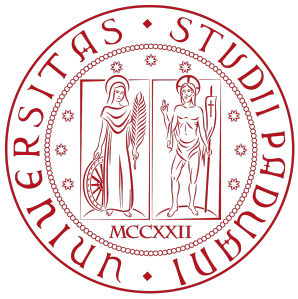
Distributed Complex Event Recognition

MASTER THESIS

in

DATA SCIENCE

by **Marco Baggio**



DEPARTMENT OF MATHEMATICS

UNIVERSITY OF PADUA

FEBRUARY 2021

Sergi Nadal
Postdoctoral researcher
Department of Service and Information System Engineering
Universitat Politècnica de Catalunya
Barcelona
External Examiner

Massimiliano De Leoni
Assistant Professor
Department of Mathematics
University of Padua
Padua
Internal Examiner

ACKNOWLEDGEMENT

This work is the result of a personal effort, made of research for motivation and getting acquainted with unfamiliar topics. I want to thank first and foremost my parents for pushing me down this path and supporting me in any possible way. I am grateful that I have only found professors with passion for their work, this helped immensely in making the subjects interesting and appealing. I want to thank my colleagues, they were always there in times of needs and always willing to help, each in his own way, they made this years unique. This has been the most enriching experience of my life and I am looking forward to see what opportunities it will open in my future.

Abstract

The goal of this thesis document can be divided in three main steps. First, summarize the current state of the art in complex event recognition (CER). Second, motivate the need for a new approach overcoming the issues posed by current solutions. Lastly propose our own solution for distributing CER systems and analyze the results against current solutions to evaluate possible gains in performances.

The ultimate goal when distributing CER is to augment the capacity of the system to deal with the ingestion of more events and/or patterns while maintaining a steady throughput, in other words to scale. We specifically focus on scale-out settings. In such cases, a cluster of interconnected machines (i.e., processing units) over the network distribute the workload so that each deals with only a fraction of it, thus overall a higher performance is achieved.

Keywords : CER, Distribution

TABLE OF CONTENTS

List of Figures

Chapter 1

Introduction

The Objective of the thesis is to design, implement and test a distributed solution for Complex Event Recognition (CER) problems in order to study the benefit of a distributed environment when facing an increasing arrival rate of events whilst maintaining a steady throughput. Having this goal in mind, we focused our efforts towards the analysis of some state-of-the-art solutions to understand what they were lacking of and where they could be adapted or integrated in order to achieve better performances. In addition to a thorough study of the workflow of a Complex Event Recognition problem, this thesis focuses on the creation of a new pipeline to takle the problem from a new perspective.

1.1 Context

Complex Event Recognition is the natural evolution of data stream processing where its goal is to detect situations of interest over data streams, commonly represented by the detection of patterns. The diversity of this topic can be captured best when looking at typical CER scenarios. An example is the use case of Network Intrusion Detection, where transfer patterns of credit-card users are analyzed and suspicious activities are detected by e.g. comparing transaction times and geographical locations. Further use cases can be found for IoT applications, healthcare, industrial control Systems or financial systems [1].

The key concept in a CER problem is the recognition on the fly of a requested group of events complying with a set of predefined rules among the events within a continuous stream of incoming information. This calls for a reliable infrastructure that is capa-

ble of enduring the influx of data regardless of its dimensions in order to avoid loosing information due to delays in the analysis caused by the lack of computational power.

If we begin analyzing a typical CER framework starting from the top level we are first introduced to two key components of a CER problem which can be identified as (1) the incoming data or stream and (2) the pattern we want to detect or, as we will call it from now on, the query. These are the defining elements of a CER problem and the points of interest of our research. All the work done to improve the framework for CER problems is a combination of digestion and manipulation of these two components in order to reach better performances.

1.1.1 The Input Stream

Regarding the input stream, we can define it as an endless sequence of information coming from possibly multiple sources at any given point in time. This information usually is represented by events. Each event represents a single piece of information at a precise moment in time from one of the selected sources. We can therefore define our event as such: (1) an atomic datum composed by an index, (2) an identifier of the type information stored, (3) the value of the measurement and (4) the time at which the measure was taken.

With these concepts at hand, the concept of a stream can be more precisely introduced: a stream is an ordered temporal sequence of single events with different identifiers, each determined by the origin where the data is coming from. The stream have no restriction on the type of data that can carry nor a limit on the number of event it is composed of. This brings us to the definition of ingestion rate, which is an essential information to have in mind when dealing with stream handling problems.

The ingestion rate in a stream processing applications is defined as the average number of events that are added to the stream at each time unit (e.g. second). The ingestion rate in other words the measure of the rate of incoming events, a fundamental measure used

to determine how much resources has to be dedicated in order to analyze the incoming stream without the creation of bottlenecks in the pipeline. In the case of a Complex Event Recognition problem the ingestion rate is very volatile since it is determined by the possibly changing number of sensors concurrently feeding information and the different rate at which different sensor may send the data.

1.1.2 The Query

The query determines what we are looking for in our stream of data. A query is in charge of recognizing a pattern of events in the input stream. A query is a collection of conditions that, if applied on a stream of events, can detect, if present, a subset of ordered events from said stream satisfying its conditions. This is called an output. The conditions a query can refer to are: (1) the number of event the query is made of, (2) the type of events, (3) the temporal order in which the types of events has to be encountered and (4) the range of values those events can assume. In the field of Complex Event Recognition a key concept is that query conditions can be applied not only on single events, but also on subsets of events that share the same type. For example, instead of having a condition on the value of a single event, we could have a condition on the mean of the values of a group of events of the same type in a fixed timelapse.

1.1.3 Running Example

Throughout the thesis, we will exemplify our techniques with a running example in order to create a common ground for future discussions about applied techniques and confrontation of different approaches.

Our running example refers to a website for online shopping. The shop owner knows that, under certain workload conditions, the server hosting his website are more likely to forcefully shutdown. Therefore, the owner of the website is constantly monitoring: the number of users concurrently on-line on the website at each moment of time, the temperature of the server facility and the stability of the power supply units. If the number of concurrent users and the temperature stay above a certain threshold, a surge

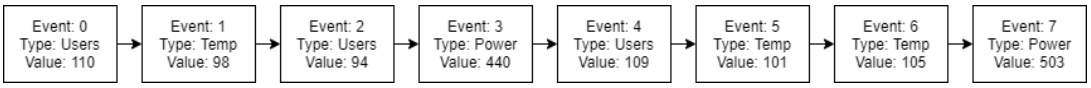


Figure 1.1: Stream Example

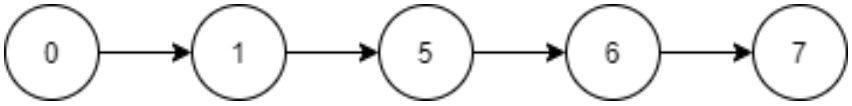


Figure 1.2: Complex Event

in power will cause the server to shutdown. In other words, the web server and the services beneath cannot deal with more than a certain workload. An example of the stream of data will look like the one in Figure 1.1

Let us suppose we want to capture a sequence of more than 90 concurrent users, a series of temperatures over 90 degrees Celsius and a surge of current over 500 W. Looking back at our example we can see a pattern satisfying our conditions represented in Figure 1.2.

It is worthwhile saying that there are more combinations of events that satisfy our query. Alternative combinations can be seen in the example in Figure 1.3

The presence of multiple available solutions is bound to the evaluation of second order predicates. In spite of being a small example, the fact that it yields already a large number of alternative solutions explains on why we propose to distribute the evaluation of second order predicates.

1.2 Motivation

As of today the majority of state of the art solutions run in a centralized fashion [2]. This is due to the fact that CER problems rely on the time structure of the stream to be consistent throughout the whole analysis process, therefore there was no incentive in added complexity derived by the addition of a system dedicated to maintain the stream synchronized throughout separate computational unit.

However this has proven to be inefficient in the ever growing field of Big Data which conversely requires a system capable of adapting the use of resources at its disposal to accommodate potentially huge streams of data, whilst maintaining a steady throughput.

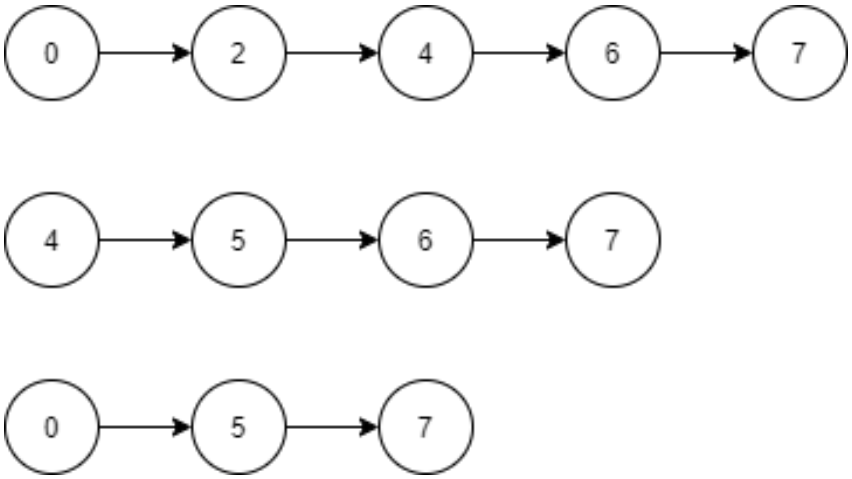


Figure 1.3: Alternative events combinations that create a Complex Event

A centralized system simply cannot be prepared to withstand a highly susceptible number of events while maintaining a reasonable cost of the infrastructure since either the computational power would be insufficient for the task at hand or, at the other side of the spectrum, the allocated resources will be mostly wasted waiting for a possible surge in data throughput. Furthermore, CER problems are inherently more and more complex as the input increases in size and/or variety. This is mainly due to the second-order nature of the query predicates and the consequent exponential growth of the problem of predicate interpretation.

This motivates the choice of distributing CER solutions. Distribution comes with a set of benefits that allow CER problems to adapt to the world of Big Data providing a system that is capable of adapting to an ever changing throughput of data in the stream whilst maintaining the allocated resources to an acceptable level at all times. These are the classic benefits that parallelism brings to the table and are better represented by two terms: *Parallelism* and *Elasticity*. The first refers to the ability of distributing the workload to different resources that can tackle the problem in parallel therefore increasing the efficiency of the system. The second is a direct consequence of the first in that when a problem can be distributed in parallel, we have power over the number of resources that work on the problem. At any given moment, based on the throughput of the data we can decide how many "workers" are needed. This prevents both bottlenecks due to an

elevated throughput with respect to the resources, and locked resources left idle waiting to be used by a process that doesn't need it.

The choice of distributing a Complex Event Recognition solutions however comes with its own set of challenges.

The first challenge we encounter is the ability to distribute the data and the query in the most optimal way possible across all processing units at our disposal. This is due to the fact that, as we said, the stream needs to maintain its time structure in order to yield useful information. Therefore the problem is not trivial since one cannot simply apply a distribution strategy to the single events in the stream since this expose each computational unit to only a fraction of the information with the consequence of loosing a great portion of information. It is not viable as solution to split the query into sub-queries to be analyzed by the various workers: this comes with its own set of challenges and problems, starting from the complexity of merging the sub-problems together once solved by the workers.

The second challenge, related to the *Elasticity* of a distributed system, refers to the well known requirement of all the distributed solutions to scale the resources allocated to the process dynamically to satisfy the request of computational power required at any time by the process whilst avoiding wasting resources when the flow is reduced and doesn't require the same amount of power used until then. While less complex as the previous challenge, it still poses a threat to the well working of a distributed CER solution in that the elasticity of a system is heavily linked to the strategy applied ahead of the distributed system.

Distributing CER solution poses a much harder challenge then distributing streaming in general due to the innate nature of the event stream and the composition of the query. All canonical distribution solutions, when applied to a Complex Event Recognition solution imply a great overhead in terms of network communication to maintain the synchronization between computational units.

1.3 Contributions

The main idea behind distributing a Complex Event Recognition problem is to minimize the overhead on one machine. This is the vision of this research project. The intention is that, while some operations need to remain centralized to maintain the integrity of the process, we aim to maximise the amount of operations that can be distributed over the nodes.

Our proposed framework aims at maintaining a clear separation between the pattern recognition and the evaluation of the restrictions over the events. While the pattern recognition has to be run in a centralized fashion, thus maintaining the temporal structure of the stream, the evaluation of the restrictions over a possible pattern satisfying the criteria of the query can be done in a distributed environment.

This structural separation of the operations allowed us to reach a middle ground where: (1) the centralized system is in charge of the lower complexity operations, such as for the pattern recognition, and (2) the different computational units of the distributed system are in charge of evaluating the second order predicates on a reduced number of outputs based on the decisions of the selected distribution strategy. This clear separation allowed us greater freedom in the choice of the distribution algorithm that could have been adapted to various possible combinations of patterns and groups of patterns. The intention is to achieve better performances with respect to different challenges that could have been presented to us.

In particular, our framework proposes a system capable of generating different versions of outputs not evaluated on their second order predicates in a centralized fashion and a selection of state of the art and novelty distribution algorithms to feed the outputs to the computational units of the distributed system in order to evaluate the second order predicates.

