SCALING PATTERN AND SEQUENCE QUERIES IN COMPLEX EVENT PROCESSING

Mohanadarshan Vivekanandalingam

(148241N)

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

June 2017

SCALING PATTERN AND SEQUENCE QUERIES IN COMPLEX EVENT PROCESSING

Mohanadarshan Vivekanandalingam

(148241N)

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

June 2017

DECLARATION

I declare that this is my own work and this MSc Research 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 other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: .....
Date: .....
Name: Mohanadarshan Vivekanandalingam

The supervisor/s should certify the thesis/dissertation with the following declaration. I certify that the declaration above by the candidate is true to the best of my knowledge and that this report is acceptable for evaluation for the MSc Research.

Supervisors

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

Abstract

Complex Event Processing (CEP) plays a major role in real-time analytics such as identifying possible frauds in credit card transactions and geospatial analysis. In CEP, events that are received from different data sources are stored in memory and processed on the fly. Scaling is one of the most important features of a CEP engine. Contemporary CEP engines provide several options to scale event processing vertically and horizontally. For example, these include scaling with Storm cluster, distributed object cache, and publisher-subscriber model, all of which come under random or attribute based partitioning. These approaches help to handle a large number of queries, queries that need a large memory, events which come in high rate, and complex queries that might not fit within a single machine. However, it is difficult to scale pattern and sequence detection in CEP for high event rate because the pattern and sequence detection depend on a set of events happened overtime. Existing scaling approaches based on random or attribute based partitioning affects the continuous event flow and event ordering, which are most important attributes for pattern and sequence matching.

We propose a novel approach to scale pattern and sequence detection queries for high incoming event rate. In the proposed approach incoming events are kept in a queue, grouped into partitions based on time interval defined in the query with some overlapping events, then the events are pushed to several CEP engines and processed simultaneously. Finally, the processed events are filtered and reordered before publishing out from the CEP. Performance analysis showed that the proposed technique increase the throughput by 800%, while increasing the per event latency from 2-3 milliseconds to 8-10 milliseconds (~400% increase) due to queuing nature of the solution.

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 us in this work. My special thanks to the University of Moratuwa for giving an opportunity to carry out this research project.

I would like to gratefully acknowledge Dr. Srinath Perera, the external project supervisor, for his continuous guidance and support throughout the whole duration of the project, under whose supervision that I gained a clear concept of what I should do. I would also like to extend my heartfelt gratitude to Dr. Dilum Bandara, the internal supervisor of the project, for sharing the experiences and expertise with the project matters. I also like to thank WSO2 Inc., which allowed me to continue with my research work and helped to balance with my official duties. Last but not least, I thank all those who like to remain anonymous although the help they provided us was valuable.

Thank you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF QUERIES

LIST OF ABBREVIATIONS

CEP	Complex Event Processing / Complex Event Processor
CPU	Control Processing Unit
DS	Data source
DSMS	Data Stream Management System
EPA	Event Processing Agent
EPL	Event Processing Language
ERP	Enterprise Resource Planning
ESB	Enterprise Service Bus
HA	High Availability
IT	Information Technology
JFR	Java Flight Recorder
JMC	Java Mission Control
JVM	Java Virtual Machine
MQ	Message Queue
MQO	Multiple Query Optimization
SOAP	Simple Object Access Protocol
WSDL	Web Service Description Language
XML	Extensible Markup Language
XSLT	Extensible Style sheet Language Transformations

1. INTRODUCTION

As the World moves to an era where data are the most valuable asset, being able to efficiently process large volumes of data in real time can help to gain a competitive advantage for businesses. Then we need a solution to deliver low latency, high volume, and scalable environment enabling data collection, both in real time and as batch analysis, and firing notifications of multiple types across numerous endpoints. Complex Event Processor is the most common enabling technology to perform real-time analytics [1]. In Complex Event Processing (CEP), event streams are processed in real time through filtering, correlation, aggregation, and transformation, to derive high-level, actionable information. CEP is now a crucial component in many business IT systems. For instance, it is intensively used in stock trading based on market data feeds; fraud detection where credit cards with a series of increasing charges in a foreign state are flagged; transportation where airlines use CEP products for real-time tracking of flights, baggage handling, and transfer of passengers [2].

CEP is a technology for extracting higher-level knowledge from situational information abstracted from processing business-sensory information [3]. Business-sensory information is represented in CEP as event data, or event attributes, transmitted as messages over a digital nervous system, such as an electronic messaging infrastructure. CEP offers the users a way to automate the detection of anomalies or other interesting phenomena. For example, correlating all the trades made by all the traders to detect all the various blunders they might have done [4].

1.1 Complex Event Processing

Most of the enterprise-level CEP engines have an architecture similar to Figure 1.1, where CEP engine receives events from different data sources in different formats as different streams. Then, events from different streams are processed based on a predefined set of queries. After processing those events, outputs are also emitted as another set of streams. Here, a query can contain rules that tell the CEP engine what needs to be done for the received events. Rules can be classified as filter, pattern,

transform, aggregate, split, compose, translate, enrich, and project. The most important rules types are filter, transformation, and pattern detection [5]. This processing happens in near real time, typically within a few milliseconds.

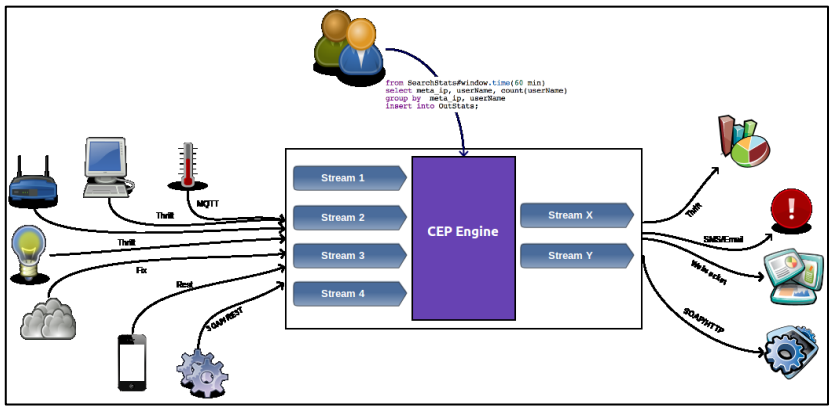

Figure 1.1: High-level overview of CEP operation.

In most of the real-world scenarios, a single CEP engine cannot process the events with limited resources. At this situation, we need to scale the CEP engine to handle [6]:

- Large number of queries.
- Queries that needs large working memory.
- Complex query that might not fit within a single machine with supporting high event rate.
- Large number of events.

Being able to scale CEP systems into multiple nodes facilitates increasing the throughput and high availability while simultaneously maintaining low response time.

1.2 Motivation

While scaling is one of the most important features of a CEP engine, it is not easy to scale CEP engines horizontally due to the in-memory states they maintain. Following approaches are being used to scale CEP engines:

- Using an Enterprise Service Bus (ESB) [7] - The key architectural insight in the system is to separate the integration functionalities of the ESB and the complex event facilities. This results in a stateless ESB, which can be scaled out by adding more processing nodes. A dedicated CEP cluster can then be tuned to handle high throughput and scaled out separately.
- Using distributed object cache [8] - This approach uses a distributed object cache for interaction between CEP engines. This is typically used for a query that needs to maintain a large window and all events in the window would need a large working memory. Distributed caching technology also increases the performance other than scalability CEP applications.
- Scale CEP on top of a Storm cluster [9] - Queries are deployed in CEP engines and run on top of a Storm deployment. Storm cluster provides the platform for inter-communication between engines. Because CEP engine runs inside storm, it allows to spawning multiple instances of CEP engines to run across a storm cluster. This results in a system that is horizontally scalable and capable of handling large volumes of events per time unit.
- Publisher-subscriber model [10] - Event publishers push events to a queue of a Message Broker, CEP engines act as subscribers, consume events from Message Broker and does the processing. Subscribers can be scaled as per the need, and message selectors that supported by Message Brokers like Kafka, AMQP, and ZeroMQ can be used to filter out the events when consuming them. This approach is based on random or attributed based partitioning strategy.

These approaches are based on a random partition, which means events are partitioned randomly without being considerate of their conditions or attributes. This is suitable for handling a large number of queries, queries that need a large working memory, a complex query that might not fit within a single machine, and high event

rate. However, these approaches cannot be applied for all possible scaling use cases. For example, they are not successful in scaling pattern and sequence detection queries and events that cannot be partitioned randomly.

Let us consider an example to understand this further. Assume a situation where a bank tries to find out possible money laundering situation by analyzing the real-time account transaction data. Consider Figure 1.2 which shows a series of a transaction from account *A* to *B*, then from account *B* to *C*, and finally, from account *C* to *A*. This is a possible money laundering activity which bank needs to detect. Following CEP Query 1.1 can be used to detect this suspicious series of events:

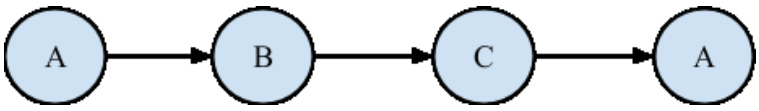

Figure 1.2: Possible money laundering transaction flow.

```
from every (a1 = transactionStream
              -> a2 = transactionStream[a1.toAccountNo == a2.fromAccountNo]
              -> a3 = transactionStream[(a2.toAccountNo == a3.fromAccountNo)   and
       (a1.fromAccountNo == a3.toAccountNo)]
within 5 min
select a1.fromAccountNo as suspectAccountNo
insert into possibleMoneyLaunderingActivityStream;
```

Query 1.1: Siddhi query to detect possible money laundering transaction.

If we adopt partition-based scalability to deal with the large number of transactions the bank may need to handle, transfer from *A* to *B* may be partitioned and send to one CEP engine while *B* to *C* transfer may be sent to another CEP engine. While the same query runs on both the CEP engine, neither will generate an alert as neither see the correlated series of events. Therefore, partitioning-based scaling fails under pattern and sequence queries.

1.3 Problem Statement

The problem that this research going to address can be stated as follows:

How to provide large-scale pattern and sequence detection in CEP while supporting high event rates?

CEP queries that contain pattern and sequence detections are unable to scale by random or attribute based partitioning because the pattern and sequence detection depend on the set of events happened overtime. Therefore, partitioning the events by attribute or random affects the continuous event flow and event ordering, which are the most important properties for pattern and sequence detection. We envision a solution where incoming events are partitioned based on time interval with some overlapping events. Then, partitioned events are pushed to CEP engines, which process events parallelly. Due to this nature, duplicated pattern detections can occur and detected events can get disordered. This would affect the "exactly one" quality of service scenario. Hence, our research should address the issue of event duplication and event reordering which are the side effects of the envisioned approach.

1.4 Objectives

The address the problem statement, following list of objectives are to be achieved:

- Design a scaling approach that is independent of the internal implementation of a CEP engine.
- Develop a suitable approach to scale CEP engine that can be effectively applied to queries that contain pattern matching and sequence conditions.
- Scale event processor to be able to handle a high rate of incoming events.
- Design an approach to overcome event duplication and event reordering that arise due to the use of multiple CEP engines.
- Analyze the performance (throughput and latency) of the proposed technique and its correctness using a set of real-world workloads.

1.5 Outline

This thesis presents the proposed approach for scaling complex event processors while especially focusing on pattern and sequence detection queries. Chapter 2 presents the literature review. It covers many areas related to CEP and scaling such as CEP characteristics, scalability attributes, common dimensions of scaling, and existing scaling approaches. Existing scaling approaches provide details about how scaling is achieved with CEP and aspects to consider when using a specific scaling approach. Chapter 3 presents the proposed solution for scaling pattern and sequence queries in CEP. It also discusses the techniques that used for scaling such as event partitioning based on time, event reordering with K-slack and event duplication handling. Moreover, details of how these techniques are implemented on top of WSO2 Siddhi CEP engine are presented. Chapter 4 presents the evaluation of the proposed approach using real-world workload. This includes details about the throughput improvement of the proposed solution, accuracy metrics, and latency variations with compared to the existing default Siddhi CEP engine. It also discusses the system resource (memory, CPU, and thread) utilization as well. Concluding remarks and suggestion for future works are presented in Chapter 5.

2. LITERATURE REVIEW

Section 2.1 provides an introduction to CEP, CEP internals, architecture, and characteristics. Section 2.2 presents some of the well-known CEP engines and their implementation. Attributes of CEP scalability is discussed in Section 2.3, while Section 2.4 discusses pattern and sequence detection. Existing scaling approaches, related pros and cons of those approaches are discussed in Section 2.5. Section 2.6 discusses various out of order event handling approaches and their similarities and differences.

2.1 Introduction to CEP

The Complex Event Processor (CEP) emerged as a solution for analyzing fast-moving Big Data and continues to grow its usage in event processing domain. At a very high level, CEP receives incoming events in *'Event Streams*' via input handlers, processes them, and notifies the output via callbacks. Here, we use the term Event Streams when the events in a particular Event Stream have a definite schema and when they are logically ordered in time.

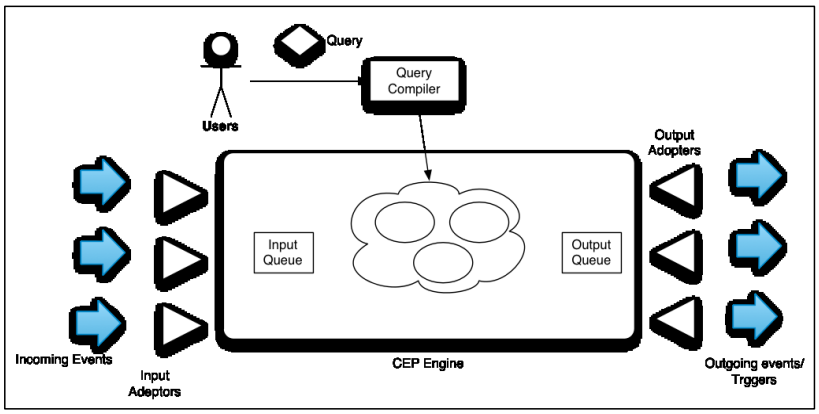

Figure 2.1: CEP internal constructs.

CEP engine typically supports four types of queries, namely Filtering, Event Windows, Event Patterns and Sequences, and Joining Events [11]. However, we are mainly focusing on event patterns and sequences

.

CEP query language has the following structure:

```
from <incoming stream>[<incoming stream filter>]#<window on the stream>
insert into <outgoing stream> <outgoing stream attributes>
```

Here, when events arrive from the incoming event streams, they are filtered, and only the success events of the filter will flow to the window. These windows, based on their configuration, sustain some of the incoming events for a period of time for further processing, like aggregation calculations. Finally, all these events will be projected on the outgoing event streams based on the defined outgoing stream attributes.

2.1.2 Characteristics of CEP Systems

Complex Event Processing (CEP) engines are designed to process a large volume of data or events by simultaneously evaluating multiple queries over event streams. Some of the unique characteristics of CEP designed to process streams in real time within a few milliseconds are as follows:

- The input to CEP systems is continuous, possibly infinite stream of events.
- Event streams require real-time processing, low latency event detection and are usually too big to be stored in their whole entirety.
- Input event streams are volatile, i.e., the arrival rate can vary, events can arrive in bursts and out of order, be lost or intentionally omitted, and timestamps may be imprecise.
- Input events usually exhibit strong temporal relationships, and come from external sources and not from a central database or permanent store.
- CEP systems must cope with a large number of submitted queries in real-time and must process a large number of events, out of which only a small percentage is of interest. As such, they are often used for monitoring.

- CEP systems are usually concerned with relationships between events and their patterns, rather than with individual events. They combine data from multiple sources and infer from it more high-level and useful information.
- The processing in CEP systems is directed by newly arrived events rather than historical data (as compared to databases).
- CEP systems follow DAHP (database active, human passive) model, in which a system does continuous processing and notifies a user. In comparison, traditional database systems employ HADP model (human active, database passive), meaning that data is simply stored and users' query it manually.

2.2 CEP Engines

There are many CEP engines exists in the CEP world [12]. Each of them has many common functionalities which are implemented and achieved in different ways [13]. This section discusses some well-known CEP engines such as Esper, Oracle CEP, Apama and WSO2 Siddhi CEP engines.

2.2.1 Esper CEP Engine

Esper is a Java or .NET library for complex event processing. It is an open-source CEP engine [14]. The main elements in the Esper architecture resemble any DSMS/CEP system. As shown in Figure 2.2, Esper Engine contains many different components like Esper Service Provider, EPL Queries, Event Objects and Subscribers to perform various operations when processing events in real-time. It processes complex queries written in a language called EPL (event processing language). Esper and Event Processing Language (EPL) provide a highly scalable, memory-efficient, in-memory computing, SQL-standard, minimal latency, real-time streaming-capable Big Data processing engine for historical data, or medium to high-velocity data and high-variety data. Esper Engine contains SQL-like language to write queries. It can process 2M events/second for simple filtering queries on a four core hardware machine.

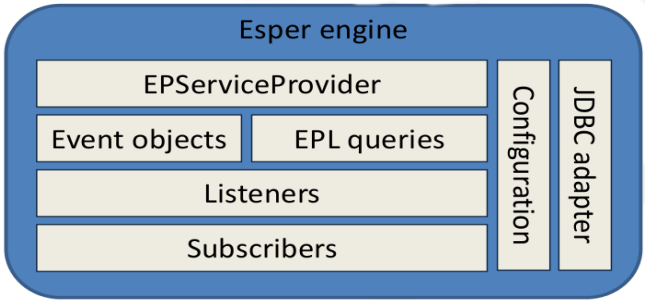

Figure 2.2: Architecture overview of Esper engine.

2.2.2 Oracle CEP

Oracle Complex Event Processing (Oracle CEP) provides a modular platform for building applications based on an event-driven architecture [15]. At the heart of the Oracle CEP platform is the Continuous Query Language (CQL) which allows applications to filter, query, and perform pattern matching operations on streams of data using a declarative, SQL-like language.

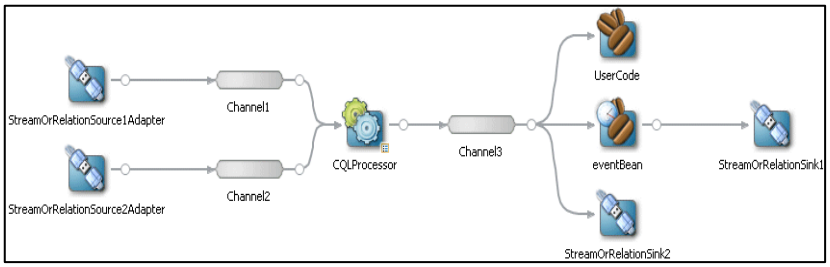

Figure 2.3: Conceptual view of Oracle CEP application.

