TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Tarvo Arikas 192597IVCM

Streaming event correlation and complex event processing using open-source solutions

Master's thesis

Supervisor: Risto Vaarandi
Ph.D

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Tarvo Arikas 192597IVCM

Sündmuste voogude korrelatsioon ja komplekstöötlus vabavaralahendusi kasutades

Magistritöö

Juhendaja: Risto Vaarandi
Ph.D

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature, and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Tarvo Arikas

13.05.2021

Abstract

There is growing amount of data being produced by growing amount of information systems in the world every year. Securing those systems partially relies on analyzing the very same data they produce and as the amount of data increases, it is getting more complex.

This thesis analyzes a specific area of event processing called complex event processing, freely available open-source solutions encompassing these concepts, and the ease of use of these solutions. Complex event processing has evolved beyond academical research and proprietary systems to the hands of users that are not fluent neither in software engineering nor event processing semantics. The aim of this thesis is to determine the current status of open-source complex event processing solutions and their suitability for non-technical end-users who are required to use such solutions in professional situations.

Based on the analysis performed for this thesis, there is currently one active open-source project and one active free to use but closed-source project available that provide the means to utilize complex event processing without the competence of technical engineer with expert knowledge on complex event processing. In addition, there are multiple projects that are developing similar solutions.

This thesis is written in English and is 90 pages long, including 6 chapters, 17 figures and 1 table.

Annotatsioon
Sündmuste voogude korrelatsioon ja komplekstöötlus vabavaralahendusi kasutades

Aina kasvav kogus uusi infosüsteeme maailmas toodab tänapäeval ka aina suuremat hulka andmeid. Kõigi nende süsteemide turvalisuse tagamine sõltub osaliselt nende samade süsteemide poolt toodetud andmete analüüsimisest. Aina suurenev andmete kogus ning aina keerukamad küberründed muudavad taolise analüüsimise ajas keerulisemaks.

Käesolev lõputöö uurib sündmuste analüüsi ühte kitsamat valdkonda – komplekssündmuste analüüsi, selle põhimõtteid rakendavaid vabavara lahendusi ning nende lahenduste kasutajamugavust. Komplekssündmuste analüüs on aja jooksul välja kasvanud ainult akadeemiliste uurimuste ning valdkonnaekspertide käest ning levib aina rohkem tavakasutajatele mõeldud lahendustes. Lõputöö eesmärk on välja selgitada praegune sündmuste komplekstöötlust rakendavate vabavara lahenduste seis ning nende sobivus lõppkasutajatele, kes ei ole tarkvarainsenerid ega eksperdid sündmuste töötlemise alal, kuid vajavad mainitud lahendusi oma igapäevaste tööalaste toimingute teostamiseks.

Magistritöö raames läbi viidud analüüsi tulemusena saab välja tuua, et on olemas üks vabavaraline aktiivne ning üks tasuta kasutatav kuid suletud lähtekoodiga projekt mis pakuvad keerukaid komplekssündmuste analüüsi vahendeid ja võimalusi läbi lihtsustatud abstraktsioonide ja kasutajaliideste, mis on sobivad kasutamiseks organisatsiooni sees ka sündmuste analüüsi ja hajusandmetöötluse detailidega mitteteadlikele kasutajatele. Täiendavalt saab välja tuua, et eksisteerib projekte mis on sarnaseid kasutajatele sobivaid lahendusi välja töötamas ja arendamas.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 90 leheküljel, 6 peatükki, 17 joonist, 1 tabel.

List of abbreviations and terms

API	Application Programming Interface
BI	Business Intelligence
CEP	Complex Event Processing
DNS	Domain Name System
DSL	Domain-Specific Language
EPS	Events Per Second
ESP	Event Stream Processing
FTP	File Transfer Protocol
GUI	Graphical User Interface
HA	High-Availability
HIDS	Host-based Intrusion Detection System
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
IT	Information Technology
JDBC	Java Database Connectivity
JMS	Java Message Service
JVM	Java Virtual Machine
NIDS	Network-based Intrusion Detection System
NOC	Network Operations Centre
PaaS	Platform as a Service
PII	Personally Identifiable Information
PoC	Proof of Concept
REST	Representational State Transfer
RFID	Radio-Frequency Identification
SCP	Secure Copy Protocol
SIEM	Security Information and Event Management
SME	Small and medium-sized enterprises
SOAR	Security Orchestration, Automation and Response
SOC	Security Operations Centre
SQL	Structured Query Language

Table of contents

1 Introduction 11

2 Background 14

2.1 History 14

2.2 Complex event processing and event stream processing 16

2.3 Problems with open-source solutions 19

3 Validating the problem 21

3.1 Prerequisites for a streaming CEP solution 21

3.1.1 Streaming data 24

3.1.2 Complex Event Pattern detection 25

3.1.3 Real-time results 25

3.1.4 Competence 26

3.1.5 Licensing 28

3.2 Alternatives 28

4 Open-source solutions 32

4.1 Architecture of a streaming CEP solution 32

4.2 Requirements for a CEP solution 36

4.2.1 Dedicated input support 37

4.2.2 Event parsing 37

4.2.3 Streaming support 38

4.2.4 Event source time 38

4.2.5 DSL 39

4.2.6 Pattern matching 40

4.2.7 Event windows 41

4.2.8 Joining multiple streams 42

4.2.9 External non-streaming data-source support 43

4.2.10 Deployment options 44

4.2.11 High-availability and horizontal scaling 44

4.2.12 Graphical user interface 45

4.2.13 Community and documentation 45

4.2.14 Enterprise support ... 46
4.2.15 Minimum baseline feature-set ... 46
4.3 Analysis of open-source CEP solution ... 47
4.3.1 Apache APEX ... 48
4.3.2 Apache Beam ... 48
4.3.3 Apache Flink ... 49
4.3.4 Apache Heron and Apache Storm ... 51
4.3.5 Apache Samza ... 52
4.3.6 Apache Spark ... 52
4.3.7 Faust ... 53
4.3.8 Elastic EQL ... 54
4.3.9 Esper ... 54
4.3.10 ksqlDB ... 55
4.3.11 Siddhi ... 56
4.3.12 Simple Event Correlator ... 57
5 Summary of CEP solutions ... 58
5.1 Custom CEP implementation ... 59
5.2 Comparison of CEP solutions ... 63
5.3 Proof of concept deployment ... 64
5.4 Graphical user interfaces ... 65
5.5 Throughput benchmark ... 68
6 Summary ... 72
References ... 74
Appendix 1 – Performance benchmark test files ... 79
Appendix 2 – Benchmark measurements results ... 80
Appendix 3 – Flink raw intestion test code ... 82
Appendix 4 – Flink parsing test code ... 83
Appendix 5 – Flink JSON test code ... 84
Appendix 6 – Flink aggregation test code ... 85
Appendix 7 – Siddhi raw ingestion test code ... 86
Appendix 8 – Siddhi parsing test code (IO module) ... 87
Appendix 9 – Siddhi parsing test code (CEP engine) ... 88
Appendix 10 – Siddhi JSON test code ... 89
Appendix 11 – Siddhi aggregation test code ... 90

List of figures

Figure 1 - Four stages of event processing history 14
Figure 2 - The Event Processing Technical Society Reference Architecture 33
Figure 3 - Proposed CEP solution architecture 34
Figure 4 - Event source time test scenario events 38
Figure 5 - Example comparison of Python code and DSL 40
Figure 6 - Comparison of two Esper pattern detection syntaxes 41
Figure 7 - Sliding and tumbling (batch) windows. 42
Figure 8 - Custom CEP implementation sample configuration 60
Figure 9 - Custom CEP implementation Windows filesystem events test scenario Esper query 61
Figure 10 - Custom CEP implementation MailCleaner test scenario Esper query 62
Figure 11 - Siddhi demo application in web-based editor 65
Figure 12 - Siddhi demo application design view 66
Figure 13 - Ververica Platform web-based SQL editor 67
Figure 14 - Siddhi demo application Business Rules Manager form 68
Figure 15 - Raw log line ingestion and parsing EPS 70
Figure 16 - JSON ingestion EPS 70
Figure 17 - Aggregation EPS 71

List of tables

Table 1 - Comparison of CEP solutions 63

1 Introduction

As there is increasing amount of data being generated in the world by ever-growing number of information technology (IT) solutions every year, interpreting this data is getting more complicated. Expectations for converting the data-stream into meaningful real-time or near real-time information are also increasing. Solutions where information is periodically aggregated into massive databases and queries on it are run daily or weekly, are not fast enough for many use-cases. Solutions where necessary information is being extracted and analyzed directly from the source do not have the luxury of correlating data from different sources. Even if these approaches could be mixed and optimized, it requires a lot of work and they are not often considered as a best practice in today's IT world anymore. In 2018 a company named Intrusion Technologies published an article where they summarized that in the field of physical security "human detection, reaction and engagement is slower than the 2 minutes needed to significantly reduce the event consequences" [1]. In the field of IT security, incidents can escalate to cause a high impact in the scale of seconds.

This thesis focuses on how concepts like complex event processing (CEP) and event stream processing (ESP) can be of assistance in today's information security field. During the first decade of the 21st century, these concepts were evolving mostly in research papers in the form of academical works and in the work of few expert engineers and most commonly out of reach for the IT department and business analysts. As time has passed and technology has progressed from the first decade of the 21st century, these concepts have had time to evolve into complete IT solutions. Some of these projects have been created by or donated to open-source community and are used by persons with no technical knowledge of event processing systems. This means that more people can use the systems that have been previously out of reach due to their complexity to regular users.

Technical aspects of CEP have been researched by many researchers more than a decade by now. Martin Hirzel from IBM T.J. Watson Research Center for example has analyzed how to parallelize CEP by partitioning events [2]. Paper from Kent State University researched how to more effectively join multiple event streams [3]. There are multiple technical papers mentioned later in this thesis that conducted technical feature comparison and throughput benchmark analysis. The main contribution of this thesis is a comparison of existing open-source solutions utilizing CEP and ESP concepts to identify which systems are easiest to use for less knowledgeable users. Novelty of the thesis is that it focuses on the ease-of-use factor. This is an important factor, especially for open-source solutions, as open-source projects tend to be more used by non-expert enthusiasts who are seeking to solve a problem.

Another important contribution of this thesis is a set of questions that evaluate if tasks at hand will justify the use of CEP and ESP solutions. As it was already mentioned, research papers so far have focused on the technical details of the solutions or the implementation specifics. With the help of these prerequisite questions, this research will provide novel insight on how to evaluate if CEP and ESP concepts are viable solution concepts to the problem.

The second chapter of this thesis will provide an overview of the subject and will describe how this area has historically evolved and will define the terms CEP and ESP for rest of the research paper. This is necessary as there are still many arguments what are the exact definitions of CEP and ESP. As this thesis focuses on open-source products, overview of common problems for open-source implementation projects is also provided.

The third chapter will define key characteristics of a problem that can be addressed using a CEP solution, how to define the problem and what pitfalls to avoid. This chapter will also bring out the important fact that not all problems should or even can be solved using the described methods. It is important to understand whether these open-source tools are the right tools for solving the actual problem. Solutions to problems that cannot or should not be tried to solve using CEP tools, are also being analyzed and a brief overview of these solutions will be provided.

The fourth chapter will be introducing different components and frameworks more in detail. The overall reference architecture is described and proposed together with different types of components in this architecture. These different types of components will be vital for designing a working architecture for a fully functional and efficient solution. As no environment is ever identical and problems to solve differ from organization to organization, so do solutions that can and should be implemented. The important properties and features of an easy-to-use CEP solution will be defined and used as the basis of analysis for the proposed solutions.

In fifth chapter there is an experiment described which aim is to understand the underlying complexities in implementing a CEP solution based on a programming language libraries. There is also thorough summary of analysis from the fourth chapter with an additional in-depth analysis of two solutions conforming to requirements set in fourth chapter.

This thesis bases on comparative study methodology and analyzes different CEP solutions based on key features gathered from analysis of previous research papers. The purpose of this thesis is not to produce a single best recommendation to use one specific tool but to give an insight into which tools are available and what are their pros and cons, focusing on the usability factor. It is important to emphasize that these evaluations are based on the currently available knowledge and are expected to change over time as new solutions will be developed.

2 Background

2.1 History

The need of processing and detecting patterns in events and reacting upon them near real time pre-dates the use of information technology (IT). As computation developed, larger amounts of information could be processed using automated IT systems. This led to more formal definitions and to a better understanding of the subject. One of the most important figures in this area has been David Luckham who has been at the forefront of event processing since the 1980s. In 2002 he published a book called "The Power of Events" [4] that has been called a "Technical manifesto for CEP" in the Information Age magazine article titled "Man of Events" by Andrew Lawrence [5]. Luckham has been active in this field since the early days of working with Rapide project under DARPA and has also been collaborating with Stanford University. Until the writing of this thesis he still publishes articles in Event Processing Community forum web portal called "Real Time Intelligence & Complex Event Processing" as an emeritus professor of electrical engineering at Stanford University [6]. His biography in Stanford University page states: "He has published four books and over 100 technical papers; two ACM/IEEE Best Paper Awards, several papers are now in historical anthologies and book collections" [7]. In one of these books, "Event Processing for Business" he has summarized the four stages of Event Processing history [8] which have been depicted by Figure 1 [8].

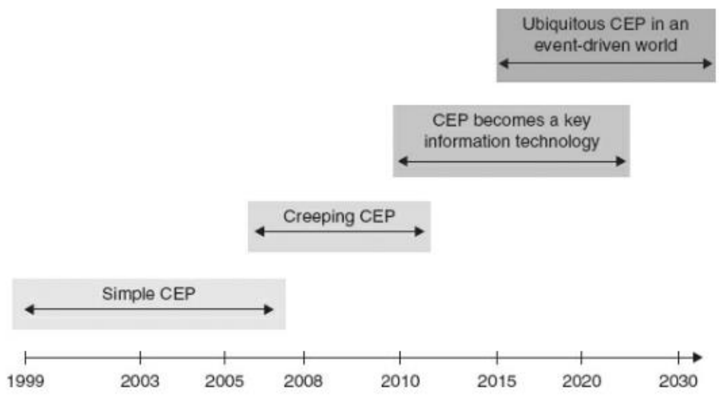


Figure 1 - Four stages of event processing history

By that definition we are currently in the third stage moving into fourth. As this book was published in 2011, there are predictions for more than a decade into the future. For example, Luckham predicts that in the third stage of CEP evolution there will be development of open-source event-processing tools with CEP capabilities. This prediction has already come into realization and has led to the writing of this thesis.

As CEP and its applications has evolved over time, so have academical papers covering this subject, but so far, they have focused on technical aspects. A paper by Gianpaolo Cugola and Alessandro Margara from the University of Milan analyzed and compared thoroughly different technical details of CEP solutions [9]. Another study from University of Stuttgart, Institute of Parallel and Distributed Systems, added features regarding parallel processing capabilities and scalability to comparison matrix [10]. A paper written in 2018 even went as far as collected, analyzed, and summarized previously written comparison papers related to CEP [11]. These papers have all one thing in common – they have all focused on very technical aspects of CEP solutions. As it was already stated in introduction, this thesis will focus on usability aspect of CEP solutions.

Over the course of years, the concepts of CEP have been utilized in many areas. In the early days, the most prominent utilizer was the stock market followed by various other financial institutions [8]. From there on, the methods of detecting complex patterns have spread into other areas. A research called VidCEP investigated spatiotemporal event patterns in video streams [12]. The researchers from WSO2 used their CEP product to analyze high-volume geospatial data for 2015 DEBS Grand Challenge [13] [14]. Even areas like healthcare and maritime security have been the research subject for applying CEP solutions into their work [15] [16]. The complex event pattern detection has even begun to enter our everyday lives without us explicitly recognizing it. Smart devices on our hands and on the walls of our homes constantly measure, analyze, and interact with us or our environment without us knowing the detailed logic behind this interaction. Luckham described this as the fourth stage in evolution of CEP. CEP is used as an integral part of many systems without users or even engineers developing these systems knowing or even realizing they are using CEP [8]. We have arrived at the stage of Ubiquitous CEP.

2.2 Complex event processing and event stream processing

The terms complex event processing (CEP) and event stream processing (ESP) have been constantly used to reference similar concepts or tools. Although these terms do define two different approaches for event processing, they are not exclusive. Moreover, they are quite often used together. To be clear and concise this thesis will define and describe these terms for the scope of this research trough the problems they aim to solve.

CEP focuses on clouds of events. To be more precise, the CEP synthesizes simple meaningful information from clouds of events based on complex patterns. These patterns try to find "causal, timing, and aggregation relationships between events" [17]**.** CEP solutions do not assume that events will arrive in correct order – in the order they have happened. This is one part that makes pattern detection in CEP complex – it introduces problem of varying event time skew. The observer must detect a pattern of two events that have happened in certain order without any certainty of order these events appear to the observer. This implies two things. First - observer must have memory, in another words, is stateful. And second - the answer to a question asked can change over time, as new events appear to the observer. Events themselves are immutable facts and stay unchanged, but their relevance in the context changes as time passes and new events are observed.

ESP focuses on stream of events. Events are processed as they arrive or in another words – are in motion. ESP solutions operate on so called unbounded or infinite datasets and do not explicitly try to overcome the problem of late arriving events. They just process events as they arrive as quickly as possible. There are no first or last events in the dataset they process – dataset is unbound, i.e. - infinite.

Like previously mentioned, those concepts are rarely used separately, at least using CEP and ESP terms. Simple event filtering or aggregation is usually just called filtering or aggregation, not complex event pattern detection. Same reasoning does apply to ESP. Simple real-time filtering or aggregation on streams is more likely to be called real-time analytics. When the terms CEP and ESP are used, it is highly likely that there is some form of complex event pattern detection that has been applied on unbounded streaming data. Throughout this paper these two terms are also referred together as just CEP if not mentioned otherwise.

By merging those two concepts together there will be a system that tries to detect timing and causal relationships in real time from events that might not arrive in the correct time and order. This task is not a trivial one and solutions for addressing this task tend to get quite complex. For this reason, most of these solutions have developed a dedicated declarative domain specific language (DSL) to express these pattern definitions in a simpler and more understandable manner. Throughout the rise of CEP, lot of pioneering work has been done by the engineers who have background in managing databases. Therefore, many solutions have built-in structured query language (SQL) like query language, and this holds true to this day. In fact, a recent paper about processing streaming datasets called "One SQL to Rule Them All" [18] states at the very beginning that SQL is the *de facto lingua franca* of real-time data analysis. This is no surprise since from the beginning of SQL, it has been heavily used to express questions about data, is well documented and has a rich history in the form of learning material. The easier it is to grasp the technical language to write a question in, the more person asking the question can focus on the question itself.

