August 2016

# Complex Event Processing as a Service in Multi-Cloud Environments

Wilson A. Higashino
*The University of Western Ontario*

Supervisor
Dr. Miriam A. M. Capretz
*The University of Western Ontario*

Graduate Program in Electrical and Computer Engineering
A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

# Abstract

The rise of mobile technologies and the Internet of Things, combined with advances in Web technologies, have created a new Big Data world in which the volume and velocity of data generation have achieved an unprecedented scale. As a technology created to process continuous streams of data, Complex Event Processing (CEP) has been often related to Big Data and used as a tool to obtain real-time insights. However, despite this recent surge of interest, the CEP market is still dominated by solutions that are costly and inflexible or too low-level and hard to operate.

To address these problems, this research proposes the creation of a CEP system that can be offered as a service and used over the Internet. Such a CEP as a Service (CEPaaS) system would give its users CEP functionalities associated with the advantages of the services model, such as no up-front investment and low maintenance cost. Nevertheless, creating such a service involves challenges that are not addressed by current CEP systems. This research proposes solutions for three open problems that exist in this context.

First, to address the problem of understanding and reusing existing CEP management procedures, this research introduces the Attributed Graph Rewriting for Complex Event Processing Management (AGeCEP) formalism as a technology- and language-agnostic representation of queries and their reconfigurations. Second, to address the problem of evaluating CEP query management and processing strategies, this research introduces CEPSim, a simulator of cloud-based CEP systems. Finally, this research also introduces a CEPaaS system based on a multi-cloud architecture, container management systems, and an AGeCEP-based multi-tenant design.

To demonstrate its feasibility, AGeCEP was used to design an autonomic manager and a selected set of self-management policies. Moreover, CEPSim was thoroughly evaluated by experiments that showed it can simulate existing systems with accuracy and low execution overhead. Finally, additional experiments validated the CEPaaS system and demonstrated it achieves the goal of offering CEP functionalities as a scalable and fault-tolerant service. In tandem, these results confirm this research significantly advances the CEP state of the art and provides novel tools and methodologies that can be applied to CEP research.

**Keywords:** Complex Event Processing, Cloud Computing, Multi-Cloud, Container Management System, Simulation, Graph Rewriting

# Co-Authorship Statement

The work presented in Chapters 4 and 5 (Attributed Graph Rewriting for CEP Management) has been developed in collaboration with Dr. Cédric Eichler from INSA Centre Val de Loire / LIFO Laboratory. Dr. Eichler's main contribution was the introduction of graph rewriting rules as a formalism to express reconfiguration actions of Complex Event Processing queries. In addition, Dr. Eichler contributed with discussions about the scope and goals of the presented formalism, as well as with his expertise about autonomic computing.

# Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Miriam Capretz for her supervision, mentoring, and for welcoming me in a new country and making me feel like home. I will always be grateful for the many opportunities you gave me and for the countless hours you spent making me a better person and professional.

I would also like to thank my co-supervisor Dr. Luiz Fernando Bittencourt for having me half way in the program, and for his support and orientation through these years. You are a great source of inspiration.

To my parents, who have always believed and encouraged me through my journey.

To my beloved Ana, who has always supported me and has never left my side even in the most difficult moments.

To my brother and my nephew, who inspires me and makes me laugh every Sunday.

To Dr. Beatriz Puzzi and Mr. Carlos Taube for your assistance and encouragement.

To Dr. Cédric Eichler, Dr. Thierry Monteil, and Dr. Ernesto Exposito for the incredible collaboration in this thesis.

To Powersmiths International Corp. for the great project in which I had the opportunity to work, and for providing the data used in many experiments from this thesis.

To all my colleagues from TEB 346. During all these years, you came from all around the world and without you this would not have been possible.

To all of you who should be thanked, but are not in this symbolic list. You all know who you are.