Oracle CEP has the capability of deploying user Java code (POJOs) which contain the business logic. Running the business logic within Oracle CEP provides a highly tuned framework for time and event driven applications. In Oracle CEP, rules are expressed as queries using the Oracle Continuous Query Language (Oracle CQL). These queries are persisted to a data store and are used for processing the inbound stream of events and generating the outbound stream of events. Queries typically perform filtering and aggregation functions to discover and extract notable events

from the inbound event streams. As a result, the number of outbound events is generally much lower than that of the inbound events. As shown in Figure 2.3 adapters, channels, processors, and business logic POJOs can be connected arbitrarily to each other, forming event processing networks (EPN). Oracle CEP can process 40,000 events/second for simple filtering queries.

2.2.3 Apama CEP

The Apama Event Processing Platform is a complete design and deployment environment for CEP applications [16]. From graphical design tools to research and backtesting utilities, the complex event processing (CEP) platform provides analysts, developers and administrators a full life-cycle design center that is optimized for CEP solutions.

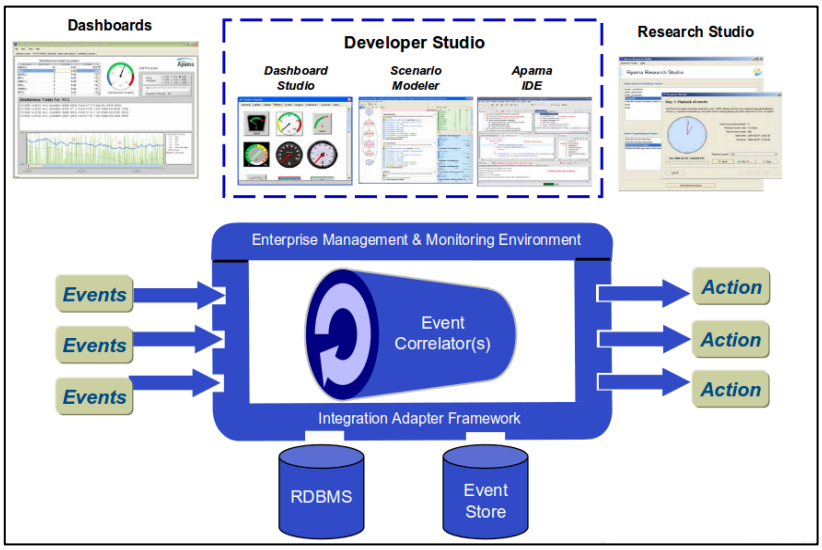

Figure 2.4: Overview of Apama architecture.

As shown in Figure 2.4 Apama CEP is built with three main components. They are Integrated Adaptor Framework, Event Correlator and Enterprise Management and Monitoring Environment. Apama Integrated Adapter Framework supports the

building of new connection, general infrastructure adapters, as well as plug-ins that are available for both Capital Markets and other applications and support a range of APIs that extends the Apama functionality via plug-ins of components for the support of specialized analytics and third party provides. Apama CEP engine contains SQL-like query language. Apama Engine can process 1Million events/day.

2.2.4 WSO2 Siddhi CEP Engine

WSO2 Complex Event Processor (WSO2 CEP) is an enterprise-grade server that integrates with various systems to collect, analyze, and notify meaningful patterns real time [17]. The core back-end runtime engine behind the WSO2 CEP server is WSO2 Siddhi. Siddhi is a lightweight, easy-to-use Open Source CEP under Apache Software License v2.0. Siddhi CEP processes events which are triggered by various event sources and notifies appropriate complex events according to the user specified queries. Siddhi Query language supports Filters, Windows, Pattern, Sequences, Joins and event aggregation. Siddhi uses an SQL-like query language, but queries are evaluated on continuous event streams.

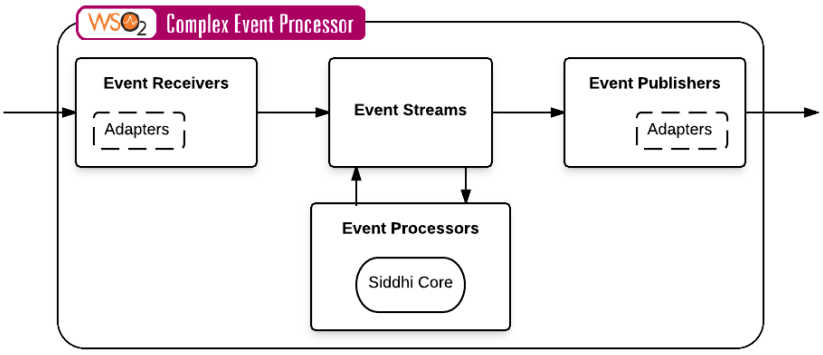

Figure 2.5: High-level architecture of WSO2 CEP engine.

As seen in Figure 2.5, at a very high-level Siddhi receives incoming events in Event Streams via Event Receivers (Input Adapters), processes them, and notifies the output via Event Publishers (Output Adapters). Here, we use the term Event Streams

when the events in a particular Event Stream have a definite schema and when they are logically ordered in time.

WSO2 Siddhi CEP architecture consists of three main components, namely input adapters, Siddhi-Core, and output adapters. Next, each of these components is discussed in detail.

Input Adapters

Input event streams to the Siddhi engine are handled by input handlers. Usually, in practical scenarios, there can be several input event streams to the event processing engine from different event sources. These different event streams can be in different forms or wrappers like XML messages, JSON messages, POJOs, emails, or proprietary binary messages. The input adapters provide an interface to these different event streams and convert them into a common easy to process representation, which is currently a tuple data structure. There are several input handler implementations to handle different event forms.

Siddhi-Core

The most important part of the Siddhi is its rule processing engine called Siddhi Core. Input events are processed according to the constructs defined by the input queries and emit detected event pattern as an output event. Siddhi core consists of several sub-components such as executors, event queues, processors and callback handlers. Normalized input events from input adapters are appended into input queues where processors fetch them from there and append resulting events to output queues. Each sub-component of Siddhi-core will be described later.

Siddhi supports an SQL-like query language called SiddhiQL to provide user queries to the processing engine. The query compiler does the validation and interpretation of SiddhiQL using ANTLR language recognizer. Validated queries are compiled into query object model, which is used by the Siddhi-core to drive its processing. Siddhi's internal data object model will be explained in next few sections.

Output Adapters

Output adapter does the reverse operation of input adapter. Once a complex event pattern is detected by the event processing engine, the resulting event is converted to a representation suitable format and notified to event subscriber (i.e. event sink) by the output adapter. There can be several event sinks who accept resulting event stream in different formats like XML messages, JSON messages, emails, SMS messages, Database updates, etc. There are separate output adapters for each of these different formats.

Apart from the above main modules, Siddhi has a pluggable user interface module which can be used to display useful statistics and monitoring tasks.

WSO2 Siddhi Query Architecture

The architecture of a basic Siddhi query (having Filter, Transform, and Window) is illustrated in Figure 2.6. Here, the events flow from the Input Handler of the incoming Event Stream to its respective Stream Junction. The Stream Junction is responsible for sending the events to all components that are registered to that Event Stream. In Siddhi, we can find two main types of Stream Subscribers; Stream Callback, which is used to notify an event occurrence on a particular stream, and Query Handler Processors - which are responsible for filtering and transforming the events for further processing. Only the event that passes the filter conditions will be outputted from the Query Handler Processor, which will indeed be fed into the Window processor where the events will be stored for time, length or uniqueness-based, or other custom processing. The events are then fed into the Query Projector to perform event attribute level processing such as average, group by and having. Finally, the output of Query Processor will be sent to its registered Query Callback and its output stream's Stream Junction where the event will be fed to all the Queries and Stream Callbacks registered to that Event Stream.

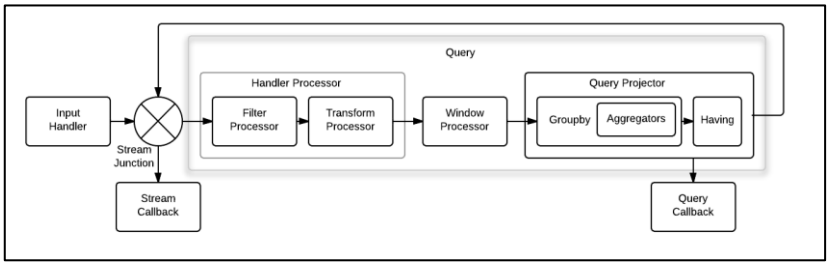

Figure 2.6: Siddhi query architecture [18].

2.3 Attributes of CEP Scalability

Complex event processing systems have a very wide variety of scalability requirements. Following lists some of the important scalability attributes as identified by Etzion and Niblett [19]:

- The volume of events - The most straightforward variable is the number of input events processed per second. It is also the most benchmarked variable in the literature [20]. Another dimension of the volume of events is the message size. The larger message size can affect the performance of some components quite a bit.
- Event processing agents - A useful CEP system must also scale to support a large collection of event processing agents. Large computations are often most easily presented as simple steps of a complex event processing network. This requires that data can be passed fast from Event Processing Agent (EPA) to next EPA or that there is some other optimization to remove this step.
- Producers and consumers - A CEP system must accommodate many producers and consumers of the data. A substantial number of individual users might want to make their data available to others. The system must also be able to send the processed data and identified complex events to the correct subscribers.
- Window size - Window size has the major role in deciding the complexity of the query. Window size affects on how many events a computation is applied at a time. For example, pattern detection might be applied for all events for the last five minutes. The window size can have drastic effects on rules if the

computational complexity grows faster than linearly with respect to the input size.

- Computational complexity - Even though the focus in this research is in scaling, the traditionally much researched challenge of computational complexity still plays a big role [19]. Most of them depends on how well the computation can be partitioned, parallelised, and distributed [21].
- Environment - The developer of a CEP system must take into account the specifics of the system. Different environments offer variable amounts of memory and CPU cycles. Some environment has limitations in power consumption. In a distributed environment message passing adds limits to latency. Sometimes the bottlenecks can even reside outside the CEP part of the system, for example in input or output channels like message queues or web services transmitting the events [19].
- Constants - There are several other factors that influence the constants of the computational requirements. In addition to the already noted variable message size, its encoding also matters much. While SOAP enveloped, XML-based messages might offer good support for enterprise integration patterns, their serialization and deserialization are computationally expensive compared to lighter and flatter representations.

2.4 Pattern and Sequence Detection

Pattern and sequence detection is the crown-jewel of CEP. Pattern matching lets a business situation be inferred or identified. Pattern detection over event streams is increasingly being employed in many areas including financial services, RFID-based inventory management, click stream analysis, and electronic health systems. A pattern query addresses a sequence of events that occur in order (not necessarily in contiguous positions) in the input stream and are correlated based on the values of their attributes. It involves combining several methods, such as grouping and correlating, as well as filtering and aggregation to identify a specific pattern or sequence of events within or across streams. In the pattern, there can be other events

in between the events that match the pattern condition, but Sequence must exactly match the sequence of events without any other events in between.

Correlations

The first step of applying a pattern or sequence is to group relevant events, forming a "window". These events are correlated using a common set of techniques called window policies; they are temporal windows, spatial correlation, and direct filters.:

1. Temporal windows, also known as time windows, can be used to do a stateful event correlation based on the event occurrence. Based on the time, a "peephole" is created in the event stream, and the state of the previous events in the stream is used with the current event's state to determine a pattern. For example, stock value declined by five percent within one hour of buying the stock.
2. Spatial correlation or dimension-based windows are similar to temporal windows. The difference is that the peephole focuses on number of events rather than time. This technique is also called count windows, as the count of events determines the window. For example, three consecutive high stock prices in the stock ticker
3. Direct filters can be applied on the attributes of the event or on aggregated events. For example, Event.CurrencyPair == EURUSD.

Event patterns are implemented using a specialized state machine approach. The state machine for event patterns listens to events, and at a given time it has a current state. Each state listens to several conditions when new events arrive, Siddhi matches those against the conditions. Then, event arrival order plays a very important role in pattern and sequence detection. State machine remembers any event that matches so far, and if there is an event that does not match, we destroy the state machine instance. When we use '*' or 'or' key works the non-matching events are not blindly dropped, but they stay in the current state till a success matching occurrence, according to the sequence definition.

Pattern and sequence queries can have many Handler Processors; here, they will have a Handler Processor for each incoming event stream [11]. After events are received by the Handler Processor, it passes them to the Inner Handler Processors; these Inner Handler Processors are responsible for processing the states in pattern and sequence queries. Here, the Inner Handler Processors contain all the events that are partially matched up to its state level, and when a new event arrives, it tries to match whether it satisfies its Filter condition along with the partially matched events. If there is a match, it passes the corresponding previously matched events and the current event to the next state (Inner Handler Processor).

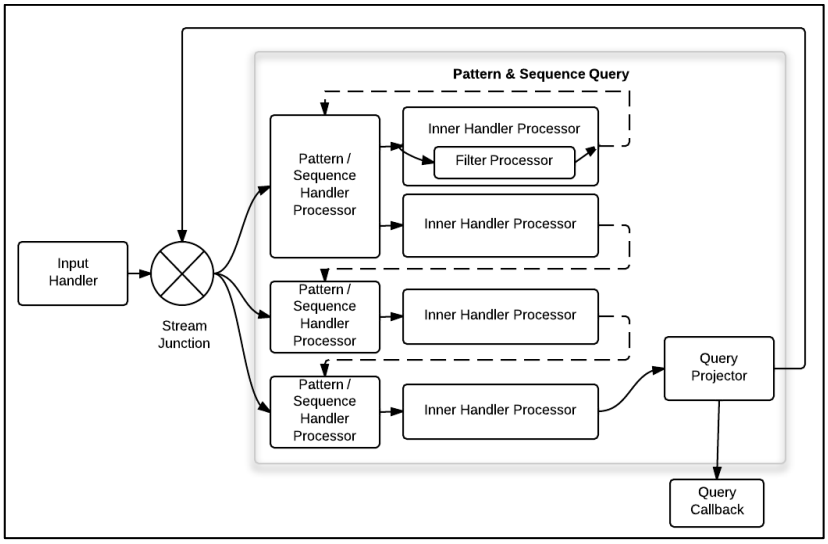

Figure 2.7: WSO2 Siddhi pattern detection flow [18].

2.5 CEP Scaling Techniques

Scaling is such an important functionality that provided by many of the CEP market players. Next, we discuss several commonly used CEP scaling approaches.

2.5.1 Common Types of Scaling

System scalability can be achieved by scaling up or scaling out. Scaling up means using a high performance, faster computer with more CPU cores and more memory power, this is also referred as vertical scaling. Scaling out means adding more computers to a cluster of computers, this is also referred as horizontal scaling. Scaling up offers some clear benefits because it allows the software to run on a single machine with shared memory. This reduces the architectural constraints imposed on the software run on the hardware. Furthermore, management of a single big machine is easier than management of a cluster of computers. Vertical scaling is not always the perfect solution. It was also observed that to efficiently use all the power in one box; it might be necessary to use similar techniques as in a distributed system [22]. This is referred as *scaling-out-in-a-box*. When considering only hardware costs, scaling out can often be cheaper. Horizontal scaling can utilize affordable commodity hardware and the buyer does not have to pay hardware vendor a premium for highly specialized business hardware. However, to leverage distributed hardware the software must be carefully designed to distribute the processing. Execution running on one of the machine instances cannot refer to memory located on another instance. Sometimes the best results can be achieved by combining both vertical and horizontal scaling. Sometimes the biggest possible single machine is not fast enough, and the system builder must resort to a distributed, scale-out architecture [22].

2.5.2 Partition-based Scaling

Partition based scaling is the commonly used approach to scale the CEP [23]. This approach involves partitioning events in a random manner or based on the attribute. Many of the contemporary CEP engines support this approach. Using this approach, we can break the query into several steps in a pipeline that matches events against some conditions and republish the matching events to steps further in the pipeline [24]. Then we can deploy different steps of the pipeline into different machines.

```
from every a1=StockStream[price > 70  symbol ='IBM'] →
             a2=StockStream[price > 70  symbol ='IBM']
      [a1.price < 1.1*a2.prize][within.time=30]
select a1.symbol, a1.prize, a2.prize
```

Query 2.1: Siddhi pattern query to detect unusual stock price change.

For example, let us consider the Query 2.1. This query matches if there are two events within 30 seconds from IBM stocks that having price greater than 70 and having a price increase more than 10%. As seen in Figure 2.8 Option 1, we can break the query into three nodes, and each node will have to republish the matching events to the next node.

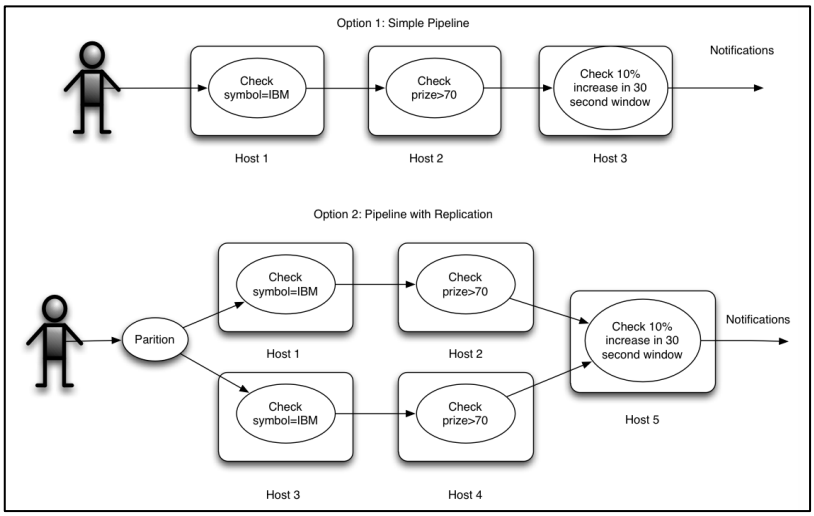

Figure 2.8: Partition-based scaling [25].

However, queries often have other properties that allow further optimization. For example, although the last step of matching price increase is stateful other two steps are stateless. Stateful operations remember information after processing an event, so that earlier events affect the processing of later events while stateless operations only depend on the event being processed. Therefore, we can add multiple instances in the place of those stateless instances using a shared-nothing architecture. For example, we can break the query into five nodes as shown by Option 2 in the bottom part of Figure 2.8.

In addition, another favorable fact is that CEP processing generally happens through filtering where the number of events reduces as we progress through the pipeline. Therefore, pushing stateless filter like operations (e.g., matching against symbol "IBM") to the first parts of the pipeline and scaling them in shared nothing manner should allow us to scale up the system for much higher event rates. For example, let's say that the StockQuote event stream generates 100,000 events per second, but only 5% of them are about IBM. Therefore, only 5,000 events will make it past the first filter, which we can handle much easier than 100k events [25].

2.5.3 Publisher-Subscriber Model

Publish/subscribe systems can typically scale to very high event rates with lots of publishers and subscribers [26], [27]. As seen in Figure 2.8, by using subscriptions events of interest can be sent from event sources to event sinks. In some systems filtering on event attributes can be done. This can be seen as a very simple form of complex event detection. For many applications though, the expressiveness of subscriptions is not enough. To correlate multiple events, applications can be built on top of publish/subscribe systems, to enable for complex event detection. These systems use the advantage of scalability and performance of a publish/subscribe middleware but are not capable of detecting a variety of complex event patterns.