From another point of view, the patterns that reveal themselves by observing multiple events can only be detected in streaming datasets using an intermediate memory and therefore this detection process is not stateless. Many tools and solutions that provide simpler stateless rule engines have the capability to use some sort of internal memory directly or execute some external command that can provide functions of a memory. These tools do not provide complex pattern detection as a separate declarative rule syntax but using these stateless rules in conjunction with some sort of stateful memory, complex patterns spanning over time and events can be detected. It will be briefly explained with a help of the example below.

The goal of the following rules is to detect the misuse of physical access control system by detecting when an access control smart card is used to enter a secure area twice whilst not exiting in the meantime.

Events:

- Event E1 - user U1 entering a secure area.
- Event E2 - user U1 leaving a secure area.

Rules:

- Rule number R1 detects E1 (user accessing a secure area) and checks the value of fact F1 for user U1 (joins the query question with external static dataset). If U1 it set as being in a secure area, anomalous pattern is detected.
- Rule number R2 detects E2 (user leaving a secure area) and checks the value of fact F1 for user U1 (join the query question with external static dataset). If U1 it set as being out of a secure area, anomalous pattern is detected.
- Rule number R3 detects E1 (user accessing a secure area) and persists the fact F1 of user U1 being in a secure area.
- Rule number R4 detects E2 (user leaving a secure area) and persists the fact F1 of user U1 being out of a secure area.

In the given scenario, observable events can arrive to the CEP system within a period spanning over days. The rules described above use basic stateless filtering function in conjunction with external memory. This kind of pattern matching can be implemented in very large variety of solutions, even in simple scripting languages such as *bash*. Some solutions do provide more specific features that simplify the process of matching complex patterns. One good example is a simple but very effective tool called Simple Event Correlator, created by Risto Vaarandi [19]. OSSEC also uses such approach with some additional limitations [20]. These and many more solutions have one similar peculiarity – individual rule definitions provide somewhat limited pattern matching capabilities. Full potential of these solutions is revealed by joining these rules together by using intermediate memory as a context around them. A disadvantage for this approach is that one logical whole is split into many parts and can eventually become hard to grasp. Most of these systems also do not have any means to manage the problem of event time skew (events can arrive to the observing system out of order of occurrence). Dedicated pattern detection engines and description languages for them address this problem. The same rule applicable to English language would be something similar to: “Event E1, followed by E1, having no E2 in between.” And in a DSL it would be something similar to: “E1 ->!E2 -> E1”.

For this reason, this research focuses more on CEP solutions that provide a dedicated declarative DSL (preferably SQL) to express the relations between events.

2.3 Problems with open-source solutions

Open-source software is often confused with free software, but the lack of pre-set pricing is just a byproduct of open-source software development principles. In fact, not all open-source software is free and not all free software is open source. There are multiple vendors who publish their source code under some open-source license but do sell the same packaged software and offer support for a price. There are also vendors who distribute their software for free, but the source code is not available for everyone to freely view, edit, or download. This kind of lack of pre-set pricing is also often misinterpreted as software without any cost and with all the positive sides of enterprise software. But there are numerous implementation hurdles that can impact the final cost of the solution - lagging development, low quality community support, inaccurate or incomplete documentation, additional requirements to the infrastructure, administration overhead, and limitations due to licensing to name a few. Del Nagy, Areej M. Yassin and Anol Bhattacherjee have described five potential problems in their 2010 article "Organizational Adoption of Open Source Software: Barriers and Remedies" [21]. These adoption barriers as they called them were knowledge barriers, legacy integration, forking, sunk costs, and technological immaturity. In his 2004 article "Open Source to the Core" Jordan Hubbard added a discussion regarding licensing, community liveliness, and security to the table [22]. Considering the speed of technological advancement, by the IT standards these are already rather old articles but the point they make is still relevant today - open-source adoption does come with many hidden caveats. A 2018 research by Ehsan Noroozi and Habib Seifzadeh collected and organized 23 different risks from academic literature related to open-source software adoption [23]. Although there were significant overlaps in topics with previously mentioned papers, they described some additional potential difficulties. For example - the fear of losing work (because their job was related to using proprietary software) or interruption in the organizational processes during open-source adoption phase can become real problems that need attention and planning in advance.

As these potential pitfalls cut through the organization – from human resources to legal department, from developers to management, from infrastructure to processes, they are not to be taken lightly. This research will cover as many nuances and important properties as possible of the projects in this comparison. However, it is important to always validate such statements before every implementation and integration of the project because solely community supported projects are prone to change their direction or retire. For example, in 2020, there were 18 different open-source projects in Apache foundation alone that were retired, or their retirement process was initiated. One of them was open-source SIEM called Apache Metron [24]. It could have been a good candidate to evaluate in this paper but was marked as being moved to Apache Attic (Apache Software Foundation retirement home) during the writing of this paper. The aim of this paper will not be to produce never-changing facts about open-source products, but to point out important information that should be considered when open-source products are considered as an option.

3 Validating the problem

As mentioned in chapter 1, there is rapidly growing amount of data being generated and in order to meet the changing needs of analyzing this dataset, the systems have become more complex. The previous chapter highlighted the fact that different solutions have been developed to meet different needs of the users. Those differences might seem subtle at first, but they can have a profound effect on implementation complexity and overall results – including end-user experience. But even before any solution must be chosen or implemented there are numerous things to consider for the solution. Not all problems can be solved using CEP solutions. Even if a problem can be solved with the chosen tool, the solution can be suboptimal and cause problems after implementation. The initial understanding of the problem domain and clear definitions of specific problems that need to be solved is a necessity, but there are also technological requirements and pre-requisites that need to be considered. These requirements in turn depend on initial problem statement and possible solution candidates. All this is an interdependent circle of information that affects the effectiveness and outcome of an open-source solution adoption project.

This chapter focuses on how to state the initial problem and determine whether organizational and technical context supports the idea of using CEP solutions. The aim is to construct a very simple validation method that can be executed beforehand starting a CEP project.

For projects that do not meet those criteria, there will be an additional analysis provided covering typical information security use-cases and open-source solution proposals without the implementation of dedicated CEP engine.

3.1 Prerequisites for a streaming CEP solution

By forcing potential users to think about some of the key features of a CEP solution, there is a good chance that the initial problem statement can be redefined or totally discarded. In order to evaluate weather implementing a CEP solution makes sense in a given organization, this thesis proposes to use a survey with five questions that all have a major effect on the outcome. These topics may seem to be obvious for IT engineers, but CEP market is expanding rapidly and open-source solutions incorporating elements of CEP are becoming more popular. This leads to a situation where CEP systems are being experimentally used by enthusiasts who do not have experience on complex event patterns nor knowledge of what is an unbounded data stream. Systems are implemented because they have been found on the first page of google search results and are being used by big international corporations to do seemingly cool things and fulfill the users every requirement. Technologies used by Google to catch cyber criminals in their networks or Netflix to detect fraud might not be the best solutions to implement in given situation and organization. All such big corporations have almost unlimited resources at their disposal to implement all these solutions. Many of these systems have even grown out of these corporations and have been open sourced afterwards. This means that they have tailor-engineered the solution that suits best for their environment. An average SME does see the need to protect its IT systems and data all the same, but it does not have the luxury to develop highly complex event processing solutions from the scratch.

Roy Schulte has discussed in his article “When do you need an Event Stream Processing platform?” about when is it reasonable to use specialized software that supports stream processing [25]. He does mention that in some situations organizations re-stream their old event data for stream processing, but this is relevant for only certain use-cases such as during development of stream processing application. But in general, there is need for streaming event sources for stream event processing. Michael Stonebraker, Uğur Çetintemel and Stan Zdonik from Brown University postulated 8 requirements of real-time stream processing in their 2005 paper [26]. As their first rule, they stated, “Keep the Data Moving” and described that streaming data is more efficiently processed with stream processing solutions and batch processing is inefficient. These rules were requirements, not prerequisites and other seven rules were directly related to stream processing engine itself, such as using SQL to query data, handling time drifts, providing high availability, and horizontal scalability. But these two papers clearly show one common thing - there must be streaming data

to use event stream processing,. This is the first prerequisite for implementing a CEP solution, more specifically in the context of this thesis, a streaming CEP solution.

In section 2.2 it was defined that CEP is distinctive from other concepts as it tries to find causal, timing, and aggregation relationships between events. If there really is the need to find such circumstances and there are no other available solutions to achieve go goal – the use of a CEP engine is justified. But as it will be shown in section 3.2, there are many use-cases that might seem to be a good fit for a CEP engine but can be solved by using more simple means even more effectively. To deduce that there is a justified reason for implementing a CEP engine with all its complexities, all other possibilities should be ruled out before that. This will be the second prerequisite.

Actual real time actions are rarely needed in case there is a human interaction link processing these results. As it was described and referenced in the introduction, human detection, reaction, and engagement time, on average, is slower than 2 minutes. Even if event streams exists and the there is a need to detect complex event patterns, but the results end up as a daily report – there are most likely other means and solutions to solve the problems. The actual need to act immediately will be the third prerequisite.

The fourth prerequisite grew out of the work of this thesis. Although the aim of the thesis is to evaluate the ease of use of the CEP systems, they still are complicated solutions and are not a standard-issued office software packages. The architecture of these setups must be carefully planned, the events to process must come from somewhere and someone must know or decide what to do with those events. This will be the fourth prerequisite – existence of necessary competence.

Open-source software is provided to the community with variety of different licenses. Not all of them allow to use the software for every purpose unconditionally. This issue must also be addressed and has been defined as fifth prerequisite.

These five influential prerequisites will be listed in the form of short questions with a supplementary longer explanation and examples describing the matter. The aim of these questions is to agitate the reader to think through all the vital issues and reasoning behind initial ambition for implementing a CEP solution.

1. Do you have streaming data?

2. Do you need complex pattern detection?

3. Do you need real-time results?

4. Do you have the necessary competence?

5. Do your goals align with the licensing policy?

3.1.1 Streaming data

Stream-processing solution is overengineering if there is no streaming data or if the streams are fabricated and re-streamed from already pre-stored datasets. The foundation of streaming data is the ability to process every single event separately. Nowadays it is quite common to store raw data using Elasticsearch clusters [27]. Elasticsearch does provide its own event pattern definition language called EQL, but this feature can only process data by executing queries on data already stored on Elasticsearch. This is very similar to executing queries on relational database, implying that no real-time event-driven processing is being done. This applies to all solutions that exercise batch processing on previously collected data. Even if batches are one minute time-periods and historic data means older than one-minute old data this still means batch processing, not stream processing. If these shortcomings do not affect the outcome of the planned system, and there already exists pre-stored data, then there should be no need to aim for more complex event-driven streaming solutions.

Streaming data is often acquired by transporting logs from various systems to a central event processing solution using Syslog protocol. This is of course not the only way and there are number of other open-source solutions that provide their own wire level formats for transporting data. Some of them are mentioned in the next chapter. But one common feature of these solutions is that the transported information does not have to be persisted to a storage medium. These solutions are designed to provide channels through which data can be transported to event processing solution. There is no explicit need to persist the data if valuable information can be extracted from data stream and raw data itself is not needed afterwards. In addition to Syslog and other log transportation protocols there are event driven messaging systems such as Kafka that are gaining popularity [28]. These messaging solutions usually do require additional agents on source systems but are ideal middleware components for stream data processing as they provide additional delivery guarantees, buffering, and backpressure support.

Solutions that transport logs in batches of information are a good indication that a stream processing solution is not suitable in the given case. Protocols such as FTP or SCP are quite common in collecting logs from systems, but these solutions do not process events as they are created, nor are they meant to move information one event at a time. Another common pattern is requesting information from its source, one common example being HTTP APIs. When an event processing system must periodically request new information from the source system, it is left with the same downsides described earlier. Events are being processed in batches not one event at a time and there can be substantial delay introduced to the system.

3.1.2 Complex Event Pattern detection

People with IT-technical background often exaggerate their problems and tend to use more complicated new tools to solve simple problems. Not always is there a justified reasoning behind these decisions. Quite many, if not most, of the questions an information security analyst might want to ask from the data, can be answered with quite simple tools that provide elementary filtering and aggregation functions. For example, a solution that searches known malicious IP addresses or domain names from firewall or DNS server logs, is not a cause to implement a CEP solution. Questions that overstep the boundaries of one event to find causal and temporal relations between multiple events are rather different and require more advanced behavior.

Let us take an example of an electronical access control system for physical secure areas. This system works with RFID cards to identify users and logs every usage of these cards.

- To check if door is unlocked during non-working hours, simple filtering and pre-configured knowledge of working hours is enough.

- To check if one specific person has entered multiple secure areas simultaneously without leaving any of them, is a much more likely candidate for complex pattern matching.

3.1.3 Real-time results

Real-time results are the product of ESP solutions. ESP engines are by design meant to process information at the same moment the information becomes available. This does not always automatically mean that answers to the questions being asked from ESP engine are available at that very same moment. This depends on the complexity of processing algorithm. For example, if an ESP engine is tasked to run a complex computation on incoming data, then the outcome of this calculation will be available sometime after the initial source information has been observed. So real-time does not mean getting instant results but rather getting answers from fresh and latest data. In another words - the trigger to calculate expected outcome is the arrival of observable source information. There are often attempts to define the term “real-time” trough time boundaries such as microseconds, seconds, or minutes. David Luckham describes this as half-life of information and states that it depends on the business situation and cannot be defined universally with this precision [20].

Real-time results are generally important if these results are acted upon automatically in such a way that something is changed or altered in the system that affects the behavior of the system. In information security field these solutions are often called security orchestration, automation, and response (SOAR) tools. Acting upon malicious or suspicious information immediately can be of topmost importance and can be time critical for the organization. Blocking an IP address after detecting an attack or locking a user account after password spraying attack detection are only a few examples when seconds count and acting upon the information an hour or even minutes later can make the difference of preventing a successful attack or dealing with the consequences of successful attack. There are of course situations where immediate human interaction and real time results are both important. For example, delay of information on 24/7 SOC or NOC monitoring dashboard can have serious consequences.

A need for a daily report to management is not a viable cause for implementing a complicated real-time stream processing information system. Even an alert to e-mail or corporate chat should not be taken as clear indication of the necessity of real-time intelligence. E-mails and chat windows are often disregarded for minutes or even hours depending on the workload. The additional complexity of implementing a stream processing and real-time analytics solution can be more of a burden than it provides in return. Simpler tools that do batch processing of information can already exist in an organization and can achieve same goals with less investment.

3.1.4 Competence

As with other specific technologies and approaches, event processing solutions also have their specific competence needs. This classification of roles presented below is not the only way and these roles do not have to be fulfilled by separate individuals. But the responsibilities described for these roles do affect all event processing systems and therefore these responsibilities must be addressed by someone.

Development and operations (DevOps) engineers

There is great difference in what solutions to choose for solving problems. These differences are also reflected in the requirements for available skillsets. Even a cloud-based platform as service (PaaS) will require initial setup and maintenance. System itself might be up and operational but the data must be configured to come from somewhere and as a result of the event processing this system must perform an action. These are mostly activities for a skilled administrator. On premise solutions usually need even more technical expertise. For example, solutions that are provided as embeddable libraries instead of standalone applications require additional development. This in turn means, that implementation project must have access to development resources and knowledge. But before all this can be deployed and run in any environment, someone must think through and design the overall architecture. This design process cannot consider only business needs and requirements – underlying technical architecture must support the final solution in technical terms also.

Data engineer

Source information that needs processing might not be available. Depending on what the goals are there might be need of cooperation with other departments of the company, other branches of the company, or maybe even with another company. For example, if the goal is to correlate physical access control logs with vacations but human resources and building administration services have been outsourced, collaboration with other companies is needed in order to obtain the required information. This is no easy endeavor and might side-rail project unexpectedly. Let us take another example. Processing of personally identifiable information (PII) or credit card information might require compliance of strict policies or regulations. Local and global standards and regulations apply to many systems and can have profound negative impact on the business in case any violation is identified by controlling body. This is rarely a task for software engineer or any other technical role. In addition to regulations themselves, there is specific knowledge required about the data that is being intended to be processed within the system.

Analyst, end-user

Finally, there are the users of the system. People who know what they expect in return from this solution and should have good understanding about underlying data. The very same data that is going to be funneled into the system and from what they are going to be asking questions from. Often these people are called analysts.

There will be little to no benefit for a system that only supports programmatic interface if the users are non-technical salespersons who lack the knowledge on how to program and operate command line utilities. This in turn is directly related to specific solutions and their capabilities. Some of them provide just a programming language interface and require the user to write quite complex code. Others provide an abstraction layer in a form of some DSL (usually SQL like). Very few of them (at least in the open-source world) provide a comprehensive user interface for applying any kind of pattern declarations on the data.

These are real potential problems that need to be understood and analyzed before any solution is chosen to be used.

3.1.5 Licensing

Although the focus of this research is to investigate different aspects of open-source projects, licensing can still be limitation depending on the usage of the product. Some open-source licenses for example forbid to provide these products as a service. This applies for example in a situation where the goal is to create an intrusion detection system and provide it as a service to a third party.

3.2 Alternatives

CEP engines are good at solving specific kinds of tasks under a specific set of circumstances. But there are lots of different use-cases where more specialized open-source solutions are available, that need less resources and less specific environment for implementation. This section mentions some common problems that do not need a CEP engine and can be solved with already existing purpose-built solutions.

These solutions and recommendations are based on author’s previous knowledge and long-term experience in software development and information security field.

Time-series data, metrics, and trends

For monitoring long-term trends of numerical metrics, doing predictive analysis, or just drawing graphs, monitoring solutions are the right choice. That is of course the option if these metrics can be collected without complex pattern detection. Whenever there is option can *grep* or *select* a numeric value and need to see it as a chart or perform trend analysis, then there is a good chance that a monitoring or even business intelligence (BI) solution is the right choice. Today's monitoring solutions are more capable of performing good number-based analysis such as predictive trend analysis or anomaly detection. To name a few popular and thriving open-source monitoring solutions - Zabbix, Nagios, Prometheus and InfluxDB.