# Contents

# List of Figures

# List of Tables

# List of Appendices

# List of Acronyms

| | |
|---|---|
| **ADAG** | Attributed Directed Acyclic Graph |
| **AGeCEP** | Attributed Graph Rewriting for Complex Event Processing Management |
| **API** | Application Program Interface |
| **CEP** | Complex Event Processing |
| **CEPaaS** | Complex Event Processing as a Service |
| **CMS** | Container Management System |
| **CQL** | Continuous Query Language |
| **CRUD** | Create Read Update Delete |
| **DAG** | Directed Acyclic Graph |
| **DBMS** | Database Management System |
| **EPN** | Event Processing Network |
| **IaaS** | Infrastructure as a Service |
| **KB** | Knowledge Base |
| **KCL** | Amazon Kinesis Client Library |
| **MAPE-K** | Monitor Analyze Plan Execute - Knowledge |
| **MQO** | Multi-Query Optimization |
| **PaaS** | Platform as a Service |
| **QAM** | Query Analyzer and Manager |
| **QLM** | Query Lifecycle Management |
| **QoS** | Quality of Service |
| **REST** | Representational State Transfer |
| **SaaS** | Software as a Service |
| **SP** | Stream Processing |
| **SQL** | Structured Query Language |
| **SQO** | Single-Query Optimization |
| **VM** | Virtual Machine |

# Chapter 1

# Introduction

The emergence of Big Data has been profoundly changing the way enterprises and organizations store and process data. Clearly, the sheer amount of data created by mobile devices, the Internet of Things (IoT), and a myriad of other sources cannot be handled by traditional data processing approaches [64]. Simultaneously, there is also a consensus that obtaining insights and generating knowledge from these Big Data can bring a competitive advantage to organizations using them. Therefore, these organizations, along with the research community, have been actively pursuing new ways of leveraging Big Data to improve their businesses.

According to the most commonly accepted definition, Big Data is characterized by four Vs [121]: volume, velocity, variety, and veracity. Volume refers to the quantity of data, and velocity concerns the speed at which data are generated and need to be processed. Variety refers to the diversity of data types and formats, and veracity relates to the accuracy and reliability of the data [65]. Datasets can be "big" in any of these directions and, most often, in more than one. For instance, volume and velocity are closely related, as fast data generation usually results in a massive amount of data to be stored and processed.

As technologies created to process continuous streams of data with low latency, Complex Event Processing (CEP) and Stream Processing (SP) have often been related to the velocity dimension and used in the Big Data context. The processing model of CEP and SP systems are both based on continuously running user-defined queries that dictate operations to be performed on fast and often distributed input streams. The goal is usually to obtain real-time insights and to enable prompt reaction to them. Because of the generality of this model, these systems have been applied to a variety of use cases ranging from simple monitoring to highly complex financial applications such as fraud detection and automated trading [65].

At about the same time, cloud computing has also emerged as a disruptive computational paradigm for on-demand network access to a shared pool of computing resources such as servers, storage, and applications [113]. From the infrastructure point of view, cloud comput-

ing environments are leveraged to provide the low-latency and scalability needed by modern applications, including CEP and SP systems [69, 128]. From the business perspective, cloud computing provides an agile way to access infrastructure resources and services without large upfront investments and preparation time.

Despite the paradigm shift brought about by cloud computing, today the CEP and SP market is still dominated by a few proprietary solutions [86, 123, 139] that require huge investments for acquisition and do not provide the flexibility that users need. Alternatively, on the other side of the spectrum many companies adopt open-source, low-level systems [17, 18, 153], which demand intense technical training and have high operating costs.

To address these problems, this research proposes the creation of a CEP system that can be offered in the *Software as a Service* (*SaaS*) model. This *CEP as a Service* (*CEPaaS*) system would enable users to access CEP functionalities on-demand, over the Internet, and with minimal management effort. However, offering such a service involves many challenges that are not addressed by current CEP state of the art. This thesis discusses these challenges further and presents a series of contributions towards the development of such a system.

## 1.1 Motivation

The use of CEP and SP solutions to analyze streaming data and obtain real-time insights has the potential to profoundly change enterprises and make them more agile and responsive. This impact has been confirmed by a recent survey, which estimated a market of $500 million in 2015 for the so-called *streaming analytics* solutions, with the potential to reach $2 billion in 2020[1]. Nevertheless, despite this growing interest, this market is still dominated by a few solutions that are costly and inflexible or too low-level and hard to operate.

The offering of *Software as a Service* (*SaaS*) is a recent paradigm shift that has been at the core of the *cloud computing* revolution. In the SaaS model, software traditionally only available as proprietary packages are now offered as services that can be consumed on-demand and with minimal management effort. Likewise, even the computational infrastructure normally required by enterprise systems can now be consumed as always-available services. This offering of *Infrastructure as a Service* (*IaaS*), in conjunction with SaaS, brings many benefits to enterprises, including reducing their capital investments, mitigating risks, and focusing on innovation and differentiation.