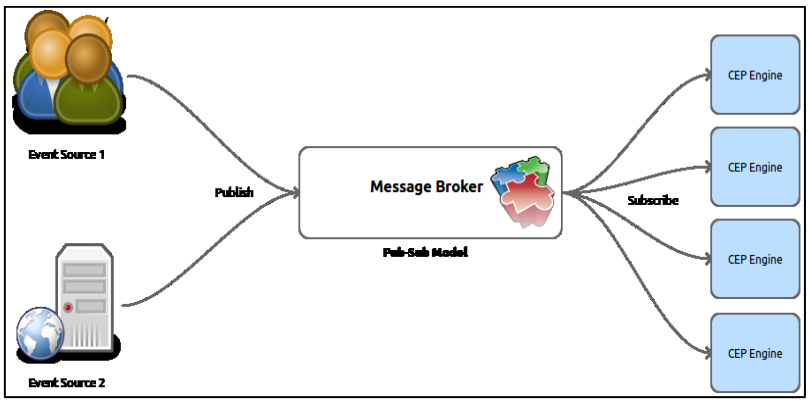

Figure 2.9: Publisher-Subscriber model for CEP scaling.

Here each CEP engine should analyze the deployed queries and subscribe to required event streams in the broker network. Generally, we match each event stream to a queue in the publish/subscribe system. In this approach, subscriber CEP engines can scale without any limitation hence it increases the overall throughput without restriction, and this scaling approach does not depend on underlying implementation of the CEP engine. However, this model increases the per event processing latency due to the Message Broker intervention. This model is not suitable for processing where events that cannot be partitioned by an attribute or random manner.

2.5.4 Storm-based Deployment

Let us discuss a little bit on Storm before moving to discuss scaling with Storm. Storm [28] is a distributed real-time stream processing platform that can be used to assemble and execute stream processing elements. The applications that run on top of Storm cluster are called topologies.

A topology in Storm is a data flow graph of computation, which consists of elements called Bolts and Spouts, connected with streams. Streams are unbounded sequences of tuples, and a tuple is a list of values of any type [29], [30]. The sources of streams are Spouts, which read data from an external source, for e.g., stock exchange, sensors, or program logs. The streams are consumed by Bolts, which do some processing and possibly emit new streams that can be consumed by further Bolts. The processing done at Bolt can be anything, from filtering or aggregation to saving tuples to a database. An example of a topology can be seen in Figure 2.10. Topology can be seen as a blueprint for a runtime computation graph. At runtime, each component of a topology will run within some tasks, which are specified by a parallelism of that component. Every task for the same component executes the same blueprint code, but is a different instance and runs in a separate thread of execution. A task receives a tuple on its input queue, processes it and may emit new tuples to its output streams.

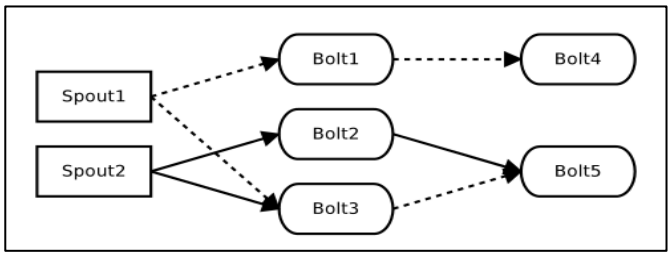

Figure 2.10: Example Storm topology.

Streams are divided between multiple tasks depending on specified stream grouping. Available stream groupings include shuffling stream in round robin fashion, replicating stream to all tasks, or shuffling stream depending on tuple attributes. An illustration of component tasks communicating over streams can be seen in Figure 2.11. Spout nodes which are the event sources push events to one or more Bolt nodes which do event processing. A Bolt can push the processed events to another Bolt for further processing as well.

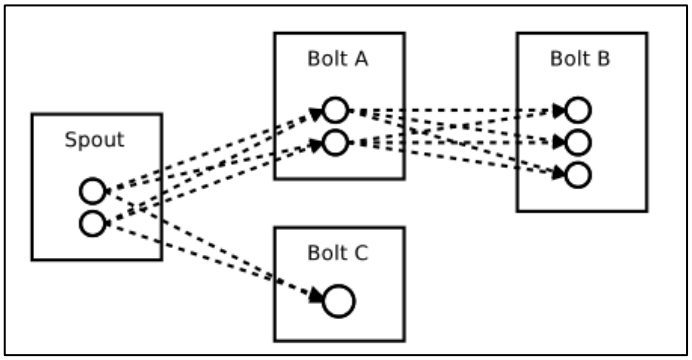

Figure 2.11: Tasks running on Storm topology.

To sum up, topology is a computation graph, where one specifies Bolt and Spout components, the stream connections between them with stream groupings, and parallelism for each component. A Storm cluster can run multiple topologies at the same time. From the time it is submitted, a topology will run forever, or until it is killed. A submitted topology is run on a cluster by specified number of worker

processes. Every worker runs an equal share of topology tasks, and the number of workers or tasks cannot change during topology run. Each Bolt task receives tuples from an input queue and can emit tuples to other Bolt tasks. Spout tasks only emit tuples. This interaction is implemented using an open-source messaging middleware, ZeroMQ [31]. ZeroMQ is a native transport layer implementation of asynchronous messaging, supporting publish/subscribe, request/reply, *N* to *N* and pipeline communication. Storm uses ZeroMQ through native Java binding library.

There are few CEP market players supports Storm-based deployment. Here, we can create queries, deploy them on top of a Storm cluster that runs a CEP engine on each of its bolts, and run it automatically. WSO2 CEP and Esper which are known CEP vendors support this capability [32], [30].

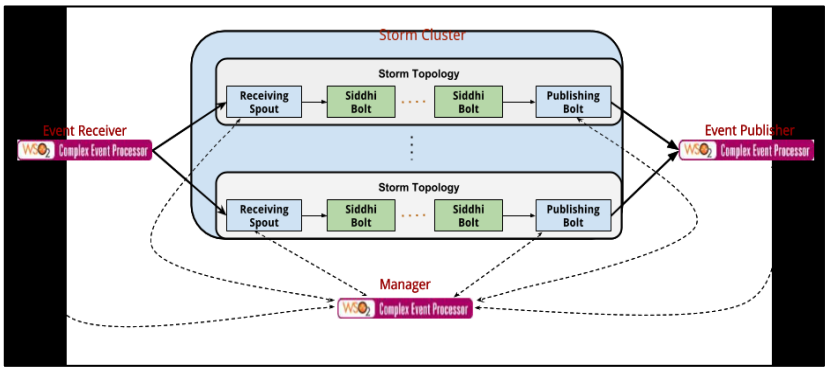

Figure 2.12: WSO2 CEP on Storm cluster.

Figure 2.12 shows a Storm based deployment with WSO2 CEP [33]. Here one or more CEP nodes will be working as Receiver Nodes, Manager Nodes and Publisher Nodes. Here we will be deploying the Siddhi Queries (Execution Plan) through the Management Node and it will take care of splitting the queries and deploying them in the configured Apache Storm cluster, at the same time it will also configure the CEP Receive and Publisher Node to route the event to and from the Storm cluster accordingly.

2.5.5 Distributed Object Cache

A Complex query that needs to maintain a large window and all events in the window would need a large working memory [8]. Then to scale this use case, we need to use a distributed cache to store the working memory. Oracle CEP engine contains this feature, where it uses distributed object cache to scale the complex event processing.

Distributed caching technology can also increase the performance other than scalability of Oracle CEP applications. For example, many event-driven applications need to join stream data with external data, such as data retrieved from a persistent store. A cache can be used to accelerate access to non-stream data, thereby dramatically increasing application performance.

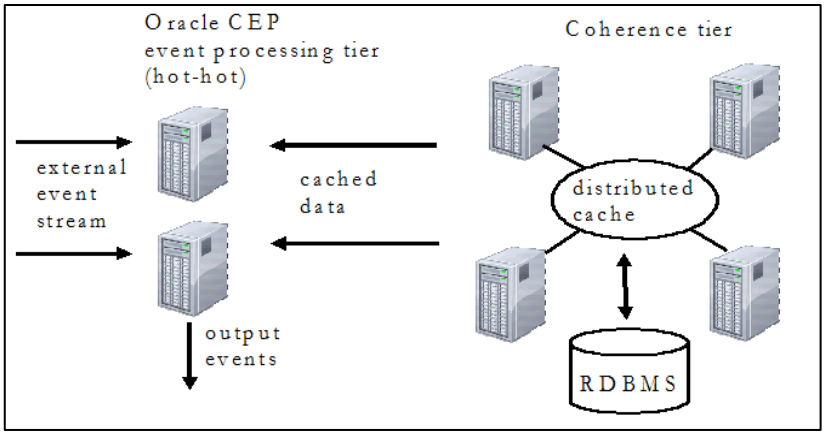

Figure 2.13: Oracle CEP with Distributed Object Cache.

Distributed cache in Oracle is called as Oracle Coherence. Figure 2.13 shows the architecture in which Coherence is combined with Oracle CEP to achieve high availability for the results computed by the Oracle CEP application. Oracle Coherence supports several basic cache configurations: replicated, partitioned, and multi-tiered. Replication is used to increase the availability of cached data. Replicated data is cached redundantly at multiple members of the Coherence cluster

so that if one member fails, another member can continue serving requests for the data. Partitioning, on the other hand, is a technique used to achieve massive scalability. By partitioning data across many machines, the size of the in-memory cache can be scaled up linearly by adding additional machines. Partitioning and replication can be combined to create caches that are both very large and highly available.

Other than in Oracle CEP, distributed cache can be combined with CEP by following some specific approaches as mentioned in [34]. Integration of a CEP engine with distributed caches can go way beyond pushing stuff from one to the other (either input or output). The integration use cases can define as follows:

1. Consume from a cache, emit to a cache. Quite trivial.
2. Deal with very large stream window: the CEP engine must allow to directly swap in the cache as an underlying stream window backend storage, possibly with overflow to disk capabilities as well to trade off latency and capacity.
3. Same as 2) but in a distributed way, so that the entire stream window is shared across several CEP engines for n+1 HA purpose.
4. Integration of streaming data with reference data that sits in the cache for a continuous join between streams and cached reference data. The CEP engine needs to provide an abstraction for that and properly support it in the event processing language.

2.5.6 Integrating with Enterprise Service Bus

Integrating with Enterprise Service Bus (ESB) is another approach used to scale the complex event processing. This project is called as MMEA which is an integration and processing platform for the environmental data [7]. The key architectural insight in the system is to separate the integration functionalities of the ESB and the complex event facilities. This results in a stateless ESB, which can be scaled out by adding more processing nodes. A dedicated CEP cluster can then be tuned to handle high throughput and scaled out separately.

ESB is an integration product [35]. It can be used to connect multiple endpoints in a heterogeneous environment. The enterprise software often uses event-driven service-oriented architecture. Many academics say that as an integration product acting as a centralized mediator for the business events, an ESB is a natural host for complex event processing [36]. The main issue is that the scalability models required by CEP and an ESB are completely different. ESB can often be completely stateless because it usually operates only on a single message at a time. However, in complex event processing, the data dependencies between events can be very complicated. Then, to address this issue, a prototype was developed for complex event processing enabled enterprise service bus, called MMEA Bus [7].

The architecture of MMEA Bus tries to answer this mismatch by deploying CEP as a separate service outside the ESB. Here, a dedicated CEP cluster was built on Storm real-time stream processing framework. The cluster performed complex event processing with multiple Esper CEP engines, which ran on different machines with their contexts. The communication between the Esper engines happens over the network by passing complex events created by the engines.

The architecture used in MMEA Bus is based on distributed, stateless instances of ESB that forward filtered events to a dedicated CEP cluster. Figure 2.14 shows the division of the responsibilities between the ESB and the CEP cluster, the physical setup and the message flows between the machines. The architecture allows exploiting the strengths of both the ESB and the CEP separately. The ESB is used to define adaptations and transports that operate only on one single message at a time. The scalability model utilizes parallel, stateless instances to the extreme, and it is very little, the ESB instances must know about each other. In CEP the data dependencies play a much bigger role. Thus want to limit all the extra work done on the CEP cluster to the minimum. This means that the events supplied to the CEP cluster are already in the correct format and free from other complications, such as access rights management and encryption. The sole purpose of the CEP cluster is to execute the pattern matching and other fundamental functions of CEP.

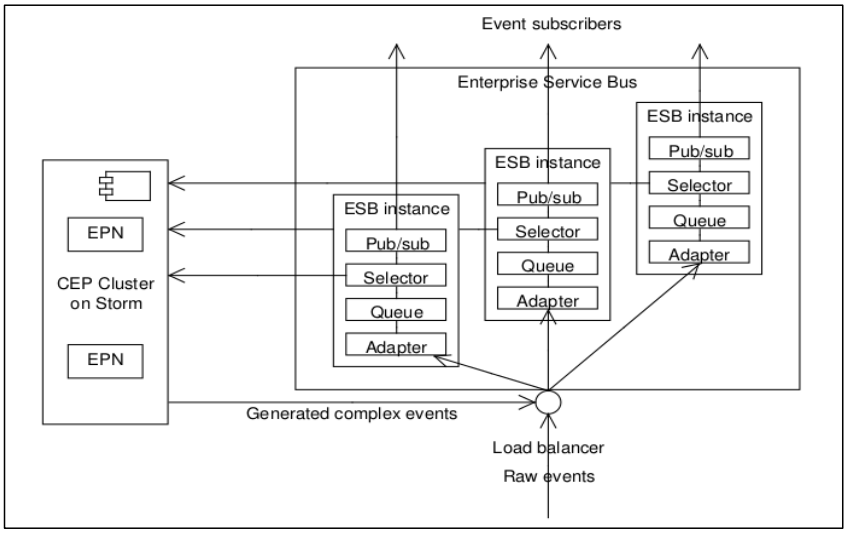

Figure 2.14: ESB in MMEA bus architecture.

Figure 2.15 depicts a logical flow of the events inside the system. Events produced by the sensors are fed to the enterprise service bus, which converts them to an internal format in the adapters. The variety of possible protocols for sending events to the ESB is wide because supporting a new protocol in the ESB only requires writing a new adapter. The currently supported protocols are SOAP over HTTP, HTTPS and Java Message Service (JMS) with various data source dependent file formats. The events received by the ESB are run through a series of filters, which act as selectors for interesting data. The filters are stateless and defined by the users of the data. By defining filters, the users can subscribe to events and event streams they are interested in.

In fact, they implement the first stage of an event processing network and can be seen as event processing agents. For example, filters could be implemented using XPath, which matches the interesting messages by some of its elements. If there are some restrictions (e.g., usage or billing limits) to the event streams the users are allowed to subscribe, they must be applied before these selectors. The selected events are forwarded to a message queue, which is read by a Storm cluster. The Storm cluster

consists of spouts, which read the message queue, and bolts, which run the data through Event Processing Language (EPL) statements (i.e., are event processing agents) or forward the events back to the ESB for further processing (i.e. act as local event sinks). The architecture fully decouples complex event processing from the enterprise service bus. This enables us to handle their scalability separately. In fact, this could even replace the whole event processing system built on Esper [14] and Storm [28] with something completely different.

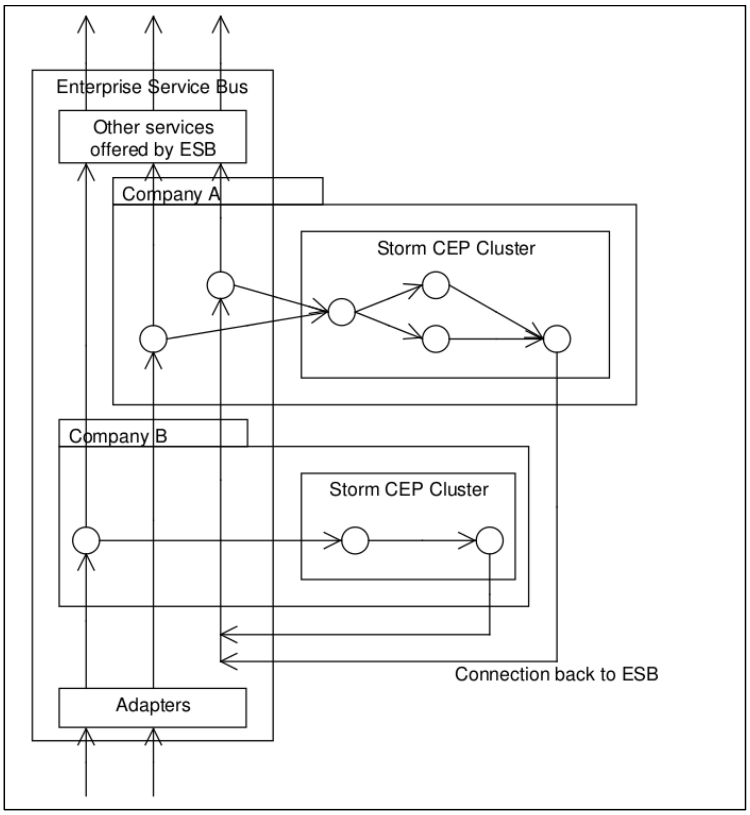

Figure 2.15: A closer look at the MMEA processing model.

The distributed CEP service aims to provide a flexible platform for users to define their event processing networks. There are three kinds of nodes in the Event

Processing Network (EPN). First, there are event producers, which read tuples from a message queue. Second, these tuples are forwarded to the Event Processing Agents (EPA) all running an instance of Esper engine. There may be several statements running on one instance, but it is discouraged, because it inhibits parallelism, as explained later. The third type is a local event consumer, which acts as a leaf node in the network and forwards the received events back to a message queue of the ESB for further processing

The selected level of distribution in this architecture is the engine level. It is a natural choice because Esper does not allow distributing a single engine on multiple machines. This decision imposes some limitations on this implementation. Engine level granularity makes multiple query optimization (MQO) less powerful [21]. MQO would help to prune unnecessary, overlapping computations when multiple queries have similar parts. Still, the queries running on the same machine could make use of MQO locally if the CEP engine supports it. Because there are multiple separate CEP engines running on different machines, the only way they can share data with each other is by messaging, that is, by creating and consuming events. This creates some overhead but makes implementation much simpler. This approach forces the user to think the event processing as a flow of events and defines the desired computations as a network.

