the event is received, the CEP receiver first orders the events of the particular event stream based on the sequence number for each event source separately. It then orders the events among multiple event sources based on the reference timestamp that was deduced by adding the time drift that was calculated during the initialization step. The global ordered events after the synchronization process are fed to the query operators. As any errors in estimating time drift could compromise the accuracy of the final global ordered events, we introduced multiple windows approach. Because the drift calculation can differ as much as the maximum delay time from the event sources to the CEP server, we produce an additional two windows that could cover this ambiguous time range. Therefore, rather than using the single aggregator operation on a single window, we can use the aggregator operator on all three windows and obtain an average value of that. This reduces possible inaccuracies in calculating the drift of the event sources. Similarly, for pattern matching query operator, we can write multiple pattern matching queries with other different combinations of the event sequence that could exist and compare the timestamp difference with max delay time to obtain the confidence level of the pattern detection.

The proposed sequence-based approach was implemented in Siddhi CEP server as an extension. Several tests were then performed to obtain its latency and accuracy of the results produced. For comparison, we also considered MP-K-Slack and AQ-K-Slack approaches which are buffer based techniques and can be used for aggregation and pattern matching query. In all cases, we observed the proposed methodology, provided very low latency. For example, the latency of the proposed approach was 9600%-15600% lower compared to MP-K-Slack and 1200% -2500% lower compared to AQ-K-Slack. We also observed that the accuracy of the proposed method was 99.97% - 99.99% in all cases analyzed, and this is relatively lower than MP-K-Slack which

always give 99.9% accuracy. However, it was greater than AQ-K-Slack which has accuracy low as 99.02% in some cases. Therefore, when we compare both accuracy and latency, the proposed sequence-based approach has a good balance compared to the other two approaches. Because MP-K-Slack and AQ-K-Slack cannot handle the time drifts, we evaluated the effect of the time drift with the sequence-based approach. With time drift calculation included, we were able to order the events to get closer to 100% accuracy compared to not having the time drift included in the system. We also looked into the possible errors that could be caused by drift calculation and the effect of those in time batch window based aggregation and pattern matching operators. We were able to reduce the difference between the aggregated time batch window and expected results by 50% after using the proposed multiple window average method that covers the ambiguous time range due to the possible inaccuracies in the drift calculation. We also observed that the pattern matching query was able to provide correct results regardless of the increased error in the drift calculation after including the ambiguous time into the consideration.

6.1. Research Limitations

The proposed approach depends on sequence number that is produced by event sources, and therefore, the solution assumes that the event sources can be modified to accurately generate and accommodate the sequence number attribute to the raw event stream.

The proposed approach is using the time synchronization technique to find the time of the event source, and we have moved the possible inaccuracies of this approach to query operators and handled them separately. Therefore, each query operator needs to be written considering this problem and handle it independently. Currently, we have only considered time batch windows, aggregator operators, and pattern matching operator, and we have not considered other windows such as sliding time window, event batch window, and other operators such as join operator.

The proposed solution only considers out-of-order events until the final output is produced by the query operators. And this has not considered the very late out-of-order event that is received after producing the result of the query operator. For example, in the time batch window case, there can be an event received after the window is elapsed.

The proposed technique includes the changes in the event sources where it should use the special client which includes the time sync process before sending the actual events. Also, this technique expects the event sources to send two additional attributes such as sequence number and source ID with the actual event. In case if the event sources cannot be modified to use such client, and send these additional attributes, then we cannot use this technique for such use cases.

This technique requires the CEP query to be written with conscious knowledge about the time drift inaccuracies with multiple sources. For example, pattern matching query should have written to consider all alternate event sequences of the pattern and trigger the pattern matched event by comparing the time of difference of the events in the matched pattern with transport delay of the event sources. This is not handled automatically by the CEP server, and users should use the information about the transport delay and rewrite the query manually to increase the accuracy of the proposed technique.

The evaluation was performed in a LAN network with simulated event sources, but in reality, the multiple event sources will belong to WLAN with different geographical regions and will have varying transport delay. We have not evaluated the behavior of with very dynamic environment, with different kind of event sources.

The datasets used in the evaluations were generated by introducing random out-of-order events to the original DEBS 2013 dataset which does not have any out of order events. Though we have introduced the delays ranging from milliseconds to seconds with different out-of-order distributions, we haven't evaluated in the real environment.

6.2. Future Work

As mentioned under research limitations, the system only considers certain query operators, and we need to extend the solution to other query operators such as sliding time window, event batch window, and join operators. We can implement the sliding time batch window and event batch window to maintain multiple windows to cover the ambiguous time range from time drift calculation as similar to the time batch window in the proposed technique. With this, when we are joining the events to another window of events, we should perform multiple joins with all active windows. With these implementations, all the operators can be used without any limitation along with this approach.

The current system can be extended to handle very late event arrival as well. For this we need to persist some information from the query operators; therefore, we could produce the corrective event once the very late event has arrived. Because we do have knowledge about when an event is missing from the sequence number order, we can decide when to store the details beforehand without storing all the information. However, this functionality needs to be handled per query operator so that it can persist its state and restore it when needed to produce the corrective event. For aggregation operators, the information to be stored might be very less, but for pattern matching operator, we may need to store the entire window which could be intense operation.

We can absorb the complexity of rewriting the queries to reduce the inaccuracies in the time drift calculations, by implementing the extended reorder based query operators. For pattern matching operator, we can implement the extended version of pattern matching query operator which can find the all alternate sequence of the events in the pattern matching internally and perform the comparison of the timestamps of the events with transport delay before triggering the matched event as proposed technique. Similarly, the aggregation operator can also be extended to get average of all windows which covers the ambiguous time range without letting the user write that logic. Therefore, the users do not have to worry about rewriting the query to increase

the accuracy, and it will be handled internally within the extended version of the query operator.