Given this scenario, it is only natural to imagine the offering of *CEPaaS* as a way to bring to CEP users the many advantages of the services model, such as:

[1] http://www.researchandmarkets.com/research/mpltnp/streaming

- No up-front investment in hardware and software infrastructure.
- Low maintenance cost, as the service model reduces the need for infrastructure monitoring and maintenance.
- Constant upgrades, mostly without interruption and at no charge.
- Ubiquitous access using the Internet.

Nevertheless, such CEP services either do not exist today or are very limited in their nature, which can be tracked to the many challenges involved in developing them and the lack of appropriate solutions in the current state of the art.

The first of these challenges is related to *understanding* current systems and *reusing* results that already exist in the form of algorithms and management procedures. The current CEP research landscape is still young and fragmented. A large variety of solutions exist, but they often use inconsistent terminology and different query definition languages. Consequently, most ongoing research is performed in the context of specific systems and languages. In particular, algorithms and procedures aimed at managing the user queries have often been developed in such a system-specific fashion that they cannot be easily generalized and applied to other contexts.

The second challenge is related to *evaluating* and *comparing* CEP query processing and management approaches. Today, this problem acquires even more challenging characteristics because most modern CEP systems use cloud environments as their runtime platform. In this type of environment, validating management procedures in the required Big Data scale is a research problem *per se*. For example, cloud environments are subject to variations that make it difficult to reproduce the environment and conditions of an experiment [56]. Moreover, setting up and maintaining large cloud environments are laborious and error-prone, and may be associated with a high financial cost. Finally, there are also many challenges related to generating and storing the volume of data required by Big Data experiments.

Finally, many technical difficulties are associated with the *design* and *implementation* of a *CEPaaS* system. For instance, low latency is essential to many CEP use cases, but it is difficult to achieve in a service environment because there is no control over the locations of event sources and consumers. Such a *CEPaaS* system is also inherently multi-tenancy, which makes fault-tolerance essential because an outage can affect many customers and damage the provider's reputation. In addition, multi-tenancy indicates that some sort of resource control and isolation is necessary to avoid interference between workloads from different queries. Finally, by offering it to anyone with Internet access, the system is expected to be highly scalable in the number of queries and to be usable by a wide spectrum of users.

## 1.2 Contributions

This research provides a *series* of contributions aimed to solve the challenges mentioned and, ultimately, to enable the development of a *CEPaaS* system.

To solve the challenge of *understanding* current systems and *reusing* existing results, this research introduces the **A**ttributed **G**raph **Re**writing for **C**omplex **E**vent **P**rocessing Management (*AGeCEP*), a formalism that provides technology- and language-agnostic representations of queries and of reconfiguration actions that can be applied to transform these queries.

In *AGeCEP*, queries are modelled as attributed graphs and described by a standard set of attributes, whereas reconfiguration actions are expressed by graph rewriting rules. In conjunction, these models provide a common foundation that can be used to represent queries written in different languages and to express generic CEP management procedures. By doing so, these procedures can be integrated into any modern cloud-based CEP system that uses *AGeCEP* as its underlying formalism. In particular, *AGeCEP* is especially suitable to represent self-management policies that can be used to manage and control autonomic CEP systems.

This research harnesses *AGeCEP* expressiveness by adopting it as the formal foundation of the other contributions. To demonstrate its feasibility, *AGeCEP* is also used to design an autonomic manager and to define a selected set of self-management policies. In addition, *AGeCEP* viability is verified through performance measurement experiments, which show that 100 queries can be processed and rewritten by graph rewriting rules in less than one second.

The second major contribution of this research is *CEPSim*, a simulator for cloud-based CEP systems. Traditionally, simulators have been used in different fields to overcome difficulties related to the execution of repeatable and reproducible experiments [33, 34, 92, 119]. *CEPSim* aims to bring simulation capabilities to CEP and to solve the challenges of *evaluating* and *comparing* different query processing and management strategies.

*CEPSim* uses a query model based on *AGeCEP* and introduces simulation algorithms based on a novel abstraction called *event sets*. *CEPSim* can model different types of clouds, including public, private, hybrid, and multi-cloud environments, and simulate execution of user-defined queries on them. Moreover, it can also be customized with various operator placement and scheduling strategies. These features enable architects and researchers to analyze the scalability and performance of cloud-based CEP systems and to easily compare the effects of adopting different query processing strategies.

A large set of experiments was executed to analyze *CEPSim*. Results show that *CEPSim* can estimate the latency and throughput of CEP queries running on a real system with less than 5% error in most cases. Moreover, results also demonstrate that *CEPSim* simulates 100 queries running for 5 minutes in approximately 7 seconds and using less than 40 MB of memory.