1.4 Structure of the Manuscript

The following chapters of the document will be structured as follows: Chapter 2 will give an insight on the background of the work that we will present in this thesis. It will set the knowledge foundations in order to comprehend what will be described in later chapters. Chapter 3 will introduce some state of the art solutions present on the market that tackle the same problem we intend to address. Chapter 4 will go into details about the ideas and the theory behind our work and will show in detail what are the reasoning behind our implementation choices. Chapter 5 shows the techniques and the means used to implement the ideas described in the previous chapter. Chapter 6 illustrates the results obtained by our implementation when tested out under different environmental settings. Finally Chapter 7 will briefly sum up our thoughts on the obtained results, the current state of our work and what are our future plans.

Chapter 2

Background

In the following section we will go through the core aspects of each component of our framework to get a better understanding of what have been used and how the whole structure works. For each module we will present the main concepts and give a formal definition for all the components that we will encounter and use throughout the whole thesis.

2.1 Complex Event Recognition

Complex event recognition (CER) is a natural evolution of data stream processing, where its goal is to detect situations of interest over data streams, commonly represented by the detection of patterns.

CER significantly differentiates itself from traditional streaming conceptualizations [3]. Classical stream querying lacks the ability to detect and infer information from a stream of different events leaving the analysis process to the client as a manual work on the results. On the contrary, CER languages allow querying for complex patterns that match incoming events on the basis of their content, sequencing and ordering relationships as well as other spatio-temporal constraints.

Numerous CER systems and languages have been proposed in the literature [2] [4]. Many CER systems provide users with a pattern language that is later compiled into some form of automaton [5]. The automaton model is generally used to provide the semantics

of the language and/or as an execution framework for pattern matching. Apart from automata, some CER systems employ tree-based models. Again, tree-based formalisms are used for both modeling and recognition, i.e., they may describe the complex event patterns to be recognized as well as the applied recognition algorithm.

2.2 Second Order Complex Event Logic

This thesis employs the second order complex event logic (SO-CEL) [5]. SO-CEL is a complex event recognition language based on second-order variables.

2.2.1 First Order Complex Event Logic

Since SO-CEL builds on the first order complex event logic (FO-CEL) introduced in [5], we start from this and later build upon. For the semantics of FO-CEL we first need to introduce the notion of match. A match M is defined as a non-empty and finite set of natural numbers. Note that a match plays the same roll as a complex event in SO-CEL and can be considered as a restricted version where only the support of the output is considered. We informally introduce FO-CEL through an example, for a more detailed explanation we address the reader to [5]:

$$[H\ as\ x; (T\ as\ y\ FILTER\ y.id = x.id)+; H\ as\ z]\ \ FILTER\ (x.hum < 30 \wedge z.hum > 60 \wedge x.id = z.id)$$

This query aims to detect a pattern of a change in temperature whenever there is an increase in humidity from below 30 to above 60. As we can see, inside the Kleene closure, y is always bound to the current event being inspected. The filter $y.id = x.id$ ensures that the inspected temperature events of type T are of the same location as the first humidity event x. Note that, in this case, the output is a match and includes in particular the positions of the inspected T events.

2.2.2 Second Order Variables

In order to understand the SO-CEL language, we first need to introduce the concept of Second Order Variable. A complex event is defined as a finite sequence of events that presents a relevant pattern for the user. By simply considering a complex event composed

of a sequence of events of increasing values followed by an event of a specific value we might want to be able to represent this two distinct groups of event with two different variables. Here is were second order variables come into play. Second order variables allow us to clearly address set of events on which we want to apply specific predicates (i.e. in this case the predicate for the second order variable is the increasing values of the events that it is composed of).

SO-CEL sets a clear collection of rules and syntax to represent a pattern that are written as such:

$$\phi := \ R \ \mid \phi \ IN \ A \mid \phi[A \ \rightarrow \ B] \mid \phi \ FILTER \ \alpha \mid \phi \ OR \ \phi \mid \phi \ ; \ \phi \mid \phi+$$

Where R ranges over the relation names in the database schema where each relation name is associated with a tuple of attributes, A and B range over labels in L which is a set of monadic second order variables containing all relation names and α is an atom over P, i.e. an expression of the form $P(A_1, \ldots, A_n)$ where P is a predicate of arity n and $A_1, \ldots, A_n \in L$.

We can now rewrite the FO-CEL example from above with SO-CEL:

$[H \ IN \ X; (T + INY); H \ IN \ Z] \ \ FILTER \ [X.hum < 30 \wedge Z.hum > 60 \wedge X.id = Y.id \wedge X.id = Z.id]$

With this set of rules we can write patterns like $\phi = (U+)[U \rightarrow us]; (T+)[T \rightarrow ts]; W[W \rightarrow w] \ FILTER \ (us.value \ \geq 100 \wedge ts.value \geq 90 \wedge w.value \geq 500)$ which is the representation in SO-CEL of the pattern proposed in our running example or even go further like $\phi = (U+)[U \rightarrow us]; (T+)[T \rightarrow ts]; W[W \rightarrow w] \ FILTER \ (us.value \ \geq 100 \wedge ts.value \ avg \ 90 \wedge w.value \geq 500)$ which is the same of our running example, but in this case we want to keep track of the average number of concurrent users across all the events that satisfy the pattern.

2.3 Unary Complex Event Automata

UCEA are a generation of match automata. The main difference is that match automata output matches, while UCEA output complex events. In particular UCEA mark events using SO variables [5].

Let R be a schema and U a set of unary predicates over R, we denote by U^+ the closure of U. A Unary Complex Event Automaton over R and U is a tuple $A = (Q, \nabla, I, F)$ where Q is a finite set of states, $\nabla \subseteq Q \times U^+ \times 2^L \times Q$ is a finite transition relation and $I, F \subseteq Q$ are the set of initial and final states, respectively.

UCEA are a generalization of the Match Automata with the only difference being that while the match automata outputs matches, UCEA outputs complex events.

The real strength of UCEA is that for each UCEA there exists an algorithm that maintains a data structure such that a set of complex events can be enumerated from data structure with constant delay and it takes $O(1)$ time to update the data structure upon the arrival of a new event.

Chapter 3

Related Work

The vast majority of the research on the field of complex event recognition focused on the centralized case. A couple of solutions only focused on distribution. In this sections we analyze these solutions and where they fail to address the demand of resources in a big data environment, whether for lack of elasticity or not being specifically designed for CER problems therefore being uneffective.

3.1 Centralized systems

As mentioned, centralized systems are unable to adapt to the size of the stream due to the technical restraints posed by a single centralized computational unit. This renders this family of solutions not a primary candidate for the future of Complex Event Recognition problems.

3.1.1 Centralized CER

In the specific field of CER streaming solutions names like SASE and T-REX start to appear. These are the state of the art solutions that can be found nowadays. Another available solution that is worth mentioning and will also be the subject of our future work is CORE.

3.1.1.1 SASE

As for our approach, SASE [6] is based on a model where the stream of input data is composed by an endless flow of events. An event is an istantaneuos and atomic occurence of interest at any point in time. SASE is a declarative language that combines filtering,

correlation and transformation of events. This language features the first instance of a flexible use of negations in event sequences, parametrized predicates for correlating events via value based constraints and a sliding window for imposing additional temporal contraints. Given a sequence of events as input the output of a SASE query is a sequence of events itself. Each result represents a unique match of the query. Where SASE fails to suits our need is in his limitation to a total order of events. In addition to that the output of a SASE query is a set of event sequences which satisfy the query requests, being as fine grained as it is this solutions lacks of elasticity when it comes to further analyze or possibly distribute the solution of this engine.

3.1.1.2 T-REX

T-REX [7] is an event detection automata that works on the embedded TESLA language. In the case of T-REX TESLA is used to define the structure of the incoming events and define the rules of the query, What T-REX does is handling the incoming flow of data from external sources by implementing a sofisticated system of queues and generating an output from the events parsed by the TESLA language that satisfy some previously defined rules to make up a complex event. As for SASE, T-REX fails under the same aspects being a centralized solution bounded by his necessity of a totally ordered sequence of events.

3.1.1.3 CORE

CORE [3] is a "still in development" engine for the recognition of complex events. Designed at the Pontifica Universitad Catolica de Chile. It is an implementation of an UCEA engine. CORE is a centralized engine that features a set of functionalities that makes it desirable to be integrated in a distributed environment.

Since this is the UCEA engine that we will use in our framework and in the implementation of our distributed environment we now introduce the basic concepts and components that we will use later.

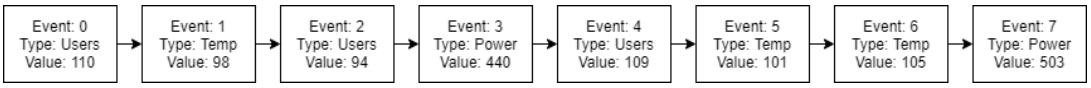


Figure 3.1: Example input for CORE output

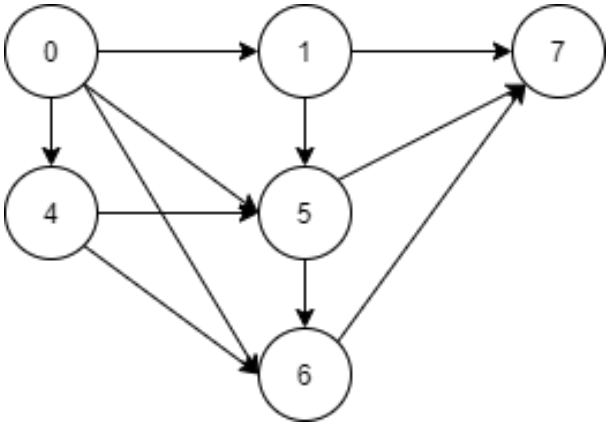


Figure 3.2: CORE output

3.1.1.3.1 Predicates

The predicates are the first components of CORE. They are the set of rules applied by the engine on the input stream in order to detect the subset of events that creates a complex event.

3.1.1.3.2 Match

What is called a Match in CORE is the collection of events from the input stream that respect the Predicates imposed by the engine and together make a solution for our problem.

3.1.1.3.3 Data Structure

Core itself does not output a list of all the matches that has found in a fixed set of the input stream but rather a specific structure that contains all the solution in a compact form and that can be parsed to extract the single matches. In Figure 3.2 we can see a representation of a data structure derived by the sample input of Figure 3.1.

Figure 3.2 shows a data structure containing all possible solutions with respect to the following set of predicates:

$$(U+\ as\ u; T+\ as\ t; P\ as\ p)\ where\ u > 100;\ t > 90;\ p > 500$$

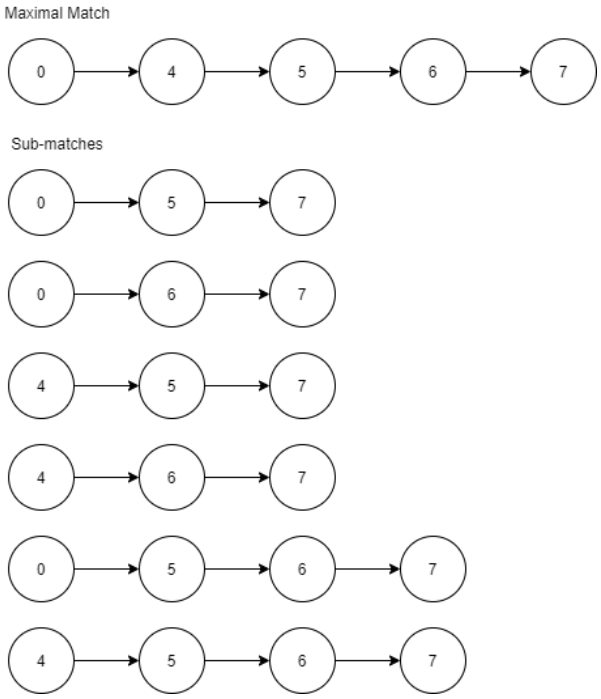


Figure 3.3: A maximal match and it's sub-matches

CORE also allows us to output directly a sub-set of this solution like only the maximal matches which are the matches using as many events as possible. We will use this feature in some of our distribution strategies since it is easy to demonstrate that a maximal match contains all the possible sub-matches in the same group.
In Figure 3.3 we can see a possible maximal match and the sub-matches that it contains.

In addition to an optimal internal structure that allows for a great flexibility the output of CORE, the "Data Structure", which contains all the Matches in this compact container, allows us to outsource the unraveling of the single Matches to the distributed computational units. This conditions gave us great freedom when it came to decide where and how to tackle this step of the pipeline.

3.2 Distributed Systems

When it comes to system distribution in the field of data streaming and complex event recognition, different alternatives are possible [8]. A natural approach to scale CER systems is to leverage on existing frameworks for parallel data processing and extend them with CER capabilities. Most of the existing frameworks such as Spark [9] or Flink

[10] are based on the dataflow model as described in section 3.2.1. One alternative to that is called *Query Partitioning* which, instead of distributing the streaming of events between workers, split the query into subqueries which can examine the data stream independently [11] [12] [13].

3.2.1 DataFlow Model

DataFlow Models base their strategy on distributing the data stream to a set of computational units based on various distribution strategies [14]. The goal is to have the workload divided as evenly as possible throughout the available resources.

3.2.1.1 Round Robin

Round Robin is the most well known distribution strategy and is capable of achieving an optimal distribution of resources. However this approach is ill suited for stateful operator where events must be collocated in a timeline.