Searching, event enrichment, and list lookups

Whether there is a need to search for some specific event or compare the incoming events to some external lookup tables (whitelists or blacklists), there are free and open-source solutions available that are more easily adoptable and less resource demanding than a CEP solution. From simple installation and easy to use user interfaces to seamless upgrade and enormous community. These are main selling points of multiple solutions that have grown to be extremely popular and thus have stepped through rigorous testing of many experts. Capability of these solutions should not be underestimated as they can solve multiple problems with ease or help to understand the problem better. If simple historic search (not days or weeks old, but minutes or seconds old) is enough and there is no explicit need to exercise per event pro-active reaction, then usually Elastic stack is one of the most popular and easy to use products to use. Free version provides its own log collection solution, event parsing and enrichment solution, modern user interface, SQL like queries, and even an event query language that provides simple pattern matching.

In 2019, Amazon announced that it had launched a new project called Open Distro for Elasticsearch [29]. In addition to previous Elasticsearch features, they added authentication, authorization, more capable SQL query language, anomaly detection engine, and multiple other features to the already existing Elastic open-source solution. Like original Elastic stack, this also provides only historical searches or scheduled jobs that can be configured to run every minute.

But if this is not sufficient and true event-driven stream processing is still needed, then one option is Graylog [30]. Graylog is like Elastic stack and its successors but does provide some additional convenience features like configuring data ingestion runtime via user interface and operating in true event driven stream mode. This means that searches and lookups are being processed real-time on events as they are ingested (before they are persisted to indexes).

These three are not the only popular solutions that are provided to the community as open-source projects and go a long way in helping an information security or any other event processing team.

Known malware-, network intrusion detection and endpoint protection

Following tools are designed to protect different IT systems but one thing they all have in common is pre-defined set of rules to detect malicious behavior. These rules are created, reviewed, and used by the very community that is driving the development of these systems.

There is no need to try building a new network intrusion system detection (NIDS) as they already exist and some of them are open source, in example solutions like Snort or Suricata [31] [32]. The open-source solutions might not have all the latest extensions or hourly signature updates but out-of-the-box they go much further than anyone can quickly implement with a CEP solution for network intrusion detection.

Other solution to a big category of problems is host intrusion detection system (HIDS). They are like NIDS systems, but focusing on operating system specifics instead of monitoring network traffic. One of the most notable ones in the open-source world are OSSEC and Wazuh [20] [33].

Protection of web applications is another big category in information security domain that has been opened up to open-source community. Projects like ModSecurity have strong community and good community-driven rule base for detecting suspicious and malicious behavior [34]. There are numerous other projects for different use-cases such as Naxsi for Nginx and WebKnight for Microsoft IIS [35] [36].

Application debugging and tracing

For debugging applications and reacting to anomalies based on these detailed tracings, there are multiple open-source solutions listed in openAPM project website [37]. As there are different solutions and approaches for applications written in different programming languages, no specific product will be listed here explicitly as there are too many variables to consider.

4 Open-source solutions

First section of this chapter will focus on describing overall architecture in a working CEP solution. Different types of components and their roles in the overall architecture will be briefly discussed. Second section will propose a list of properties that are inherent to a CEP engine and that increase the ease of use for end-users. From those, a minimum baseline will be selected for highlighting the systems that meet the most important requirements. Third section will analyze a list of CEP systems based on the requirements set in the second section.

4.1 Architecture of a streaming CEP solution

An end-to-end streaming CEP solution in information security field can rely on multitude of different inputs starting from raw port mirror from central switch to physical access control system logs. None of those inputs are irrelevant. This diversity brings complexity to any solution that tries to correlate events from these sources and synthesize new and meaningful events. As the aim is to process different logs in one system, first challenge is to centralize the necessary information. This alone can be a time-consuming and expensive endeavor. Then there is the question of understanding the gathered information. To this day vital information is produced as unstructured text. In order to make it machine readable, it must be parsed. During or after this some additional filtering might be needed to discard raw data with no value. As the focus of this research is on CEP engines and their usability, only a brief overview of event collection and parsing important features and existing solutions will be provided.

Only after collecting, parsing, and optionally filtering input data, it is time to analyze them. This is where CEP solutions come in. As defined in the prerequisites of implementing a CEP solution, results must be real-time (or at least near real-time). Because of that there must be a real need to do something with these new synthesized events at the time they are created. It would be meaningless to implement a complex real-time system and then email the report of the findings to security analyst in the morning.

There was an expert group who founded The Event Processing Technical Society in 2008. Although the society as a formal group does not exist anymore, they did publish multiple reference architectures for CEP systems. One such example was published in the summary of 5th International Symposium, RuleML, in 2011 [38]. This reference architecture as they called it, is an abstract view of a CEP system and its life cycle (Figure 2 [38]).

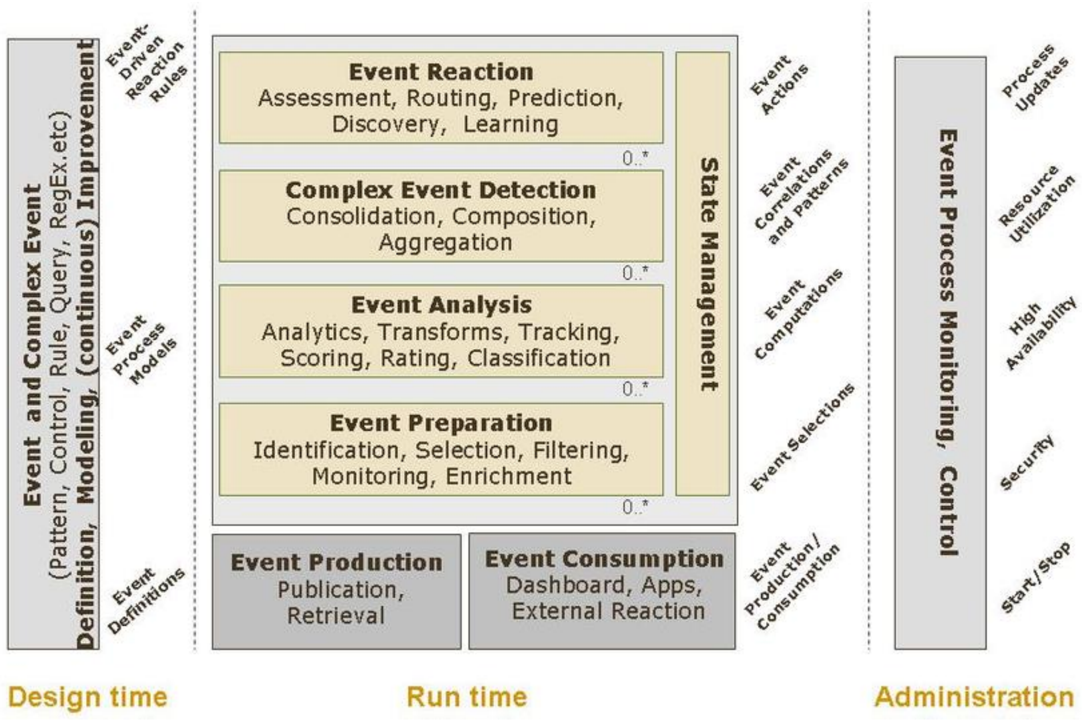

Event and Complex Event
(Pattern, Control, Rule, Query, RegEx.etc)
Definition, Modeling, (continuous) Improvement
Event-Driven Reaction Rules
Event Process Models
Event Definitions
Event Reaction
Assessment, Routing, Prediction, Discovery, Learning
0..*
Complex Event Detection
Consolidation, Composition, Aggregation
0..*
Event Analysis
Analytics, Transforms, Tracking, Scoring, Rating, Classification
0..*
Event Preparation
Identification, Selection, Filtering, Monitoring, Enrichment
0..*
State Management
Event Production
Publication, Retrieval
Event Consumption
Dashboard, Apps, External Reaction
Event Actions
Event Correlations and Patterns
Event Computations
Event Selections
Event Production/ Consumption
Event Process Monitoring, Control
Process Updates
Resource Utilization
High Availability
Security
Start/Stop
Design time
Run time
Administration

Figure 2 - The Event Processing Technical Society Reference Architecture

Similar diagrams and architectural overviews that focus on CEP solution have been published in many research papers and blog posts. But in real life these systems do not exists just by themselves. Data does not just come into existence and does not go into nothingness. Depending on the final setup, additional components might be needed to provide event collection and parsing, buffering, high-availability, authentication and authorization, or a user interface for managing some of these components.

There are lots of different products available that solve one or two of these problems. Some of them even solve multiple problems very effectively. But due to the complexity of these setups, fully featured drop-in products are unfortunately not available for free. Enterprise SIEM solutions mostly fulfill all these needs and add even some additional value, but these solutions come with a high price tag. Open source SIEM offerings are simplified or have restrictions that limit their usage. Some examples are Splunk, IBM QRadar CE, Wazuh, OSSIM and Prelude OSS. Splunk and QRadar Community Edition are offered for free with a data ingestion cap. This means that for simple testing or proof of concept they are acceptable, but these limitations will most likely get in the way of real-world production usage. Wazuh, OSSIM and Prelude OSS on the other hand are offered for free without any data ingestion limits, but they lack additional features. All three of them embed their event rule engine from the same open-source solution OSSEC. The limitations of OSSEC were briefly described in introduction chapter of this thesis. OSSIM and Prelude OSS just use OSSEC as a foundation and do not provide any additional functionality to the CEP engine part. Wazuh does extend some OSSEC capabilities, but it uses the same rule engine and same rule syntax, and this does not allow much more than simple filtering and grouping. OSSEC is not meant for this kind of event processing.

To be able to collect, parse and analyze and react upon events and their relations, a high-level architecture overview is proposed in this paper (see Figure 3). This is a high-level overview of *possible* components that a working solution is comprised of. The word *possible* must be emphasized here as not all components are required for all use-cases and some of these parts can be abstracted away by the technical solution.

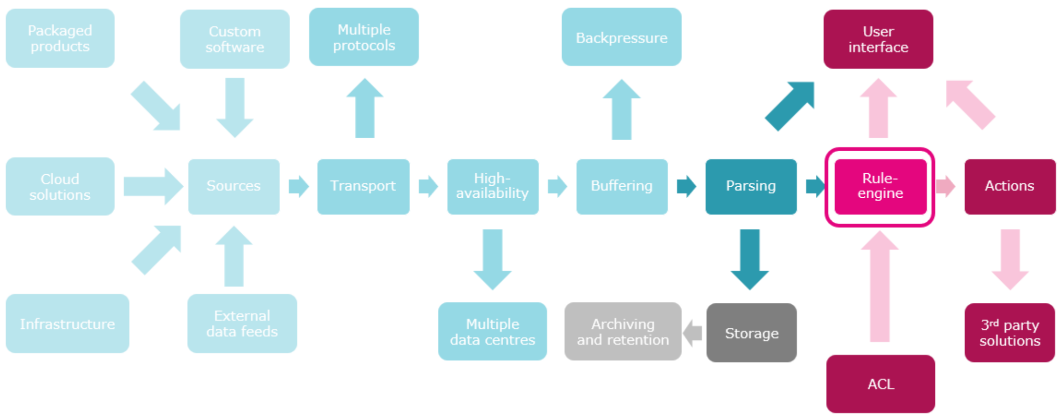


Figure 3 - Proposed CEP solution architecture

Event collection is an essential part of any centralized event processing solution and very few CEP solutions provide these features out of the box. In case of CEP engines that do not have the capability to ingest event streams or lack event parsing features, additional solutions must be used that provide collection, parsing and transformation features.

Event collection systems have different properties and features that define the reasonability of their usage. Support for different input types such as raw files, network traffic, Windows Event Logs or capturing database change records differ from solution to solution. This is in part related to support for different architectures because there might be need to collect logs from systems with different architectures and operating systems. Ability to parse plain text events prior or after transportation can also be a vital feature if the event pipeline after the collection process cannot parse raw text. To transport events, different solutions use different methods. Some publish events to message brokers using standardized protocols, some can act both as a client and server and use their proprietary protocol for event transportation.

There are multiple active and popular open-source solutions that provide different options and possibilities. Components of Elastic stack, Fluentd and Fluent Bit, NXLog, Rsyslog, Syslog-ng and Vector are examples of systems that can behave both as a client and server [27] [39] [40] [41] [42] [43] [44]. Some of them have the capability to send events to an external messaging system. Debezium for example is dedicated to collecting database change records and sending them to Kafka [45]. Benthos on the other hand provides good message parsing capabilities [46]. There are more solutions out there that can be considered as a viable option for final architecture but research of these is not the scope of this thesis.

It is common to use message brokers such as Apache Kafka or RabbitMQ as an intermediate event channel [28] [47]. Message brokers are designed to decouple software architecture by permitting asynchronous communication between parts of the architecture using publish-subscribe paradigm [48]. Guruduth Banavar, Tushar Chandra, Robert Strom, and Daniel Sturman from IBM T. J. Watson Research Center called it the glue technology in their paper "A Case for Message Oriented Middleware" published in 1999 [49]. Message brokers provide the means to decouple event collection from event processing and add an additional fail-safe buffer for events to be stored in case of consumption of events by CEP has stopped due to outage.

In the context of CEP solution architecture, message brokers provide the means for event buffering and optionally high availability. As event processing systems need to be upgraded, or if an unexpected downtime occurs, message brokers can act as persistent message buffers and store all the events they receive. When event processing engine is up and running, it can process these buffered events without losing any info.

High availability is important in event collection because even if there is no unexpected outage, components still need to be updated and periodically restarted but events from remote systems are being sent constantly. Depending on the CEP solution, it can be easier to implement high availability for log collection at the message broker, not in the CEP solution.

And thirdly, message brokers provide a common protocol and unified way for CEP solution to consume events. Remote systems can have different architectures and solutions for shipping events but if they can be centralized into one message broker, then there is less need for CEP engine to have connectors to different types of external systems.

Only after event collection process there is opportunity to correlate different streams of events, detect patterns and react on them.

4.2 Requirements for a CEP solution

CEP engines have numerous different properties and features that affect its implementation, utilization, and ease of use. This section will analyze some of these features in more detail and explain their importance.

These features have been selected by analyzing academical papers that have performed exhaustive research on important features of CEP engines and solution-comparison oriented papers like this. As focus of these papers differ in detail, only common features from them have been selected as analysis criteria for this work. List of papers that have been used as source are as follows:

1. Processing Flows of Information: From Data Stream to Complex Event Processing [9]
2. A Comprehensive Survey on Parallelization and Elasticity in Stream Processing [10]
3. Recent Advancements in Event Processing [11]
4. One SQL to Rule Them All: An Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables [18]

As this work focuses on the ease of use from the perspective of the end-user, some features have been emphasized or added that affect the research and testing of the system and most importantly the usage of the system by end-users. As the solutions being evaluated and analyzed in this thesis are open source, characteristics of open-source projects have been also added as affecting properties.

After all features have been described, there will be a proposed subset of them as minimum baseline feature set. CEP engines that pass this baseline feature set, will be suggested to use as simplest to use in active projects.

4.2.1 Dedicated input support

Dedicated support for ingesting events from different sources presents the opportunity to include less components into the final architecture and thus reducing the overall complexity of the system. Let us call these inputs and outputs using different protocols as input-output connectors. Lack of these so-called existing connectors increase the probability of the need for development know-how or the need to incorporate additional components to the final system.

For initial proof of concept (PoC) implementations such existing connectors provide an easy way of testing the CEP engine itself without having to develop new custom connectors or set up a complex architecture. Every additional step in preliminary testing phase requires additional time, additional resources and is prone to errors. Those errors can affect the outcome and decisions made after reviewing PoC results.

These inputs can serve another purpose. As CEP engine is used, it is likely that the user of the system will need to test changes of the CEP rules before deploying them on real production CEP engine. Lightweight setup which does not depend on existing event collection infrastructure is essential for developing new and testing changed CEP rules.

4.2.2 Event parsing

Events from remote systems are usually collected using different components, but this is not always the case. If this is true, there should be a way to parse events that are represented as plain text log lines. This is not a very common feature for a CEP engine to have, but it does help to raise usability. For example, a bundled package that can be executed and tested or even used for CEP query development on local workstations without the need for additional log transportation and parsing infrastructure, is a very welcomed feature.

4.2.3 Streaming support

CEP solutions are designed to process large amounts of events, a lot more than any average company can produce, let alone meaningfully process. But these events do not have to be all permanently stored. As CEP engines process events, they must keep some events in running memory to meet the query conditions, but after an event has been discarded from all event windows and potential pattern matches, it is no longer needed and can be discarded. This is the potential resource saving aspect of stream processing.

These high throughputs are achieved partly because opposed to batch processing, there is no need to process massive amounts of previously ingested, stored, and indexed data. Batch processing takes huge amounts of data and processes it periodically, causing temporary potential resource bottlenecks and producing results with a delay. This leads us to latency – stream processing will achieve almost instantaneous results, depending on the processing complexity. In systems where automated actions are needed in order to respond to event patterns, this is a critical feature.

4.2.4 Event source time

The ability to understand the actual source event time during event processing provides the opportunity to find correct temporal relations between events even in the case where events are ingested out of order by CEP engine. As it was described in the CEP and ESP introduction, in section 2.2, stream event processing means that a CEP pattern rules are evaluated for every new event that has been observed by the system. Combining these two features means that it is possible that a pattern is detected, but after observing additional events this pattern does not correspond to a match anymore. In another words, the answer is changed after being answered. Let us see this through three scenarios from the example of physical access control events example previously described in this paper (see section 2.2). Events used in these scenarios are described in Figure 4.

	Scenario no 1	**Scenario no 2**	**Scenario no 3**
1	E1 user enters secure area	E1 user enters secure area	E1 user enters secure area at 00:00:00
2	E2 user leaves secure area	E1 user enters secure area	E1 user enters secure area at 01:05:00
3	E1 user enters secure area	E2 user leaves secure area	E2 user leaves secure area at 01:01:00

Figure 4 - Event source time test scenario events

In scenario 1, standard and expected operation of the system is observed. In scenario 2, user enters the secure area second time before leaving it. Without the notion of when these events happened, there is no way of knowing for certain if the scenario 2 is a possible malicious activity or event E2 just happened to arrive to CEP engine after the second E1 event. In scenario 3, there is additional information about when these events took place. If CEP engine is tasked to find pattern of consecutive events E1 without E2 in between then CEP engine will detect this pattern after observing the first two events in scenario 3, even if CEP engine understands the notion of event source time. At the time second E1 event arrives, CEP engine has no knowledge of the event E2. It is only after observing the third event when engine can properly order these events by their time and deduce that there is no anomalous pattern match. This is a simple explanation how answers to questions being asked can change after they are answered. To overcome this obstacle, CEP engines can be tasked to wait for some user configured amount of buffer time to compensate this variable time skew.