The last major contribution of this research is the *design* and *implementation* of a *CEPaaS* system. The proposed design leverages multi-cloud environments to increase the system availability and to explore the geographical diversity of cloud datacentres, creating the possibility of strategic deployment in which system resources are positioned close to event producers and consumers. Moreover, the design also explores container-based virtualization and container management systems (CMS) to control the deployment and execution of system components.

In the *CEPaaS* system, every component, including user queries, is encapsulated in an application container that is managed and scheduled by a CMS. By doing so, it is possible to have a fine control over the resource usage of the components and to isolate their execution. Moreover, the CMS also handles fault-tolerance and scalability of the containers, facilitating the implementation of these requirements at the system level and simplifying the system operation. Finally, the proposed *CEPaaS* system explores the idea of vertex and query templates as a way to define queries and to enable the definition of custom event processing logic. In practice, queries defined in such a way are transformed into an *AGeCEP*-based representation and executed by an actor-based execution engine.

By putting all contributions together, the *CEPaaS* system was designed over a strong formal foundation and, at the same time, based on efficient algorithms and strategies that have been tested and evaluated in simulations. This approach, in tandem with the chosen architecture, enabled the creation of a robust and scalable *CEPaaS* system that successfully brings the advantages of the services model to CEP. Experiments executed to validate the system show that the strategic deployment enabled by the multi-cloud architecture reduces query latency up to 60%. Further experiments also indicate that queries are properly isolated from each other and can quickly recover from failures.

Notwithstanding, note that each one of the contributions presented are valuable by themselves and can be used separately from the others. Therefore, either by considering these contributions in isolation or together, this research significantly advances the CEP state of the art and provides novel tools and methodologies that can be applied in the context of CEP research and development.

## 1.3 Thesis Organization

This thesis is organized as follows:

- Chapter 2 presents core background concepts that are necessary to understand the remaining text. It starts with definitions of stream processing and complex event processing and a discussion about how they differ from each other. Following that, it presents the

nomenclature used in this research and the query lifecycle management concept. In the second part, this chapter also discusses cloud computing and system architectures based on multiple clouds. Finally, it concludes with a examination of container-based virtualization, application containers, and container management systems. These are recent technological trends that are used by the *CEPaaS* implementation.

- Chapter 3 presents an extensive review of research related to this thesis. It starts with a review of traditional historical systems, which established most of the basic concepts and terminology used by current CEP research. Next, it discusses the plethora of modern systems, dividing them into MapReduce-based, open source, and cloud-based systems. Each system is discussed briefly and its main contributions are highlighted. Moreover, the chapter discusses current systems that offer CEP-related services, or are based on multi-cloud architectures. Finally, the chapter concludes by discussing CEP formal models and cloud computing simulators, which are related to the *AGeCEP* and *CEPSim* contributions.

- Chapter 4 presents the main concepts of the *AGeCEP* formalism. First, it examines the *AGeCEP* assumptions and design principles. Second, it presents a novel classification of CEP operators focused on their reconfiguration capabilities. This classification serves as the basis for the standard set of operator attributes used by *AGeCEP* and constitutes another major contribution of this research. Finally, Chapter 4 discusses the formalism itself, including the notation used and examples that illustrate its basic concepts.

- Chapter 5 presents a thorough evaluation of the *AGeCEP* formalism. First, *AGeCEP* is used to design an autonomic manager. Based on this design, a generic procedure to express operator placement procedures and a selected set of self-management policies are discussed. Finally, the performance of graph rewriting rules is assessed by experiments that analyze the time needed to reconfigure queries.

- Chapter 6 presents the *CEPSim* simulator. It starts with a discussion about the basic *CEPSim* architecture and how *AGeCEP* is used to internally represent the simulated queries. Following that, it explains in detail how the simulation algorithms work, and how the simulator can be customized with user-defined operator scheduling and operator placement algorithms. Finally, this chapter presents a series of experiments that validate *CEPSim* in different scenarios, assess the execution time and memory consumption of simulations, and analyze the effects of various parameters in the simulator performance.

- Chapter 7 presents the design and implementation of the *CEPaaS* system. First, a system overview is presented, including discussions about the main system components

and about how the system leverages CMS and multi-cloud architectures to provide fault-tolerance and scalability. Next, this chapter discusses the concept of vertex and query templates, and how they are employed by users to define queries. Following this, implementation details are presented, including specifics of the query execution engine. Finally, the *CEPaaS* system is evaluated regarding the effects of multi-cloud placement in the end-to-end query latency and regarding the fault-tolerance provided by the CMS.