However we will not discard this solution right from the get go but rather we will change the elements that Round Robin will distribute so that the process will not break the continuity of data while still exploiting most of the powerful features Round Robin has to offer.

3.2.1.2 Hashing

Hashing or Field partitioning works with a so called hash function which is in charge of assigning a key to each event of the stream [15] [16] . Such key will later be used to choose the computational unit the event will be sent to. This approach succeed in grouping similar event into the same computational unit allowing the analysis of patterns in them. Where Hashing fails is when the input data is highly skewed, in which case most of the input data will be assigned to the same unit creating a much dreaded bottleneck and leaving most of the resources idle.

3.2.1.3 Double-Hashing

Double-Hashing comes in the middle ground for what regards distribution balance and event aggregation[17][18]. What the algorithm does is enhance the behaviour of the Field algorithm by adding a second hashing function which computes a different key for each event [19]. In this way every time an event has to be assigned to a computational unit the dispatcher can choose between two different destinations and select the one with the least overhead. Double Hashing solves one of the biggest issues of Field partitioning but still fails to solve the same problems Round Robins fall into.

3.2.2 Query Partitioning

Another possible approach is the automata-based query partitioning strategy. This method is based on the idea of distributing through-out all computational units the automata in charge of detecting the pattern. Each unit will then use the automata to catch a different portion of the match sequence. In order to achieve this, the whole stream sequence will have to be broadcasted to all units which will ignore most of it. This comes with a set of problems: (1) broadcasting events is significantly less efficient than distributing them and (2) once every computational unit has evaluated its part of the sequence the results have to be merged together which generates further aggregation costs.

3.2.3 Conclusions

As we saw the available solutions fail in one way or another to tackle with success the problems presented by distributing a complex event recognition task. Not only that but all the aforementioned solutions do not take in considerations the massive overhead that is the final step or merging all the results from each computational unit in what can be the desired output respecting the constraints of the query.

Chapter 4

Techniques for Distributed Complex Event Recognition

This section reports on the framework designed in this thesis. The framework has a structure that streamlines and sharply separates the processes of pattern recognition and predicates evaluation so that it is easy to change algorithms and functionality inside the different modules. In particular it allows us to swap between different distribution strategies within each execution of the program and compare the results obtained in terms of performances and ultimately evaluates the pros and cons of each one of them. In this chapter we will go in depth about all the solution we have tested by starting with the generic distribution ones and then looking into the three CER specific we have devised.

Our framework is organized into a sequential set of operations that starts from the input data, i.e. the input stream and the query, up until the distribution of the matches to different computational units for the final verification of their second order variables. It is imperative to understand that the sequentiality of our framework is linked to the intrinsic nature of Complex Event Recognition processes, the detection of patterns is inherently sequential, which does not really make sense to distribute. What this framework aims to achieve is to increase the performance of CER processing by splitting up two subroutines of CER:

- The pattern recognition and First Order predicates evaluation;
- the Second Order predicates evaluation which can be of arbitrary complexity with respect to the FO.

The first subroutine will be done in a centralized fashion for the reasons mentioned in Chapter 3 about the sequentiality of the detection of patterns. This also comes with a set of advantages that mostly lies in the cutting of all overheads deriving from: distributing the engine in charge of finding the pattern required from the query, distributing the whole stream of events to each computational unit and to remove the final overhead that comes from the aggregation of the solution of each computational unit which must not be ignored. In addition to all this reasons it is of utmost importance to remember the constant delay enumeration property of UCEA engines. To be more specific, constant delay enumeration means that, given an UCEA engine $\mathcal{A}$ and a stream S, there is always an algorithm A that maintains a data structure D such that there exists an enumeration routine that takes D as input and generates all the possible events in $[A]_n(S)$ (stream S parsed with the UCEA engine A up until the event n) without repetitions such that the time it takes for the enumerate process is always constant. Moreover, it takes $O(1)$ time to update D at the arrival of the event $n + 1$.
What we will be distributing is the second subroutine of CER processes, i.e. the evaluation of Second Order predicates. The reasoning behind this decision was driven by the fact that the evaluation of second order predicates is computationally heavier with respect to all the other sub-routines in the framework. This way, once the UCEA engine has completed the operation of pattern matching and First-order predicates evaluation, each computational unit in the distributed environment is in charge of evaluating the Second-order predicates and return as results the patterns respecting the predicates, discarding the others.

This way we achieved a framework in which the most intense operations are distributed so as to gain some of the advantages of parallel computing, without the overhead related to aggregation costs or to duplication of streams. In Sections 4.1, 4.2 and 4.3 we will go in depth in each step of our framework from the pattern recognition of the UCEA engine to the evaluation of the Data Structures by the single computational units. Each process will be analyzed and we will see all the alternatives in distribution strategies implemented. The evaluation of the second order predicates will not be analyzed and

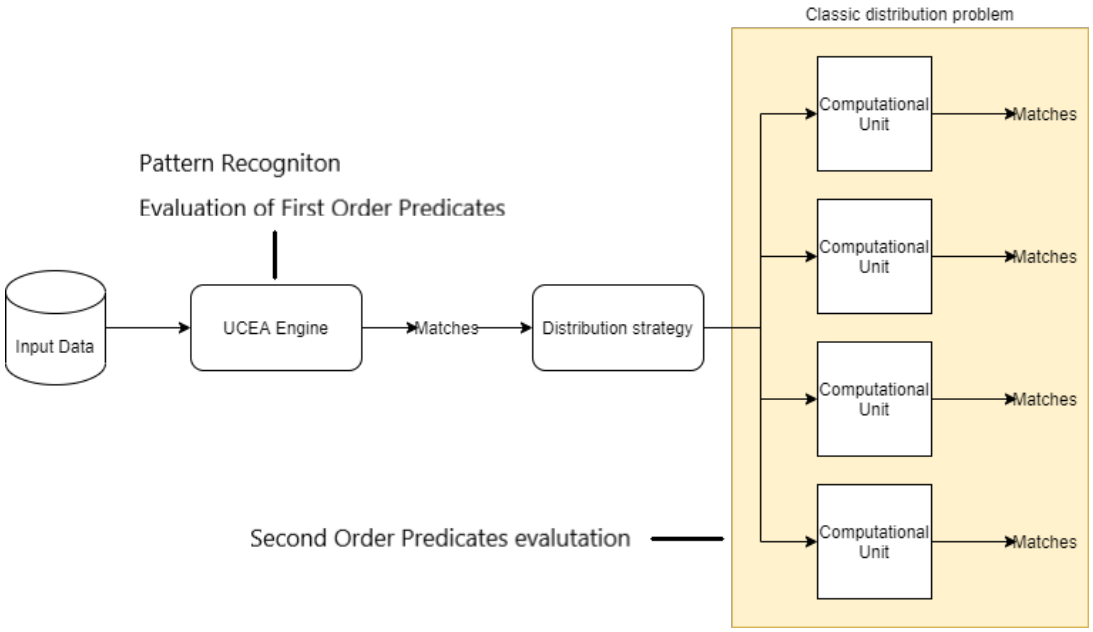


Figure 4.1: Techniques for Distributed CER framework

only simulated in this implementation because having an approximated idea of the time taken by the computational units to evaluate Second-order predicates is sufficient for the scope of the thesis.

In the following figure we can have a preview of the structure of our framework.

4.1 UCEA Engine

The first component of the pipeline is the recognition of patterns a task that is delegated to the UCEA engine. At this stage of the pipeline the input is composed of the raw data stream and the query. CORE, our UCEA engine, receives as input the query and then proceeds to parse the input stream. When CORE encounter a set of events that matches the query it proceeds to generate a collection of matches as output that contains all the possible matches encountered in the segment of data stream analyzed up until now.
It is important to understand that this stage is still running in a centralized fashion due to the intrinsic constraints of Complex Event Recognition. The main difference between a typical centralized solution and our distributed one lies in the fact that we specifically prevent our engine from analyzing the second order predicates of the query, thus reducing the computational effort for the centralized computational unit, and delegating it to the

distributed component of the pipeline. As a result, the group of matches generated by the engine is not actually the Data Structure that CORE outputs containing all the matches that satisfies the query predicates, but rather a selection of all the ordered set of events from the input stream that satisfy the first order predicates and may also respect the second order predicates constraints thus being candidates to become matches. In this environment the output of CORE may consist of a greater number of what we might call "match candidates" (i.e. sets of events evaluated only on their first order predicates) as opposed to the canonical "Data Structure" that CORE outputs normally, nonetheless it is still a more convenient solution due to the reduced effort required to the centralized single computational unit.

4.2 Data Structure Manipulation

Once the collection of matches is generated the novelty component of our pipeline comes into play. Before we distribute our matches to the various computational units in order to be evaluated on the second order predicates we need to select what we will call a grouping strategy.

CORE data structure offers a compact representation of all the matches obtained from the first step of our framework. This structure, while compact, cannot be split into sub-groups, but rather has to be digested into all the single matches. This poses the question wether we want to distribute the data structure as is with a reduced overhead of the central unit or rather compute all the matches in order to obtain a more fine grained distribution and therefore a more balanced workload between computational units.

The decision has to be made considering that while the separation of the groups into their singular matches gives us the best distribution balance between the computational nodes as opposed to distributing the groups which may greatly vary in dimension, the pre-processing operation has to be done in the centralized unit thus further burdening it.

4.3 Choice of a Distribution Strategy

The last step of the pipeline aims to choose the appropriate strategy for distributing our input from the previous step.

In order to test our algorithms, we also implemented some of the consolidated state of the art distribution algorithms so that we would have meter of comparison.

4.3.1 Generic Solutions

We will first have a look at the generic solutions that we used in our framework. All this distribution strategies has been applied to both the processed output from the CORE engine, i.e. the single matches, and the groups of matches themselves.

4.3.1.1 Centralized

While not a distribution strategy, we have tested the whole pipeline in a centralized fashion in order to have a baseline from which to compute the improvement of each and every distribution strategy proposed below.

4.3.1.2 Round Robin

The first distribution algorithm is well known and it distributes the CORE output to the various computational engines in the most even way possible in relation to the number of outputs and computational engines. While very simple in the structure, Round Robin still proves to be a solid algorithm and that is why we have used it as a measure of quality for our novelty solutions.

4.3.1.3 Double Hashing

Double hashing is an evolution of the hashing distribution. Hashing distributions work with a key that is used to map each input to a specific computational unit based on the result of a computation between the hashing key and a pre-determined identifier of the input. This method guarantees that the same input will always be sent to the same computational unit every time it is evaluated with the same hashing key. The problem with the hashing algorithm is that it is impossible to choose a key that guarantees an

even distribution on input without knowing ahead of time the set and frequency of input identifiers. This could result in bottlenecks as a consequence of too many inputs being routed to the same computational unit. To solve this problem the double hashing algorithm uses two different hash keys to evaluate an input and then chooses to assign the input to one of two possible computational units based on which is less loaded.

Since this algorithm currently have no way to communicate with the computational units in order to have a live information of their workload, it estimates a value for the added workload a match or group of matches represent by using the length of the match itself and the dimensions of its Kleene closures. This produces a number which we will use as weight in order to determine the amount of workload we are assigning to the computational unit.

4.3.1.4 Conclusions on Generic Solutions

This state of the art solutions, with a particular focus on round robin and double hashing, will always perform in any environment with acceptable rates. However they neither exploit nor adapt to the environment they are put into. Even thought their performances remains in the range of acceptable results as we will see in Section 6 they do not consider some key factors about distributing CORE data structures like the intrinsic complexity of the structure itself or the difference in matches contained in each data structure up to the cost of evaluation of few matches with multiple second order predicates as opposed to a large number of simpler matches. That's why we decided to make a tentative to create and test some distribution strategies dedicated for CER environments.

4.3.2 CER Specific Solutions

By taking into account the possibility to obtain only the maximal matches as an output from CORE engine we devised two novelty distribution techniques. These techniques work on the premise that every maximal match contains all the possible sub-matches inside that sub-string of the stream. The advantage of computing only the maximal matches is that it is much faster then computing all the matches from a match group.

One major difference between the first two algorithms and the third is the presence of duplicates in the matches generated in the different computational unit. Brute Force Enumeration and Fine Grained Enumeration work on the sub-matches contained in all the maximal matches produced by CORE. Although this is a more efficient way of distributing the matches between the computational units it is important to notice that different maximal matches may contain the same sub-match that ultimately will be analyzed twice by the different computational units that are in charge by the maximal matches in question. As opposed to this the third algorithm is inspired on the algorithm proposed by Brute Force Enumeration but goes a step further eliminating the duplicates at the cost of computational complexity while the first two were deliberately left unchecked to compare the efficiency of a simpler solution as opposed to a more sophisticated one.

4.3.2.1 Brute Force Enumeration

Brute Force Enumeration is the first CER Specific distribution algorithm. As previously said this algorithm is based on the distribution of the maximal matches from a group of matches obtained by CORE. The idea behind Brute Force Enumeration is a simple Round Robin distribution of the maximal matches between the different computational units. This way each computational unit will be in charge of handling only the sub-matches generated from the maximal match they are being assigned.
This strategy on paper promises to yield an improved efficiency with respect to the centralized environment. However some flaws are noticeable from the get go, for instance:

- it is prone to bottle-necks due to the different number of matches contained in different maximal matches thus creating an unbalance between different computational units;
- it shows its potential only in presence of streams that generates multiple maximal matches otherwise it is prone to agglomerate all the solutions in one maximal match rendering the solution on par with the centralized alternative.