The two parts of MMEA Bus can be scaled out separately by adding more processing nodes in a cloud computing environment. The performance tests that performed show that one ESB instance can mediate 1,750 messages of 470 bytes scaled linearly by adding more instances. The throughput of the CEP cluster depends a lot on the computational requirements of pattern detection. Nevertheless, in a simple real-life example case, the throughput was 28,000 events per second on a cluster with eight worker nodes [7]. The latency of the system was very low; usually less than 10 ms. The implemented platform fits its integration purposes well because the ESB product offers a wide variety of different adapters and mediation patterns. But, above performance level is not sufficient for a real complex event scenario because many

CEP engines can perform processing in the range of millions in a second. By fronting CEP by an ESB cause a bottleneck in the processing as well.

2.5.7 Comparison of Solutions

While there are many scaling approaches exist in CEP engines, a single scaling approach can only handle one or few scaling use cases. Some of the scaling approaches can handle a large number of queries, some of them are used for queries that need a large memory, some of them are used to handle queries which are not fit into a single machine and finally some scaling approaches are used handle events which come in high event rate.

Partition-based scaling is the commonly used scaling approach in current CEP engines. This approach can handle a large number of events as well as a complex query that might not fit within a single machine. Through this approach, events are partitioned in random or based on an attribute and processed by several CEP engines parallel. Even though this approach can be applied to many real-time CEP queries, it cannot be applied for the pattern and sequence detection queries where events cannot be partitioned. Publisher-Subscriber model based scaling is also another simple approach which can be applied without much effort. In the approach, numbers of subscribers are increased to improve the scalability. However, this approach can only apply to a query that does not depend on event flow because we cannot make sure that all the related events are subscribed by same CEP engine. Then, this model cannot be applied to a pattern or sequence detection queries.

Storm-based scaling is a novel approach which provides a distributed real-time stream processing platform to improve the scalability for CEP. Storm-based CEP scaling is another variation of partition-based scaling where events are partitioned automatically according to the CEP queries. This approach also contains the same bottleneck that partition-based approach has. Distributed cached based scaling is used by CEP engines like Oracle CEP to improve the scalability. In a CEP engine, a complex query that needs to maintain a large window and all events in the window would need a large working memory. Then to scale this use case, we can use a

distributed cache to store the working memory. This approach is somewhat feasible to handle any CEP scaling use case if there are more computer resources available. However, the important concern of this approach is the complexity of the deployment.

ESB-based scaling is another unique way of scaling CEP compares to other approaches. This approach provides an easy integration or scalable solution for existing CEP engines without much effort. The key architectural insight in the system is to separate the integration functionalities of the ESB and the complex event facilities. This approach does not solve all the limitation that exists in other scaling approaches but provides easy integration with any heterogeneous environment because of ESB support. The main limitation of this approach is performance. Typically, a CEP engine can process millions of events per seconds for simple filter queries, and ESB can only handle few thousands of events per second then by putting ESB in front of the CEP engine causes the bottleneck for CEP processing as well.

2.6 Out of Order Event Handling Approaches

Out-of-order event arrival is present in general data stream processing applications. Handing the disorder consists of a trade-off between result accuracy and result latency. There are several known approaches to handle disordered events as given below.

2.6.1 Buffer-Based Out of Order Event Handling

The key idea in buffer-based [37] approach is to use a buffer to sort tuples from the input stream in ascending timestamp order before presenting them to the query operator. However, due to buffering and sorting, processing of the input tuples gets delayed, and it increases the latency.

2.6.2 Punctuation-Based Out of Order Event Handling

Punctuation-based [38] approaches depend on special tuples (i.e., punctuations) within the data streams. This is to indicate that no future tuples with timestamps smaller than the timestamp of punctuation are expected. Punctuations explicitly inform a query operator when to return results for windows. Therefore, query operator can consume out-of-order input directly. Heartbeats and partial order guarantees are two example punctuation types.

2.6.3 Speculation-Based Out of Order Event Handling

In Speculation based approach [39] results are produced speculatively with or without applying a compensation technique to correct early emitted inaccurate query results when late arrivals are observed. Buffer-based and punctuation-based approaches are conservative approaches. They wait for late arrivals to avoid producing inaccurate results. Speculation-based approaches are aggressive. They assume in-order arrival of tuples and produce the results of a window immediately when the window is closed. When a late arrival event *e* is detected, previously emitted results which are affected by event *e* are invalidated. New revisions of these results are produced by taking *e* into account. However, for data streams that are highly out-of-order, one query result may be revised many times before the final exact revision is produced. This exhausts CPU may cause high result latency.

2.6.4 Approximation-Based Out of Order Event Handling

Approximation based approach [40] is related to computing approximate aggregates over data streams. These techniques summarize the raw data stream with a particular data structure (histograms, q-digests) and produce approximate aggregate results based on these summaries. Current approximation-based approaches follow an aggressive strategy. The difference from speculation-based techniques is that when a late arrival is received, approximation-based approaches only ensure that this late arrival is accounted for in future aggregate computations. However, it does not correct previously emitted results. For queries with small windows, this strategy leads to significant amount of query results with noticeable errors.

2.6.5 K-Slack Based Out of Order Event Handling

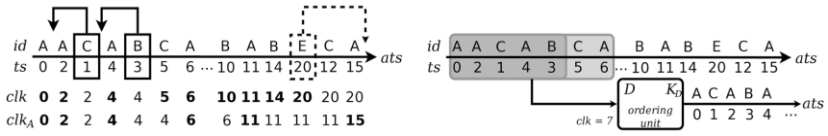

Figure 2.16: Event reordering.

K-slack transparently buffers and reorders events before they are processed by event detectors. It uses a buffer of length K to delay an event e_i for at most K time units (K must be known a priori). It dynamically adjusts the buffer size to a big-enough value to accommodate all late arrivals, aiming to provide near exact query results. Although K-slack has originally being designed for non-distributed, single-threaded stream applications, it has been used in distributed environments as well [41], [42].

2.7 Comparison of CEP Engines

In Section 2.2, we discussed various CEP engines. A summary of findings is listed in Table 2.1. Based on the properties we decided to go for an open-source CEP engine because it will be easier to get the source code and also developer community support for the research if needed. Then, we have drilled down both Esper and WSO2 Siddhi CEP engines from above mentioned CEP products because they are well-known open-source CEP engines.

We focused on some other constraints like easy usability, consumability, high performance and public availability of online resources when making the final decision. Then, we selected WSO2 Siddhi CEP engine because it can process five million events per second for simple filtering queries (as shown in Figure 2.15). This event rate is far higher than the Esper CEP engine's event processing rate [43], [44]. In addition to performance WSO2 Siddhi CEP engine contains many publicly available online resources, which provide more understanding about the product and

its functionalities. In addition, the simplicity of the Siddhi queries provides an added advantage for the research.

Table 2.1: Comparison of CEP engines.

Feature / CEP Engine	Esper	Oracle CEP	Apama	WSO2 Siddhi
Open-Source	Yes	No	No	Yes
Performance (events/second)	2M	0.04M	0.0002M	5M
Query Language	SQL like	SQL like	SQL like	SQL like
Online Resources	Many	Few	Few	Many
Developer Community	Yes	No	No	Yes
Easy Use	Better	Good	Better	Best

2.8 Summary

In this chapter, we have provided an introduction to CEP engine and discussed the features and architecture of some well-known CEP engines like Esper, Oracle, Apama, and WSO2 Siddhi CEP engine. We also discussed the attributes of CEP scalability and important of the pattern detection and how it works as well. In the later part of the chapter, we have provided information about some commonly used and well-known scaling techniques like partition based scaling, publisher-subscriber based scaling, Storm based scaling, distributed object cache based scaling and scaling through ESB. Here, we have compared the pros and cons of existing scaling approaches that mentioned above. We have also done some comparison of the CEP engines and decided to go with WSO2 Siddhi CEP engine due to the open source nature and performance capabilities of the engine.

3. METHODOLOGY

We propose a scalable solution for the pattern and sequence detection queries that can be implemented on most CEP engines. The proposed approach is independent of the internal implementation of a CEP engine, which allows incorporating with most of the CEP engines. Section 3.1 presents details of the proposed solution and discusses the key technologies and features of the solution. Implementation of the proposed solution on top of WSO2 Siddhi CEP engine is explained in Section 3.2. This section is also discussed on how events are partitioned with time and how pattern detected events are reordered and duplications are handled through the solution.

3.1 Proposed Solution

Among many approaches to scale Complex Event Processing (CEP), partition-based scaling is the most popular and accepted approach in the industry [45]. While partitioning is typically based on attribute value or multiple attribute values, we propose a technique based on time, where events are partitioned based on the *within* time that we define in the query. *within* time variable defines the time window that should be considered for detecting a pattern or sequence query.

Figure 3.1 illustrates the key stages of the proposed solution. Initially, incoming events are partitioned based on time. Then the pattern is detected within a partition. Finally, pattern detected events are pushed to next stages to remove duplicated events and reorder them. This approach enables us to scale pattern and sequence detection operations, which are required to perform a whole set of events without grouping by an attribute. Because the events are partitioned based on time with some overlapping events, as well as processing occurs simultaneously in multiple CEP engines, this may result in duplicate pattern detections and leads to duplicate output events. Moreover, events could reorder as patterns can be detected at different times due to parallel processing of partitions by different CEP engines.

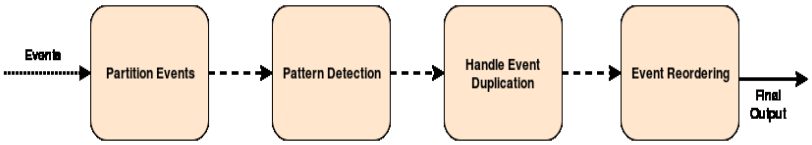

Figure 3.1: Overview of the solution.

Next, we provide more details about the proposed technique. Here, we discuss the solution, side effects that arise when applying the suggested solution, and possible ways to overcome those issues.

3.1.1 Partitioning Events by Time

Events are pushed from various event sources to the CEP engine with a high throughput. In this case, incoming events get queued at the entry to the CEP engine. Then events in the queue are partitioned based on time values. Then each partitioned event group is pushed to one of the parallelly running CEP instances. To understand further, let us consider the following example Siddhi query for pattern illustrated in Figure 3.2:

```
from every h1 = hitStream -> h2 = hitStream[h1.pid != pid and h1.tid == tid] -> h3
= hitStream[h1.pid == pid]
within 5 seconds
select h1.pid as player1, h2.pid as player2, h3.pid as player3, h1.tsr as tStamp1 ,
h2.tsr as tStamp2 , h3.tsr as tStamp3
insert into patternMatchedStream;
```

Query 3.1: Siddhi query to detect ball passes between the players.

Above Sidhi pattern query works on events that come through a stream called *hitStream*. *hitStream* contains the event related to a ball hit during a football game. Here we are looking for following three states,

1. Ball hit from a player *x* of team 1
2. Then, a ball hit from another player *y* of opponent team *2*
3. Finally, a ball hit from the same player *x* who hit first.

Moreover, these three states need to happen within five seconds.

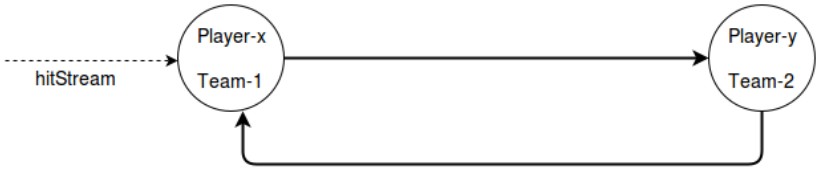

Figure 3.2: Example pattern query.

The three states are not required to occur one after another. As it is a pattern query (and not a sequence query), there could be zero or more intermediate states. To be relevant, these state transitions should happen within a time interval (as indicated by the value of "within" variable). For example, as shown in Figure 3.3, if within value is five seconds then we partition the events as 10 seconds batches by overlapping five seconds event groups of two consecutive partitions. By partitioning the events into batches of twice the *within* time, we make sure that no patterns or sequences are missed due to event partition and processing them individually.

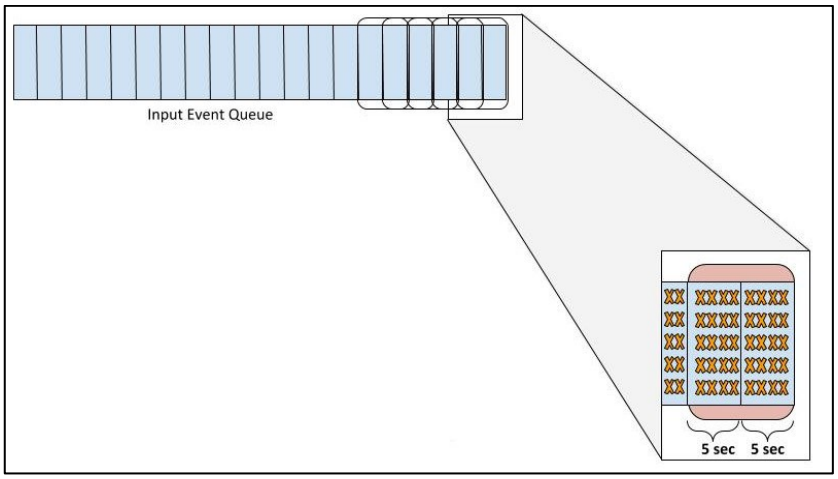

Figure 3.3: Partitioning input events by timestamp.

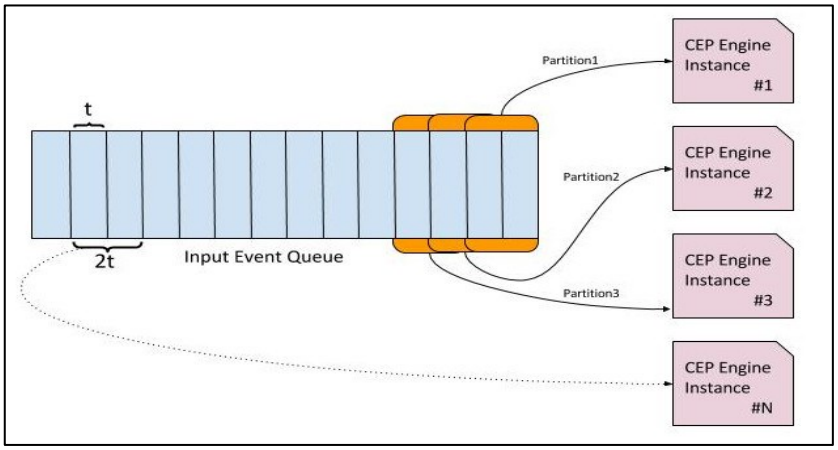

Figure 3.4: Partition distribution among CEP instances.

As shown in Figure 3.4, once the events are partitioned based on timestamp, each partition is pushed to one of the CEP instances and processed parallelly. Here, CEP instance is shut down once processing is completed and re-initiated when it is required but number of CEP instance count will be less or equal to the user-defined value. We can instantiate a new runtime instance and scale up the processing automatically based on the input rate of events. Then each runtime instance can process partitioned event groups in parallel and output those processed events. Because of this parallelism, CEP can accept and process events at a higher throughput. It is a good approach to have a pool of Siddhi CEP engines and use them for event processing rather spawning a new Siddhi engine instance for each event partitions. But, we have to reset the state of the CEP engine before using it for processing of another event partition because if not it can produce some unexpected patterns or sequences. But as per the implementation of WSO2 Siddhi CEP engine, there is no way to flush the current state of the engine and reset it. Due to this, we have to spawn a new Siddhi CEP instance to process each event partitions.

3.1.2 Handling Event Duplication and Reordering

As we partition events by time interval with some overlapping of events between partitions, it could result in event reordering and event duplication that we need to consider and solve. Next, we discuss a possible solution to address event duplication and reordering.

Event duplication can be handled using a HashSet-based data structure [46]. HashSet creates a collection that uses a hash table for storage. Hash table stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code. The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically. Here, we have written the hash function of the event which returns the hash code by considering the attributes of the event. If attributes of events are equal, then hashcode of those events will also be equal. The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.

Figure 3.5 illustrates an example scenario of out-of-order events arrival. The event IDs are shown above the arrow and their corresponding event generation timestamps are shown below the arrow. In this example event e_{10} and e_{13} arrive late in the event streams. As shown in Figure 3.3 events are pushed to multiple Siddhi engines parallelly. In this case, there is a high possibility where events can get out of order.

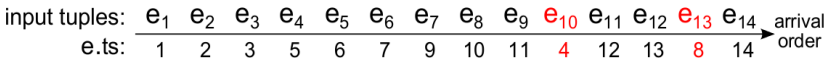

Figure 3.5: Out of sequence events.

As discussed in Section 2.6, there are several known approaches available to handle out of order events. They are buffer based approach, punctuation based approach, speculation based approach, approximation based approach and K-Slack based approach. Here, we have used K-Slack based implementation to overcome event

disordered issue. K-Slack based implementation is straightforward compared to the other scaling approaches that we discussed in Section 2.6. Even though K-Slack based approach increases the latency due to buffering and sorting delays when processing input events it has an important advantage where it dynamically adjusts the buffer size to a big enough value to accommodate all late arrivals, aiming to provide near exact query results. And also, K-Slack based approach does not need to keep entire history of the query results as like punctuation based event reordering approach. And also it does not require high CPU for processing and cause high result latency as like speculation based approach. The HashSet based data structure incorporated with the K-slack based implementation to handle the event duplication and event reordering at the same time to avoid unnecessary latency increment when detecting pattern or sequences.

3.2 Implementation

The proposed solution is to be implemented on the WSO2 CEP Siddhi engine. Siddhi is selected as it is open-source, has low latency, and capable of analyzing millions of events per second [47]. As shown in Figure 3.6, following steps are followed while implementing the proposed solution on Siddhi:

- Buffer the incoming events in a queue to partition them.
- Events are partitioned based on the time interval using the event timestamp. Here, partition time interval depends on the within time that mentioned in the query. For e.g., if within time is t seconds then partition time is $2t$ seconds.
- Instantiate a set of Siddhi CEP engines and push event partitions into it.
- Then at each CEP engine pattern recognition is performed based on the event arrival time to the Siddhi CEP engine.
- Pattern detection happens simultaneously in different siddhi engines and results will be pushed to another processing layer to perform reordering and duplication handling.

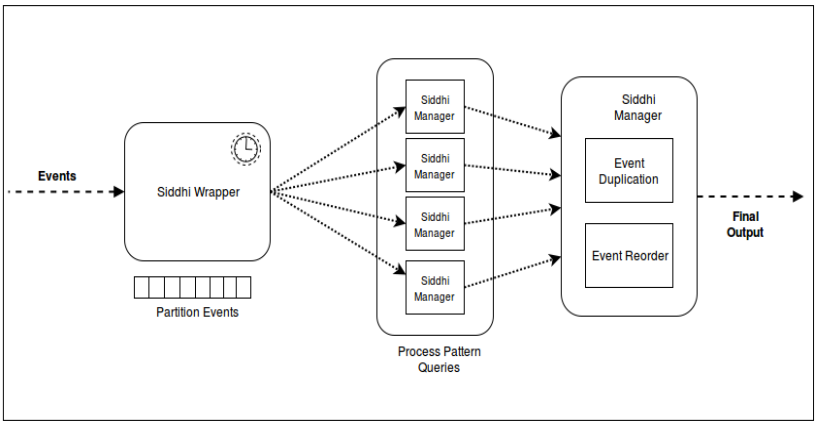

Figure 3.6: Overall processing in Siddhi CEP engine.