Another useful side-effect of being able to understand event source time is the option to replay old events to a CEP engine. David Luckham describes this as running the system offline [50]. This is useful because it provides the opportunity to test the system and complex event pattern rules on historical or even generated and purpose-built test data. In that case time is passing differently for the CEP engine - time and its progress are being dictated to the engine by the events themselves.

4.2.5 DSL

DSL in CEP engine is a way of simplifying rule definitions. They abstract away the specifics of the programming language in what given CEP engine is implemented in. They provide a unique opportunity to involve non-technical personnel as the end-users. This can drastically increase the productivity of the system. This will be shown with a simple example in Figure 5. Esper Event Processing Language (EPL) and simple Python code equivalents will be compared that output the average value for the last five observed events for every new observed event.

Python

```python
def event(evt):
    global mem
    if len(mem) > 5:
        mem.pop(0)
    mem.append(evt)
    print("avg:",sum(mem)/len(mem))
```

Esper EPL

```
select avg(num) from stream#length(5)
```

Figure 5 - Example comparison of Python code and DSL

There are features in CEP engines which will help in numerous complex CEP specific tasks and thereby shorten and simplify the code, but it still must be a valid code describing event processing steps in the programming language at hand. Thus, these CEP rules must adhere to programming language syntax, deployment, and runtime models. DSLs in CEP engines on the other hand are custom tailored to define rules – questions asked from the data. Using a DSL can express more with less.

For non-technical personnel this is a welcomed feature because they have more time to focus on their task – analyzing events.

4.2.6 Pattern matching

Pattern matching is one of the key capabilities of a CEP engine. In a paper presenting Siddhi CEP engine the authors analyzed thoroughly different internal aspects and design decisions of CEP systems and they stated that a CEP engine without the capability to express queries that span multiple input events should not be considered as a CEP engine [51].

As we saw in the section 4.2.5, these queries can be expressed in a generic programming language or in a DSL designed for a specific CEP system. As the aim of this research is to evaluate the usability of CEP engines, option to express these event-spanning patterns in a dedicated DSL is very influential.

In 2006, a feature called row pattern recognition (match recognize) was introduced to the SQL standard [52]. This feature is specifically designed to describe patterns of events. This is not the only way and some CEP engines have implemented their own pattern matching syntax. Esper has even implemented both. Example in Figure 6 shows SQLs match recognize (Esper implementation) and Esper EPL Pattern syntax implementing the same use-case. This use-case is the same as used previously throughout this work (detecting anomalous usage in physical access control system).

SQL match recognize in Esper	Esper EPL Patterns

`match_recognize(`
`  measures e1.data as e_data`
`  pattern (e1 not_e2* e1_succ)`
`  define`
`    e1 as e1.action = 'E1',`
`    not_e2 as not_e2.action != 'E2',`
`    e1_succ as e1_succ.action = 'E1'`
`)`
	`pattern[`
	`  every e1=events(action="E1") -> (`
	`    events(action="E1")`
	`    and not events(action="E2")`
	`  )`
	`]`

Figure 6 - Comparison of two Esper pattern detection syntaxes

Both syntaxes are describing following pattern of events – two instances of event E1 must appear in the event stream without having event E2 in between. Putting it into context of the use-case, these patterns detect the anomalous entrance into secure area.

The example above could be implemented in a generic programming language quite easily but as these patterns grow more complex, advantages of these dedicated pattern matching syntaxes become more noticeable. For the rest of this thesis, ability to express these pattern definitions in a declarative way within an event query is assumed. Full analysis and explanation with complex use-cases of these syntaxes is not the scope of this work.

4.2.7 Event windows

Similar to the pattern matching, event stream windowing capability is considered as one of the key features of a CEP engine [51]. Something without what the system should not be considered as a fully featured CEP engine. There are different explanations to windows, but this thesis views them as subsequences of zero or more events from event streams. These events are related to each other by relations defined by the type of the window. If event streams are abstract constructs that represent infinite flows of events, then windows represent a selection of events from that stream in a specific point of time. For example, a window of length 1 event holds 1 event in any given time. But a window of length 1 minute holds all events from the stream within that one-minute period. A window of 5 events having same property value holds 5 events that does not have to be a contiguous subsequence of events from the event stream, they must have equal valued property. Depending on the CEP engine, different types of windows are available for use. All of them simplify event processing in specific use-case types.

One typical way to categorize different windows is by time and length. Time based windows retain events from a certain amount of time and windows of length type retain certain number of events. Both window types can also be represented as a sliding or tumbling (also called batch) window. Sliding windows can be thought as windows with constantly changing contents. For example, a time-based sliding window of length one minute contains all events one minute into the past – meaning that the start and end of this window changes as time flows. But a tumbling time-based window with same length always retains its start and end times, even if the end is in the future. As one tumbling window ends, another one starts and next one starts one minute after that, and so on. Differences of these types are shown visually on Figure 7 [53].

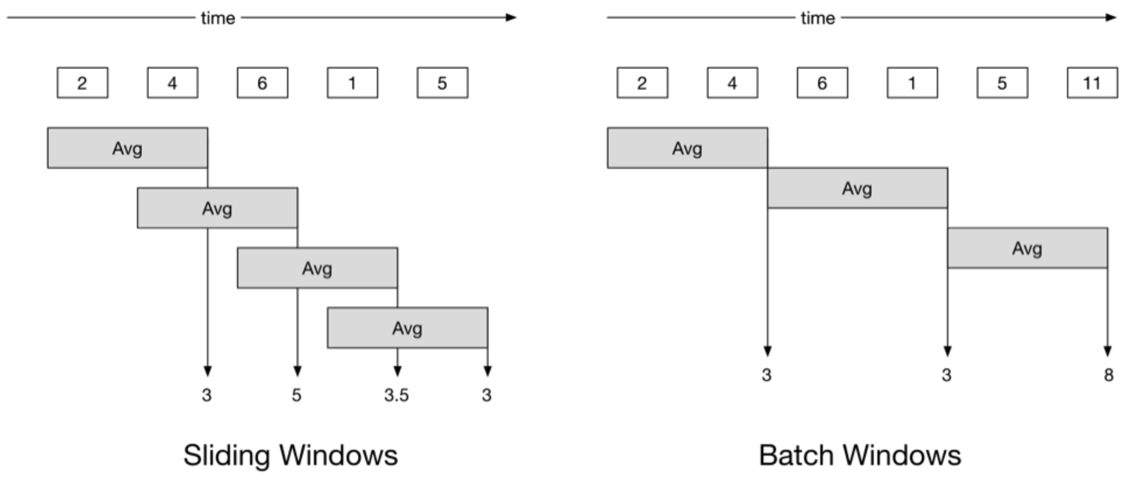


Figure 7 - Sliding and tumbling (batch) windows.

In addition to window types mentioned so far, there are other specific window types implemented by CEP systems. Some of them providing sorted view of the events within the window. Each one helping to solve some problem more easily. As it was already mentioned – windows are essential part of a CEP engine and the more functionality and features they provide, the easier it is to ask complex questions from event streams.

4.2.8 Joining multiple streams

Joining multiple streams means joining events from one stream with events from another stream, typically based on some properties of these events. This is very similar to joining multiple tables in a query from relational database. But on streaming datasets concepts of joining are a slightly different. For example, when two streams are joined, then the timing of the arrival of records plays an important role. This research will not go into the details of explaining complexities in joining streams, but will quote 3 important problems highlighted by Srinath Perera, Vice President of WSO2 [53]:

1. “The first challenge of joining data from multiple streams is that they need to be aligned as they come because they will have different time stamps”.
2. “The second challenge is that, since streams are never-ending, the joins must be limited, otherwise the join will never end”.
3. “The third challenge is that the join needs to produce results continuously as there is no end to the data”.

But it is important in CEP engines to be able to join different streams because complex events often span over the scope of one stream.

4.2.9 External non-streaming data-source support

Enriching events as they are ingested or in any other step within the CEP engine can provide necessary context around the event. By having the option to look up and add GeoIP information for IP addresses or match usernames to user IDs might be essential for detecting a crucial complex event. Such datasets are often accessible via different means and from different systems. Possibility to join information not just from another event streams, but other external datastores with bounded data is a very useful feature in a CEP engine.

Second side of the same feature is updating these external data-sources with information from event streams. External bounded datasets are not only useful for reading but in certain cases they need to be updated as well. Let us take an example - keeping the state of last active country of a logged in user in an external database for additional analysis. For this solution we must be able to read stream of login events containing user ID and source IP, join external information to identify country and username and write the last known country for that user into another database to be used by another applications.

The more external sources can be directly communicated with, the more effective a CEP solution can be. This is especially important as many use-cases need immediate and automatic reaction if some specific complex event has been detected. In the context of this thesis, during comparison, it is assumed that the engine itself has the built-in connectors and capabilities to connect and communicate with these external non-streaming data sources.

4.2.10 Deployment options

Whether the CEP engine can be used just as a library in a specific programming language, run simply as a command line utility or started as a standalone server will decide the requirements for implementation team and infrastructure. There are many ways how a CEP engine can be distributed and made available:

1. As a programming language library
2. As a Docker container
3. As a command-line runnable utility
4. As a server with an API or GUI

This is by no means an exhaustive list, but it will illustrate that different solutions can come in many forms - as an archived patch of code, or as a polished GUI with all the extensive features. Regarding the ease of use, writing custom code, building, and deploying it on some cluster-manager via command line interface is not considered as an easy-to-use solution. What defines an easy-to-use solution are projects that offer a bundled local test-environment with examples that can be run in regular desktop computer and in addition provide seamless installation for different server and container distributions.

4.2.11 High-availability and horizontal scaling

Business critical processes require high-availability (HA) support for minimizing downtime. As implemented solutions evolve in time and more use is found for them, criticality of these solutions raises. As usage and use-cases change, so must the underlying architecture. Not all solutions support zero downtime upgrades or any kind of multi node failover installations. This is something that must be thought trough before initial implementation and potentially implemented from the beginning. Another caveat that might arise as solutions evolve in time – a computational resource choke. A single installation, even backed up by a failover instance, might not be sufficient to handle the load. Like with HA, support for scaling event processing horizontally, is not something that is built into every solution.

4.2.12 Graphical user interface

As one of the aims of this research is to focus on the ease of use for CEP solutions, existence of a GUI is a vital property. Like with DSL, GUI lets users concentrate on what is important, depending on the business need. There are many people who have good analytical thinking and domain knowledge but lack the skill to operate on Linux command line or write Java code. In installations targeted for non-technical users, GUI is a welcomed enabler and productivity amplifier. Like with Windows operating system or with the rise of web browsers, graphical interfaces have brought more people to the userbase and by doing that, they have also helped in the further development of the solutions. Considering ease of use, existence of a usable GUI that provides DSL development and deployment is one of the most important features.

4.2.13 Community and documentation

For open-source projects, size and liveliness of the community are good indicators of a healthy product. Even the number of issues or bug reports is a good indicator that there are many users of this product and they are concerned for the well-being of the product. Number of discussions in different forums and blog posts about implementation details, and even likes in the source repository are all indicators that there are users who have invested their time to at least trying to implement the solution, and to seek help from the community instead of discarding the solution. Thoroughness and readability of documentation is another clear sign of a well operating project.

There is no clear baseline or evaluation method that would say which project is bad or good based on the criteria mentioned above. But outliers are clearly visible, and this research is focused only on bringing out all these outliers. Lack of proper documentation, no activity, or no easily available information at all are the tell-signs to be reported and considered.

4.2.14 Enterprise support

Availability of enterprise support should be considered as good backup option if open-source implementation does not go the way intended or even primary plan if the necessary expertise is not available within the organization. The enterprise support is generally offered to solutions that have backing and lead development role by an organization.

4.2.15 Minimum baseline feature-set

So far, many important and useful features of CEP engines have been covered. To be able to clearly define and highlight the systems that conform to all the minimum CEP requirements and that are most easy to use, key characteristics will be defined that increase the ease of use and set minimum requirements for a CEP engine. Importance of all the previously described requirements was already described, but reasons why some requirements have been left out from minimum baseline, will be explained in the end of this section.

1. CEP engine must be a fully featured CEP, meaning:
    - 1.1. It must be able to stream process events (see section 4.2.3).
    - 1.2. It must be able to understand event source time (see section 4.2.4).
    - 1.3. It must have dedicated event pattern matching solution (see section 4.2.6).
    - 1.4. It must be able to connect to non-streaming data stores (see section 4.2.9).
2. CEP solution must be easy to use, meaning:
    - 2.1. CEP engine must have a DSL and support main CEP features, referenced in requirements 1.1-1.4, in DSL.
    - 2.2. System must have GUI for developing and managing CEP queries in DSL (see section 4.2.12).
3. Open-source project must not be discontinued or dead, meaning:
    - 3.1. Project is not marked as discontinued (see section 4.2.13).

3.2. Project has no indications of being a negative outlier in community (see section 4.2.13).

Systems that correspond to these requirements, will be highlighted as most easy to use, and having all the features of a CEP solution.

Dedicated input and parsing support were not selected because connecting to different data sources and parsing unstructured text are not the primary goals of a CEP engine. At first and foremost, a CEP engine must be able to process streaming data and detect patterns of complex events. As an alternative, there is always option to use other tools to collect events, intermediate the connection to specific data sources and parse unstructured text.

Although support for different event window types and joining of different streams are very important features of CEP, they are not as critical as the capability to define and detect patterns of events and their temporal relations. If patterns spanning over multiple streams must be detected, forcefully joining streams before funneled into CEP can be used as an alternative approach.

Possibility to choose from different deployment options and design a horizontally scalable CEP solution with high availability should not be considered as irrelevant features. But a system can still be set up even if there is only one supported deployment model and high availability for log collection can be ensured by adding HA message brokers to the architecture. And to certain extent, scalability can be addressed by manually installing multiple instances and manually splitting the event streams to process events separately.

4.3 Analysis of open-source CEP solution

CEP solution described here are responsible for event correlation and pattern matching. These are the engines that derive meaningful information from user-defined rules based on the events that they ingest. This list of CEP engines is compiled based on information gathered from online information and academic articles used throughout this research. These CEP engines will be analyzed based on the requirements defined in section 4.2.

Information about these solutions has been acquired from the official documentations of these systems, if not stated otherwise.

4.3.1 Apache APEX

Apache APEX was one of the earliest Apache foundation projects to incorporate event stream processing concepts [54]. It is a platform and framework built on Hadoop for building distributed stream and batch processing applications. As Apache APEX has been recently retired to Apache Attic and is no longer in active development it will not be thoroughly covered [55].

4.3.2 Apache Beam

Apache Beam provides programming-language agnostic unified model for batch and stream data processing (the name Beam comes from Batch and strEAM) [56]. Beam is an orchestration and management layer coordinating other popular streaming processors as individual processing jobs, called runners [57]. These runners are processing events defined in the Beam data flow description, called pipeline. Currently, execution engines supported are Apache Flink, Apache Nemo, Apache Samza, Google Cloud Dataflow, Hazelcast Jet, Twister2 and Apache Beams internal Direct Runner. Beam API for defining these pipelines is provided in Java, Python and Go.

In addition to local file system Beam provides built in ingestion support for files from filesystems such as Hadoop, Google Cloud Storage and Amazon S3. Beam also supports reading events from various message brokers and databases. Most popular of them are RabbitMQ, Kafka, MQTT, Amazon Kinesis, Amazon SQS, Google PubSub, Cassandra, HBase, Elasticsearch, MongoDB, Redis and any database with a JDBC driver support.

As a quite recent addition (initial introduction in late 2017) Beam provides support for using SQL queries to process data [58]. From then on, this feature has been developed further and has gained some momentum, but there are still quite restrictive limitations. Although there is an interactive shell for executing SQL queries without any additional programming, it is just meant for ad-hoc querying for batch data. Stream data processing jobs on unbounded streams must be defined through programming API using Java or Python. Beam has included support for two different SQL dialects. One is provided by Apache Calcite project and another one is called ZetaSQL and is more focused on querying Google BigQuery solution. But neither of them yet have the support for declarative pattern detection and have limited support for reasoning about time using event windows. There was an effort in 2020 Google Summer of Code to implement match recognize as a CEP feature for Beam, but this feature is not yet implemented [59]. Joining unbounded and bounded streams in SQL is supported, but not all join types are implemented.

There is an additional extension that allows to define external data resources declaratively through SQL. This would be useful for enriching stream data but many connectors for this are still marked as experimental or have not yet been implemented.

Support for event source time is built into Apache Beam. Beam tries to account for late arriving data by keeping record of a watermark timestamp. This watermark is systems best guess when all data in a certain window should have arrived. When defining Beam's pipelines in a programming language, this watermark estimation algorithm can be changed to better suit the ingested events, but this is not an option while using plain SQL queries.

In conclusion, Apache Beam is a comprehensive framework for managing complicated stream processing topologies. It does not provide all the fancy convenience features in SQL query language, it does not provide any drag and drop user interface, but it does have an unique feature to run complex stream and batch event processing topologies on different CEP engines in a highly distributed form. New features such as declarative SQL support do make entry barrier a bit lower, but for now there are still many features that are missing or need additional development before Beam CEP engine could be fully used by these SQL dialects. Apache Beam is a good option when there is need to run complex event processing scenarios but for some reason different queries and pattern detections need to be executed in different CEP engines. There does not seem to be official enterprise support for Apache Beam, but as Beam originates from Google, Beam pipelines can be run in Google Cloud Dataflow service [60].

4.3.3 Apache Flink

Before it moved under the Apache foundation and named Apache Flink (in 2004), it was a research project named Stratosphere in Germany as a collaboration of multiple universities [61] [62]. Since then, it has become a widely adopted stream processing engine used by Alibaba, Amazon AWS, Uber, Ebay and many more. Flink is designed to be batch and stream data processing engine capable of using virtually unlimited amounts of resources due to high parallelization support. Flink engine can be executed within resources managers like Hadoop YARN, Apache Mesos and Kubernetes. For data ingestion, Flink includes in addition to local files connectors to various external messaging and storage systems such as Kafka, Cassandra, Kinesis, Easticsearch, RabbitMQ, Google Cloud PubSub and data-sources with JDBC drivers. Projects Apache Bahir and Community Packages for Apache Flink provide some additional connectors [63] [64].

Like Beam, Flink has built in support for event source time, and it handles lateness of events via similar watermark mechanism as in Beam.