4.3.2.1.1 The input

As we talked in the introduction of CER specific solution the input of Brute Force Enu-

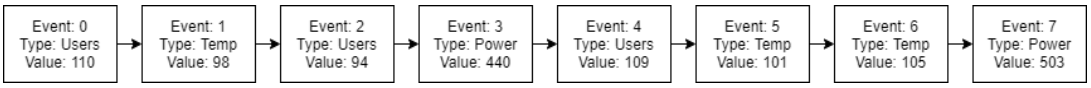


Figure 4.2: Example input for CORE maximal matches output

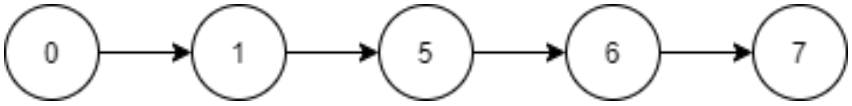


Figure 4.3: Example of maximal match

meration is a CORE Data Structure containing all the maximal matches. We use maximal matches as a rough expedient to group the matches into sub-groups that are then distributed to different computational units. This way we obtain a distribution of matches which is not optimal, but reduces the effort on the central computational unit to a minimum.

4.3.2.1.2 The Distribution

The maximal matches contained in the CORE Data Structure are distributed to the computational units Starting with a Round Robin strategy for the first matches until each unit has received at least a match to analyze, the the following matches are distributed to the computational unit with the least amount of workload.

4.3.2.1.3 Computation of Sub-matches

Once a computational unit is assigned a maximal match it computes all the possible sub-matches in order to evaluate the second order predicates on each one of them.

In order to compute all the sub-matches we used combinatorial calculus. What is important to notice in a maximal match is that, while single events are constant in each sub-match, what makes the different sub-matches is the combination of all possible events being part of a Kleene-closure in the maximal match. Therefore, in order to compute all the sub-matches Brute Force Enumeration computes the power set of each Kleene-closure in the maximal match, then to compose the sub-matches it combines all the different sets of matches created by performing a Cartesian product of the power sets of all the Kleene-closures. Below in figure 4.4 we can see an example of all the combinations generated by

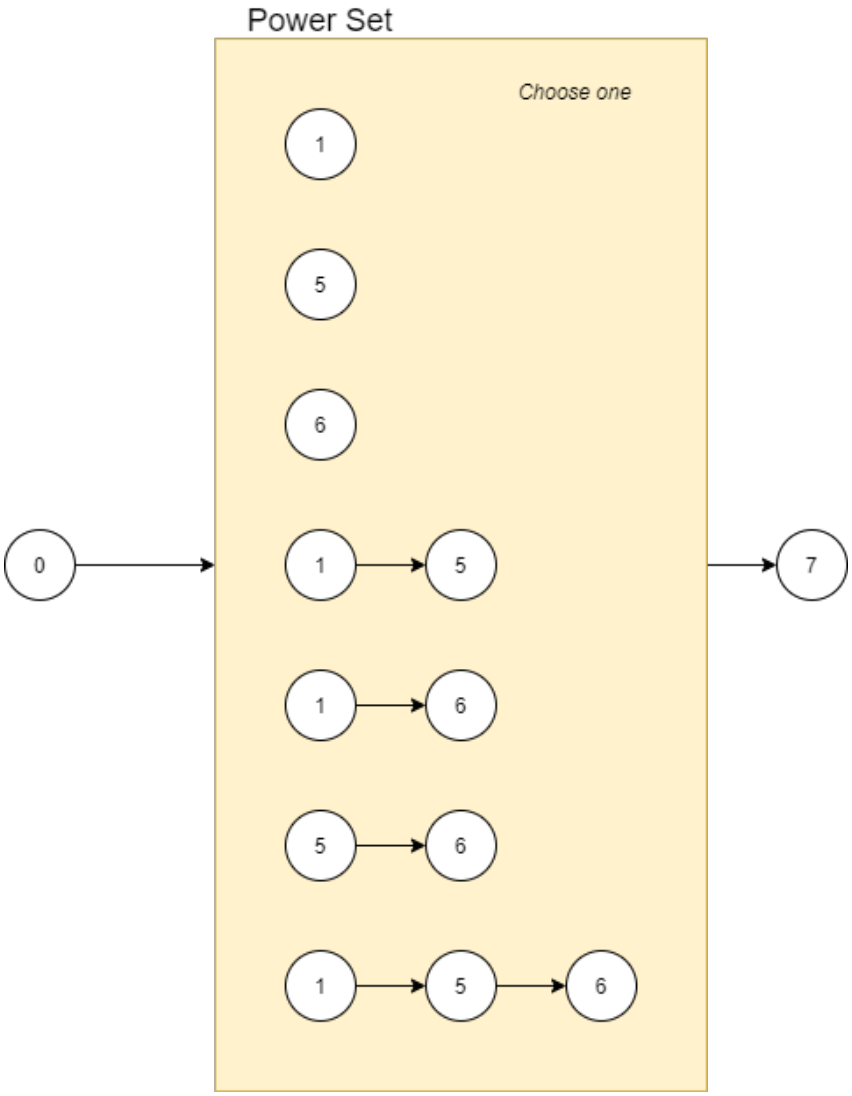


Figure 4.4: Brute Force Enumeration sub-matches from a maximal match

Brute Force Enumeration given a maximal match.

4.3.2.2 Fine Grained Enumeration

Fine Grained Enumeration tries to touch a middle ground between Brute Force Enumeration and Round Robin. Fine Grained Enumeration generates all the sub-matches from the maximal matches before distributing them. The strategy focuses on the dimension of the generated sub-matches grouping them by length. Once the maximal matches are parsed, the sub-matches are produced and grouped by length, then the algorithm proceeds to merge these groups in a way such that each final group of matches is as even as possible with respect to the others. In Digure 4.6 below we can see an example of the groups generated by our example.

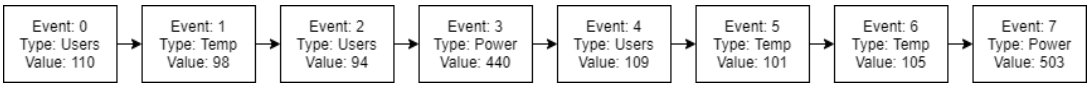


Figure 4.5: Example input for Fine Grained Enumeration Grouping

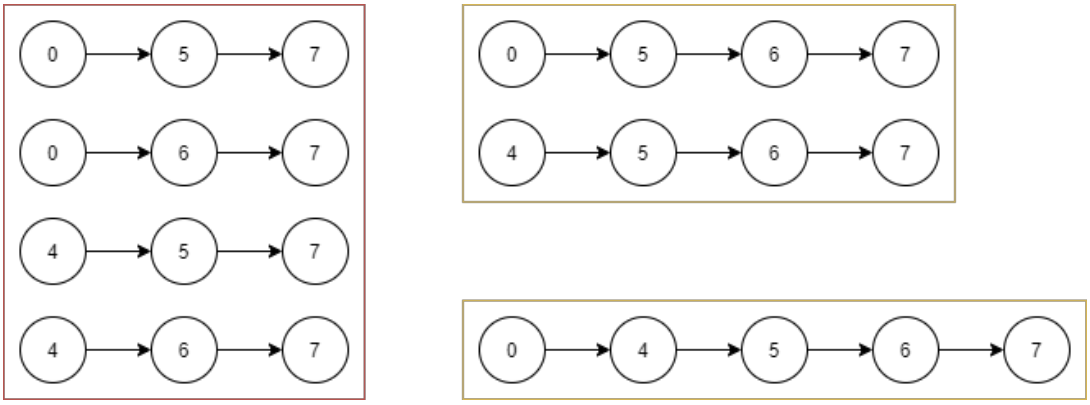


Figure 4.6: Fine Grained Enumeration Grouping

In Digure 4.6 we can see that, as a result of Fine Grained processing, the matches are grouped by size. Each group is represented by the rectangle containing the matches of size x, the color of the group is the subsequent grouping made by the algorithm to even the distribution of matches between the computational units.

4.3.2.2.1 Maximal Matches

As for Brute Force we start our process by the maximal matches received from CORE.

4.3.2.2.2 Sub-matches Configuration

In order to distribute the various matches with our Fine Grained Enumeration system the first step we take is to create a structure in which we store the matches ordered by their length produced by each maximal match. This structure is called Sub-matches Configuration.

4.3.2.2.3 Bin Packing problem

Once the configuration is generated the Algorithm proceeds to solve a classic bin packing problem with the set of matches generated. In our specific case the algorithm first gives a set of matches to each available computational unit, then proceeds to distribute the

remaining to the computational unit with the minimum workload of the available. The workload is computed based on the dimension of the previously assigned sets of matches.

4.3.2.3 Hyper-graph Enumeration

To tackle the issue of duplicates in the matches of each computational unit we devised a new version of the brute force enumeration algorithm which is based on Hyper-graph like structures to relieve the algorithm from unwanted duplicates without the need of cross-validation between the computational units and without producing unnecessary matches.

The whole idea is based around the Data Structure containing all the matches generated by CORE from a stream segment. In Figure 4.7 we can see an example of representation of this structure to better visualize its composition. For this case, we will use an ad-hoc example to better represent the situation, for ease of representation we will omit the value of each event assuming that they will respect the query restriction.

$$The\ Query:\ A+; B+; C+; D$$

$$The\ Stream:\ A_0B_0A_1B_1C_0C_1C_2A_2A_3B_2C_3D_0$$

Each node represented in this graph contains information about an event and its eligible following events that can participate in creating a match. Each maximal match is stored in this representation and can be extracted into forming a sub-section of the graph above containing the information about itself and all the possible sub-matches contained in it. In the figure below we can see an example of maximal match contained in the data structure above.

By analyzing the representation of two maximal matches from the example above it is easy to see that duplicates may arise in different computational units with different maximal matches(e.g. the match $A_0B_1C_3D_0$ is present in both maximal matches).

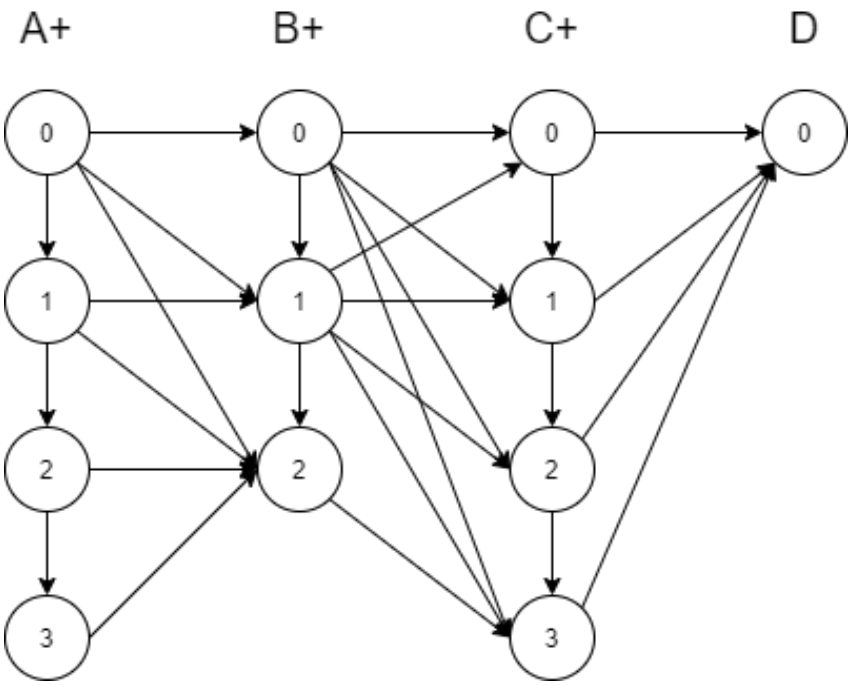


Figure 4.7: Example of Data Structure Graphical Representation

The focus of this algorithm was to find a way to restrict the rules with which the computational unit generated the sub-matches from a maximal match without having to compare the results with the ones of each other unit and without generating all the possible sub-matches to then discard the duplicate ones. However, in order to achieve this objective the algorithm has to increase in complexity since every maximal match has to be sent to every single computational unit which, in order to determine what matches to analyze and what to avoid needs to compute each one with the algorithm we will see below.