Next, we explain the detailed implementation on top of WSO2 Siddhi CEP Engine. Siddhi Wrapper component act as the overall processing manager, which controls overall scaling and processing of events. Siddhi Wrapper performs two primary operations as follows:

- Partitioning the events based on “within” time interval of the query.
- Instantiate multiple Siddhi manager instances and push events to them.

Here, we used a queue-based data structure to store the event partitions; internally each event partition is also an event queue. When events are received at high arrival rate, they are queued and partitioned based on *within* time interval value that is mentioned in the Siddhi pattern or sequence query.

Let us consider the following example scenario. If within time value is five seconds then we will partition the events by ten seconds (5 seconds × 2). Here time duration is calculated based on the timestamp value tagged in the event, but not with the system timestamp. As shown in Figure 3.7, events are partitioned as ten seconds event batches by combining two successive five second batches based on the timestamp defined in the event itself. Based on the inputs that were shown in the figure, they are divided into three partitions.

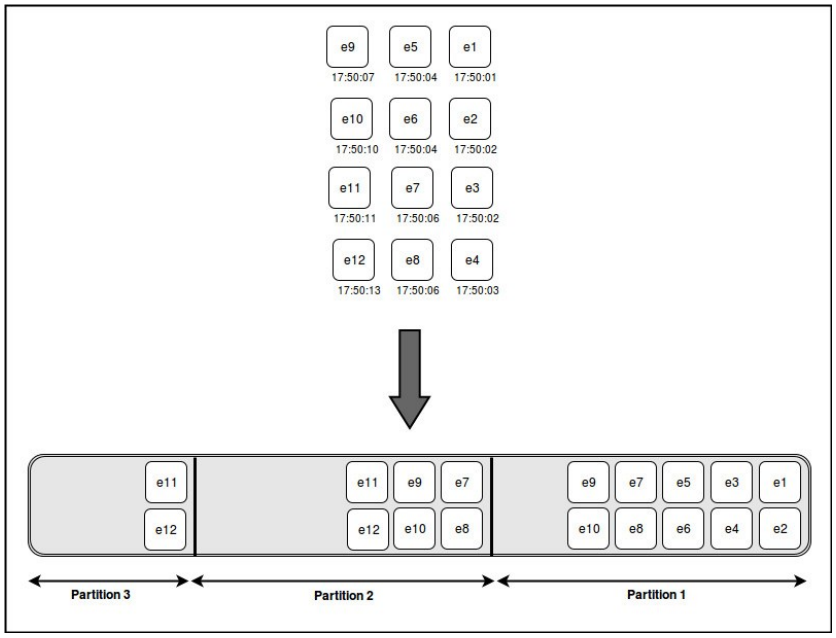

Figure 3.7: Partitioning events based on within time interval.

In the meantime, Siddhi Wrapper instantiates necessary Siddhi processing engines instances in a proactive manner. Then, it pushes the event partition to Siddhi manager instance for processing. Each Siddhi manager instance runs similar query as defined in Query 3.1 and process incoming events. Number of Siddhi engine instances is a user configuration parameter of our implementation. Users can configure an appropriate value for the parameter by considering the input event rate and machine hardware resources.

As illustrated in Figure 3.6, after processing the event partitions, output events of the pattern query is sent for further processing to another Siddhi processing instance to remove the duplicated events and rearrange the order of them.

Siddhi CEP engine process events based on system timestamp, and it does not have the capability to do the processing using the user-provided external timestamp

(attribute of an event which specifies the timestamp of event origination). Let us consider the following scenario. If there is a pattern query which spans through (or within time interval) each five seconds, then events are pushed to multiple Siddhi CEP engines as shown in Figure 3.8.

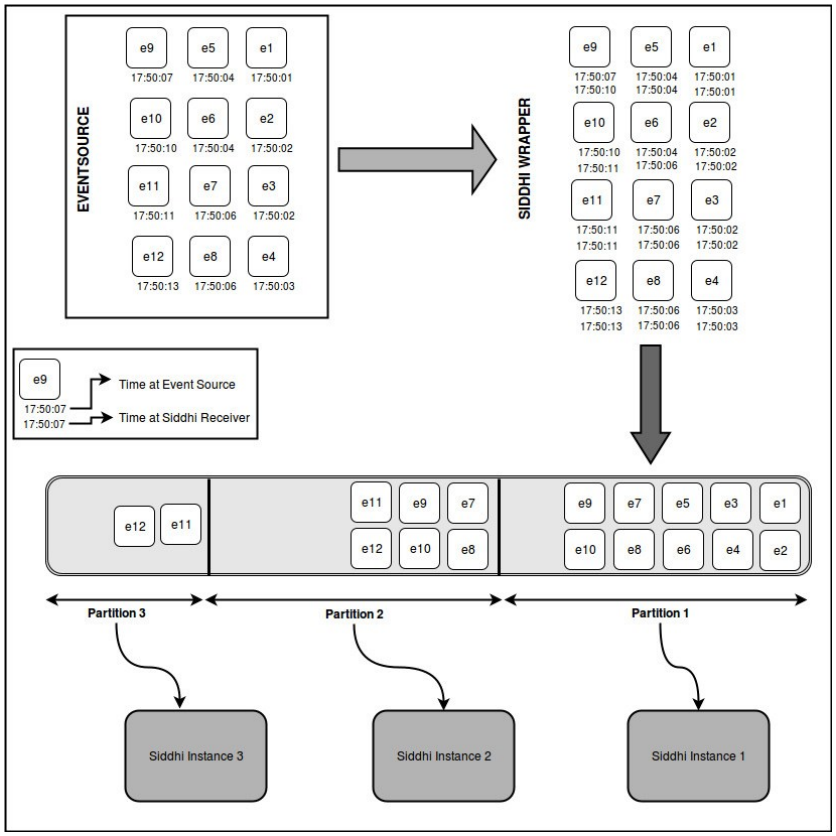

Figure 3.8: Partitioning and processing based on external time.

If we do the processing based on external timestamp (which is an attribute), then we will consider the time that event is originated at event source for the processing. As illustrated in Figure 3.8, event partition is based on the time that an event gets originated from an event source. If partitioning is done based on the system

timestamp (or time that event arrives at Siddhi engine), there could be some differences in events that are in the partition as shown in Figure 3.9. This has an impact on overall processing and results as well, if events are partitioned based on system timestamp then the grouping of events depend on some external factors like event source efficiency, network delay, and IO operation. These external factors can change the way that an event gets grouped, due to this reason pattern detection is also get affected and leads to wrong outputs.

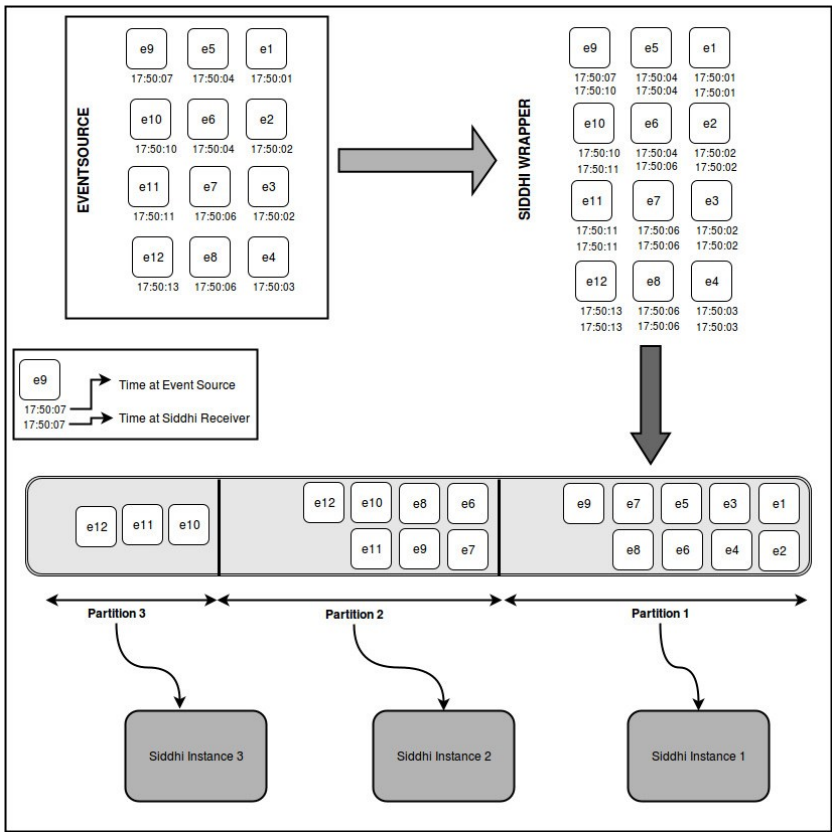

Figure 3.9: Partition events based on System Time.

To overcome this behavior, some changes are required at Siddhi core to compare the timestamp of the event for the time-related operations rather considering system timestamp. Due to this change, overall processing would be more accurate as network/transport level delays or publisher level delays are neglected. This helps the evaluation/benchmarking process as well because we do not have to worry about keeping same publisher frequency and other external changes.

After processing the events by multiple Siddhi engines parallelly, the output will be pushed to another Siddhi engine instance to handle event duplication and event reordering. Here, Siddhi engine runs a query which is similar to the following:

```
define stream patternMatchedStream (player1 string, player2 string, player3 string,
tStamp long, tStamp1 long, tStamp2 long); ");

from patternMatchedStream#window.kslack(10000) select *
insert into filteredOutputStream;
```

Query 3.2: Siddhi query which reorders and remove duplicate events.

In above, events are sent to a Siddhi extension called K-slack to perform event reordering and to remove duplicated events. This K-slack based extension is a custom Siddhi window processor extension which buffers events and processes them based on provided configuration.

Figure 3.10 illustrates the time batch window based K-slack implementation. Incoming events are buffered and reordered based on the timestamp of the event, at this situation K-slack Siddhi extension also verifies the hash value of the event to detect duplicated events and remove them; after that, those reordered events will be sent out based on fixed timer event.

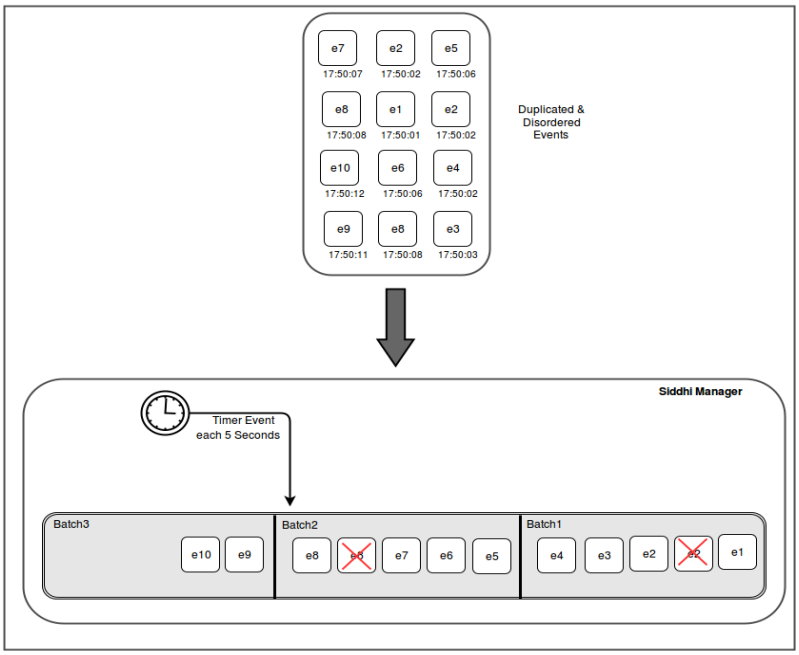

Figure 3.10: K-slack based extension of Siddhi.

As specified in Query 3.2, K-slack based extension accepts one parameter which is the fixed timeout value (specified in milliseconds) set at the beginning of the process. This extension uses the corresponding timestamp of the event for ordering the events. Once the timeout value expires, the extension drains all the events that are buffered within the reorder extension to outside. Internally the timeout has been implemented using a timer. Each time when the timer ticks, the events which are buffered within the extension are released.

3.3 Summary

We discussed the implementation of the proposed solution and how it was developed on top of WSO2 Siddhi CEP engine. Our proposed approach contains three primary steps which are partition events by time, handling event duplication and event reordering. In our proposed approach incoming events are sent to the Siddhi wrapper component which partition events based on event timestamp and push those

partitioned groups to multiple siddhi engines which are running parallel. Each Siddhi engine executes the pattern or sequence queries and sends the output to another Siddhi instance to perform event reordering and to remove duplicated events. Here, we have used K-slack based window implementation to reorder events and HashSet based data structure used to remove duplicated events.

4. EVALUATION

This section provides the details about performance analysis of the proposed time-based partition scaling approach. Section 4.1 discusses the benchmark and its features that used for the evaluation. Section 4.2 presents the experimental setup and hardware configurations. Section 4.3 analyzed the throughput improvement of the proposed solution with compared to the existing default Siddhi CEP engine. System resource (memory, CPU, and thread) utilization details are discussed in Section 4.4. Section 4.5 presents more in-depth information on accuracy metrics while latency variations are discussed in Section 4.6.

4.1 Soccer Monitoring Benchmark

Soccer monitoring benchmark is based on the DEBS (Distributed Event Based Systems) 2013 Grand Challenge [48], which we referred to as SMB2013 benchmark. SMB2013 benchmark focuses on conducting real-time streaming analytics on a soccer game. Figure 4.1 illustrates the flow of data in SMB2013. As shown in the Figure, incoming events are sent through a stream called *sensorStream* and it is get partitioned based on the *within* time that defined in the query then it goes through few set of queries to perform initial filtering and events are output through a stream called *hitStream*. Pattern detection occurs in the next step by looking the events in *hitStream* and pattern detected events are output through a stream called *patternMatchedStream*. Finally, event reordering and duplication handling occur for the events of *patternMatchedStream* and final output sends through *filteredOutputStream*.

The data used for this benchmark is collected by the real-time locating system deployed on a football field of the Nuremberg Stadium in Germany and it contains 47 Million rows events. Data originates from sensors located near the players' shoes (1 sensor per leg) and in the ball (1 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.

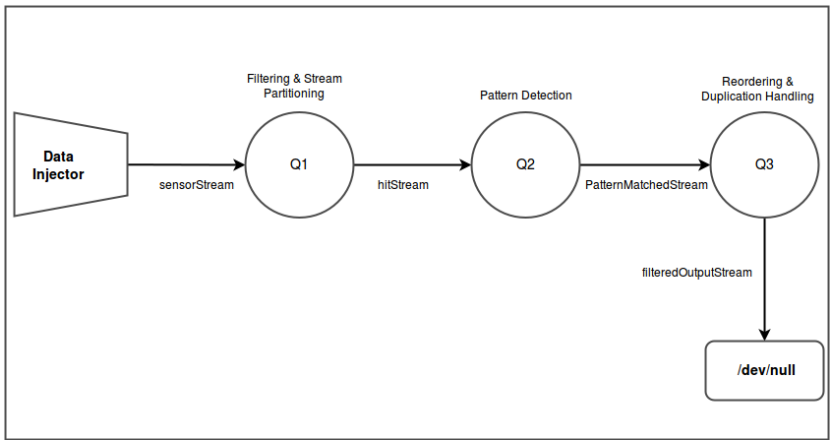

Figure 4.1: Soccer monitoring benchmark event flow.

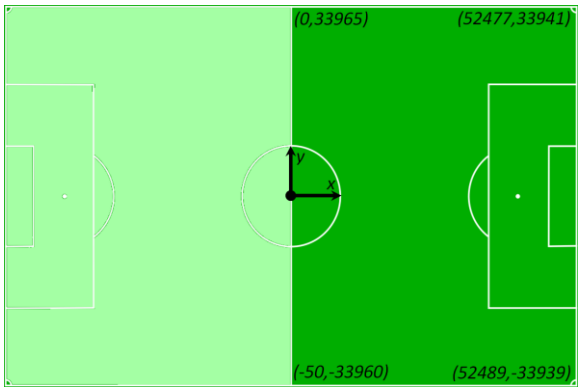

Figure 4.2: Playing field and its dimensions [48].

The total data rate reaches roughly 15,000 position events per second. Every event describes a position of a given sensor in a three-dimensional coordinate system. The center of the playing field is at coordinate (0, 0, 0). See Figure 4.2 for the dimensions of the playground and the coordinates of the kickoff. The event schema is following:

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

Table 4.1 describes the attributes of the event schema.

Table 4.1: Description of event attributes.

Symbol	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

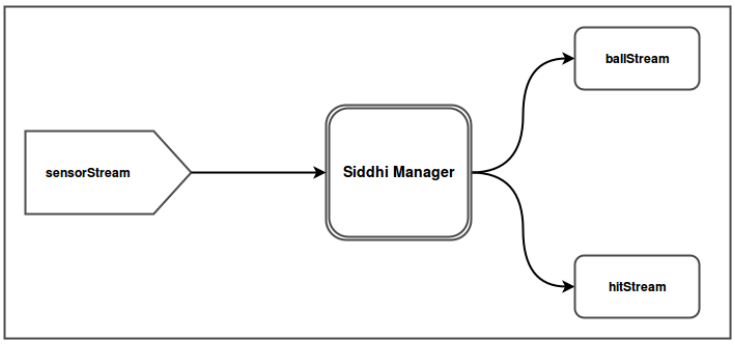

Figure 4.3: Stream partitioning of data.

Here, the data file contains events that generated from sensors. In the benchmark, these events are pushed to sensorStream; then those events will be divided into ballStream and hitStream as shown in Figure 4.3. This benchmark uses the hitStream and find out the following pattern (logic) and notify:

1. Ball hit from a player-x of team-1
2. Then, a ball hit from another player-y of opponent team-2
3. Finally, a ball hit from the same player-x who did it first.

The corresponding Siddhi query is as follow:

```
from every h1 = hitStream -> h2 = hitStream[h1.pid != pid and h1.tid == tid] -> h3
= hitStream[h1.pid == pid]
within 2 seconds
select h1.pid as player1, h2.pid as player2, h3.pid as player3, h1.tsr as tStamp ,
h2.tsr as tStamp1 , h3.tsr as tStamp2
insert into patternMatchedStream;
```

Query 4.1: Siddhi pattern query used for evaluation.

Following tests are conducted for various combinations to analyse the throughput and accuracy of the proposed solution:

1) Throughput against number of CPU cores.
2) Throughput against different within time (i.e., batch time).
3) Accuracy against number of Siddhi instances. Accuracy is defined in Section 4.5.
4) Accuracy against within time (i.e., batch time).