- Chapter 8 finalizes the thesis by summarizing its contributions and discussing areas for further research.

# Chapter 2

# Background

This chapter introduces background concepts used in the thesis. It starts with an overview of complex event processing and stream processing, and how they relate with each other. Section 2.2 introduces cloud computing concepts, with emphasis on multiple cloud architectures and how they can be used to improve the quality of services offered to users. In the same context, container-based virtualization and container management systems are discussed in Section 2.3. Containers are an essential part of the *CEPaaS* system discussed in Chapter 7. Finally, Section 2.4 presents autonomic computing concepts and the MAPE-K framework. This framework is used for defining an autonomic manager in Chapter 5 and is also the basis of the *CEPaaS* system management module presented in Chapter 7.

## 2.1 Event Processing

In recent years, many applications that require processing of high-volume continuous streams of data have emerged. These applications range from simple alarm mechanisms to highly complex trading systems that analyze thousands of transactions per second. For many years, these applications have been implemented using *ad-hoc* solutions, which have led to high development and maintenance costs and limited reuse opportunities.

It is in this context that Complex Event Processing (CEP) and Stream Processing (SP) technologies have emerged. CEP and SP share similar goals, as both are concerned with processing continuous data flows coming from distributed sources to obtain timely responses to queries [41]. Nevertheless, they have been simultaneously developed for years by researchers with different backgrounds [41], a situation that has resulted in duplicated and inconsistent vocabularies as well as fuzzy distinctions among their concepts.

In order to overcome these inconsistencies, the next two subsections present conceptualizations of SP and CEP. Following that, the main differences between them are discussed, and the

terminology used in this research is introduced.

### 2.1.1 Stream Processing

Stream Processing (SP) or Event Stream Processing is a set of techniques aimed at processing continuous and potentially unbounded streams of data within strict time constraints through long-running and continuous (standing) queries [58]. SP systems are also known as Stream Processing Engines or Data Stream Management Systems; their origin is often associated with the Database Management Systems (DBMS) research community, which created the first SP systems as a response to stream-processing requirements that could not be satisfied by traditional relational databases.

The traditional DBMS paradigm is based on queries that are explicitly initiated by users and applications, whereas the SP paradigm requires active update of continuous query results. In addition, DBMSs store data before processing them, which may limit dataset size and may incur additional latency in the processing pipeline. On the other hand, most SP applications are not interested in persisting streams and require low-latency response to queries.

Early research projects, such as the Aurora [1] and STREAM [19] systems, established the basis of the discipline and have influenced most subsequent research. Later, Stonebraker *et al*. [142] listed the eight main requirements of real-time stream processing systems:

1. Process data on-the-fly, using an active query model;
2. Use a query language based on SQL, with additional constructs appropriate to stream processing;
3. Handle data streams containing delayed, out-of-order, or lost items;
4. Generate predictable and repeatable outcomes;
5. Integrate streaming data with state and historical data;
6. Guarantee data safety and availability;
7. Partition and scale applications automatically;
8. Process data and respond instantaneously.

Most of these requirements still serve as guidelines for modern SP research and development, yet they acquired even more challenging characteristics with the advent of Big Data.

### 2.1.2 Complex Event Processing

Complex Event Processing (CEP) was originally defined as "*a set of tools and techniques for analyzing and controlling the complex series of interrelated events that drive modern distributed information systems*" [106]. CEP systems aim to detect complex patterns of events to identify important situations and react promptly to them. These systems normally accept user definitions of patterns that express complex relationships among events, including the use of aggregation, correlation, and time-sequencing operators.

The term CEP was first used in 2002 in the seminal book by Luckham [106], which justified the need for CEP by noting that existing tools and techniques could not manage and understand the numerous flows of information (events) that were driving enterprise systems of that time. To enable better understanding of events generated by these systems, two main concepts were introduced:

- *Event causality*: some events cause others, and tracking these relationships helps to determine the root cause of events. According to the nature of events involved in such relationships, causality can be further classified as *horizontal* or *vertical*. The former refers to causality between events at the same conceptual level; for example, an email causes a response message, and a ping network packet causes a reply. The latter refers to the fact that events generate other events at a "lower-level" layer; for instance, a business process generates requests to many systems, which in turn generate many network packets.