In order to understand how the algorithm works we need to take a step back and analyze the input stream in order to determine a new structure that allows us to move from the simple graphical representation as seen until now to an Hyper-graph structure that enables the correct functioning of the algorithm itself.

$$A_0 B_0 A_1 B_1 C_0 C_1 C_2 A_2 A_3 B_2 C_3 D_0$$

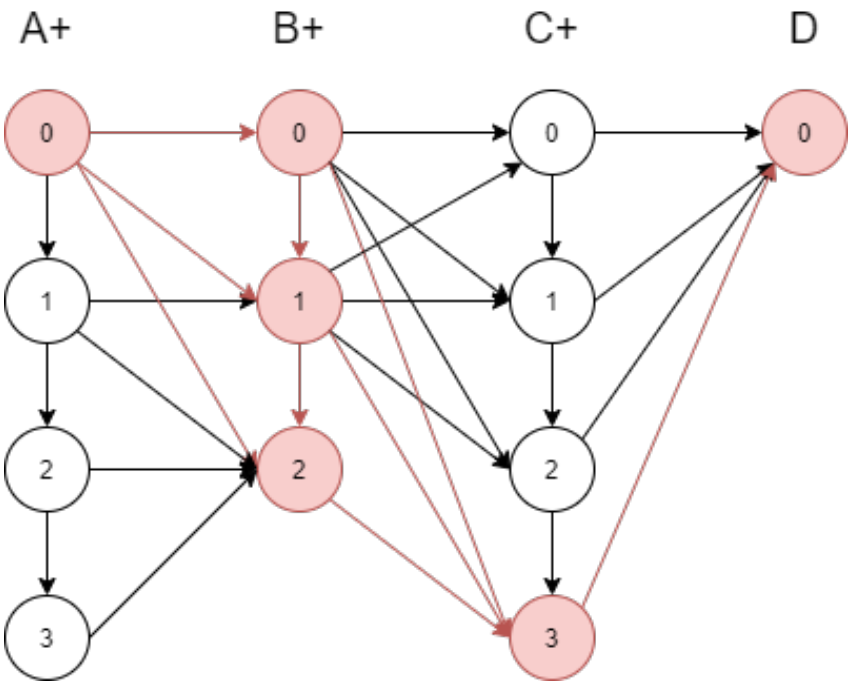


Figure 4.8: Maximal Match Graphical Representation

As we can see in the Example Stream above there are some clusters of Events of the same type which for the purpose of creating a match respecting a query can be treated as a single entity. This entity is called an Hyper-edge. This concept of hyper edge is crucial when it comes to instructing the algorithm on the available steps it can take in order to create a match from the maximal matches it has been assigned.

We can represent the newly generated Hyper-graph as in the picture below.

Once the Hyper-graph structure is determined the algorithm to generate sub-matches from a Maximal match hyper-graph is as follow:

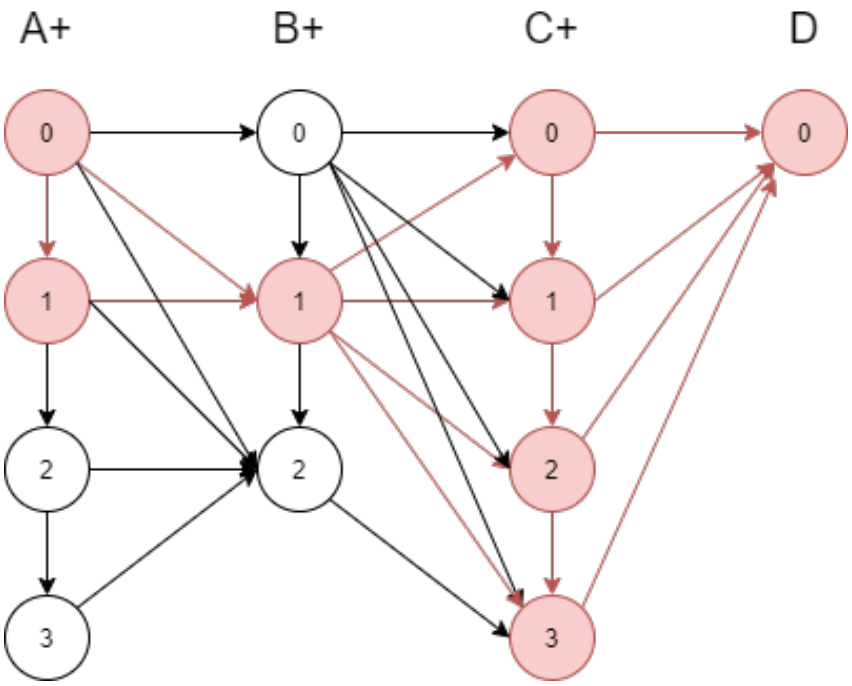


Figure 4.9: Second Maximal Match Graphical Representation

Algorithm 1 Hyper-graph Enumeration algorithm

Data: Maximal match Hyper-graph
Result: Sub-matches from maximal match
for *Node in the first group of the Hyper-graph* **do**
  Add node to new **match**
  while *Node* ***not*** *in final group* **do**
    Select new node from:
    - Subsequent node from same group
    - A node in the first Hyper-edge of the following group
    Add node to match
    Repeat from new node until condition is met
  end
  Return match
end

This simple steps ensure that each sub-match generated is unique and will not be reproduced by any other maximal match.

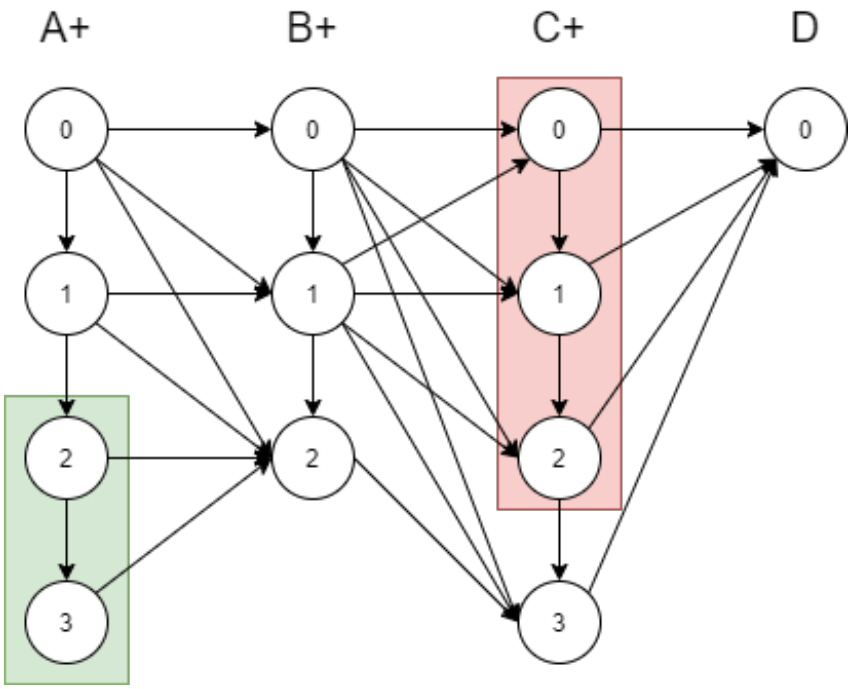


Figure 4.10: Hyper-graph Representation

Chapter 5

Implementation

5.1 Pipeline

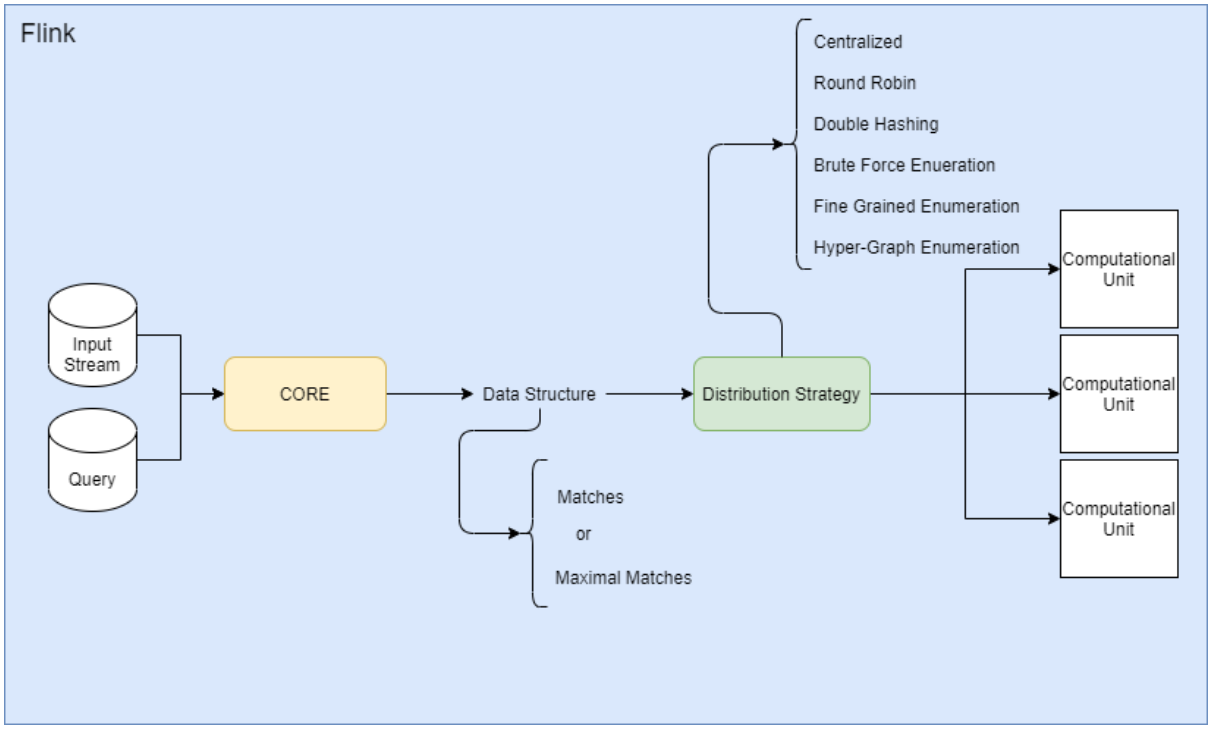


Figure 5.1: DCER Pipeline

In the above figure we recall the pipeline for our Distributed Complex Event Recognition framework. The implementation was composed of three main step. First, integrate our UCEA engine in a distribution environment. Second, adapt the engine to output a structure with the information needed to implement all our distribution strategies and lastly implement the distribution strategies in order to apply them to the parsed output of the UCEA engine. We will now go into the detail of the components that we chose for

the implementation.

5.2 CORE

CORE is the UCEA engine that we will use in order to evaluate predicates on a stream of events. CORE has a neat structure that allows us to work with a more intuitive writing of the queries and an easily extensible structure of the output as well as the single events. This allowed us to modify the output content to contain additional information useful for our tailor-made algorithms.

5.2.1 The Query

The first part of the input for CORE is the definition of the stream in order for the engine to be able to recognize and discern the event from one another. This is simply expressed as shown below

```
1    DECLARE EVENT T(t int)
2    DECLARE EVENT U(u int)
```

As we can see each event is defined with a type and a variable name. This way we can execute the next step which is defining the composition of the stream like so

```
1    DECLARE STREAM S(T, U)
```

The stream is declared with a variable name and the name of the events it is composed of.

Finally CORE requires as input a query which is the pattern of events we want to find in our stream. This query is expressed in a language very similar to SQL. Below we can see an example of query

```
1    SELECT *
2    FROM S
3    WHERE (T as t1; U+ as us; U as u1)
4    FILTER (t1[t < 0] AND us[u < 60] AND u1[u >60])
```

Now let's analyze each component of the query.

```
1    SELECT *
```

The first row gives us restriction on the actual matches that we find in the input stream. The * shown in the example means that we accept every match we find in the stream, an alternative could be MAX which will only accept the longest match we find.

```
1    FROM S
```

The second row is in charge of telling the engine which stream to analyze. This is useful in case there are different streams as input.

```
1    WHERE (T as t1; U+ as us; U as u1)
```

The third row sets the structure of the match. In this line we set the sequence of events we expect and we set a variable name to every event that matches a particular step. The language is pretty simple, when composing a pattern we write the type of event we are expecting and with the *as x* writing we assing his own variable. In case we want to match a Kleene closure we need to add a + sign after the type of event we are looking for.

```
1    FILTER (t1[t < 0] AND us[u < 60] AND u1[u >60])
```

The last row set the boundaries on the variables that we've set in the previous row. in this line, for each variable we want to put a constraint on we define a boundary that will be applied to each variable that will satisfy the conditions of the third row. This conditions are called predicates.

5.2.2 The Stream

Once the query is set and the engine is instructed to recognize the chosen pattern in a stream the input representing the stream has to be represented as such.

```
    ("S", "T", -2),
    ("S", "U", 30),
    ("S", "U", 20),
    ("S", "T", -1),
    ("S", "U", 27),
    ("S", "T", 2),
    ("S", "U", 45),
    ("S", "U", 50),
    ("S", "T", -2),
    ("S", "U", 61)
```

The example above shows a sample of what could be the input stream for the query that we've defined before. Each event has to be labeled for the stream it is part of, the type of event it represents and the value it took. This way our engine can analyze the input stream and look for patterns it was instructed to recognize. For example in this case the sequence of events 1-2-3-5-7-8-10 is a valid match for the query defined before.

5.2.3 The Output

In the example above the sequence 1-2-3-5-7-8-10 is not the only valid match for our query. In order to represent all the solutions available CORE has devised his own structure to represent all the matches. This data structure is a compact representation of all the matches in a single compact container which can be parsed to produce the single matches. We can see a graphical representation of the Data structure in the image below.