The technique should be evaluated with real-world dynamic environments with multiple event sources, and the performance and accuracy of the technique should be evaluated.

REFERENCES

[1] D. Luckham and R. Schulte, “Event processing glossary-version 1.1,” Event Processing Technical Society [Online] Available: http://complexevents.com/wp-content/uploads/2008/08/epts-glossary-v11.pdf [Accessed: 23rd Dec 2016]

[2] K. Finkenzeller, “RFID Handbook: Radio-frequency identification fundamentals and applications,” New York: Wiley, 1999.

[3] C. Mutschler and M. Philippsen, “Distributed Low-Latency Out-of-Order Event Processing for High Data Rate Sensor Streams,” in IEEE 27th Int. Symp. on Parallel and Distributed Processing, 2013, pp. 1133 – 1144.

[4] Y. Ji, J. Sun, A. Nica, Z. Jerzak , G. Hackenbroich, and C. Fetzer, “Quality-driven Processing of Sliding Window Aggregates over Out-of-order Data Streams,” in 9th ACM Int. Conf. on Distributed Event-Based Syst., 2015, pp. 68 - 79.

[5] J. Li, D. Maier, D. Maier, K. Tufte , V. Papadimos , and P. A. Tucker, “Semantics and Evaluation Techniques for Window Aggregates in Data Streams,” in ACM SIGMOD Int. Conf. on Manage. of Data, 2005, pp. 311 - 322.

[6] J. Li, K.Tufte , V. Shkapenyuk, V. Papadimos, T. Johnson , and D. Maier, “Out-of-order Processing: A New Architecture for Highperformance Stream Systems,” in VLDB Endowment, vol. 1, no. 1, 2008, pp. 274- 288.

[7] R. S. Barga, J. Goldstein, M. Ali and M. Hong, “Consistent Streaming Through Time: A Vision for Event Stream Processing,” in 3rd Biennial Conf. on Innovative Data Syst. Research, 2007, pp. 412 - 422.

[8] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber, “Speculative Out-of- order Event Processing with Software Transaction Memory,” in 2nd Int. Conf. on Distributed Event-based Syst., 2008, pp. 265 - 275.

[9] S. Tirthapura and D. P. Woodruff, “A General Method for Estimating Correlated Aggregates Over a Data Stream,” in 28th IEEE Int. Conf. on Data Eng. (ICDE’ 12), Washington, DC, USA, 2012, pp. 162–173.

[10] M. Li, M. Liu, L. Ding, E.A. Rundensteiner and M. Mani, “Event Stream Processing with Out-of-Order Data Arrival,” in 27th Intl. Conf. Distrib. Comp. Systems Workshops, 2007, pp. 67–74

[11] U. Srivastava and J. Widom, “Flexible Time Manage. in Data Stream Syst.,” in 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Syst., 2004, pp. 263–274.

[12] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, and A. Li, “Continuous Analytics over Discontinuous Streams,” in 2010 ACM SIGMOD Intl. Conf. on Manage. of Data, 2010, pp. 1081 - 1092.

[13] L. Wang, G. Luo, K. Yi, and G. Cormode, “Quantiles over Data Streams: An Experimental Study,” in 2013 ACM SIGMOD Int. Conf. on Manage. of Data, 2013, pp. 737–748.

[14] J. G. Ziegler and N. B. Nichols, “Optimum Settings for Automatic Controllers,” J. Dyn. Sys., Meas., Control, vol. 115, no. 2B, 1993, pp. 220–222.

[15] Y. Xiao, T. Jiang, Y. Shen and H. Deng, “Efficient Strategy for Out-of-Order Event Stream Processing,” in J. Appl. Sci. and Eng., vol. 17, no. 1, 2014, pp. 73-80.

[16] Y. Ji, A. Nica, Z. Jerzak, G. Hackenbroich, and C. Fetzer, “Quality-driven Disorder Handling for Concurrent Windowed Stream Queries with Shared Operators,” in 10th ACM Int. Conf. on Distributed and Event-based Syst., 2016, pp. 25–36, 2016.

[17] V. Jacobson, “Congestion avoidance and control,” in SIGCOMM '88 Symposium proceedings on Communications architectures and protocols, Stanford, 1988, pp. 314-329

[18] David L. Mills, “Internet Time Synchronization: The Network Time Protocol,” in IEEE Transactions on Communications, vol. 39, no. 10, 1991, pp 1482 - 1493

[19] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara, “Siddhi: A Second Look at Complex Event Processing Architectures,” in ACM Workshop on Gateway Computing Environments, 2011, pp. 43-50.

[20] Anonymouse, “Performance Analysis Results” [Online]. Available: https://docs.wso2.com/display/SP400/Performance+Analysis+Results [Accessed: 29-Dec-2018]

[21] Anonymous, “Writing Extensions to Siddhi” [Online]. Available: https://docs.wso2.com/display/CEP420/Writing+Extensions+to+Siddhi. [Accessed: 10-Jan-2017]

[22] Anonymouse, “Siddhi Query Guide” [Online]. Available: https://wso2.github.io/siddhi/documentation/siddhi-4.0/#extensions [Accessed: 29-Dec-2018]

[23] Anonymous, “Siddhi-io-http” [Online]. Available “https://github.com/wso2-extensions/siddhi-io-http” [Accessed: 2-Dec-2018]

[24] Anonymous, “Netty Project,” [Online]. Available” https://netty.io/ [Accessed: 2-Dec-2018]

[25] Anonymous, “DEBS 2013 Grand Challenge: Soccer monitoring” [Online]. Available http://debs.org/debs-2013-grand-challenge-soccer-monitoring/ [Accessed: 2-Dec-2018]