4.2 Experimental Setup

We used two different computers with different CPU configurations. One of them contains 32 core CPU, and other one contains 16 core CPU. Each computer instance was pre-configured with Oracle JDK 1.7.0_79-b15. All the tests are carried out by allocating -Xms16g -Xmx18g for memory and we used the Siddhi Engine version 2.2.1.

4.2.1 Prototype

Tests were carried out using the implementation in Section 3.2. We used Siddhi version 2.2.2 to implement the proposed solution. To simulate the Siddhi manager instance, we spawned separate threads and partitioned data pushed to them. The component that was implemented is wrapped as a separate jar file with other dependencies and ran with JDK 1.7_79.

Here, number of Siddhi engine instances and batching time are configurable and can be passed as Java arguments to the test class that implemented. Usage of such prototype made it possible to simplify the testing process by reducing the testing time. Tests were conducted by setting multiple possible values for Siddhi instance count and batching time to get the better understanding about the performance and usability of the implementation.

4.2.2 Hardware Configuration

As our primary goal is to scale the pattern and sequence queries to handle a large number of events per second (TPS), we have to run multiple Siddhi CEP engine instances to verify the solution. Therefore, we used two high-end machines for tests. Most of the tests were carried out on a machine with 32-core Intel^{R} Xeon^{R} CPU E5-2470. Based frequency of the processor is 2.30 GHz and turbo frequency is 3.10 GHz. It has 20 MB L3 cache. For all the tests, we allocated 16 GB of minimum memory and 18 GB of maximum memory by setting the Xmx and Xms values for Java Virtual Machine (JVM). To verify the throughput against the number of CPU core, we used another machine with 16 core CPU which has the same processor.

4.3 Throughput of Scaled Solution

As our primary objective is to achieve high throughput, we compare the performance of the proposed technique with the performance of default WSO2 Siddhi CEP engine. Same workload and set of queries were used in both cases. It makes sense to get statistics with default Siddhi engine and compare those values with the statistics that we got with our implemented solution because our solution is implemented on top of Siddhi CEP engine and we can clearly compare the performance results between those in a straightforward manner by following this approach. We also made sure that both tests were carried out with same system environmental parameters and configurations.

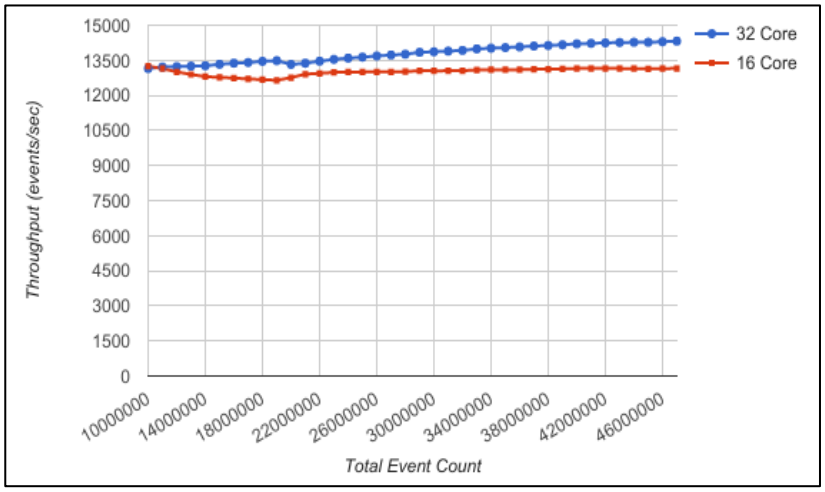

Figure 4.4: Throughput of the default Siddhi CEP engine.

The initial test was carried out against the default WSO2 Siddhi CEP engine to measure throughput with the SMB2013 benchmark. As seen in Figure 4.4, 47 Million events were pushed to the CEP engine and throughput was calculated for each batch of one Million events of the distribution. As shown in figure initial throughput of the first set of one Million events is 12,509 events/sec. After that throughput steadily increased for next set of one Million events and went up to nearly 14,000 events/sec and gets stabilized in the 32 core CPU machine. And the similar behavior was observed in 16 core CPU machine as well where throughput was stabilized at 13,200 events/sec. We can consider the initial phase of the test as the warm up period which allows to byte-code optimization. Overall throughput was distributed between 12K to 14K events/sec for all the set of events. In Figure 4.4, there were some small hiccups for the initial set of events that pushed. We believe this is due to the warm-up of the performance test.

As shown in Figure 4.4, there is no much difference observed in throughput when running the SMB2013 benchmark in 16 core and 32 core CPU machines. WSO2 Siddhi CEP engine uses around from six to eight threads when processing a pattern or sequence query. To serve this requirement, it is required to have a four core CPU

machine. Even the processing happens on a machine which has more CPU cores; it does not use CPUs more than four cores since the implementation of Siddhi does not require it. As per the pattern and sequence design of Siddhi, events are processed sequentially then it does scale even more hardware resources are available.

As you can see from Figure 4.5, tests were carried out on two different machines which have 32 core and 16 core CPUs. As per the proposed solution we enabled multiple Siddhi CEP engines. Statistics related to throughput was collected across all Siddhi Instances.

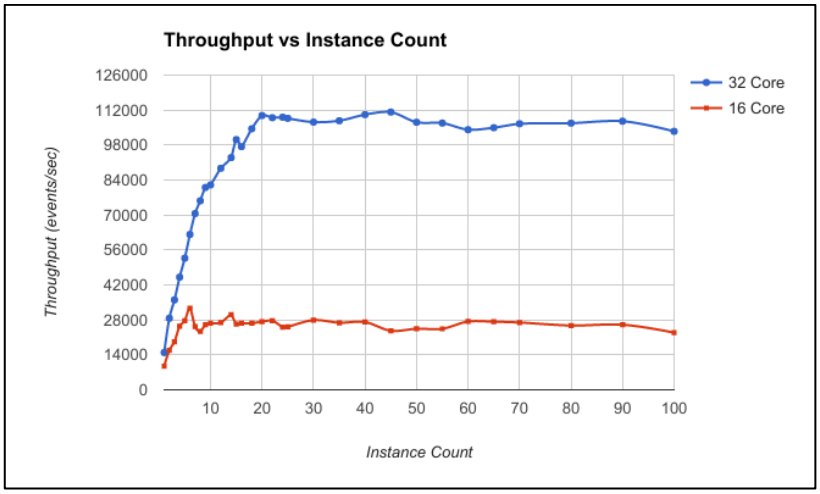

Figure 4.5: Throughput in multi-core machines of the proposed solution.

Throughput gradually increased from 14,921 events/sec to 109,815 events/sec when Siddhi instances are added one by one up to 20 from one. After that, throughput gets stabilized and continued between 100K to 120K events/sec until 100 Siddhi instances are spawned. While a similar pattern was observed on 16-core CPU machine, the resulting throughput was considerably lower. In this test, throughput gradually increased up to 32,621 events/sec from 9,488 events/sec when Siddhi instance count was increased to six from one. Throughput stabilizes between 25K to 28K events/sec.

This was the same behavior that observed when running the solution with parallel Siddhi CEP engines which are counted from seven to 100.

As we are starting each Siddhi engine on a different CPU level thread, multiple threads are created and they can execute in parallel on available CPU cores. Consequently, the machine with 32-core CPU outperforms 16-core machine. Therefore, we can get better throughput by increasing the number of parallel CEP engines. As mentioned in the methodology section, in the proposed solution events are partitioned based on within time interval given in the query. For example, if within time interval is given as 30 seconds then event are partitioned by 60 seconds (30 seconds x 2) with overlapping events between the partitions. Because events are accumulated in the memory for some time, it has an impact on overall throughput and processing of the solution.

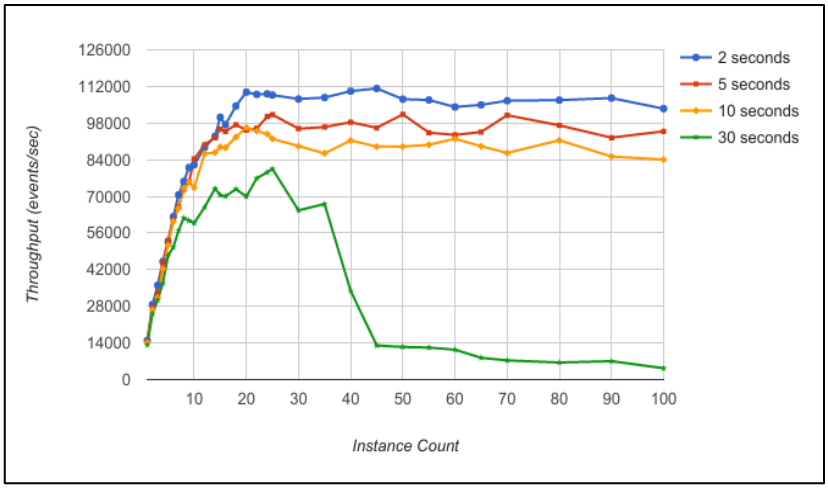

Figure 4.6: Throughput in multi-core machines of the proposed solution.

Tests were carried out with four different possible within time values as 2, 5, 10, and 30 seconds by changing the Siddhi query that used for pattern detection in the benchmark. When within time was set as two seconds, throughput is increased from 14,921 events/sec to 109,815 events/sec when 20 Siddhi engines are initialized. After

that throughput is stabilized, and it was between 100K events/sec to 110K events/sec range. When within time is five seconds, throughput is 14,428 events/sec when single Siddhi engine was processing events. However, it gradually increased up to 95,280 events/sec when 20 Siddhi engine instances are processing events in parallel. After that throughput is stabilized and settled to a value between 85K and 95K events/sec. If within time is 30 seconds, throughput is 13,267 events/sec when single Siddhi engine instance is initialized. It then gradually increased as multiple Siddhi engine instances were added. Throughput reached 80,482 events/sec with 25 Siddhi engine instances. However, there is a drop in the throughput and it decreased suddenly to a low value. After initializing 45 Siddhi engines, throughput is decreased to 13K events/sec, and it is reduced further when adding much more Siddhi engine instances for processing.

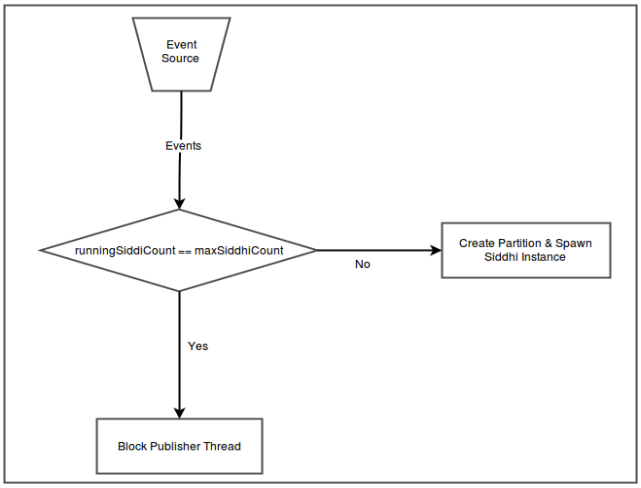

Figure 4.7: Event partitioning logic.

When the within time is 2, 5, and 10 seconds throughput gradually increased and get stabilized. However, when within time is 30 seconds even though throughput is increased gradually, it does not become stabilized rather it get dropped to a lower value. If within time is 30 seconds then events are get partitioned as 60 seconds batches. That means it contains many hundred thousands of events in a single

partition. As shown in Figure 4.7, if the maximum number of Siddhi engine instance count is high then there is a possibility where multiple partitions are get created and kept in the memory. In this situation, 60 seconds partition runs out of memory and starts swapping to disk space; this drastically reduces overall throughput.

4.4 Resource Utilization

We have gathered necessary system statistics to get a better understanding about the system resource (memory, CPU, and thread) usage. We used Java Flight Recorder to retrieve system resource related information and analyzed it using the Java Mission Control tool [49]. Below diagrams, which are represented by Figures 4.8, 4.9, 4.10, 4.11, 4.12 and 4.13 are derived from that.

We have used the same SMB2013 benchmark for below evaluation purposes as well. As defined in the query *within* time is set as two seconds for both evaluations of default Siddhi CEP engine and our proposed solution. The graphs that are related to the proposed solution are related to the processing while having 20 parallel WSO2 Siddhi engines. We have used 20 parallel Siddhi engines for the evaluation of the proposed solution because we have achieved maximum throughput when having 20 parallel Siddhi CEP engines as shown in Figure 4.6. As per our tests, we have observed 800% of throughput improvement in our proposed solution compared to the default Siddhi CEP engine.

Figure 4.8 shows the CPU utilization by the default WSO2 Siddhi CEP engine. It has consumed nearly 5% of the CPU consistently at all the time while processing events. This is expected as we are running a single instance of the Siddhi engine. Figure 4.9 shows the CPU usage when running our proposed solution. As per the figure, you can see it has consumed 60% of the CPU when processing events in most of the time and there is a reduction in the CPU usage time to time as well. This behavior is expected since our solution instantiated 20 Siddhi engines simultaneously and there are around 28-30 active threads at most of the time. This means it required around more than 15 cores to run these threads as each CPU core can handle two threads at a

time as per the system configuration. Once processing is completed each thread which runs the Siddhi CEP engine is stopped, and another thread recreated to instantiate another Siddhi engine instance, that is why there is a sudden reduction in CPU usage in some situations. This behavior is consistent across the complete processing time of our proposed solution.

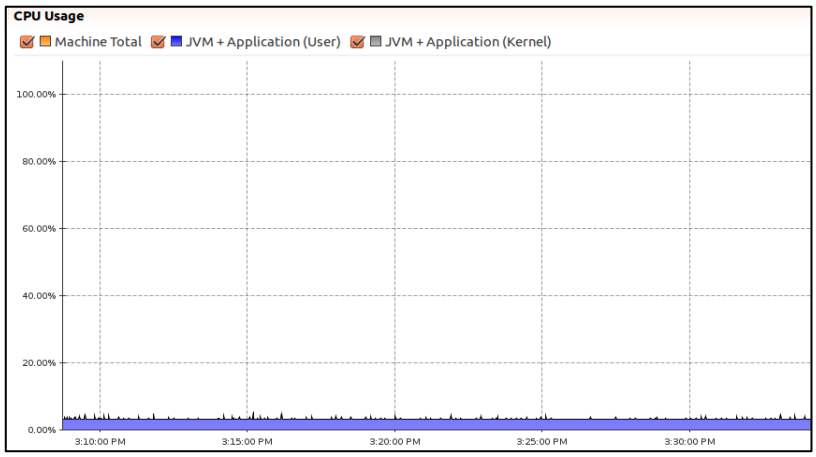

Figure 4.8: CPU usage in default WSO2 Siddhi engine when processing.

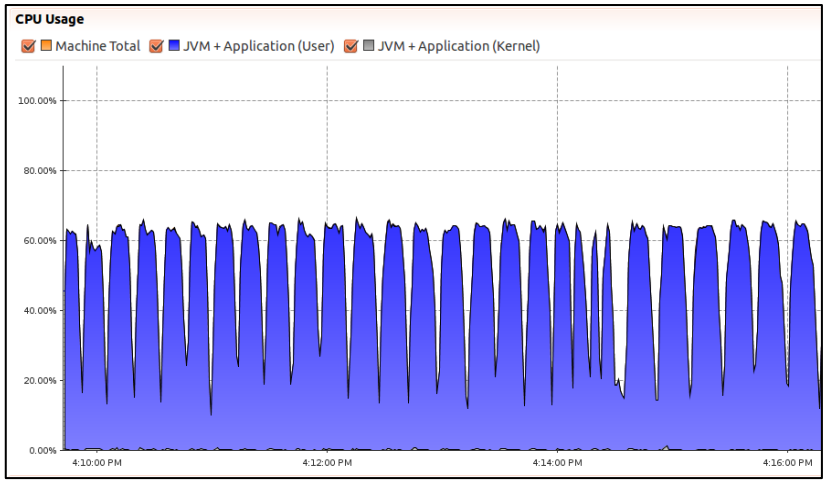

Figure 4.9: CPU usage in the proposed solution.

Figure 4.10 shows the thread count distribution when processing events. As you can see above, there are seven active threads used consistently. We have used a separate thread to push events, another thread to receive processed events from Siddhi and some threads are used internally in Siddhi for processing of pattern and other Siddhi queries. Figure 4.11 shows the usage of threads when processing is done with our proposed solution. Here you can see there are around 28 active threads and there is some sudden drops time to time. This pattern repeatedly continues for whole processing time. Because each Siddhi CEP engine runs in a separate thread, there will be multiple active threads at a time. Once processing is completed corresponding Siddhi engine instance is stopped and thread gets terminated as well. That is why there are some sudden drops in the total number of active threads in some situations.

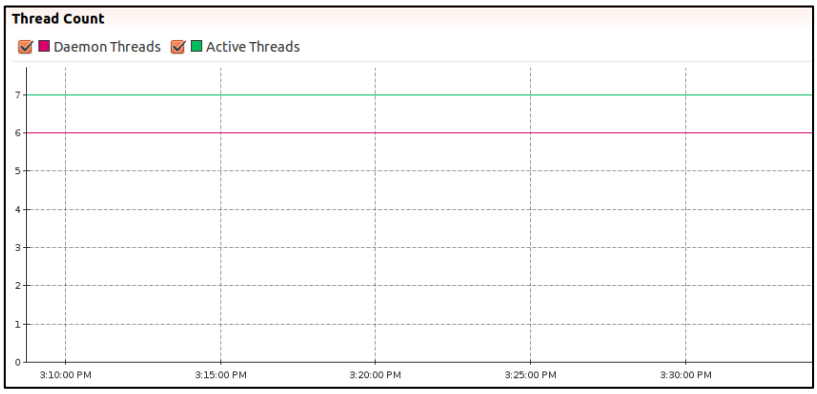

Figure 4.10: Thread count in default WSO2 Siddhi engine when processing.

The machine that we used for testing contains 64 GB of total memory. But we have only allocated 16 GB as the minimum memory (committed memory) and 18 GB as the maximum memory (reserved memory) to provide a controlled environment for testing scenarios of default Siddhi engine and proposed solution. As you can see in Figure 4.12, memory usage is consistent and it is around 6 GB across all the time when processing was done. Figure 4.13 shows the memory usage when processing events with our proposed solution. Here, you can see memory usage is gradually increased and gone up to 16 GB which is the committed memory size; then its usage

is suddenly dropped and incremented again. This behavior is observed because after the committed memory is consumed by the events that are partitioned, garbage collector runs and get cleared the memory. We can see the same pattern of the memory usage continued in at all the time of the event processing.