But unlike Beam, Flink has implemented a lot more features into its SQL query language based on Apache Calcite project. As of latest version (v1.12) comments about the SQL query solution not being feature complete have been removed from its official documentation and it is considered as production ready. Although Apache Calcite is heavily being used to provide SQL capabilities to many event processing projects, Flink has implemented many features that are of importance to CEP. Most notably, a pattern matching feature called match recognize that was described in chapter 4.2.6. In addition to that, Flink has also implemented good support for joining streams and windowing in its SQL syntax. Some of the built-in input-output connectors can be directly used from SQL. For example to create a Flink SQL table directly backed by Kafka topic, Elasticsearch index or any other database through JDBC connection. It is very similar to Beam-s implementation, but like other facets of SQL, it is much more feature complete and documented as production ready.

As a project of Apache Software Foundation, Flink is provided for use as Java and Python libraries and as a packaged runtime that can be used to run prebuilt Flink jobs. However, this distribution does require the user to develop and package these jobs in advance in a generic programming language, even if SQL queries are executed within this job.

However, a company named Ververica founded by original authors of Flink, does provide a pre-packaged Flink with GUI and other additions called Ververica Platform Community Edition [65]. This GUI has a Flink SQL editor for developing Flink applications and it allows full control over lifecycle management of these applications. Ververica Platform runs on Kubernetes cluster. It does not have locally runnable server but if for an already existing Kubernetes cluster, it is relatively easy to get the platform up and running. Ververica also offers an option to purchase enterprise features and support. But it must be mentioned that this platform does not have an open-source license. Product is free to use, but source code is not available, and redistribution or hosting it as any kind of service is forbidden.

4.3.4 Apache Heron and Apache Storm

Both have been originally developed and then open sourced by Twitter [66]. Heron was built as the successor of Storm to overcome Twitter-scale Storm shortcomings. Although Heron project is still in incubation stage Apache Software Foundation, software itself is considered as production ready. Storm and Heron both run applications, that are called topologies. These topologies are directed graphs that consist of inputs (spouts) and processors (bolts) and are written in Java or Python, no special declarative language is supported.

As Heron was built to be backwards compatible and it can run Storm topologies, further analysis will concentrate on Heron.

Heron does provide a GUI and a command line interface, but both are just for interacting with topologies and managing the cluster. Local, single node test environment is possible, but by documentation it requires building Heron from source and is not a straightforward task. As a production system, Heron is highly distributed and currently supports multiple cluster scheduling options such as Aurora, Slurm, Hadoop YARN and Kubernetes. Heron does not come with a good selection of existing spouts (data sources), as the only non-incubation stage data source is Kafka.

There is no direct documentation about the support for understanding source event time. There are references to watermarking (indication of the support) in Java API, but these are also in packages containing the suffix test. Joining streams and supporting event windows is supported but as there is no DSL, this is only applicable if programmatically writing applications to run on Heron’s cluster.

Apache Heron (and Storm) is a capable stream processing system that scales well horizontally and supports different cluster management systems, but it is not a CEP engine as such.

4.3.5 Apache Samza

Apache Samza is Apache Software Foundation top-level project since 2014 and has its roots in LinkedIn [67]. While Samza has connectors to various streaming messaging services such as Kafka, Azure EventHubs and AWS Kinesis, it has limited integration with bounded data-sources – only Elasticsearch is documented. Samza has added a DSL query language powered by Apache Calcite but this only supports limited set of stateless functions for now. This means no event windows, no joining, and no event pattern matching. However, documentation does state that stateful operations are in the roadmap. Samza also does not support processing events by their source time natively – this is supported only trough integration to Apache BEAM.

Although Samza has been around for a few years there seem to be no enterprise support options available, and no dedicated GUI for managing applications. When running Samza applications on YARN cluster, there is YARN interface for managing these applications, but this does not provide insight into the application itself. Samza fits better into architecture where stream processing applications can be written in Java, streams are very high-volume and event processing needs to be highly distributed – this sort of horizontal scaling is something that Samza supports well.

4.3.6 Apache Spark

Apache Spark is a large-scale data processing engine developed by UC Berkeley's AMPLab in 2009, and open sourced in 2010 as an Apache Software Foundation project [68]. Spark can be run locally for testing or for running example applications in Windows and in Linux systems where appropriate JVM version exists. For production-grade systems multiple clustering solutions are supported such as Apache Mesos, Hadoop YARN and Kubernetes. Spark is a solution that does not fit clearly into batch nor stream processing categories – it is a micro-batch event processing framework. Spark ingests data like typical event stream processing solutions, but before processing them, it buffers events and creates immutable micro batches - Resilient Data Sets. Although this batch size can be configured, there are recommendations on how to configure and limits what Spark can do. So true event at a time stream processing is not achievable with Spark.

In other aspects of a CEP engine – Spark understands event source time, provides event windows, stream join operations and all those features also in a SQL compatible DSL. But from the viewpoint of this research, the downside is that there is no support for event pattern matching in this DSL and there is no GUI for developing these Spark applications using this DSL. One detail rules out Apache Spark as a CEP engine, and another reduces the usability aspect.

Spark does also not excel in built in connectors area – only Kafka and Amazon Kinesis are listed as available stream input options. There is a separately managed community site called SparkPackages that provides some extra options, but these are not validated by and tested by Spark release pipeline [69].

As a big-data scale event processing engine, Spark meets all the requirements by being highly scalable, with vibrant community and enterprise offerings by Cloudera [70].

4.3.7 Faust

Faust is a stream processing library written in Python and designed to be used only in Python, getting its inspiration, and sharing many similarities with Kafka Streams project [71]. As opposed to Kafka and its Streams subproject, Faust does not have so vibrant and active community and there are no enterprise support offerings.

It has elementary event window and stream join support, but no DSL for querying event streams. This means that all event processing must be defined with python code. Although it is possible to write python code that mimics stateful event pattern matching that spans over time and events, there are no dedicated features provided to declare such temporal relations. Like Samza, Faust is oriented to just stream processing – there are built in connectors to Kafka, RabbitMQ and few other messaging middleware, but no direct support for additional external non-streaming data-sources.

Faust is a good option to try stream processing if there is existing python environment and knowledge to be used, as it does provide some stream processing features. But due to the lack of complex event pattern matching features, Faust does not qualify as CEP engine. Nor does it provide any GUI, high-availability, or horizontal scaling.

4.3.8 Elastic EQL

Previously known as Endgame EQL (before acquired by Elastic) [72] [73]. This solution provides a limited pattern detection (called sequences in Elastic EQL) on data stored in Elasticsearch database. EQL is not a dedicated CEP engine as such. It is an additional query syntax in Elasticsearch that provides some features of a CEP engine. But these queries must all be executed periodically on already collected and stored information (in Elasticsearch indexes), there is no way to continuously run these queries on event streams.

Another important limitation is that due to implementation specifics sequence queries fail to detect all potential matches. This is also described in the documentation and reasoning is that it would be too resource intensive for large event data sets. This is most likely the outcome of processing historical bounded datasets instead on processing streaming data.

Because EQL is meant to be used on already stored events, functions such as event source time or different event window types, that are typical to CEP systems are not relevant.

Although Elastic has a very vibrant community, easy to use and polished GUI, support for horizontal scaling and high-availability and enterprise offerings, no support for stream processing and lack of decent pattern detection capability prevents it from being classified into full CEP solution category.

4.3.9 Esper

Esper is a CEP engine developed by EsperTech [74]. Esper offers very little help for connecting to different message brokers or databases, it is provided as embeddable CEP engine libraries for Java and .NET runtimes. There is a subproject called Esper-IO that adds some input and output features to and from Esper, but there is still need for custom programming to utilize them. No pre-packaged command line runtime, server or any solution that would contain GUI is provided. This means that even testing and creating a proof of concept does come with the requirement of development skillset on either of those languages.

But CEP Engine features are well developed - Esper is the only fully featured CEP solution covered in this research that provides a SQL based DSL language (Event Processing Language or EPL) with support for two distinct language constructs to define event patterns (EPL patterns and a more standardized match recognize). Esper also has built in support for handling event source time, connecting to JDBC enabled data-sources directly and using them from within SQL, 21 different event window types and possibility to use different join types. All the necessary features are there and often more advanced than in other CEP engines. Esper EPL is designed to be the primary usage method for Esper, not an additional extra feature as it is for other CEP engines. And that reflects also in the documentation – full reference documentation for Esper is over one thousand pages long. As Esper is provided only as a library, there is the additional API documentation. This exhaustive documentation is also necessary because there is not so much community activity as for other CEP engines.

As it was already mentioned, Esper open-source offering does not come with the support for high-availability, horizontal scaling nor any GUI. But all these features are provided in the Enterprise Edition.

To summarize - entry barrier for using Esper should be considered at least average if not hard compared to other CEP solutions covered in this research. But for solutions where a comprehensive CEP engine must be embedded into a Java or .NET application, Esper should be considered as an option.

4.3.10 ksqlDB

This is a product built by Confluent on top of Kafka [75]. Previously know just as KSQL, it was rebranded in the end of 2019 [76]. It started as a simple SQL query language extension to Kafka topics operating on top of Kafka Streams but eventually grew into a separate product. ksqlDB provides multiple features that are inherent to a CEP system such as understanding event source time semantics, using event windows, and joining multiple streams within a query. But it lacks one vital feature to be a fully featured CEP system – the actual complex pattern detection.

Because ksqlDB is directly related and relies upon Kafka, it leverages its vast ecosystem of input and output connectors provided by Confluent. Due to this, ksqlDB has by far the largest different input and output connector support in CEP systems covered in this research. Both bounded and unbounded event sources are supported and can be leveraged to join multiple streams or to do event enrichment using external sources.

ksqlDB itself runs as a server that connects to a Kafka cluster and provides a simple shell for executing queries. Confluent also provides a GUI component called Control Center with their Confluent Platform that is meant to manage Kafka management and to develop, view, and run ksqlDB queries.

All the Confluent products, especially Kafka, have very active and thriving community. Although ksqlDB and Kafka both support complex fault-tolerant and high-availability setups as a free offering, there is additional enterprise support option to be used. Kafka is also the most widely supported messaging platform within the CEP engines and event collection solutions covered in this research. In a situation where Kafka is already used to transport events, then ksqlDB is an easy to implement addition that provides lots of easy-to-use stream processing features. But it is not a full CEP solution as it does not support any event pattern matching. This must still be implemented using Kafka Streams to develop custom applications using Java or Scala programming language.

4.3.11 Siddhi

Siddhi is a CEP engine and runtime developed under the WSO2 corporation [77]. It started as a final year project of the CSE department at the University of Moratuwa in 2011 [78] [51]. After that it became one of the main projects in WSO2 and by now Siddhi’s engine is used in multiple streaming analytics products within the company. Siddhi has been also used by Uber, EBay and PayPal throughout its history.

Siddhi on its own, like Esper for example, is provided as a programming library for Java and Python. But unlike Esper, Siddhi has additional companion solutions that provide a prepackaged runtime and additional tooling with a user interface. Siddhi’s runtime and its user interface provide quick way of running sample applications, editing, and running Siddhi queries. Siddhi applications are files containing Siddhi queries and all the necessary definitions for data input and output.

Like Esper, Siddhi has a SQL-like DSL that was designed to be the primary event processing tool. So, all the implemented features are available directly in this DSL. Support for handling event source time, event pattern matching (match recognize), 14 different event window types, joining of streams any many other features a built into the engine. But there are numerous additional extensions provided by WSO2 that extend and add to these features.

Siddhi's application declaration syntax allows to declare more than just this query language – these application definition files could be compared to Apache Beam's pipelines or Apache Heron's topologies. They include the event ingestion definition, processing, output configuration and can be ran with Siddhi's command line runner, via GUI or using REST API. This is possible as Siddhi provides list of input and output connectors as well, covering most of the popular technologies such as Kafka, JMS, HTTP, and plain text files. Siddhi can also handle events containing custom unstructured text by using regex patterns to extract information.

As it was already mentioned, siddhi CEP engine is part of other WSO2 solutions, and that Siddhi has additional tooling to aid development. For example, WSO2 Streaming Integrator (successor of WSO2 Stream Processor) wraps Siddhi in an enterprise environment supporting high-availability clusters and multiple graphical and command line interfaces for managing various aspects of software integration within an organization. For stream- and complex event processing, Stream Integrator adds additional components such as Streaming Integrator Tooling, Business Rules Template Editor and Business Rules Manager. These tools are provided as web-based user-interfaces for developing and managing the lifecycle of Siddhi applications, creating templated Siddhi applications, and creating web-based forms for executing new Siddhi applications based on the templated parameters by just filling a web-form. This is a powerful set of convenience tools for an open-source CEP solution.

To summarize, Siddhi is a very well-equipped CEP engine that can be embedded into other programs if needed. But if used with additional tooling provided by WSO2, Siddhi's CEP engine can be turned from a complex tool for expert engineers to a simple to use web application for non-technical end-user.

4.3.12 Simple Event Correlator

Simple Event Correlator (SEC) is exactly what the name says – a simple tool for event correlation [79]. SEC applications are configuration files consisting of some number of rules. Although it is possible to use this tool to process event streams and even find patterns and temporal relations related to those events, it all comes in a bit more differently than on other previously mentioned solutions. Those streams can be files or named pipes, not some messaging middleware such as Kafka. Temporal relations and patterns can be detected by chaining multiple rules together and using SEC-s built in context feature as an intermediate memory, not by using a dedicated DSL that provides pattern description through declarative means. By the requirements set in section 4.2.15, SEC is not a true CEP engine.

Although SEC is a very flexible tool, it is provided only as command line utility and can ingest files (including stdin) and named pipes. For this reason, it is more suitable for correlating events directly in remote systems before they are collected into one central event processing solution.

5 Summary of CEP solutions

CEP solutions analyzed in chapter 4.3 give an overview of current open-source solutions, their capabilities and ease of use. It turned out, that many of systems selected from various academical papers covering different aspects of CEP and streaming systems did not even qualify as true CEP systems as defined by this research. Lack of options besides raw programming language to declare and detect patterns of events and their temporal relations is inherent to multiple such systems. As these systems, such as Apache Spark and Apache Beam, are indeed popular and widely used by many big corporations, it can be deduced that declarative complex event pattern matching is not yet very widely spread, at least not in the open-source world.

Another important class of systems are the ones that do conform to the CEP requirements but provide their CEP engine as a programming language library. A separate experiment was conducted for implementing one such solution. Specifics of this experiment will be covered in section 5.1. Although such solution provides the most flexibility, it also comes with a responsibility to invest quite a lot of resources into development. This path is most definitely not the easy-to-use solution.

In the middle, there are number different solutions that qualify as CEP engines, but do not provide convenience features such as DSL, or GUI for developing and managing DSL based CEP applications. Existence of GUI application for managing actual CEP rules that are applied to processed events is a huge step forward in lowering the learning curve.

5.1 Custom CEP implementation

To fully comprehend the complexity of using a CEP system as a programming language library, one CEP solution was selected for experimental test implementation. As existence of GUI and deployable server are not important in such scenario, availability of SQL and its features were basis of the final solution selection. As Esper is the only CEP solution in this comparison to support multiple pattern definition syntaxes and has the widest selection of different window types, it was chosen for this task. Esper is provided in the form of Java and .NET libraries. JVM execution environment and Java libraries were chosen to better align with other solutions as most of them are based on Java. Set of goals was established for this proof-of-concept application:

- Must be command-line runnable.
- Must be configured through external configuration.
- Must have modular input-output system.
    - Must at least be able to read regular files as input.
    - Must at least be able to write results to regular files.
    - Additional input-output connectors must be added easily.
- Must be able to ingest and parse unstructured data.
- Must be able to connect to non-streaming external data sources.
- Must be able to “play back” old data and use source event time.

The result was a roughly 10k lines long modularized Java application that could be run from a command line with an external configuration, defining four key parameters such as data source, data parsing definitions, Esper EPL queries and result output destination. One working simple example configuration (a runnable application definition) is shown on Figure 8.

```
# Sample data:
#128   first line
#1024  second line

# input definitions (read contents of file 'logfile.log')
input: logfile.log

# how to parse the input (fields 'seq' and 'data')
pattern: ^%{INT:seq}\t%{NOTSPACE:data}$

# esper statements on parsed data
query: select seq,data from events where seq < 4

# output definitions (output resulting json to file 'outputfile.out')
output: outputfile.out
```

Figure 8 - Custom CEP implementation sample configuration

To facilitate real-world scenarios, additional features for developing and testing applications were added.

Depending on the specific requirements, working application can be achieved with much less code. In the case of this prototype application, it was made to handle as many use cases as possible with just changing external configuration. Good portion of the code was written to be able to understand different specifics of the CEP engine and how to use it correctly. It is worth mentioning that this solution does not provide any user interface or any other simplifying components for a non-technical user.

Throughout development of this solution, it was tested in multiple scenarios with real streaming data (using Redis as messaging middleware). Two successful scenarios describing the analysis of Windows filesystem audit logs and MailCleaner e-mail gateway server logs will be described in more detail. These scenarios were chosen so that complex pattern matching capabilities could be used. Both tested log sources provide information about logical events in such a way that information about these logical events is scattered in multiple log lines. These raw events can be unordered in a way that pieces of information from different logical events can be mixed.

Microsoft Windows filesystem audit logs do not provide clear and explicit events for when file system objects are read, written, moved, or deleted. Evidence of these logical events are scattered through multiple different atomic events representing actions with file handles having different access masks. During testing, CEP queries searching for specific patterns to detect these logical events on file systems were successfully ran on streams reaching well over 1 000 events per second (EPS).

For this, at first information about logged in users must be collected and kept in a separate lookup window because events regarding file system activities only log a generated logon identifier. This was achieved by keep record of active logons to map logon identifiers to actual user – this was implemented within the same application and using Esper stream queries. For detecting actual file system events, a pattern of events for each logical activity was created that when matched, resulted an output that used previously mentioned user information along with the data of the action itself. A sample from file system object rename detection pattern is shown in Figure 9. Insignificant parts have been redacted for better readability.

```
@Name('FSEvents')
insert into FSEvents(/* redacted */)
select
    /* redacted */
    "RENAME" as ActionType
from
    pattern [
        every init=win_evt(
            EventID = 4663 and
            AccessMask? = '0x10000' and
            HandleId? is not null
        ) -> act=win_evt(
            EventID = 4663 and
            AccessMask? = '0x80' and
            HandleId? = init.HandleId?
        ) where timer:within(5 sec)
    ] as patmatch;
```

Figure 9 - Custom CEP implementation Windows filesystem events test scenario Esper query

Second test scenario dealt with processing of MailCleaner email gateway server logs. MailCleaner provides detailed logs about every e-mail that it processes but it writes multiple lines per e-mail providing different information on each line. In this test scenario, a CEP query was defined that searched for patterns in events that formed full information about an e-mail. Example of an email detection pattern is shown in Figure 10. Insignificant parts have been redacted for better readability.

```
@name("mc-emails")
insert into mc_emails(/* redacted */)
select
    /* redacted */
from pattern [
        every-distinct(mc_from.key) mc_from=mc_from(
            key is not null
        ) -> mc_to=mc_to(
            key = mc_from.key
        ) -> (
            timer:interval(3 minutes) or
            (
                mc_completed=mc_completed(key=mc_to.cid)
            )
        )
    ] as pat unidirectional
        left outer join
            mc_dmarc#time(3 minutes) mc_dmarc
            on mc_dmarc.key = mc_from.key
        left outer join
            mc_spamc#time(3 minutes) mc_spamc
            on mc_spamc.cid = mc_to.cid
        left outer join
            mc_trustedsources#time(3 minutes) mc_trustedsources
            on mc_trustedsources.cid = mc_to.cid
        left outer join
            mc_newsletter#time(3 minutes) mc_newsletter
            on mc_newsletter.cid = mc_to.cid;
```

Figure 10 - Custom CEP implementation MailCleaner test scenario Esper query

Firstly, these test scenarios illustrate that quite complex patterns can be expressed by using these SQL based declarative pattern matching syntaxes.