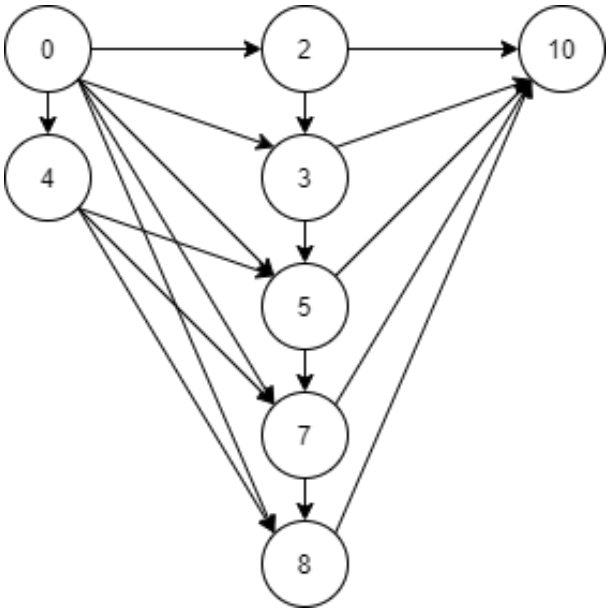


Figure 5.2: CORE Data Structure

5.2.4 Our Modifications

To better visualize the additional information that we've added to the CORE Data Structure we can represent the query as an automata like in figure 5.3

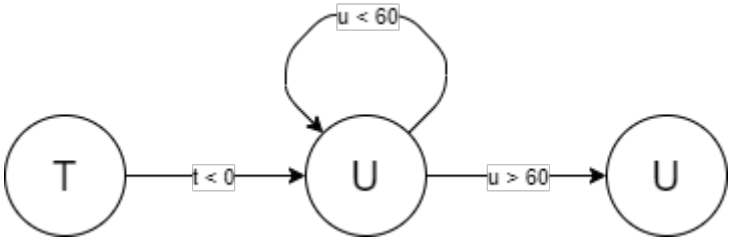


Figure 5.3: Automata representation of the query

In order to compute the complexity of a match before evaluating his second order predicates (i.e. predicates applied to Kleene Closures) we added to each event saved in the Data Structure the identifier of the node the event triggered in the automata in order to remember to which part of the query the event belongs to. This way we can easily distinguish between single events and sets of one or more events inside the match. This additional information will be used when computing the "weight" of the match when the routing algorithm decides to what computational unit it will be sent.

5.3 Flink

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams [10]. In this project we have decided to rely on Flink to handle the distribution side of our project.

Flink allowed us to create our own distributed environment with our custom structures to handle the engine, the data stream and set our custom algorithms for the distribution strategies.

Flink provides us with a structure called Operator which takes as input a stream of data and applies a custom function over it in a distributed fashion. In order to deal with the data stream with our own UCEA engine we've created our own Flink Operator. Having our own Operator allowed us to pipeline all the operations from instructing the engine with the new query, to parsing the stream with the engine while maintaining a non distributed environment up to applying different distribution algorithms to the output of CORE and then subsequently store metrics on the execution of the pipeline itself.

Below we can see a typical pipepile in Flink-Core.

```
1    FlinkCore.pattern(nonPartitionedInput, query, ROUND_ROBIN)
2              .selectMatches(SelectMatchesFunction()).forceNonParallel()
3              .map(new Distribution_Function())
4              .map(new Matches_Function())
5              .map(m->m.f1)
6              .filter(new SecondOrderPredicate(complexity));
```

In the first row we feed the Flink Operator with the input stream, the query and the distribution strategy we want to be applied. Then, once the Operator has created an istance of the CORE engine instructed to recognize the query on the input stream we execute the parsing operation of the input stream in search for matches. As previously said this operation has to be done in a centralized structure, therefore we force it to be non parallel. The following steps are in charge of taking metrics on the execution up until

line 6 where we simulate the computation of the second order predicates with the given complexity.

5.3.1 Operator

Flink Operator is in charge of handling the distribution aspect of the pipeline.

When the program gets executed and the Operator is instantiated an open function is in charge of creating the engine from the CORE query and a instantiate a partitioner with the right distribution strategy chosen for the execution.

```
1    public void open() throws Exception {
2        engine = engineFactory.createEngine();
3        dcer_partitioner = new Partitioner(systemMaxParallelism);
4    }
```

Once the engine is up and running the Operator starts processing the input elements one by one (in our case the single events from the input stream). The events are sent to the engine which returns a Data Structure once the received event triggered the closing condition of the Query.

```
1    public void processElement(StreamRecord<IN> element) throws
     Exception {
2        Tuple3 T = (Tuple3)element.getValue();
3        Event event = new Event(T.f0,T.f1,T.f2);
4        DataStructure result = engine.new_sendEvent(event);
5        ...
```

Then, once a Data Structure has been created the Operator proceeds to assign the partition to the result based on the distribution strategy previously chosen. For example

```
1    result.forEach(match -> {
2        userFunction.processMatch(dcer_partitioner.partition(match));
3        ...
```

5.3.2 Distribution Strategies in Flink

Flink allow the user to setup his own custom distribution function. This allowed us to easily implement and integrate our own distribution strategies like so

```
1    .partitionCustom(new PartitionerFromTuple(), new MatchKeySelector()
     )
```

In the line above PartitionerFromTuple() is in charge of routing the current element of the stream to the computational unit indexed by the key fetched by MatchKeySelector().

5.3.2.1 Centralized

Regarding the Centralized solution the idea was to simply force the entire pipeline in a non parallel state. This was achieved in Spark by setting the number of available units to 1.

```
1    env.setParallelism(1); env.setMaxParallelism(1);
```

5.3.2.2 Round Robin

Round Robin was also pretty simple to implement. The distribution strategy was given a counter that increased with each subsequent element that was distributed. The key is then computed by simply computing the counter modulo the number of computational units available.

```
1    public RoundRobinPartitioner(int systemMaxParallelism) {
2        super(systemMaxParallelism);
3        this.roundRobinCounter = 0;
4        ...
5        match = new Tuple<>(match, this.roundRobinCounter);
6        this.roundRobinCounter++;
7        ...
```

5.3.2.3 Double Hashing

For Double Hashing the strategy becomes a bit more complex. As we mentioned before, we modified the structure of CORE output in order to include information about the Kleene closures. This information allows us to make a conjecture on the complexity of the match we are distributing so as to create information about the overall workload assigned to the available computational units. To compute the final key for each match, the distribution algorithm uses a function offered by Google APIs [20] that takes a string version of the match as input and returns the key computed with Murmurhash algorithm from Google guava [20]. This procedure is repeated with two different seeds in order to obtain two different keys. Once the keys are generated the algorithms checks an internal memory of the workload of the computational units indexed by the selected keys and then assign the match to the one with less load. The load of the current match is then

added to the computational unit that took in charge the match.

First we set the two hashing function chosen

```
1    HashFunction h1 = Hashing.murmur3_128(13);
2    HashFunction h2 = Hashing.murmur3_128(17);
```

Then we proceed to compute the key for the two computational units based on the selected hash functions and the key representing the match which is a String representation of the match itself

```
1    int firstChoice = h1.hashBytes(key.getBytes()).asLong() %
     partitions;
2    int secondChoice = h2.hashBytes(key.getBytes()).asLong() %
     partitions;
```

Finally, based on the two computed choices we select the one linked to the computational unit with less workload and then we add the weight of the distributed match as a measure of workload to the select unit.

```
1    int selected = workload.get(firstChoice) > workload.get(
     secondChoice) ? secondChoice : firstChoice;
2    workload.set(selected, workload.get(selected) + weight);
```

5.3.2.4 Brute Force Enumeration

Brute Force Enumeration in order to distribute the maximal matches as a first step enumerates the number of sub-matches contained in each maximal match in order to create a measure of weight of the maximal matches that will be used later on to assign them to the least loaded computational unit.

```
1    public static double countAllSubmatches(Match m) {
2        List<Set<Event>> groupedEvents = eventGrouping(m);
3        double count = Math.pow(2,groupedEvents.get(0).size())-1;
4        for (int k = 1; k < groupedEvents.size(); ++k) {
5            count *= Math.pow(2,groupedEvents.get(k).size())-1;
6        }
7        return count + 1;
8    }
```

The number of sub-matches is obtained by computing the power set as explained in paragraph 4.3.2.1.

```
1    int K = super.getSystemMaxParallelism();
2    double[] cummulatedWeight = new double[K];
```

Then, the first matches are added following a Round Robin algorithm to the available computational units with the respective workload value.

```
    for (int i = 0; i < K && i < submatches.length; ++i) {
        partitionPerMatch.put(submatches[i].f0,i);
        cummulatedWeight[i] = submatches[i].f1;
    }
```

Finally, each subsequent maximal match and its workload value is added to the computational unit with the least amount of assigned workload.

```
    for (int i = K; i < submatches.length; ++i) {
        int idx = 0;
        for (int j = 1; j < K; ++j) {
            if (cummulatedWeight[idx] > cummulatedWeight[j]) idx = j;
        }
        cummulatedWeight[idx] += submatches[i].f1;
        partitionPerMatch.put(submatches[i].f0,idx);
    }
```

5.3.2.5 Fine Grained Enumeration

Fine Grained Enumeration, in order to distribute the matches between the computational units has to count and group all the matches with the same size.

```
    for (submatch : SubmatchEnumerator.countAllSubmatchesOfCertainSize(
m)) {
        SubmatchesPerMaxMatchAndConfiguration.add(getAllMaxatches(),
Integer.valueOf(i)),submatch.f1));
    };
```

m is a maximal match from which we retrieve all the possible sub-matches marked for their size and then we proceed to store the sub-match along with their size in a container.

Afterwards, once the sub-matches are grouped by size, the algorithm proceeds to solve the bin packing problem as follow

```
    int K = super.getSystemMaxParallelism();
        double[] cummulatedWeight = new double[K];

        for (int i = 0; i < K && i < submatches.length; ++i) {
            partitionPerMatchAndConfiguration.put(submatches[i].f0,i);
            cummulatedWeight[i] = submatches[i].f1;
        }

        for (int i = K; i < submatches.length; ++i) {
            int idx = 0;
            for (int j = 1; j < K; ++j) {
                if (cummulatedWeight[idx] > cummulatedWeight[j]) idx =
j;
            }
            cummulatedWeight[idx] += submatches[i].f1;
```

```
15              partitionPerMatchAndConfiguration.put(submatches[i].f0,idx)
    ;
16          }
```

5.3.2.6 Hyper-Graph Enumeration

Hyper-Graph Enumeration is also RoundRobin-based in the sence that for each maximal match produced the partition is chosen under Round Robin strategy. The differences between Hyper-Graph Enumeration, Round Robin and the other novelty solutions lies after the partition has been chosen. For each maximal match the whole set of maximal matches is sent to the assigned partition along with a copy of the query. The algorithm then proceed to extract the query structure in order to recreate the graphical representation we have seen in Chapter 4.

Subsequently, once the query structure has been retrieved the algorithm then proceeds to find the hyper-edges in the input by using the maximal matches by following this structure:

```
1     hyper_edge = [A=0, B=0, C=0]
2     for event in M_4:
3       event.hyper_edge = hyper_edge[event.type]
4       If event.type != event.next.type:
5         hyper_edge[event.type]++
6       else:
7         for match in [M_0,M_1,M_2,M_3]:
8           for event2 in match:
9             if event2 == event:
10              if event2.next != event.next:
11                hyper_edge[event.type]++
12                break
13            if event2 == event.next:
14              hyper_edge[event.type]++
15    break
```

Once the query structure and the hyper edges have been generated the last step is to generate the hyper-graph from the obtained information.
The Hyper-graph is stored in memory as a list of all the events in the maximal match each with the list of alpha and beta child at his disposal in order to generate a sub-match.

At this point the algorithm last duty is to generate all the sub-matches starting from the Hyper-graph structure generated following the rules explained in Chapter 4. Below we can see the pseudo-code behind the generation of the submatches.

```
1     data_structure = []
2     for event in M_4:
3       node = new node(event)
4       for event2 in M_4 [event:]:
5         if event2.type == event.type:
6           node.add_beta_child(event2)
7         elseif event2.type == query[event.type].next.type:
8           if event2.hyper_edge == 0
9             node.add_alpha_child(event2)
10    data_structure.append(node)
```

Chapter 6

Experiments

This chapter reports on the experiments with CORE engine, which was integrated with flink and the distribution algorithms. The experiments aim to compare the efficiency of the distribution solution with respect to a centralized solution. We ran two types of experiments. The first type evaluated the various distribution strategies under random settings to test the efficiency of each algorithm. The second type focused on a specific query with an input stream designed to generate an increasing number of maximal matches. This aimed to test the capacity of our algorithms versus the state of the art when dealing with a set of inputs that are better handled by our algorithm by design.

An important thing to be noted is that all the tests have been executed in a local environment. A typical setting for Flink should be a cluster, but a local setting emulating separate workers was sufficient in order to obtain a scaled down overview of the problem.

6.1 Hypothesis

The expectation from these experiments is a significant boost in performance if compared to a centralized solution.