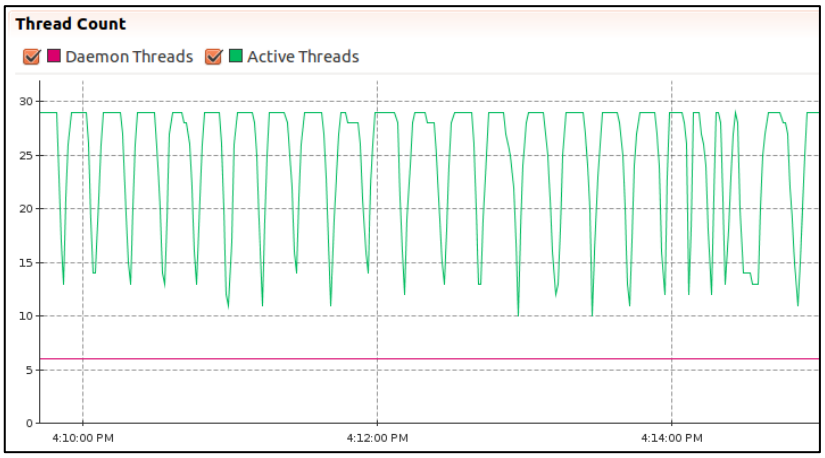

Figure 4.11: Thread count in the proposed solution.

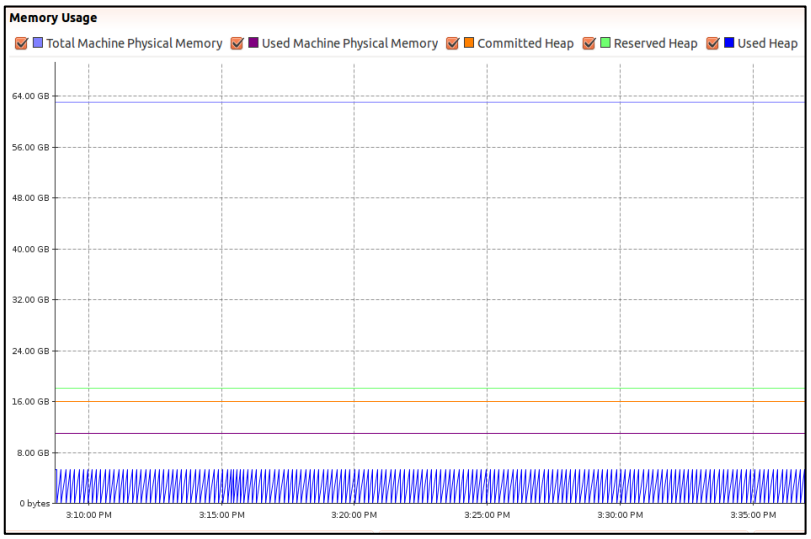

Figure 4.12: Memory usage in default WSO2 Siddhi engine when processing.

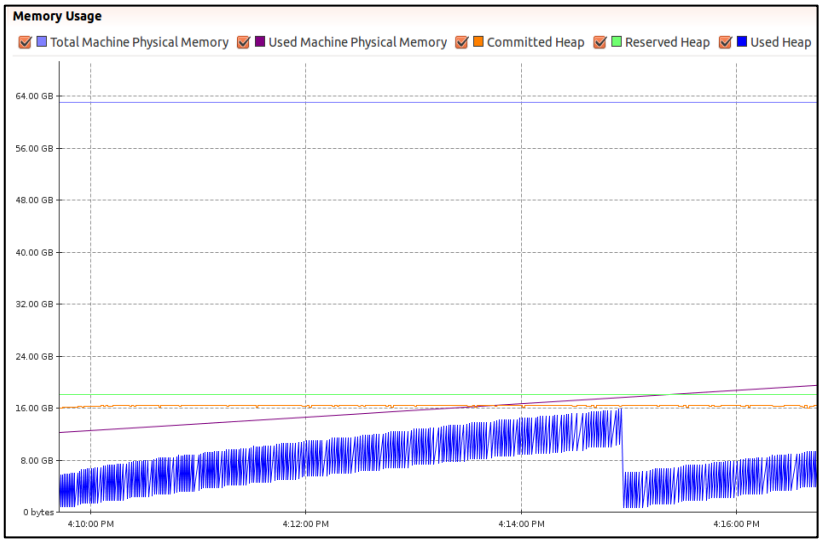

Figure 4.13: Memory usage in the proposed solution.

4.5 Evaluation of Accuracy

Several rounds of testing were carried out to verify and analyze the accuracy of our proposed solution. We can analyze the accuracy at two different levels. First, we have used the SMB2013 benchmark without Query 3.2, which does event reordering, and event duplication in a default WSO2 Siddhi CEP engine and got the necessary output of pattern-detected events. This output is considered as the template to verify the accuracy of our solution. Then accuracy can be defined into two main levels.

1. Whether our proposed solution detects all possible patterns
2. When level (1) is fulfilled, whether duplicate events are get removed and reordered.

Due to the nature of tests, our proposed solution detected all possible patterns which are identified the default Siddhi CEP engine, and this is expected and proved through

our tests that conducted. This means our solution is achieved 100% accuracy for level-1.

Figures 4.14 and 4.15 illustrate accuracy level-2. These tests were conducted to verify the effectiveness of both reordering and duplicated event handling approaches that we discussed in Section 3.1.2. These tests will provide the necessary expectation of reordering and duplicate handling capabilities of our proposed solution.

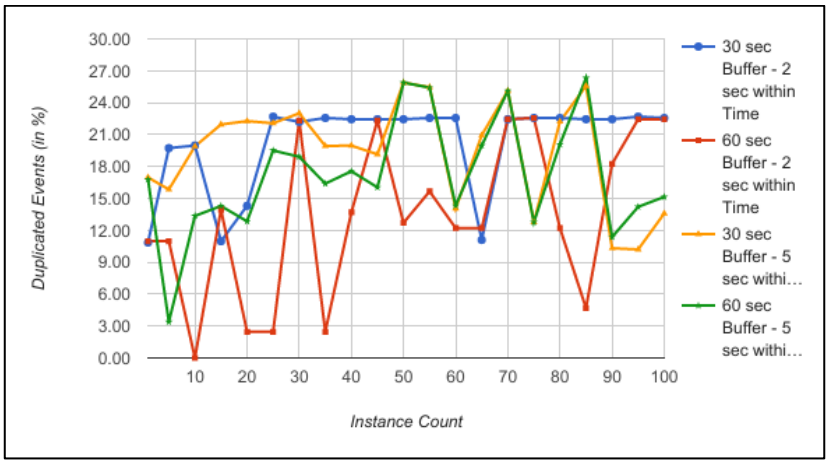

Figure 4.14: Duplicated events (in %) vs. Siddhi instance count.

We have conducted the accuracy tests with few different combinations. The tests were conducted with a various number of Siddhi instance count and buffer time. As we mentioned in the previous section events are get reordered using K-slack based implementation and Hash-based data structure used to avoid duplicated events. After events are get processed and output by multiple parallel Siddhi engines, pattern detected events are sent to another Siddhi engine which runs our K-slack based implementation to reorder the events and get removed the duplicated events at the same time. Here, events are buffered for some amount of time as batches to perform reordering and to remove duplicated events. Duplicated events were calculated by comparing with the pattern detected events of the default WSO2 Siddhi CEP engine.

As you can see in Figure 4.14, tests were conducted for two different buffer times which are 30 seconds and 60 seconds and also for two different *within* times which are two seconds and five seconds. Here, duplicated events are in between the range of 10% - 25% of the actual pattern detected events. And this percentage is changed as per the Siddhi instances count, different buffer time intervals and *within* time. We have conducted the accuracy tests with Siddhi instances from 1 to 100. When tests were conducted with 30 seconds buffer and two seconds *within* time, it was observed 10.85% duplicated events as the minimum when Siddhi CEP instance count was one, and it was observed 22.69% of duplicated events as the maximum when Siddhi instance count was 25. With 60 seconds buffer time and two seconds *within* time, no any duplicated events are found while running with 10 Siddhi engine instances. As the maximum, 22.56% of duplicated events found when there were 75 Siddhi engine instances when buffer time was 60 seconds and *within* time was two seconds. When tests were conducted with 30 seconds buffer and five seconds *within* time, it was observed 10.20% duplicated events as the minimum when Siddhi CEP instance count was 95, and it was observed 25.89% of duplicated events as the maximum when Siddhi instance count was 50. With 60 seconds buffer time and five seconds *within* time, 3.39% duplicated events are found while running with five Siddhi engine instances. As the maximum, 26.31% of duplicated events found when there were 85 Siddhi engine instances when buffer time was 60 seconds and *within* time was five seconds.

As expected, duplicated event count is low when running with 60 seconds buffer. If buffer time is high, then more events get buffered in the memory; this increases the possibility of duplicated events gets removed as described in Figure 3.13.

Figure 4.15 provides statistics of the tests that we conducted to analyze the disordered events. From above graph, you can see tests are conducted with 30 seconds buffer and 60 seconds buffer with two different *within* times which are two seconds and five seconds. In scenarios where *within* time was two seconds, disordered event percentage was less than 7% compared to the patterned detected events in a default Siddhi CEP engine. When buffer size was 30 seconds and *within*

time was two seconds, it was observed 1.73% of disordered events as the minimum with 35 parallel Siddhi CEP engines. And 6.91% of disordered events are observed as the maximum with 95 parallel Siddhi CEP engines. When buffer size was 30 seconds and *within* time was five seconds, it was observed 0.35% of disordered events as the minimum with one parallel Siddhi CEP engine. And 22.07% of disordered events are observed as the maximum with 100 parallel Siddhi CEP engines.

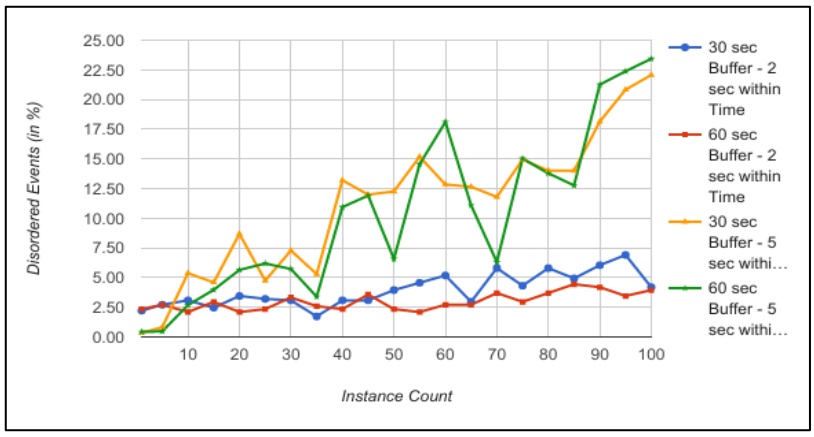

Figure 4.15: Disordered events (in %) vs. Siddhi instance count.

With 60 seconds buffer time and two seconds *within* time, 2.10% of disordered events are found while running with 10 Siddhi engine instances. As the maximum, 6.91% of disordered events found when there were 95 Siddhi engine instances when buffer time was 60 seconds and *within* time was two seconds. It was observed 0.43% of disordered events while running with one Siddhi engine instance when buffer time was 60 seconds and *within* time was five seconds. As the maximum, it was observed 23.43% of disordered events as well.

As expected disordered events are less when buffer time was 60 seconds in our tests as per Figure 4.16. We can understand that buffer time has a direct impact on the total number of duplicated and disordered events based on above analysis.

4.6 Latency Evaluation

Our proposed solution has an impact on overall latency on pattern detection as expected, but this totally depends on the buffer time that we used for reordering and duplication handling. It is a known fact that, if there is a queue or buffer used then it will impact the latency undoubtedly. However, calculating latency for pattern related queries is complex as output event count (pattern detected event count) is not matched with the input event count. Then, it is not possible to do the direct calculation of the per event latency. However, what we can do is calculating the delay in time on finding the pattern in own proposed solution compare to the default Siddhi CEP engine.

First, we test with default WSO2 Siddhi CEP engine to identify the event latency. Here, we used the same Siddhi query that used for other tests. As per the query, we need to have three events to find out the pattern that is mentioned by the Siddhi query. The pattern is detected only after matching attributes in three different events. Then, to calculate the latency of the pattern detected output event we have used the timestamp of the first event that initiated the pattern. Based on that we have calculated the per event latency for all the output of the pattern query. There were 811 patterns detected for our Siddhi query and latency for those events are given in Figure 4.16. As per Figure 4.16, maximum per event latency is 2.0305 milliseconds and the minimum latency is 0.0027 milliseconds. Event latency is less than 0.25 milliseconds for more than 60% of the pattern detections.

Figure 4.17 shows that there is a linear increment in latency when the number of Siddhi instances increase in our solution. This is expected due to our solution's design, in our proposed solution when events are received; they are get partitioned based on within time interval at the entry phase. Here, event partitioning happens by considering the timestamp of the event and not the system timestamp as shown in Figure 3.11.

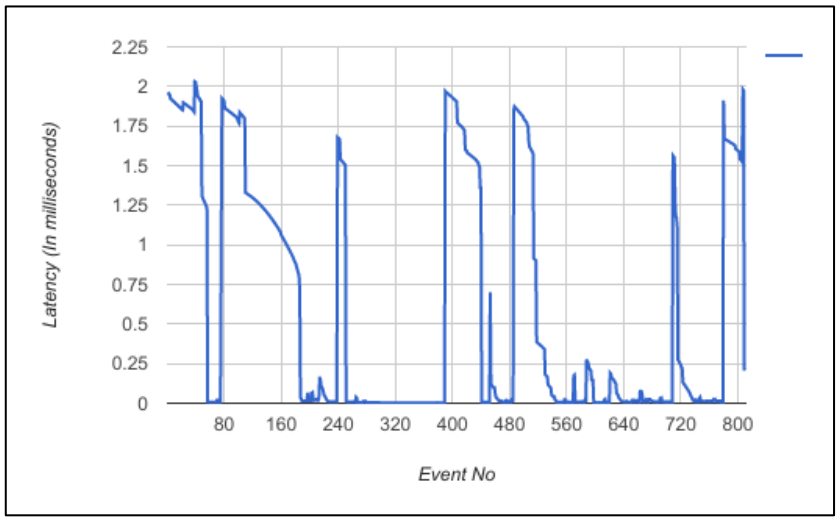

Figure 4.16: Latency in default WSO2 Siddhi CEP engine.

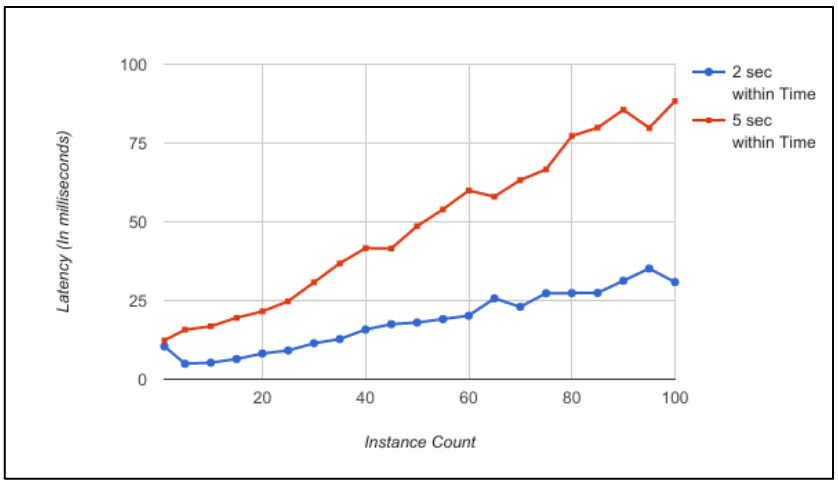

Figure 4.17: Latency in the proposed solution.

As per our tests, throughput is gradually increased when Siddhi instances are increased and latency is also increased as shown above in Figure 4.17. Due to the queue and partition approach of the solution, there is a possibility that partitioned

events are get queued for some amount of time until a Siddhi instance is available to process the partitioned events. If Siddhi instances are high, then no of event partitions are also will be high as per the implementation. Then this will also impact the latency of an event.

In above we have provided necessary statistics about the throughput improvements and how it influenced in overall accuracy. As we mentioned in above, we have used the Soccer Monitoring 2013 (SMB2013) benchmark for the evaluation purposes. As per that, we were able to achieve 13,638 events/sec throughput in a default Siddhi CEP engine. We have used the same SMB2013 benchmark to verify our proposed solution as well. Table 4.2 provides throughput related improvement details of the proposed solution for the SMB2013 benchmark.

As shown in Table 4.2, we could achieve 815% of improvement in overall throughput compared to the default Siddhi CEP engine when within time interval is defined as two seconds. If we have five seconds batch, then we could achieve 743% of throughput improvement and we could see 706% of throughput improvement with 10 seconds batch. However, it reduces to 506% as the throughput improvement when the batch time is defined as 15 seconds. This implies if batch time increases then it affects the throughput improvement of the proposed solution. This is mainly due to the memory that allocated for the processing. Since we have allocated a fixed size memory as the maximum (18 GB) and batch time increases, we have to store millions of events in the memory; this leads to frequent GC (garbage collection) and I/O operations and results huge impact on overall throughput of our proposed solution.

Table 4.2: Test results summary on throughput improvement.

Metric	Throughput (events/sec)	Percentage Improvement
32 core Machine (2 seconds Batch)	111,207	815%
32 core Machine (5 seconds Batch)	101,323	743%
32 core Machine (10 seconds Batch)	96,247	706%
32 core Machine (15 seconds Batch)	80,482	590%

As we discussed above, there are some side effects due to the proposed approach, which are disordering and duplication of events. To overcome this, we have suggested an approach which is based K-slack and hash-based implementation. This implementation has helped to reorder the events and remove duplicated events at the same time. Table 4.3 summarizes the proposed solution's overall accuracy based on batching time interval and the number of parallel Siddhi CEP instances as well.

Table 4.3: Test results summary on accuracy.

Metric	Value
Duplicated Events - 30 seconds Buffer & two seconds within Time	20.22%
Duplicated Events - 30 seconds Buffer & five seconds within Time	19.39%
Duplicated Events - 60 seconds Buffer & two seconds within Time	13.31%
Duplicated Events - 60 seconds Buffer & five seconds within Time	17.13%
Disordered Events - 30 seconds Buffer & two seconds within Time	3.94%
Disordered Events - 60 seconds Buffer & two seconds within Time	2.98%
Disordered Events - 30 seconds Buffer & five seconds within Time	11.00%
Disordered Events - 60 seconds Buffer & five seconds within Time	10.32%

As per Table 4.3, if the buffer is defined as 30 seconds and the *within* time in the query was defined as two seconds then there were 20.22% duplicated pattern detected events while there were 19.39% of duplicated events when the *within* time was five seconds. However, when the buffer time was increased, the total number of duplicated events reduced as well; according to our tests when the buffer size is

defined as 60 seconds, duplicated event percentage reduced to 13.31% when the *within* time was two seconds and 17.13% when the *within* time was five seconds. When the *within* time was two seconds 3.94% disordered events when we had 30 seconds buffer, whereas this was reduced to 2.98% when the buffer is increased to 60 seconds. The similar behavior was observed when *within* time was five seconds where disordered events reduced when buffer time was increased.