Second outcome of this proof-of-concept application for this research was that any CEP implementation that builds upon a programmatic interface, is likely to be relatively complex and time-consuming endeavor. Aforementioned development consumed in an estimate of one-month regular dedicated work hours, including research of necessary documentation. And by no means is this a production ready or easy to use solution by the definition of this research.

This proof-of-concept solution will be published on Github source repository before final submission of this thesis [80].

5.2 Comparison of CEP solutions

	Input	Streaming support	DSL	Event source time	Event windows	Stream joins	Event pattern matching	Event parsing	Non-stream data-source	Library distribution	Server runtime	Command line utility	High availability	Horizontal scalability	GUI	Community	Enterprise support
Apache APEX	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	-	o
Apache Beam	x	x	x	x	x	x	-	-	x	x	x	x	x	x	-	x	-
Apache Flink	x	x	x	x	x	x	x	x	x	x	x	x	x	x	-[1]	x	-[1]
Apache Heron	x	x	-	-	x[2]	x[2]	-	-	-	x	x	-	x	x	-	x	-
Apache Samza	x	x	x	-	x[2]	x[2]	-	-	-	x	x	x	x	x	-	x	-
Apache Spark	x	x[3]	x	x	x	x	-	-	x	x	x	x	x	x	-	x	x
Faust	x	x	-	x	x[2]	x[2]	-	x[2]	-	+	-	x	-	-	-	x	-
Elastic EQL	x[4]	-	x	-	-	x	x[5]	x[4]	-	-	x	-	x	x	x	x	x
Esper	-	x	x	x	x	x	x	-	x	x	-	-	-	-	-	x	x
ksqlDB	x	x	x	x	x	x	-	x		x	x	x	x	x	x	x	x
Siddhi	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
SEC	-	x	-	-	x	x	-	x	-	-	-	x	-	-	-	x	-
Custom CEP	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o

Table 1 - Comparison of CEP solutions
x = supported, - = not supported, o = irrelevant, no explicit answer

Custom CEP implementation answers in comparison table were marked as irrelevant because some of them directly depend on the library chosen for implementation and other features such as GUI must be developed during the implementation process and are not provided by the CEP library itself.

[1] Provided by Ververica Platform, free to use but not open source.

[2] Only via programming, no DSL support.

[3] Micro-batches, no true streaming support.

[4] Provided by Logstash.

[5] Partial support.

The only CEP engine to pass the minimum set of requirements proposed in section 4.2.15 was Siddhi. At first Flink with the combination of Ververica Platform seemed to have similar features, but Ververica offers its Community Edition only with a proprietary community edition license, which is not an open-source license. Software itself is provided for free, but no source code is available. This conflicts with the initial goals of this research. Software provided by Ververica itself is very well equipped with useful features and is free to use, but it is advised to thoroughly read through the license before deciding to use it.

This leaves Siddhi CEP engine and WSO2 Streaming Integrator bundle as the only truly open-source CEP solution that passes all the minimum requirements set for an easy-to-use CEP solution. But as Ververica Platform is still free to use then both solutions were compared in more detail.

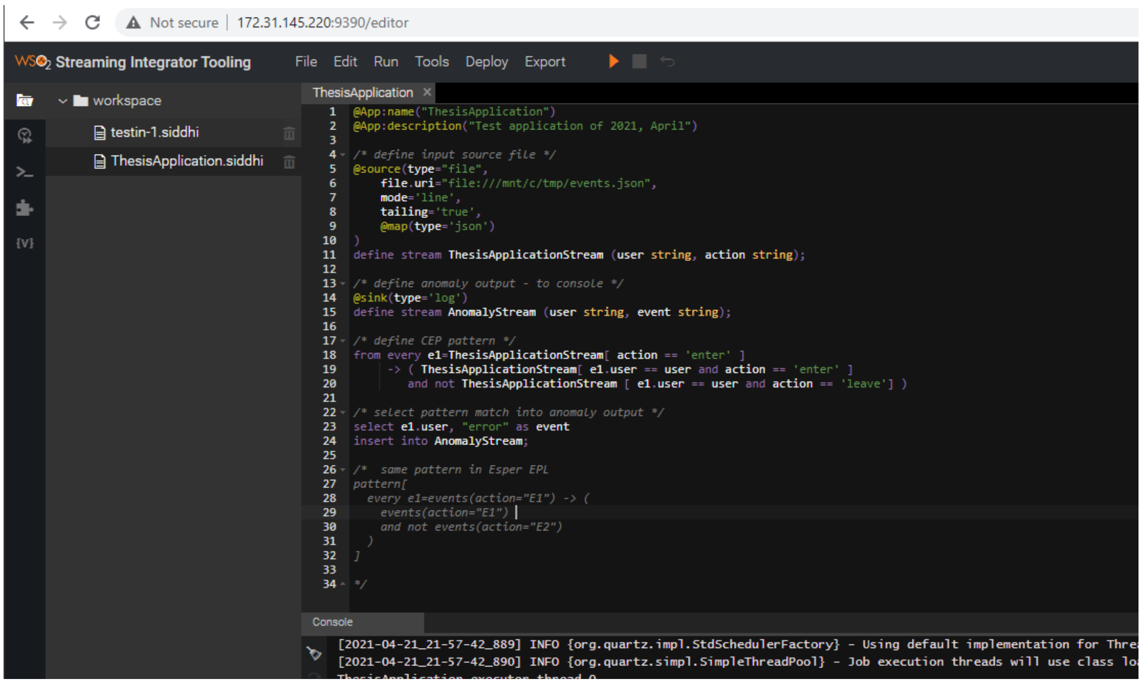
5.3 Proof of concept deployment

Initial installment of Siddhi is straightforward on both Windows and Unix (Linux and Mac are supported) based systems. By following the quick start guide Siddhi engine with additional web-based editor toolset can be set up in minutes just by downloading and executing one command to start the server. Although documentation about additional tools is somewhat confusing due to recent product name changes, they are no harder to set up if the right product package has been found. All that is needed (except Java runtime) to try out these solutions are pre-packaged and do not need any additional infrastructure.

Installation of Ververica Platform is not that straightforward. Although quick start guide for initial setup in the documentation is clearly written there are additional infrastructure needs. There are installation guides for these additional components, but they do need to install Kubernetes cluster, Helm package manager and MinIO object storage server. After two installation attempts on a VMWare ESXi Hypervisor, Ververica Platform was up and running but it took multiple hours and not all the features did work and failed with Kubernetes error messages. GUI with Flink SQL editor was working but no Flink SQL application could be submitted to processing. These errors could be solved by experienced systems administrator and additional time, but the aim of this thesis is focused on the ease of use of the system. Due to this, Ververica Platform cannot be considered as a very simple system to quickly test.

5.4 Graphical user interfaces

For a simple GUI PoC, goal was to run the scenario of physical access control system that has been used throughout this paper. Goal of the application is to detect a user entering a secure area twice without leaving meanwhile (Figure 11).



Not secure | 172.31.145.220:9390/editor
WSO2 Streaming Integrator Tooling
File Edit Run Tools Deploy Export
workspace
testin-1.siddhi
ThesisApplication.siddhi
ThesisApplication
Console

```
@App:name("ThesisApplication")
@App:description("Test application of 2021, April")

/* define input source file */
@source(type="file",
    file.uri="file:///mnt/c/tmp/events.json",
    mode='line',
    tailing='true',
    @map(type='json')
)
define stream ThesisApplicationStream (user string, action string);

/* define anomaly output - to console */
@sink(type='log')
define stream AnomalyStream (user string, event string);

/* define CEP pattern */
from every e1=ThesisApplicationStream[ action == 'enter' ]
     -> ( ThesisApplicationStream[ e1.user == user and action == 'enter' ]
        and not ThesisApplicationStream [ e1.user == user and action == 'leave'] )

/* select pattern match into anomaly output */
select e1.user, "error" as event
insert into AnomalyStream;

/*  same pattern in Esper EPL
pattern[
  every e1=events(action="E1") -> (
    events(action="E1")
    and not events(action="E2")
  )
]

*/
```

```
[2021-04-21_21-57-42_889] INFO {org.quartz.impl.StdSchedulerFactory} - Using default implementation for Thre
[2021-04-21_21-57-42_890] INFO {org.quartz.simpl.SimpleThreadPool} - Job execution threads will use class lo
```

Figure 11 - Siddhi demo application in web-based editor

As Siddhi's web-based editor provides syntax highlighting and contextual help, simple applications or minor changes to sample applications can be done by an inexperienced user. As Siddhi's pattern matching syntax is very similar to Esper, with minor changes it was possible to use Esper EPL example described in 4.2.6. The very same text file that defines this Siddhi application can be used to run in this web-based editor, started on a local machine using command line utility or deployed it to a separate production cluster using this same web-based editor interface. Source of the events for the application can simply be changed to Kafka or any other supported external system by changing the parameters in the same text file. In addition to textual view, graphical design view is also provided (Figure 12).

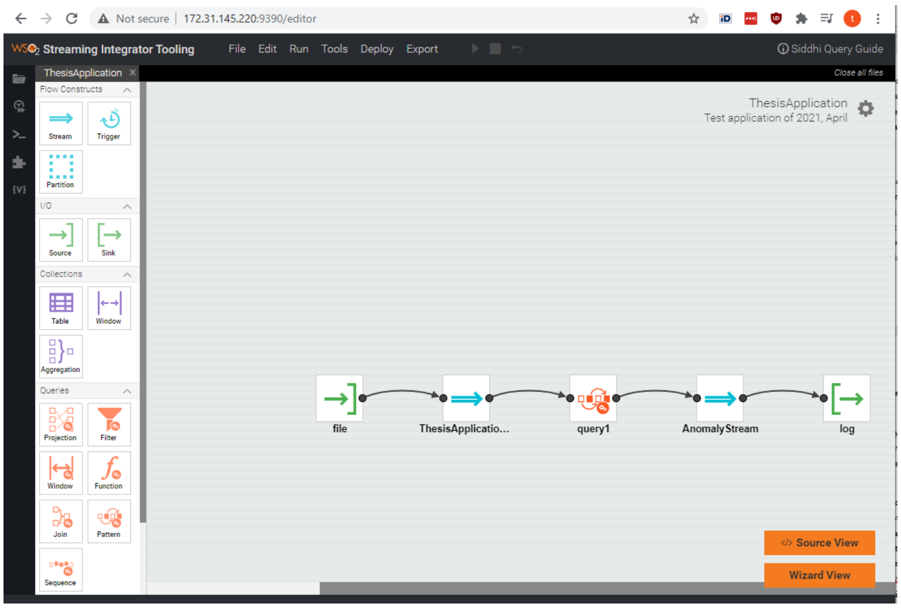
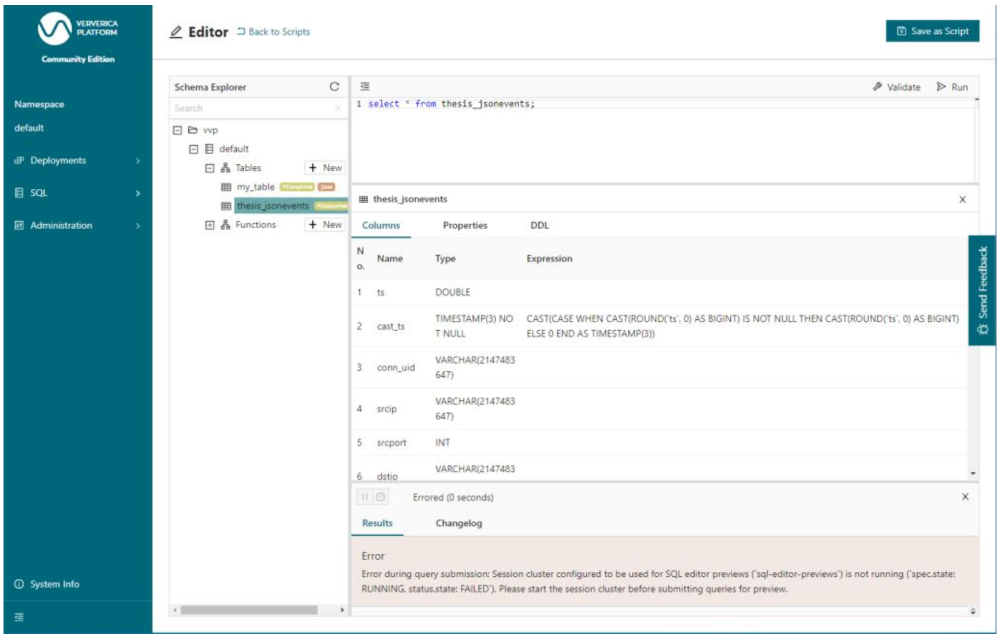


Figure 12 - Siddhi demo application design view

Considering ease of use, this is a huge step forward compared to writing Java code, building this code with Maven into a jar file and deploying it to a CEP cluster running on some Linux distribution.

Ververica Platform interface looks similarly featured as it provides syntax-highlighted SQL editor and syntax validation but lacks the graphical view (Figure 13). In Ververica Platform (and in Flink general), data definition language (DDL) scripts defining new streams and queries on them must be executed separately. This means that it is not possible to combine one logical set of SQL queries into one text file like in Siddhi. All the stream and view definitions must be executed separately and then queries must be executed and submitted to the server one by one, clearing the editor in the mean time. For testing and developing straming CEP applications it is much more inconvenient than the Siddhi’s approach.



```
select * from thesis_jsonevents;
```

thesis_jsonevents

Columns | Properties | DDL

No.	Name	Type	Expression
1	ts	DOUBLE	
2	cast_ts	TIMESTAMP(3) NOT NULL	CAST(CASE WHEN CAST(ROUND(`ts`, 0) AS BIGINT) IS NOT NULL THEN CAST(ROUND(`ts`, 0) AS BIGINT) ELSE 0 END AS TIMESTAMP(3))
3	conn_uid	VARCHAR(2147483647)	
4	srcip	VARCHAR(2147483647)	
5	srcport	INT	
6	dstip	VARCHAR(2147483	

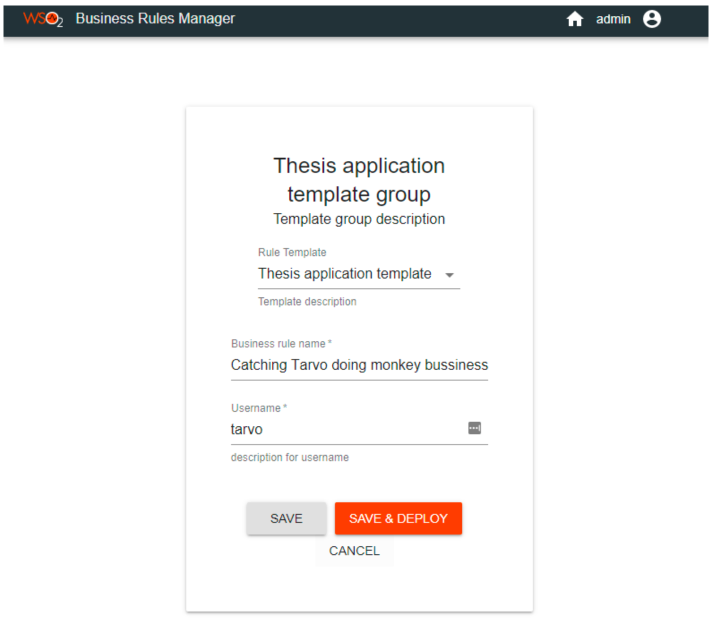
Errored (0 seconds)

Results | Changelog

```
Error
Error during query submission: Session cluster configured to be used for SQL editor previews ('sql-editor-previews') is not running ('spec.state: RUNNING, status.state: FAILED'). Please start the session cluster before submitting queries for preview.
```

Figure 13 - Ververica Platform web-based SQL editor

Ververica Platform does not provide additional features for its Community Edition. But WSO2 SI goes a step further and lets users define templated applications, where specific values within the application can be selected and submitted by using a web form generated by WSO2 Business Rules Manager. Altering one line in the application code shown before and turning it to a WSO2 Business Rules Manager template yields a web form (Figure 14) that can be used to deploy new CEP application that captures only the user defined in the form.



Thesis application template group
Template group description

Rule Template
Thesis application template
Template description

Business rule name *
Catching Tarvo doing monkey bussiness

Username *
tarvo
description for username

Figure 14 - Siddhi demo application Business Rules Manager form

Behind this simple web form, that can be filled by anyone who can use a computer, can lie a complex CEP enabled application, searching temporal relations in events by joining multiple event streams with a throughput reaching up to thousands or even tens of thousands of events per second.

5.5 Throughput benchmark

Although high throughput of these solutions is not a quality of easy-to-use solution, it is vital to verify that these solutions are capable of processing enough events in a single node to at least facilitate initial PoC testing. No in-depth benchmark tests were conducted that would cover all the features as this is not the aim of this thesis. To measure event throughput as accurately as possible and to eliminate any deployment related performance deviations, throughput benchmark tests were conducted as a Java application by using Java libraries provided by the solutions. Because Siddhi platform has a requirement that version 1.8 JDK must be used, both solutions were tested on 1.8 JDK. To eliminate any network interference, tests were conducted locally, using local filesystem as input data sources. Publicly available Bro (now rebranded to Zeek) connection logs were used as source data [81]. For testing throughput, four simple use-cases were implemented on both solutions, by utilizing only the features of the DSL languages of the solutions. This means that defining event sources and parsing unstructured text was done using only the features of DSL.