In particular, the expectation is that such canonical distributed algorithms as round-robin will not perform satisfactory when the number of complex events rises. While Double-hashing algorithm can show improvements because of its ability to evaluate the weight of the input, when the numbers starts to rise it may show poor performances to avoid bottle-necks. Indeed it is practically impossible to choose a perfect hashing function

that completely avoids favoring one unit over another. On the other hand, we expect a sub-par performance from Brute Force Enumeration since it distributes the maximal matches in a round robin fashion, therefore completely ignoring the dimension or the number of sub-matches contained in the maximal match, nonetheless the objective of Brute Force Enumeration is to minimize the effort done by the centralized unit. Furthermore, while the performances might be drastically worse with a small sample of inputs, the difference in performances might drop once the number of inputs scales a lot. Finally, we expect that Fine Grained Enumeration, with the right composition of settings, will eventually surpass round robin due to his intrinsic ability to reach a great balance in weight between the distributed inputs.

6.2 Experimental Settings

All the experiments were conducted on randomized streaming settings and query compositions. Below we will see how each component has been generated and what variables has been taken in consideration.

6.2.1 Stream

The Event Stream for the first experiment has been generated in a completely random fashion with each event being select between all the available events options with a random value ranging from -50 to 50. The length of the test streams was set dynamically during the execution with a criteria we will analyze later.

For the second experiment the Stream was generated in a specific way. The Stream was composed of increasing number of event couples respecting the query and closed by a final event for completing the query requirements. For example, with the query $A+; B+; C$ the stream was composed by and indefinite number of AB couples and terminated with a C event. This particular stream gave us control over the number of maximal matches since for each AB pair in the stream we have a new maximal match as a result.

6.2.2 Query

The Query for each algorithm execution in the first experiment has been also generated at random under some constraint based on the following set of variables:

1. **Events**: this variable was in charge of determining the number of unique events that composed the query;
2. **Size**: this variable determined the length of the query, or, in other words, the number of nodes in our automata;
3. **Closure**: the last variable was in charge of determining the number of Kleene-closures contained in our query. This number was bounded to be lower equal the size of the query.

On the other hand for the second experiment we have already introduced a candidate query. The query was composed of three different event, two alternating in order to generate a controlled number of maximal matches, and the third one as the closing event. For example $A+; B+; C$

6.2.3 Second order Predicates Complexity

Since CORE engine is not yet ready to be used in a distributed environment, during our experiments we were bound to simulate the execution of the analysis of second-order predicates inside each computational unit by executing simple dummy functions that were supposed to emulate the various possible computational complexities.

6.2.4 Distribution Strategies

All the proposed distribution strategies were tested against all possible configurations for the stream and query that has been proposed before.

6.2.5 Execution Scripts

In order to test all the strategies with a multitude of input streams and queries an execution script has been devised. The script was in charge of generating a stream and a

query and then test those inputs against all the possible distribution algorithms. The algorithm takes as inputs a set of variables to generate the inputs and then proceeds to save the results in a .csv file for future evaluation.

In our tests the values for the variables are the following:

1. **Events**: 3;
2. **Sizes**: 4, 6, 8, 10;
3. **Kleene Closures**: start from 2 increasing by 2 up to size.
4. **Complexity**: Linear

The input stream had a dynamical generation. Since with the increasing of the query structure the probability of finding a match in a stream with a low number of events decreases, the stream was setup to adapt to the query at hand.

The algorithm was instructed that the query had a nominal starting dimension of zero events and, if the result from the execution of Flink-Core didn't yield any result then the dimension of the input query has to be increased by a great quantity. On the other hand once the algorithms start to produce results, then the input stream will begin to increase by a small amount between each set of test for every distribution algorithm.

Finally, the testing algorithm had a time limit bound on the execution timer for Flink-Core and was instructed to stop incrementing the dimension of the input query once the execution with each distribution strategy didn't manage to yield any result before the time limit runs out and to pass to the next query and input settings.

For the second Experiment the execution script was simply in charge of increasing the stream length after generating the results with the current stream against all the strategies until one of the strategies failed to meet the time requirements.

6.3 Experimental Results

The meters of evaluation we have used to compare the results obtained with the execution of Flink-CORE with different distribution strategies are:

- **Execution Time**: This is the time taken by Flink-CORE to output the results under a certain set of variable settings;
- **Coefficient of Variation**: In probability theory and statistics, the coefficient of variation (CV) is a standardized measure of dispersion of a probability distribution or frequency distribution. It is often expressed as a percentage, and is defined as the ratio of the standard deviation σ to the mean μ. In our case we use it to evaluate the uniformity of the distribution of elements between the computational units.

In the graphs presented below the following acronyms were used to represent the various distribution algorithms:

- **COREC**: Centralized;
- **CORERR**: Round Robin;
- **COREDH**: Double Hashing;
- **BFEFMM**: Brute Force Enumeration;
- **FGEFMM**: Fine Grained Enumeration;
- **NCBFE**: Hyper-Graph Enumeration;

For the first experiment we have compared the combined results of each distribution algorithm over all the tests.

Figures 6.1 and 6.2 show a box-plot of the results obtained from the execution of the script. Figure 6.1 shows the execution times of the first batch of experiments grouped by distribution algorithm. In order to correctly read the graph it is important to make clear that each algorithm was executed until the input settings were too demanding (i.e. the execution took longer than an assigned time limit). Therefore, not every algorithm was tested on the same amount of CORE outputs. This ensured that more efficient

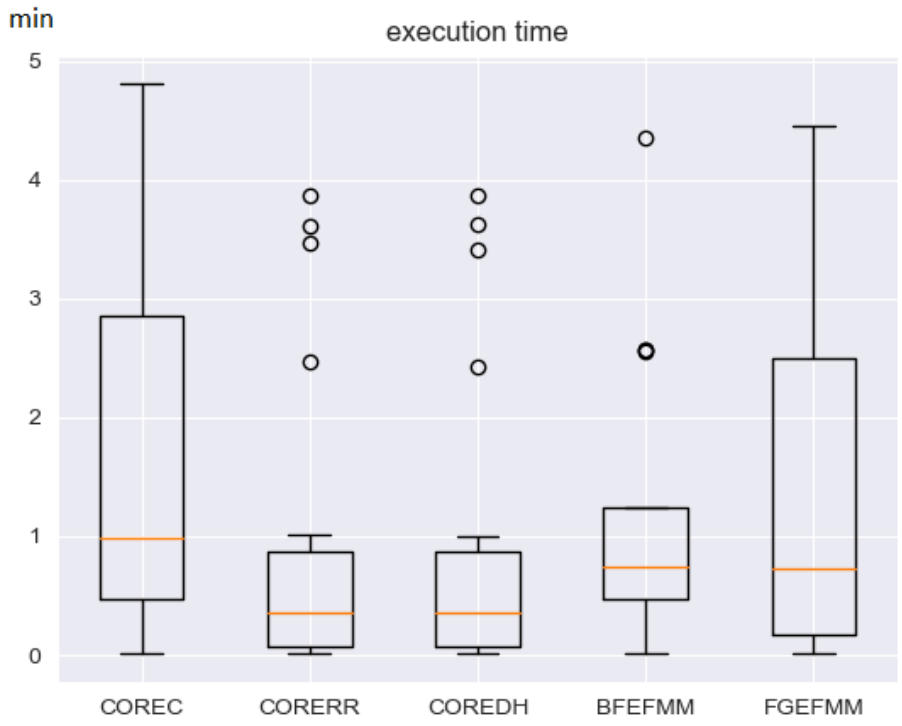


Figure 6.1: Execution Time Results

algorithms could be tested with more demanding settings. Note how the centralized solutions were expectedly the first to fail. While this may obscure the real difference between the various algorithms, the results obtained under the first experiment condition matched our expectations. When dealing with a small random input stream the number of maximal matches cannot grow to a point where algorithms like Brute Force Enumeration and Fine Grained Enumeration can make use of all the computational units at hand, therefore their lack in performances was not linked to a design flaw, but rather it was caused by an unfavourable set of inputs.

In fact, even if they were able to withstand heavier tasks than a centralized solution, their average execution time was not satisfactory and far away from the average time of a good solution like Round Robin or Double Hashing. The struggle of Fine Grained Enumeration is also reflected by his below average coefficient of variation as shown in fig. 6.2 which as we will see later should be almost on par with Round Robin.

As a result of the lack in number of maximal matches in the input stream generated

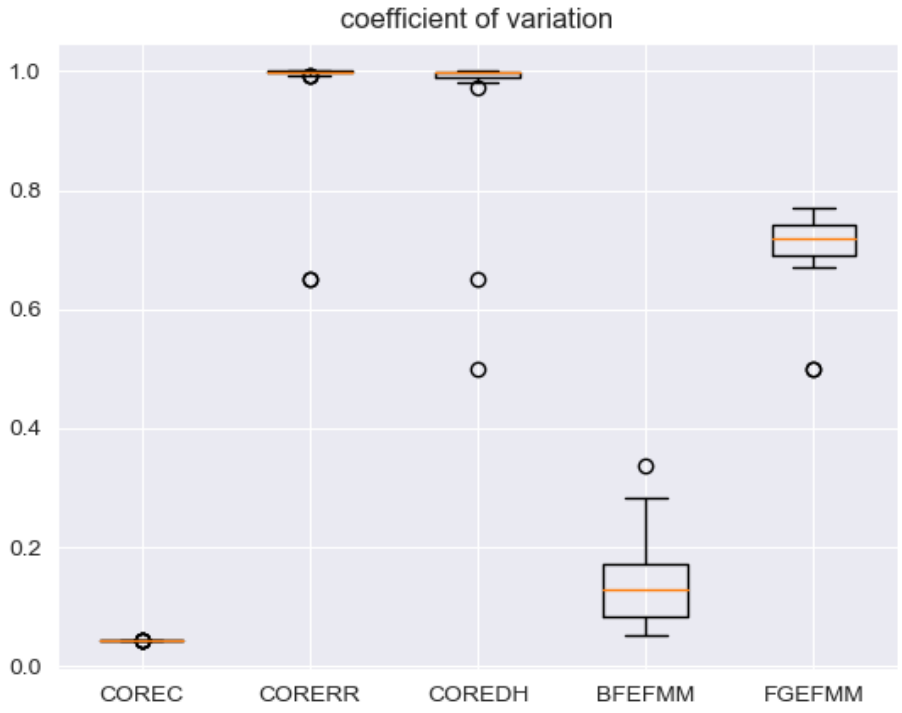


Figure 6.2: Coefficient of Variation Results

at random that didn't manage to fully exploit the potential of algorithms like Brute Force Enumeration and Fine Grained Enumeration the first round of tests was not completely satisfactory. In the second experiments we compared the results obtained by each algorithm when dealing with an increased amount of maximal matches. Then we also combined the results over all the tests to compare the overall performance of each algorithm.

Figures 6.3 and 6.4 show a more predictable behaviour of the execution time and coefficient of variation (Y axes) of all the algorithms when the number of maximal matches (X axes) in the inputs stream grows. The first observation is that the results show the expected exponential growth in execution time for each algorithm as a natural consequence of the growth of the complexity of the input. The measure of quality is the speed at which this exponential curve grows, i.e. the more maximal matches it takes to grow the better. Figure 6.3 shows that the centralized solution scale least efficiently; second in the list is Brute Force Enumeration which suffers from the reduced options in distribution control since the non digested data structures generated from CORE force it to distribute

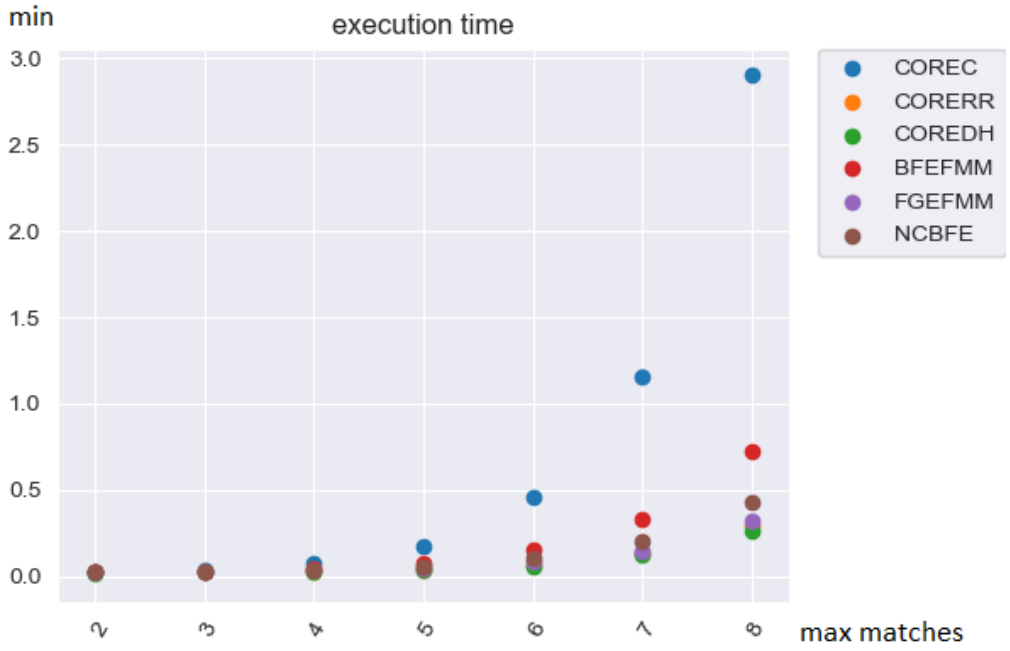


Figure 6.3: Execution Time Results

entire groups of matches without controlling the actual number of them in each group. On the other hand the other algorithms perform very similarly, thus proving that our max-match-based algorithms are a good contender to Round Robin and Double Hashing.