We have done performance tests with different within time intervals to prove the capability of our proposed solution. As well as our accuracy tests gave some understanding on possible scenarios that we can use our solution. Now, we can get a clear understanding of the capabilities and limitations of our proposed solution from above provided test results. If there is a pattern detection scenario where duplicates are allowed (or not an issue), then our proposed solution would be ideal. Also our implementation is fit for use cases where false positives are acceptable. However, through our reordering and duplication handling approaches, we could achieve a considerable amount of accuracy in overall based on our evaluation.

4.7 Summary

Chapter 4 discussed the details related to the evaluation and the results that obtained through tests. We have used DEBS 2013 dataset as the benchmark for our evaluation process. The dataset that we used as the benchmark contains 47 Million events related to a football game. As per the test that we have done, we were able to achieve more than 800% of throughput improvement compared to the default Siddhi CEP engine. Throughput improvement is also varied based on 'within' time defined in the query, resources (memory and number of cores) allocated for the processing and number of parallel Siddhi CEP engines. We have also done tests to measure the accuracy and latency of our solution as well. Accuracy is determined based on whether our proposed solution detects all possible patterns and when all are detected, whether duplicate events are removed and reordered. In option 1, our proposed solution showed 100 % of accuracy compared with default Siddhi CEP engine. However, with option 2, there is some tradeoff with accuracy. We have observed

some latency increment on our proposed solution. This is due to the queueing and partitioning nature of our proposed solution.

5. SUMMARY

5.1 Conclusion

Scaling the Complex Event Processing is an essential requirement in the data analytics space. Scaling enables handling a large number of CEP queries, running queries that need large working memory, handling a large number of events, complex queries that might not fit within a single machine, and handling a large number of events. In this thesis, we have discussed the most common and available scaling approaches that can be used for CEP scaling purposes. They are, scaling through attribute-based partitioning, Publisher-Subscriber approach, Storm-based deployment, distributed cache, and scaling by integrating with Enterprise Service Bus. We have evaluated each of these scaling approaches and analyzed their pros and cons. Based on the analysis, we identified several bottlenecks while scaling Pattern and Sequence queries. Scaling pattern and sequence queries are bit tricky as in-memory states need to be maintained between the nodes in the cluster.

We proposed time-based partition to scale pattern and sequence CEP queries. In the proposed solution, incoming events are partitioned based on the within time that is specified in a CEP query. Those partitioned events are sent to parallelly running Siddhi CEP engines for pattern and sequence matching as defined in the query. Here, partitions are created by overlapping with each other, which could lead to duplicated output events as same patterns can be detected in two different Siddhi CEP engines. Due to the parallel processing by multiple CEP engines, it leads to the event disorder as well. Then, pattern detected output events are pushed to another Siddhi CEP engine to discard event disorder and event duplication issues.

We conducted various tests to verify the performance of the proposed solution with compared to the default Siddhi CEP engine. The tests were carried out with different combinations of time batches of partitioned events and number of Siddhi CEP instances by allocating resources in various levels. Based on this analysis, we were able to achieve more than 800% of improvement in throughput. Through our tests, it was observed that throughput has increased when the batch time is reduced and the

number of CPU core is increased. As expected, we also observed an increase in per event latency with our proposed solution due to the queueing and partitioning nature of the proposed solution. We further conducted tests to verify the accuracy. The proposed solution was able to detect all possible patterns that are detected by a default Siddhi CEP engine which means we were able to achieve 100% accuracy. However, 13% - 20% of events got duplicated and 3% - 11% of events get disordered compared to patterns detected by the default Siddhi CEP engine. Duplicated and disordered events depend on batch time interval and number of parallel Siddhi CEP engines.

5.2 Limitations

In the proposed solution, incoming events are buffered in a queue and then partitioned before sending to multiple Siddhi CEP engines for event processing. Due to this buffering, the proposed solution needs more memory. Here, buffering and partitioning happen based on the time interval defined in the query. Then it is not ideal to use our proposed solution for patterns and sequences, which require correlating events within a large time window, e.g., minutes to hours. In the proposed solution, multiple Siddhi CEP engines are spawned in the same machine to process the partition, but this requires a large number of CPU cores. This issue can be overcome by implementing the same approach across multiple machines rather using a single machine for processing.

The proposed solution increases the per-event latency compared to the pattern detection in default Siddhi CEP engine. Moreover, this latency depends on the within window time. Due to this, our solution might not be the most suitable to detect pattern instantly. However, in most cases the latency is increased by only a few millisecond ranges; hence, the proposed solution is still applicable in many applications that need near real-time event detection

Due to the design and implementation of the proposed solution, pattern detected events might get duplicated or/and reordered. Then if there are any use case where

'exactly one' quality of service is expected then our solution is not the ideal answer. However, in most cases, patterns and sequences are used to find out some abnormal behaviors and react to them then in such cases duplicated or disordered pattern detection does not have a considerable impact on the overall solution.

5.3 Future Work

Above proposed solution is implemented in such a way, which can be deployed in a single node where communication between the Siddhi instances happens through Java method communication. However, this solution can be improved where it can be run in a distributed environment. Rather running Siddhi CEP instances on the same machine, it could be deployed on multiple machines as well. As seen in Figure 5.1 events can be partitioned on a node and those partitioned events can be then pushed to CEP engine instances running on multiple machines. Due to this behavior, we do not require a single machine, which has a high number of CPU cores and memory that capable of running multiple Siddhi CEP engines.

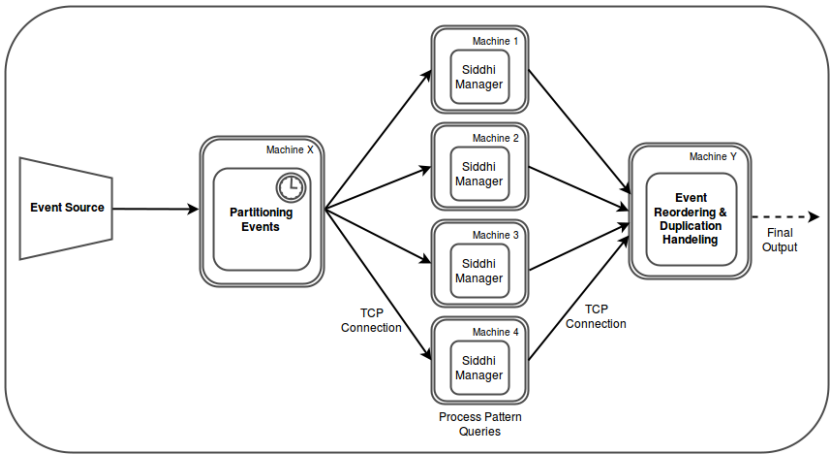

Figure 5.1: Distributed deployment of the proposed solution.

In our implementation, we used buffer-based disorder handling (also known as K-slack) approach because its implementation is straightforward compared to the other

scaling approaches that discussed in detail under Section 2.6. Even though K-Slack based approach increases the latency due to buffering and sorting delays when processing input events it has an important advantage where it dynamically adjusts the buffer size to a big enough value to accommodate all late arrivals, aiming to provide near exact query results. However, we can focus on other approaches like punctuation-based, speculation-based or approximation-based disorder handling approaches as well to reorder events in a more efficient manner for some specialized use cases. For example, speculation based approach is suitable for cases which have less out of ordered events and required output instantly at least in a partial manner. We could achieve 'exactly one' quality of service by investing more effort on this.

As mentioned in Section 3 in our implementation, number of Siddhi instances is a user-defined configuration parameter. However, this could be improved in such a way that our solution decides the number of Siddhi instances in a self-calculated manner and tune itself automatically. The optimal parameter value needs to be calculated based on the hardware resource consumption and other factors like throughput and latency.

Our proposed solution buffer incoming events in-memory and partition them based on within time defined in the query. Due to this, it is impossible to use our approach for queries which has longer within time because if event rate is high and within time is also high then we'll end up storing millions of events in the memory, and this leads to an issue of high memory usage and eventually goes to out of memory situation. Then to cater the requirement of having longer within time, we may need to increase the memory allocated to the physical machine. However, this will not be a scalable solution in the long, run and we need to look for options on scaling pattern and sequence queries which have longer within time interval. Above limitation can be solved up to some extent if we go for a distributed deployment as shown in Figure 5.1. But there are many factors that we need to consider here, for example limiting the number of partitions get created in the initial step and increasing the number of machines that run Siddhi instances to process partitioned events. By following this approach, we can reduce the time to keep the partitioned events in the initial

processing machine. But, delays introduced by the network to send events between the machines and time taken for data transformation are also influential in overall performance but yet another challenging area of research on scaling the pattern and sequence queries.

REFERENCES

Commented [1]: Double check all references. They must be in IEEE style. I did a few as examples

[1] Oded, *Complex Event Processing (CEP): A key for real-time, context aware, analytics* [online]. Available: https://community.hpe.com/t5/Telecom-IQ/Complex-Event-Processing-CEP-A-key-for-real-time-context-aware/ba-p/6793862#.WMf87iYvDCJ. [Feb. 24, 2017].

[2] D. C. Luckham, *Event Processing for Business: Organizing the Real-Time Enterprise*, Wiley., Hoboken, New Jersey, 2012, pp. 16-21.

[3] T. Bass, *What is Complex Event Processing?* [online]. Available: http://www.thecepblog.com/2007/05/14/what-is-complex-event-processing-part-1/. [Dec. 23, 2014].

[4] D. C. Luckham and B. Frasca, "Complex event processing in distributed systems," Technical Report, Program Analysis and Verification Group Computer Systems Lab, Stanford University, 1998.

[5] C Li, "Performance Management of Event Processing Systems," Ph.D. Dissertation, School of Engineering and Applied Science., Aston University, Birmingham, 2013.

[6] S. Perera, *Handling Large Scale CEP Usecase with WSO2 CEP* [online], Available:http://srinathsview.blogspot.com/2014/07/handling-large-scale-cep-usecase-with.html. [Dec. 23, 2014].

[7] A. Aalto, "Scalability of Complex Event Processing as a part of a distributed Enterprise Service Bus," Ph.D. dissertation, Dept. Science., Aalto University, Espoo, 2012.

[8] Magmasystems Blog, *CEP Engines and Object Caches* [online]. Available: http://magmasystems.blogspot.com/2008/02/cep-engines-and-object-caches.html. [Dec. 23, 2014].

[9] T. Dudziak, *Storm and Esper* [online]. Available: https://tomdzk.wordpress.com/2011/09/28/storm-esper/. [Feb. 24, 2017].

[10] Oracle, *Fusion Middleware Developer's Guide for Oracle Complex Event Processing* [online], Available: https://docs.oracle.com/cd/E23943_01/dev.1111/e14301/scalunder.htm#CEPED1975. [Mar. 12, 2017]

[11] WSO2 Inc, *WSO2 Siddhi CEP Query Language Model* [online]. Available: https://docs.wso2.com/display/CEP300/Language+Model. [Dec. 25th, 2016]

[12] P. Vincent, *CEP Tooling Market Survey 2014* [online]. Available: http://www.complexevents.com/2014/12/03/cep-tooling-market-survey-2014/. [Dec. 23, 2014].

[13] M. Gualtieri and J.R. Rymer, "Complex Event Processing (CEP) Platforms," The Forrester Wave, Cambridge, MA 02139 USA, 2009.

[14] EsperTech, *Event Processing with Esper and NEsper* [online]. Available: http://esper.codehaus.org/. [Jan. 09, 2015].

[15] Overview of Oracle CEP, *Oracle CEP Getting Started* [online]. Available: http://docs.oracle.com/cd/E16764_01/doc.1111/e14476/overview.htm. [Jan. 30th, 2015]

[16] M. Palmer, *Progress Apama* [online]. Available: http://library.corporate-ir.net/library/86/869/86919/items/236187/section4.pdf?q=apama. [Jan. 30th, 2015]

[17] WSO2 Inc, *WSO2 Complex Event Processor* [online]. Available: http://wso2.com/products/complex-event-processor/. [Jan. 30th 2015]

[18] S. Suhothayan, *Understanding How Siddhi Powers WSO2 Complex Event Processor 2.x* [online]. Available: http://wso2.com/library/articles/2013/06 /understanding- siddhi-powers-wso2-cep-2x/. [Jan. 30th, 2015].

[19] O. Etzion and P. Niblett, *Event Processing in Action*, 1st ed., Manning Publications Co., Greenwich, CT, USA, 2011, pp. 265-266.

[20] M. R. N. Mendes, P. Bizarro, and P. Marques, "A framework for performance evaluation of complex event processing systems," In Proc. 2nd Intl. Conf. on Distributed event-based systems (DEBS '08), New York, NY, USA, Jul 2008, pp. 313–316.

[21] T. Heinze, "Elastic complex event processing," In Proc. 8th Middleware Doctoral Symposium (MDS '11), New York, NY, USA, Nov 2007, pp. 4:1–4:6.

[22] M.M. Michael, J.E. Moreira, D. Shiloach, and R.W. Wisniewski, "Scale-up x scale-out: A case study using Nutch/Lucene," In Proc. 21st International Parallel and Distributed Processing Symposium, IPDPS '07, Long Beach, California, USA, Mar 2007, pp. 1–8.

[23] R. Mayer, B. Koldehofe, and K. Rothermel, "Meeting Predictable Buffer Limits in the Parallel Execution of Event Processing Operators," In Proc. IEEE BigData '04, Washington, USA, Oct 2014, pp. 402–411.

[24] Y. Wang and K. Cao, *A Proactive Complex Event Processing Method for Large-Scale Transportation Internet of Things* [online]. Available: http://www.hindawi.com/journals/ijdsn/2014/159052/. [Jan. 05, 2015].

[25] S. Perera, *How to scale Complex Event Processing (CEP) Systems?* [online]. Available: http://srinathsview.blogspot.com/2012/05/how-to-scale-complex-event-processing.html. [Dec. 23, 2014].

[26] Sybase CEP, *Sybase* [online]. Available: http://www.sql-ag.de/uploads/media/Sybase_CEP_Overview_ds.pdf. [Jan. 05, 2015].

[27] V. Govindasamy and P. Thambidura, "An Efficient and Generic Filtering Approach for Uncertain Complex Event Processing," In Proc Intl. Conf. on Data Mining and Computer Engineering, Thailand, Bangkok, Dec 2012, pp. 211-216.

[28] Apache Software Foundation, *Apache Storm* [online], Available: http://storm.apache.org/documentation/Home.html. [Jan. 03, 2015].

[29] K.A. Nagy, "Distributing Complex Event Detection," M.S. thesis, Dept. Computing., Imperial College of Science, Technology and Medicine, London, 2012.

[30] T. Dudziak, *Storm & Esper* [online], Available: https://tomdzk.wordpress.com/2011/09/28/storm-esper/. [Jan. 06 2015].

[31] Zero MQ, *ZeroMQ, The Intelligent Transport Layer* [online]. Available: http://www.zeromq.org/. [Jan. 03, 2015].

[32] S. Ravindra, *WSO2 CEP 4.0.0 in Distributed Mode* [online]. Available: http://sajithr.blogspot.com/2015/09/wso2-cep-400-in-distributed-mode.html. [Feb. 23, 2017].

[33] WSO2 Inc, *Configuring WSO2 CEP to run with Apache Storm* [online]. Available: https://docs.wso2.com/display/CEP400/Configuring+WSO2+CEP+to+run+with+Apache+Storm. [Jan. 04, 2015].

[34] G. Sharon and O. Etzion, "Event-processing network model and implementation," IBM Syst. J., 2008, pp. 47(2):321–334.

[35] A. Thomas, *Enterprise Service Bus: A Definition* [online]. Available: https://www.gartner.com/doc/1405237/enterprise-service-bus-definition. [Jan. 12, 2015]

[36] J. L. Marechaux, "Combining service-oriented architecture and event-driven archi-tecture using an enterprise service bus," IBM Syst. J., 2006.

[37] C. Mutschler and M. Philippsen, "Distributed low-latency out-of-order event processing for high data rate sensor streams," In Proc. 27th Intl. Conf. Parallel and Distributed Processing Symposium, Boston, USA, May 2013, pp. 1133-1144.

[38] J. Li, D. Maier, K. Tufte, V. Papadimos and P. A. Tucker, "Semantics and Evaluation Techniques for Window Aggregates in Data Streams," In Proc. 5th SIGMOD Intl. Conf. Management of data, Baltimore, USA, Jun 2005, pp. 311-322.

[39] A. Brito, C. Fetzer, H. Sturzrehm and P. Felber, "Speculative Out-Of-Order Event Processing with Software Transaction Memory," In Proc. 2nd Intl. Conf. on Distributed event-based systems, DEBS '15, Rome, Italy, Jul 2008, pp. 265-275.

[40] S. Tirthapura, B. Xu and C. Busch, "Sketching asynchronous streams over a sliding window," In Proc. 25th annual ACM symposium on Principles of distributed computing, Denver, USA, Jul 2006, pp. 82-91.

[41] M. Li, M. Liu, L. Ding, E. A. Rundensteiner and M. Mani, "Event Stream Processing with Out-of-Order Data Arrival," In Proc. 27th Intl. Conf. on Distributed Computing Systems Workshops, Toronto, Canada, Jun 2007, pp. 67.

[42] Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich and C. Fetzer, "Quality-driven processing of sliding window aggregates over out-of-order data streams," In Proc. 9th ACM Intl. Conf. on Distributed Event-Based Systems, Oslo, Norway, Jul 2015, pp. 68-79.

[43] S. Perera, *CEP Performance: Processing 100k to Millions of events per second using WSO2 Complex Event Processing (CEP) Server* [online]. Available:

http://srinathsview.blogspot.com/2013/08/cep-performance-processing-100k-to.htm. [Dec. 23, 2014].

[44] EsperTech, *Performance-Related Information* [online]. Available: http://esper.codehaus.org/esper/performance/performance.html. [Jan. 09, 2015].

[45] Oracle, *Oracle Complex Event Processing High Availability* [online]. Available: http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/ocephawhitepaperbenchmark-193519.pdf. [Dec. 20, 2016].

[46] S. Mittal, *Internal implementation of Set/HashSet (How Set Ensures Uniqueness)* [online]. Available: http://javahungry.blogspot.com/2013/08/how-sets-are-implemented-internally-in.html. [Feb. 23, 2017].

[47] WSO2 Inc, *WSO2 CEP in Action - An Analysis of Use in Real-World Applications of Different Domains* [online]. Available: http://wso2.com/library/articles/2014/08/wso2-cep-in-action-an-analysis-of-use-in-real-world-applications-of-different-domains/. [Jan. 05th, 2017].

[48] DEBS Org, *DEBS 2013 Grand Challenge: Soccer monitoring* [online]. Available: http://debs.org/?p=41. [Jan. 05th, 2017].

[49] M. Hirt, *Life, Java and Everything* [online]. Available: http://hirt.se/blog/. [Jan. 05th, 2017].