1) Ingesting raw log lines

This test just reads all the lines from source into CEP engine, one line as one event.

2) Parsing raw log lines into structured stream

This test parses the Bro connection log lines into 19 different fields that can be used in CEP queries.

3) Ingesting JSON formatted data

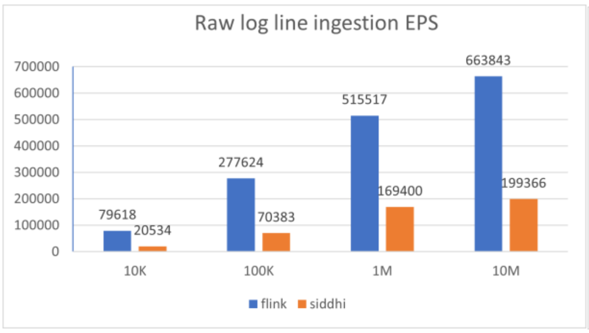
This test uses CEP engines native JSON parsing capability to send structured events to CEP engine. Bro connection log entries were parsed and converted to JSON for this test beforehand.

4) Executing an aggregation

Aggregation test searched for events with unique combination of source and destination IPs and selected only these IP pairs that had exchanged at least 25 000 bytes per second. This test used external time event as the time source through CEP engine windows and aggregations on those windows.

SQL code for all these test-cases is listed in appendixes 3-11.

To determine if the solutions had any warm-up period such that for processing small datasets the average events per second (EPS) was lower than for big datasets, first three tests were conducted on different sized datasets. The aim was to determine source dataset size for what throughput fluctuations would be as small as possible compared to smaller sized dataset. Differently sized datasets started from 10 000 lines and increased by multiplying the size by 10. All tests were executed at least 6 times from what average of 5 last runs was used to calculate throughput EPS. Files and their sizes used in the tests are listed in Appendix 1. Raw tests results are show in Appendix 2.



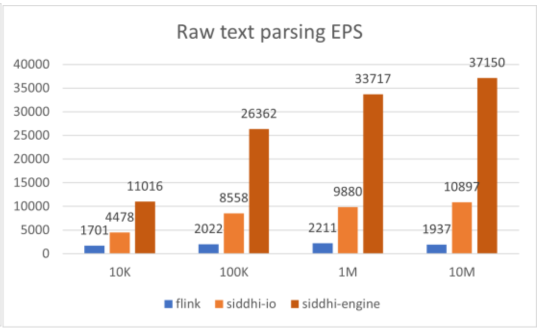


Figure 15 - Raw log line ingestion and parsing EPS

For raw log line ingestion, Flink outperformed Siddhi's engine over three times but in parsing these lines into structured fields Siddhi was considerably faster than Flink (Figure 15). As Siddhi provided two separate ways of parsing text, both were tested, and both were over 5 times faster than Flink. As it was suspected, EPS did increase with source dataset size but between one and ten million entries fluctuations were still over 10%.

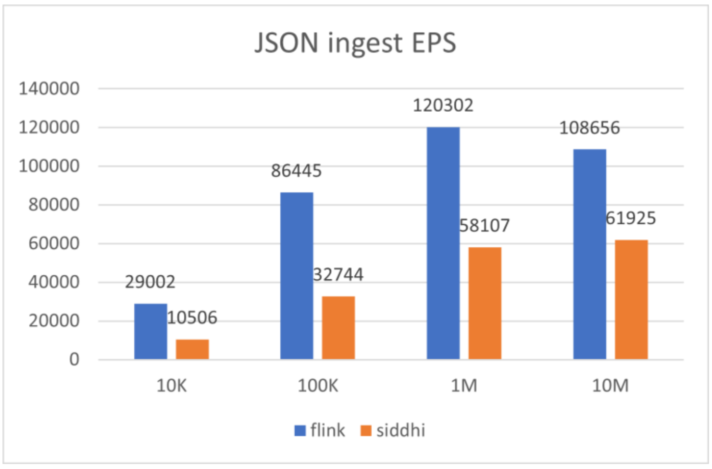


Figure 16 - JSON ingestion EPS

In JSON parsing test, Flink proved to be roughly 1,5 times faster than Siddhi (Figure 16). Fluctuation between 1 million and 10 million lines achieved the point were for Flink, throughput decreased and for Siddhi the increase fell to 6%. As other tests resulted in bigger fluctuations, JSON formatted data and source datasets of size 1 million and 10 million lines were chosen for fourth test.

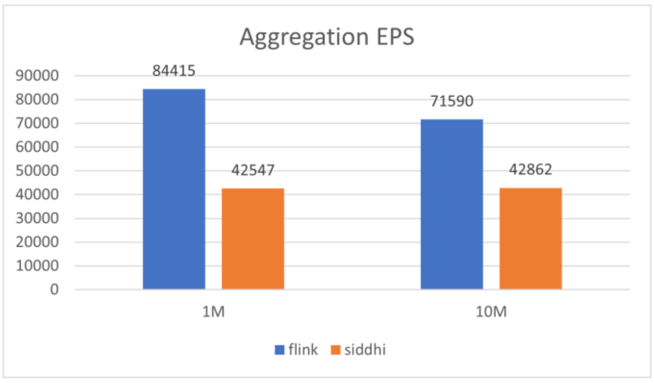


Figure 17 - Aggregation EPS

In aggregation query test, Flink processed events almost two times faster than Siddhi's engine but compared to just ingesting JSON data, changes in EPS differed no more than 4% (Figure 17). This significant difference just came from the ingestion and JSON parsing process and might differ based on the source of the data. This implies that aggregation features of CEP engine have similar throughput on these solutions.

In conclusion, throughput of these systems varies greatly even on a single node set-up depending on what they are doing with the events. But throughput of even tens of thousands of events per second is enough to execute a proof-of-concept test and decide if this system is suitable. As business needs change, horizontally scaled clusters can be deployed to provide more computational power.

6 Summary

There are as many unique solutions as there are unique problems to solve. But not all solutions are suitable for solving all problems. Keywords like big data, stream processing, event correlation, and even complex event processing are often used for marketing and to popularize open-source projects. But not all solutions behind those keywords are the same and easy to implement as they initially seem to be from the presented marketing.

As complex event processing and solutions encompassing these concepts have evolved over time, usability of these systems has increased. Systems that once belonged to the narrow area of academic research and systems managed by expert software engineers are now becoming available to all users and usable for people who are not experts in information technology and complex event processing details.

One contribution of this thesis is a set of questions that define prerequisites that must be evaluated before any CEP implementation project. The goal of these questions is to force engineers to be critical in their reasoning if they are thinking about implementing a CEP solution. Overview of typical use-cases where CEP systems are not applicable, and short introduction to possible alternative solutions was provided to supplement these questions.

The other contribution of this thesis is the analysis of existing research papers, available literature on the topic, and solutions themselves to find out the current state of open-source CEP systems. As more people can and will use these systems for their benefit, the ease of use of these systems is of critical importance. The focus of this research is to highlight CEP systems, that provide the most opportunities for simplifying testing, implementation, and most importantly the usage of the system.

Implementing a CEP system can become a complex and resource consuming endeavor and in the world of open-source community-based development, things can change unexpectedly without any certain control over these changes. But for those who have thoroughly weighed their requirements there are projects that have solid backing from the user community and have lower entrance level than a requirement for expert software engineers.

This research found that currently there is at least one open-source CEP engine, WSO2 Siddhi, that can be used from web-based GUI to develop and manage lifecycle of CEP applications. In addition to that, WSO2, creator of Siddhi, provides additional web-based companion applications providing easy to use features to create web-based forms for creating instances of CEP application based on templated approach. Ververica Platform, based on Flink CEP engine, does provide similar features and is free software. But Ververica Platform is provided as a prepackaged solution without the access to source-code and with a proprietary license that limits the usage of the solution. Although these solutions were the only one that conformed to requirements set by this thesis, there are other open-source projects that are already developing similar features.

The future work can be done to analyze and test possible use-cases for CEP usage in the field of information security. Another important aspect to consider in future analysis is additional in-depth throughput testing, as in information security field event sources such as firewalls can produce high volume of events.

References

[1] “120 Seconds - Every Second Saves Lives,” Intrusion Technologies, 06 2018. [Online]. Available: https://medium.com/@intrusiontechnologies/120-seconds-every-second-saves-lives-2534dc2ab8bf. [Accessed 04 2021].

[2] Martin Hirzel, “Partition and Compose: Parallel Complex Event Processing,” 2012. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/2335484.2335506. [Accessed 04 2021].

[3] Weilong Ren, Xiang Lian and Kambiz Ghazinour, “Efficient Join Processing Over Incomplete Data Streams,” 11 2019. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3357384.3357863. [Accessed 04 2021].

[4] D. Luckham, The Power of Events, 2002.

[5] A. Lawrence, “Man of Events,” *Information Age,* no. August, 2004.

[6] D. Luckham, “Real Time Intelligence & Complex Event Processing,” [Online]. Available: https://complexevents.com/.

[7] S. University, “Biography of David Luckham,” [Online]. Available: https://profiles.stanford.edu/david-luckham.

[8] D. Luckham, Event Processing for Business: Organizing the Real-Time Enterprise, 2011.

[9] Gianpaolo Cugola and Alessandro Margara, “Processing Flows of Information: From Data Stream to Complex Event Processing,” 06 2012. [Online]. Available: https://www.researchgate.net/publication/258243344_Processing_Flows_of_Information_From_Data_Stream_to_Complex_Event_Processing. [Accessed 04 2021].

[10] Henriette Röger and Ruben Mayer, “A Comprehensive Survey on Parallelization and Elasticity in Stream Processing,” 04 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3303849. [Accessed 04 2021].

[11] Miyuru Dayarathna and Srinath Perera, “Recent Advancements in Event Processing,” 02 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3170432. [Accessed 04 2021].

[12] Piyush Yadav and Edward Curry, “VidCEP: Complex Event Processing Framework to Detect Spatiotemporal Patterns in Video Streams,” 2019. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/2007/2007.07817.pdf.

[13] Sachini Jayasekara, Srinath Perera, Miyuru Dayarathna and Sriskandarajah Suhothayan, “Continuous analytics on geospatial data streams with WSO2 complex event processor,” June 2015. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/2675743.2772585.

[14] Aug 2016. [Online]. Available: https://debs.org/grand-challenges/2015/.

[15] Di Wang, Elke A. Rundensteiner, Han Wang and Richard T. Ellison III, “Active Complex Event Processing: Applications in real-time health care,” September 2010. [Online]. Available: https://dl.acm.org/doi/pdf/10.14778/1920841.1921034.

[16] Juan Boubeta-Puig, Inmaculada Medina-Bulo, Guadalupe Ortiz and Germán Fuentes-Landi, “Complex Event Processing Applied to Early Maritime Threat

Detection," September 2012. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/2377836.2377838.

[17] David Luckham, ""Complex" or "Simple" Event Processing," January 2005. [Online]. Available: https://complexevents.com/2005/01/31/complex-or-simple-event-processing/.

[18] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight and Kenneth Knowles, "One SQL to Rule Them All: One SQL to Rule Them All Management of Streams and Tables," 2019.

[19] R. Vaarandi, "Simple Event Correlator," [Online]. Available: https://simple-evcorr.github.io/.

[20] OSSEC. [Online]. Available: https://www.ossec.net/.

[21] Del Nagy, Aree M. Yassin and Anol Bhattacherjee, "Organizational Adoption of Open Source Software: Barriers and Remedies," no. https://dl.acm.org/doi/10.1145/1666420.1666457, 2010.

[22] Jordan Hubbard, "Open Source to the Core," May 2004. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/1005062.1005064.

[23] Ehsan Noroozi and Habib Seifzadeh, "Proposing novel measures to alleviate the risks of migration to open source software," 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3177457.3177478.

[24] "Apache Metron," [Online]. Available: https://metron.apache.org/.

[25] Roy Schulte, "When do you need an Event Stream Processing platform?," [Online]. Available: https://complexevents.com/when-do-you-need-an-event-stream-processing-platform/.

[26] Michael Stonebraker, Uğur Çetintemel and Stan Zdonik, "The 8 requirements of real-time stream processing," 12 2005. [Online]. Available: https://cs.brown.edu/~ugur/8rulesSigRec.pdf. [Accessed 04 2021].

[27] "Elastic," [Online]. Available: https://www.elastic.co/elastic-stack.

[28] "Kafka," [Online]. Available: https://kafka.apache.org/.

[29] "New – Open Distro for Elasticsearch," Amazon, 2019. [Online]. Available: https://aws.amazon.com/blogs/aws/new-open-distro-for-elasticsearch/. [Accessed 04 2021].

[30] "Graylog," Graylog, Inc, [Online]. Available: https://www.graylog.org/. [Accessed 04 2021].

[31] "Snort," Cisco, [Online]. Available: https://www.snort.org/. [Accessed 04 2021].

[32] "Suricata," [Online]. Available: https://suricata-ids.org/. [Accessed 04 2021].

[33] "Wazuh," Wazuh Inc, [Online]. Available: https://wazuh.com/. [Accessed 04 2021].

[34] "ModSecurity," Trustwave, [Online]. Available: https://github.com/SpiderLabs/ModSecurity. [Accessed 04 2021].

[35] "Naxsi," [Online]. Available: https://github.com/nbs-system/naxsi. [Accessed 04 2021].

[36] "WebKnight," Aqtronix, [Online]. Available: https://www.aqtronix.com/?PageID=99. [Accessed 04 2021].

[37] Novatec, "OpenAPM," [Online]. Available: https://openapm.io/. [Accessed 04 2021].

[38] Frank Olken, Monica Palmirani and Davide Sottara, “Rule-Based Modeling and Computing on the Semantic Web,” 2011. [Online]. Available: https://link.springer.com/content/pdf/10.1007%2F978-3-642-24908-2.pdf. [Accessed 2021].

[39] “Fluentd,” [Online]. Available: https://www.fluentd.org/. [Accessed 04 2021].

[40] “Fluent Bit,” [Online]. Available: https://fluentbit.io/. [Accessed 04 2021].

[41] NXLog, “NXLog,” [Online]. Available: https://nxlog.co/. [Accessed 04 2021].

[42] “Rsyslog,” [Online]. Available: https://www.rsyslog.com/. [Accessed 04 2021].

[43] “Syslog-ng,” [Online]. Available: https://www.syslog-ng.com/. [Accessed 04 2021].

[44] Timber, Inc, “Vector,” [Online]. Available: https://vector.dev/. [Accessed 04 2021].

[45] Debezium, “Debezium,” [Online]. Available: https://debezium.io/. [Accessed 04 2021].

[46] A. Jeffs, “Benthos,” [Online]. Available: https://www.benthos.dev/. [Accessed 04 2021].

[47] VMware, Inc., “RabbitMQ,” [Online]. Available: https://www.rabbitmq.com/. [Accessed 04 2021].

[48] Vineet John and Xia Liu, “A Survey of Distributed Message Broker Queues,” 04 2017. [Online]. Available: https://arxiv.org/abs/1704.00411. [Accessed 04 2021].

[49] Guruduth Banavar, Tushar Chandra, Robert Strom and Daniel Sturman, “A Case for Message Oriented Middleware,” 1999. [Online]. Available: https://link.springer.com/content/pdf/10.1007%2F3-540-48169-9_1.pdf. [Accessed 04 2021].

[50] D. Luckham, “Causality in Event Stream Analytics,” 8 2020. [Online]. Available: https://complexevents.com/2020/08/27/causality-in-event-stream-analytics/. [Accessed 4 2021].

[51] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera and Vishaka Nanayakkara, “Siddhi: a second look at complex event processing architectures,” 11 2011. [Online]. Available: https://dl.acm.org/doi/10.1145/2110486.2110493.

[52] “What’s New in SQL:2016,” 2017. [Online]. Available: https://modern-sql.com/blog/2017-06/whats-new-in-sql-2016. [Accessed 04 2021].

[53] S. Perera, 05 2019. [Online]. Available: https://medium.com/stream-processing/stream-processing-101-from-sql-to-streaming-sql-44d299cf38aa. [Accessed 04 2021].

[54] Packt Expert Network, “Apache Apex in a Nutshell,” 02 2018. [Online]. Available: https://medium.com/@PacktExprtNtwrk/apache-apex-in-a-nutshell-c4d5be3d5d83. [Accessed 04 2021].

[55] “Apache Apex,” [Online]. Available: https://attic.apache.org/projects/apex.html. [Accessed 04 2021].

[56] The Apache Software Foundation, “Apache Beam,” [Online]. Available: https://beam.apache.org/. [Accessed 04 2021].

[57] C. Zhao, “Reading Apache Beam Programming Guide — 1. Overview,” [Online]. Available: https://medium.com/@chengzhizhao/reading-apache-beam-programming-guide-1-overview-3adde0898b02. [Accessed 04 2021].

[58] “Apache Beam v2.2.0 release notes,” 12 2017. [Online]. Available: https://issues.apache.org/jira/secure/ReleaseNote.jspa?projectId=12319527&version=12341044.

[59] M. Zeng, “Pattern Matching with Beam SQL,” 08 2020. [Online]. Available: https://beam.apache.org/blog/pattern-match-beam-sql/. [Accessed 04 2021].

[60] K. Tzoumas, “Why Apache Beam?,” 03 2016. [Online]. Available: https://www.ververica.com/blog/why-apache-beam. [Accessed 04 2021].

[61] “Apache Flink,” [Online]. Available: https://flink.apache.org/. [Accessed 04 2021].

[62] “Stratosphere,” [Online]. Available: http://stratosphere.eu/. [Accessed 04 2021].

[63] The Apache Software Foundation, “Apache Bahir,” [Online]. Available: https://bahir.apache.org/. [Accessed 04 2021].

[64] Ververica, “Community Packages for Apache Flink,” [Online]. Available: https://flink-packages.org/. [Accessed 04 2021].

[65] Ververica, “Ververica,” [Online]. Available: https://www.ververica.com/. [Accessed 04 2021].

[66] B. Bhäte, “Twitter Storm Vs Heron,” [Online]. Available: https://medium.com/@vagrantdev/twitter-storm-vs-heron-12774056f45b. [Accessed 04 2021].

[67] J. Venkatraman, “Samza 1.0: Stream Processing at Massive Scale,” 11 2018. [Online]. Available: https://engineering.linkedin.com/blog/2018/11/samza-1-0--stream-processing-at-massive-scale. [Accessed 04 2021].

[68] “Apache Spark,” [Online]. Available: https://spark.apache.org/history.html. [Accessed 04 2021].