Figures 6.5 and 6.6 show a Box-plot of the same data analyzed before in order to better visualize the difference between the various distribution algorithms in performances. The analysis of Figures 6.5 and 6.6 shows that Hyper-graph algorithm does not feature the best coefficient of variation, but it shows an average execution time very close to the one of Fine Grained Enumeration. From this we can conclude that combining the good distribution algorithm of Fine Grained Enumeration with the capabilities of Hyper-Graph enables performances comparable with the current state of the art.

6.4 Discussion

The results of the first Experiment Round-Robin proves once again to be the best distribution algorithm despite his simplicity. Double-hashing is second with slightly lower

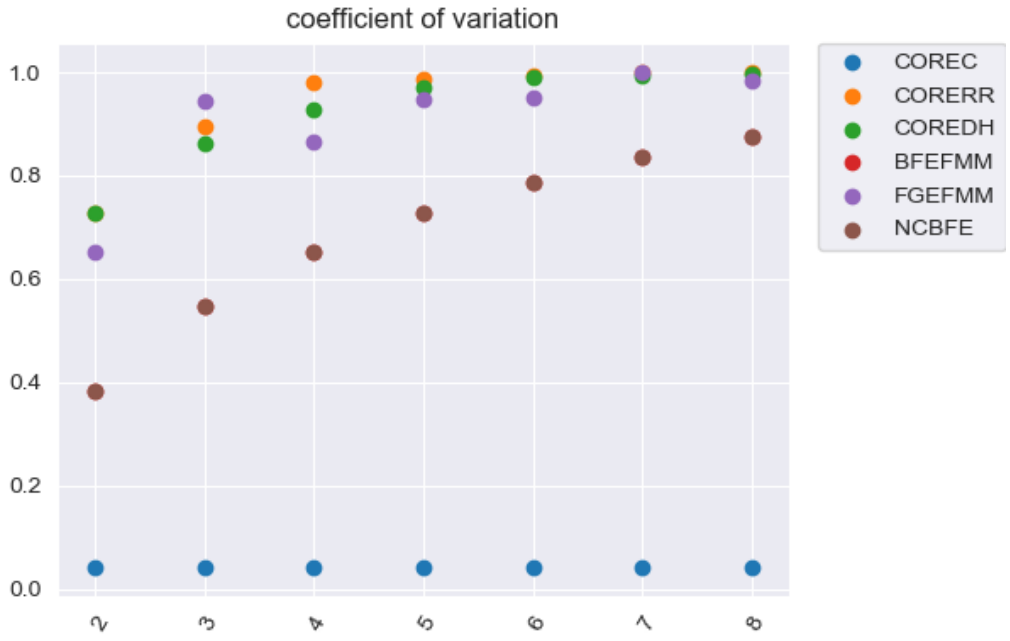


Figure 6.4: Coefficient of Variation Results

performances. As expected Round-Robin maintains a perfect coefficient of variation, since it distributes the matches among computational units as evenly as possible. The same can be told for Double-Hashing, which has proven to be capable of handling bottlenecks pretty well, on the basis of a dynamic allocation on input weight. Fine Grained Enumeration shows potential in that his coefficient of variation is always well above 0.6 while Brute Force Enumerations suffered greatly from the low number of maximal matches found in the small testing environment mostly behaving like a centralized solution which incidentally has been outclassed by all the other distribution strategies proving that Complex Event Recognition Solutions can benefit greatly from a distribution environment.

If we take a look at the execution timers we can see that Fine Grained Enumeration is largely a great contender for Round-Robin. The second position is probably due to the small testing environment. We saw in fact, from the test results, that Fine Grained enumeration shows a fast increase in performances, up to a competitive level, once the number of maximal matches begin to surpass the number of computational units avail-

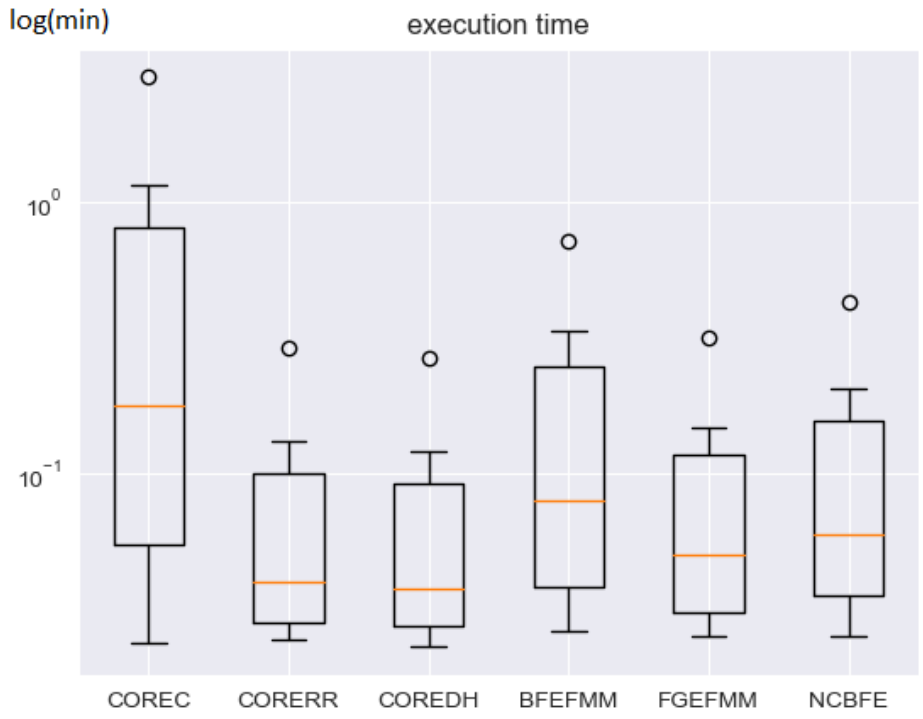


Figure 6.5: Execution Time Comparison

able, before that point the element balancing structure of Round Robin is superior.

The second set of experiment, was in a controlled environment, thus highlighting the potential of the algorithms based on maximal matches. We observed that maintaining a good coefficient of variation is a key factor for boosting the performances of the chosen distribution algorithm, but with Hyper-Graph generation we also showed that avoiding duplicates is not a feature to ignore when looking for best performances since it managed to achieve a good execution time with a less then desirable coefficient of variation.

The final lesson learnt is that, duplicate generation has a substantial impact on the performances of a distribution algorithm and coefficient of variation reflects pretty accurately the ability to perform well of a distribution algorithm. Therefore, focusing our attention on this two factors for the fine tuning of maximal-match-based distribution algorithms we can strive to outclass the current state of the art in the field of Distributed Complex Event Recognition.

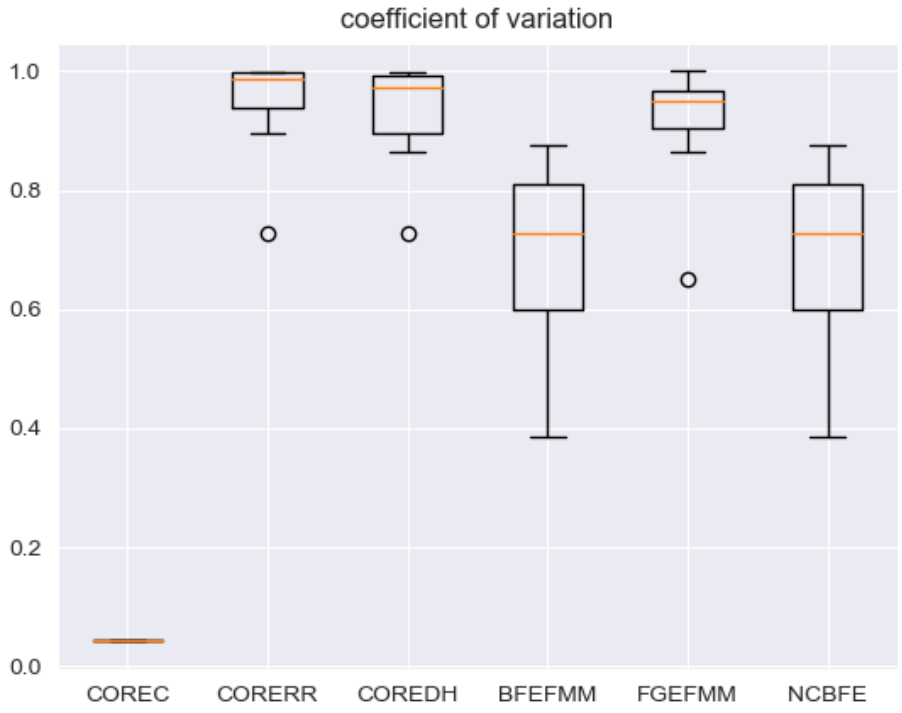


Figure 6.6: Coefficient of Variation Comparison

Chapter 7

Conclusion and Future Work

In a world where the datum is gaining a foothold when it comes to business analysis, and the amount of data produced by almost the majority of nowadays processes is growing exponentially, the field of big data is more relevant then ever before. Complex Event Recognition aims at addressing the problem of the heterogeneity of this data and therefore it covers a very important role in this field. The current state of the art focuses on dealing with this heterogeneity, but fails to address the problems related to the context it is immersed in, i.e. the great amount of data to deal with at any given point in time. Our contribution aims at adapting what are the current state of the art CER solutions in a way such that they can be used in a distributed environment in order to address the problems related to a big data environment with a set of novelty algorithms in order to maximize the synergy between the two worlds. The experiments that we ran so far on our framework have given us the confidence to say that our contribution to the field can improve the performances over the state of the art, and that it is worth extending our research and plan future, more comprehensive, tests.

In the future we aim at analyzing the current results in order to take away where we improved over the current state of the art and what can be done to further improve our algorithms so that we can strengthen their reliability. A second direction of future work is to test Flink-Core in a larger environment. This way we step away from an experimental setting and move onto what can be viewed as a truthful simulation of a possible real world problem. This is done in order to use tools like Flink in their natural habitat and get an

unbiased view of their behaviour. Lastly we hope to cohoperate with the creators of the CORE engine in order to adapt it to a distributed environment so that the process of analyzing the second order variables in the distributed computational units can be done by the same engine that we have used in the centralized portion of our framework.

Bibliography

[1] Marie Heinrich, Joan Tiffany, and Ong Lopez. *Complex Event Processing in the Big Data Era*.

[2] Giampaolo Cugola and Alessandro Margara. *Processing Flows of Information From Data Stream to Complex Event Processing*. ACM Computing Surveys, 2012.

[3] Alejandro Grez, Cristian Riveros, and Martín Ugarte. "A Formal Framework for Complex Event Processing". In: *22nd International Conference on Database Theory (ICDT 2019)*. Ed. by Pablo Barcelo and Marco Calautti. Vol. 127. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 5:1–5:18. ISBN: 978-3-95977-101-6. DOI: 10.4230/LIPIcs.ICDT.2019.5. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10307.

[4] N Giatrakos et al. *Complex Event Recognition in the Big Data era: a survey*. Springer-Verlag, 2019.

[5] Alejandro Grez et al. *A Second-Order Approach to Complex Event Recognition*. 2017.

[6] Eugene Wu, Yanlei Diao, and Shariq Rizvi. *High-Performance Complex Event Processsing over Streams*. SIGMOD, 2006.

[7] Gianpaolo Cugola and Alessandro Margara. *Complex event processing with T-REX*. 2012.

[8] Nikos R. Katsipoulakiss, Alexandros Labrinidis, and Panos Chrysanthis. *A holistic view of stream partitioning costs*. Proceedings of the VLDB Endowment, 2017.

[9] Apache. *Spark*. URL: https://spark.apache.org/docs/latest/index.html.

[10] Apache. *Flink*. URL: https://flink.apache.org/.

[11] Lars Brenna et al. *Distributed event stream processing with non-deterministic finite automata*. 2009.

[12] Lars Brenna et al. "Cayuga: A High-Performance Event Processing Engine". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. Beijing, China: Association for Computing Machinery, 2007, pp. 1100–1102. ISBN: 9781595936868. DOI: 10.1145/1247480.1247620. URL: https://doi.org/10.1145/1247480.1247620.

[13] Cagri Balkesen et al. "RIP: Run-Based Intra-Query Parallelism for Scalable Complex Event Processing". In: *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*. DEBS '13. Arlington, Texas, USA: Association for Computing Machinery, 2013, pp. 3–14. ISBN: 9781450317580. DOI: 10.1145/2488222.2488257. URL: https://doi.org/10.1145/2488222.2488257.

[14] Tyler Akidau et al. *The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.* Proceedings of the VLDB Endowment, 2015.

[15] Buğra Gedik. *Partitioning functions for stateful data parallelism in stream processing.* The VLDB journal, 2014.

[16] Yossi Azar et al. *Balanced Allocation.*

[17] Anis Nasir et al. *When two choices are not enough: Balancing at scale in distributed stream processing.* 2016.

[18] Gianmarco De Francisci et al. *The power of both choices Practical load balancing for distributed stream processing engines.* 2015.

[19] Nicolo Rivetti et al. *Efficient Key Grouping for Near-Optimal Load Balancing in Stream Processing Systems.* 2015.

[20] Google. *Guava.* URL: `https://github.com/google/guava`.