[69] Databricks, “SparkPackages,” [Online]. Available: https://spark-packages.org/. [Accessed 04 2021].

[70] Cloudera, “Cloudera,” [Online]. Available: https://www.cloudera.com/. [Accessed 04 2021].

[71] “Faust,” [Online]. Available: https://github.com/robinhood/faust. [Accessed 04 2021].

[72] Elasticsearch , “EQL for the masses,” 11 2018. [Online]. Available: https://www.elastic.co/blog/eql-for-the-masses. [Accessed 04 2021].

[73] Elasticsearch, “Introducing Event Query Language,” 06 2019. [Online]. Available: https://www.elastic.co/blog/introducing-event-query-language. [Accessed 04 2021].

[74] “Esper,” EsperTech Inc, [Online]. Available: https://www.espertech.com/. [Accessed 04 2021].

[75] “ksqlDB,” [Online]. Available: https://ksqldb.io/. [Accessed 04 2021].

[76] “Introducing ksqlDB,” Confluent, Inc, [Online]. Available: https://www.confluent.io/blog/intro-to-ksqldb-sql-database-streaming/. [Accessed 04 2021].

[77] “Siddhi,” WSO2, [Online]. Available: https://siddhi.io/.

[78] M. Vivekanandalingam, “Long Story Short: Siddhi - A view of a contributor,” 1 2020. [Online]. Available: https://medium.com/@mohan.vive/long-story-short-siddhi-a-view-of-a-contributor-91aa747f3546. [Accessed 04 2021].

[79] R. Vaarandi, “Simple Event Correlator,” [Online]. Available: https://simple-evcorr.github.io/. [Accessed 04 2021].

[80] T. Arikas, “Comet,” 2021. [Online]. Available: https://github.com/arikastarvo/comet.

[81] “SecRepo.com - Samples of Security Related Data,” [Online]. Available: https://www.secrepo.com/maccdc2012/conn.log.gz. [Accessed 04 2021].

[82] L. S. Sterling, The Art of Agent-Oriented Modeling, London: The MIT Press, 2009.

[83] NXLog, “Event Correlator (pm_evcorr) | Log Collection Solutions,” [Online]. Available: https://nxlog.co/documentation/nxlog-user-guide/pm_evcorr.html.

[84] D. Luckham, “Introduction To Real-Time Intelligence,” [Online]. Available: https://complexevents.com/2013/09/17/understanding-real-time-intelligence/.

[85] David Luckham, “What’s the Difference Between ESP and CEP?,” June 2020. [Online]. Available: https://complexevents.com/2020/06/15/whats-the-difference-between-esp-and-cep-2/.

[86] “Treasure Data,” [Online]. Available: https://www.treasuredata.com/. [Accessed 04 2021].

[87] “Flume,” [Online]. Available: https://flume.apache.org/. [Accessed 04 2021].

[88] Elasticsearch , “Elastic Beats,” [Online]. Available: https://www.elastic.co/beats/. [Accessed 04 2021].

[89] Elasticsearch, “Logstash,” [Online]. Available: https://www.elastic.co/logstash. [Accessed 04 2021].

[90] Logalyze, “Logalyze,” [Online]. Available: https://www.logalyze.com/. [Accessed 04 2021].

[91] “rsyslog,” [Online]. Available: https://www.rsyslog.com/. [Accessed 04 2021].

[92] Srinath Perera, “A Gentle Introduction to Stream Processing,” 04 2018. [Online]. Available: https://medium.com/stream-processing/what-is-stream-processing-1eadfca11b97. [Accessed 04 2021].

Appendix 1 – Performance benchmark test files

filename	**format**	**size**	**lines**
conn-10K.log	Bro log format	1.3M	10 000
conn-100K.log	Bro log format	12M	100 000
conn-1M.log	Bro log format	117M	1 000 000
conn-10M.log	Bro log format	1.2G	10 000 000
conn-10K.json	json	3.8M	10 000
conn-100K.json	json	38M	100 000
conn-1M.json	json	370M	1 000 000
conn-10M.json	json	3.7G	10 000 000

Appendix 2 – Benchmark measurements results

Test	Flink raw ingestion test					Flink parsing test					Flink JSON test			
Event count	**10000**	**100000**	**1000000**	**10000000**		**10000**	**100000**	**1000000**	**10000000**		**10000**	**100000**	**1000000**	**10000000**
AVG	**125.6**	**360.2**	**1939.8**	**15063.8**		**5877.8**	**49455.6**	**452196.4**	**5164000**		**344.8**	**1156.8**	**8312.4**	**92033.4**
EPS	**79617.8**	**277623.5**	**515517.1**	**663843.1**		**1701.3**	**2022**	**2211.4**	**1936.5**		**29002.3**	**86445.4**	**120302.2**	**108656.2**
Measurement values in milliseconds	122	383	1823	14391		6233	50019	463743	5149587		307	1223	8334	92986
	127	437	2306	15367		5872	49564	436968	5155043		419	1094	8338	91560
	131	369	1930	15119		5667	50212	468929	5202447		344	1172	8193	93232
	129	301	1828	15260		5724	49384	454502	5175143		377	1258	8152	90408
	119	311	1812	15182		5893	48099	436840	5137781		277	1037	8545	91981

Test	Flink aggregation test	
Event count	**1000000**	**10000000**
AVG	**11846.2**	**139683.6**
EPS	**84415.3**	**71590.4**
Measurement values in milliseconds	12055	138708
	12464	139554
	12071	139894
	11183	140489
	11458	139773

Test	Siddhi raw ingestion test					Sisddhi IO module parsing test					Siddhi CEP engine parsing test			
Event count	**10000**	**100000**	**1000000**	**10000000**		**10000**	**100000**	**1000000**	**10000000**		**10000**	**100000**	**1000000**	**10000000**
AVG	**487**	**1420.8**	**5903.2**	**50159**		**2233.4**	**11684.6**	**101210.2**	**917678.2**		**907.8**	**3793.4**	**29658.4**	**269177.8**
EPS	**20533.9**	**70382.9**	**169399.6**	**199366**		**4477.5**	**8558.3**	**9880.4**	**10897.1**		**11015.6**	**26361.6**	**33717.3**	**37150.2**
Measurement values in milliseconds	529	1328	6476	50936		2177	11885	99266	920188		951	3800	29791	269230
	445	1391	6011	49870		2157	12013	98200	922464		909	3751	29917	270130
	539	1340	5951	49897		2154	11683	106342	916818		924	3712	29333	269873
	505	1540	6053	50120		2264	11356	99841	919404		864	3894	29824	268502
	417	1505	5025	49972		2415	11486	102402	909517		891	3810	29427	268154

Test	Siddhi JSON test					Siddhi aggregation test	
Event count	**10000**	**100000**	**1000000**	**10000000**		**1000000**	**10000000**
AVG	**951.8**	**3054**	**17209.6**	**161486.2**		**23503.2**	**233309.2**
EPS	**10506.4**	**32743.9**	**58107.1**	**61924.8**		**42547.4**	**42861.6**
Measurement values in milliseconds	944	3182	17184	161938		23424	238168
	910	2818	17359	166142		23527	233317
	935	2936	16681	159047		23191	231404
	945	3160	17314	162467		23529	231540
	1025	3174	17510	157837		23845	232117

Appendix 3 – Flink raw intestion test code

```
# DDL statement
CREATE TABLE rawevents (
	log string
) WITH ('connector' = 'filesystem', 'path' = 'conn-10M.log', 'format' = 
'raw')

# Query statement
SELECT log FROM rawevents
```

Appendix 4 – Flink parsing test code

```
# DDL statement
CREATE TABLE rawevents (
      log string
) WITH ('connector' = 'filesystem', 'path' = 'conn-10M.log', 'format' =
'raw')

# Query statement
SELECT
      CAST(REGEXP_EXTRACT(log, '(.+?)\t', 1) as DOUBLE) as ts,
      REGEXP_EXTRACT(log, '(.+?\t){1}(.+?)\t', 2) as conn_uid,
      REGEXP_EXTRACT(log, '(.+?\t){2}(.+?)\t', 2) as srcip,
      CAST(REGEXP_EXTRACT(log, '(.+?\t){3}(.+?)\t', 2) as INT) as srcport,
      REGEXP_EXTRACT(log, '(.+?\t){4}(.*?)\t', 2) as dstip,
      CAST(REGEXP_EXTRACT(log, '(.+?\t){5}(.+?)\t', 2) as INT) as dstport,
      REGEXP_EXTRACT(log, '(.+?\t){6}(.+?)\t', 2) as proto,
      REGEXP_EXTRACT(log, '(.+?\t){7}(.+?)\t', 2) as service,
      REGEXP_EXTRACT(log, '(.+?\t){8}(.+?)\t', 2) as duration,
      REGEXP_EXTRACT(log, '(.+?\t){9}(.+?)\t', 2) as orig_bytes,
      REGEXP_EXTRACT(log, '(.+?\t){10}(.+?)\t', 2) as resp_bytes,
      REGEXP_EXTRACT(log, '(.+?\t){11}(.+?)\t', 2) as conn_state,
      REGEXP_EXTRACT(log, '(.+?\t){12}(.+?)\t', 2) as local_orig,
      REGEXP_EXTRACT(log, '(.+?\t){13(.+?)\t', 2) as missed_bytes,
      REGEXP_EXTRACT(log, '(.+?\t){14}(.+?)\t', 2) as history,
      REGEXP_EXTRACT(log, '(.+?\t){15}(.+?)\t', 2) as orig_pkts,
      REGEXP_EXTRACT(log, '(.+?\t){16}(.+?)\t', 2) as orig_ip_bytes,
      REGEXP_EXTRACT(log, '(.+?\t){17}(.+?)\t', 2) as resp_pkts,
      REGEXP_EXTRACT(log, '(.+?\t){18}(.+?)', 2) as resp_ip_bytes
FROM rawevents
```

Appendix 5 – Flink JSON test code

```
# DDL statement
CREATE TABLE jsonevents (
	ts double,
	conn_uid string,
	srcip string,
	srcport int,
	dstip string,
	dstport int,
	proto string,
	service string,
	duration string,
	orig_bytes string,
	resp_bytes string,
	conn_state string,
	local_orig string,
	missed_bytes string,
	history string,
	orig_pkts string,
	orig_ip_bytes string,
	resp_pkts string,
	resp_ip_bytes string
) WITH ('connector' = 'filesystem', 'path' = 'conn-10M.json', 'format' =
'json')

# Query statement
SELECT * FROM jsonevents
```

Appendix 6 – Flink aggregation test code

```
# DDL statement
CREATE TABLE jsonevents (
	ts double,
	`cast_ts` as cast(coalesce(cast(round(ts, 0) as bigint), 0) as
TIMESTAMP(3)),
	conn_uid string,
	srcip string,
	srcport int,
	dstip string,
	dstport int,
	proto string,
	service string,
	duration string,
	orig_bytes string,
	`cast_orig_bytes` as IF(IS_DECIMAL(orig_bytes), cast(orig_bytes as
int),0),
	resp_bytes string,
	`cast_resp_bytes` as IF(IS_DECIMAL(resp_bytes), cast(resp_bytes as
int),0),
	conn_state string,
	local_orig string,
	missed_bytes string,
	history string,
	orig_pkts string,
	orig_ip_bytes string,
	resp_pkts string,
	resp_ip_bytes string,
	WATERMARK FOR cast_ts AS cast_ts
) WITH ('connector' = 'filesystem', 'path' = 'conn-10M.json', 'format' =
'json')

# Query statement
SELECT
	TUMBLE_START(cast_ts, INTERVAL '1' SECOND) as cast_ts,
	srcip,
	dstip,
	(srcip || ' -> ' || dstip) as srcdstip,
	count(*) as cnt,
	SUM(cast_orig_bytes) as orig_bytes,
	SUM(cast_resp_bytes) as orig_bytes
FROM jsonevents
GROUP BY TUMBLE(cast_ts, INTERVAL '1' SECOND),srcip,dstip
HAVING SUM(cast_orig_bytes) > 25000 OR SUM(cast_resp_bytes) > 25000
```

Appendix 7 – Siddhi raw ingestion test code

```
@App:name("thesis-raw-read")

/* input definition */
@Source(type="file",
    file.uri="file://tmp/conn-siddhi.log",
    mode='line',
    tailing='false',
    @Map(type='text', fail.on.missing.attribute = 'false')
)
define stream Events (data string);

/* query */
from Events;
```

Appendix 8 – Siddhi parsing test code (IO module)

```
@App:name("thesis-parse-io")

/* input definition */
@Source(type="file",
    file.uri="file://tmp/conn-siddhi.log",
    mode='line',
    tailing='false',
    @Map(type='text', fail.on.missing.attribute = 'false',
regex.A='(.+?)\t(.+?)\t(.+?)\t(.+?)\t(.+?)\t(.+?)\t(.+?)\t(.+?)\t(.+?)\t(.+?)
\t(.+?)\t(.+?)\t(.+?)\t(.+?)\t(.+?)\t(.+?)\t(.+?)\t(.+?)\t(.+?)',
       @attributes(
              ts = 'A[1]',
              conn_uid = 'A[2]',
              srcip = 'A[3]',
              srcport = 'A[4]',
              dstip = 'A[5]',
              dstport = 'A[6]',
              proto = 'A[7]',
              service = 'A[8]',
              duration = 'A[9]',
              orig_bytes = 'A[10]',
              resp_bytes = 'A[11]',
              conn_state = 'A[12]',
              local_orig = 'A[13]',
              missed_bytes = 'A[14]',
              history = 'A[15]',
              orig_pkts = 'A[16]',
              orig_ip_bytes = 'A[17]',
              resp_pkts= 'A[18]',
              resp_ip_bytes = 'A[19]'
       )
    )
)
define stream ConnEvents (
       ts double, conn_uid string, srcip string, srcport int, dstip string,
dstport int, proto string, service string, duration string, orig_bytes
string, resp_bytes string, conn_state string, local_orig string, missed_bytes
string, history string, orig_pkts string, orig_ip_bytes string, resp_pkts
string, resp_ip_bytes string
);

/* query */
from ConnEvents;
```

Appendix 9 – Siddhi parsing test code (CEP engine)

```
@App:name("thesis-parse-cep")

/* input definition */
@source(type="file",
    file.uri="file://tmp/conn-10.log",
    mode='line',
    tailing='false',
    @map(type='text', fail.on.missing.attribute = 'true', regex.A='(.*)',
@attributes(data = 'A[1]'))
)
define stream RawEvents (data string);

/* query */
from RawEvents
select
      cast(regex:group('(.+?)\t', data, 1), 'double') as ts,
      regex:group('(.+?\t){1}(.+?)\t', data, 2) as conn_uid,
      regex:group('(.+?\t){2}(.+?)\t', data, 2) as srcip,
      cast(regex:group('(.+?\t){3}(.+?)\t', data, 2), 'int') as srcport,
      regex:group('(.+?\t){4}(.+?)\t', data, 2) as dstip,
      cast(regex:group('(.+?\t){5}(.+?)\t', data, 2), 'int') as dstport,
      regex:group('(.+?\t){6}(.+?)\t', data, 2) as proto,
      regex:group('(.+?\t){7}(.+?)\t', data, 2) as service,
      regex:group('(.+?\t){8}(.+?)\t', data, 2) as duration,
      regex:group('(.+?\t){9}(.+?)\t', data, 2) as orig_bytes,
      regex:group('(.+?\t){10}(.+?)\t', data, 2) as resp_bytes,
      regex:group('(.+?\t){11}(.+?)\t', data, 2) as conn_state,
      regex:group('(.+?\t){12}(.+?)\t', data, 2) as local_orig,
      regex:group('(.+?\t){13}(.+?)\t', data, 2) as missed_bytes,
      regex:group('(.+?\t){14}(.+?)\t', data, 2) as history,
      regex:group('(.+?\t){15}(.+?)\t', data, 2) as orig_pkts,
      regex:group('(.+?\t){16}(.+?)\t', data, 2) as orig_ip_bytes,
      regex:group('(.+?\t){17}(.+?)\t', data, 2) as resp_pkts,
      regex:group('(.+?\t){18}(.+?)', data, 2) as resp_ip_bytes;
```

Appendix 10 – Siddhi JSON test code

```
@App:name("thesis-json")

/* input definition
@Source(type="file",
    file.uri="file://tmp/conn-siddhi.json",
    mode='line',
    tailing='false',
    @Map(type='json', fail.on.missing.attribute = 'false')
)
define stream ConnEvents (
	ts string,
	conn_uid string,
	srcip string,
	srcport string,
	dstip string,
	dstport string,
	proto string,
	service string,
	duration string,
	orig_bytes string,
	resp_bytes string,
	conn_state string,
	local_orig string,
	missed_bytes string,
	history string,
	orig_pkts string,
	orig_ip_bytes string,
	resp_pkts string,
	resp_ip_bytes string,
	event_type string
);

/* query */
from ConnEvents;
```

Appendix 11 – Siddhi aggregation test code

```
@App:name("thesis-aggregation")

/* input definition and additional ddl statements */
@Source(type="file",
    file.uri="file://tmp/conn-siddhi.json",
    mode='line',
    tailing='false',
    @Map(type='json', fail.on.missing.attribute = 'false')
)
define stream ConnEvents (
	ts string, conn_uid string, srcip string, srcport string, dstip
string, dstport string, proto string, service string, duration string,
orig_bytes string, resp_bytes string, conn_state string, local_orig string,
missed_bytes string, history string, orig_pkts string, orig_ip_bytes string,
resp_pkts string, resp_ip_bytes string, event_type string
);

define stream ConnWindow(ts long, srcip string, dstip string, orig_bytes int,
resp_bytes int);

define stream ConnAggregation(ts long, srcip string, dstip string, srcdstip
string, count long, orig_bytes_sum long, resp_bytes_sum long);


/* queries */
from ConnEvents
select
	convert(convert(ts, 'double')*1000, 'long') as ts,
	srcip,
	dstip,
	convert(orig_bytes, 'int') as orig_bytes,
	convert(resp_bytes, 'int') as resp_bytes
insert into ConnWindow;

from ConnWindow#window.externalTimeBatch(ts, 1 second)
select
	min(ts) as ts,
	srcip,
	dstip,
	str:concat(srcip,' -> ', dstip) as srcdstip,
	count() as count,
	sum(orig_bytes) as orig_bytes_sum,
	sum(resp_bytes) as resp_bytes_sum
group by srcip,dstip
having orig_bytes_sum > 25000 or resp_bytes_sum > 25000
insert into ConnAggregation;
```