- *Event aggregation*: low-level events can be aggregated into higher-level business-related events. The motivation for aggregation is twofold: first, many monitoring tools can only observe low-level events and analyze them in isolation, providing very little information for business-level decision making; second, many events are not explicitly generated, and their occurrences must be inferred from other events. For example, policy or regulation violations are very important for enterprises, but can be detected only if lower-level events fail to satisfy specific rules.

According to Luckham, there are two main differences between CEP and SP [107]:

- SP systems process data streams, or sequences of events ordered by time, whereas CEP can process partially ordered sets (*posets*) of events. These *event posets*, also known as *event clouds*, can be simultaneously generated by many IT systems and sources. Therefore, an event cloud can potentially include many event streams. For example, the temperature readings of a specific weather station form an event stream, but the whole

weather forecasting system generates an event cloud composed of readings from many stations and sensors, other systems, and analysis results.

- Most SP systems use SQL-like queries aimed at fast processing and at performing calculations on data streams. CEP systems, on the other hand, are more focused on detecting complex patterns of events that include the notions of causality and aggregation.

Other researchers, such as Bass [26] and Cugola and Margara [41], have made similar distinctions, yet they all acknowledge the similarities between CEP and SP.

### 2.1.3 Concepts and Terminology

This research defines CEP as the "*processing of continuously flowing data from geographically distributed sources with unpredictable rate to obtain timely responses to complex queries*" [41]. This is a broad definition that encompasses both CEP and SP, and was originally presented by Cugola and Margara to describe *Information Flow Processing* systems. In addition, this research uses a terminology based on the Event Processing Technical Society (EPTS) glossary [108] and Etzion *et al*. [50], which originated from the CEP literature. This terminology has been chosen because its terms are also broadly defined and encompass most SP concepts. Moreover, it prevents the creation of new terms for established ideas. Most terms are used as is, but some are redefined to avoid conflict with other concepts presented in this research.

Figure 2.1 shows the main components of a system based on an *event processing architecture*. *Event producers*, also known as *sources*, introduce events into the *CEP system*. Conversely, *event consumers*, or *sinks*, receive events from it. Here, the term *event* is used very broadly as the computational representation of something that happened in the context of interest. For instance, an event can represent a sensor reading, the CPU load of a server, or the creation of a new user on a website.

The CEP system is the main component of the architecture, and its goal is to act upon input events to produce output events according to user-defined *queries* or *processing rules*. Collectively, producers, consumers and the CEP system form an *event processing network* (EPN).

Queries[1] represent the processing that takes place between producers and consumers. For instance, a query can detect anomalies in sensor readings and warn building administrators, or refresh a dashboard with new CPU load data. Logically, queries are implemented by a flow of *query operators* that receive one or more event streams as input and generate other streams as

[1]This research uses the term *queries* instead of *processing rules* to avoid conflict with graph rewriting *rules*, presented in Chapter 4.

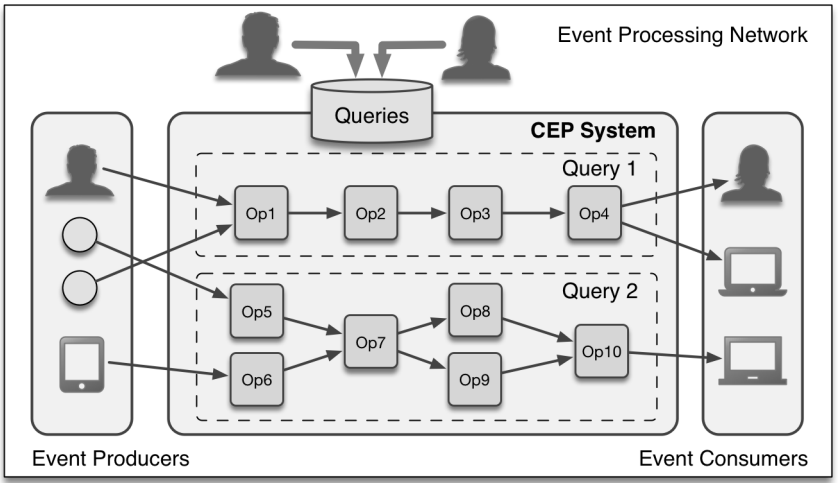



Figure 2.1: CEP terminology.

output. Depending on its goals, a query operator can represent different kinds of processing logic, such as filters, joins, or anomaly detectors.

### 2.1.4 Classification of CEP systems

The myriad of CEP systems currently available differ in many aspects. This section elaborates on three classification criteria that are especially important for this thesis: deployment model, interaction model, and query definition language. A more complete classification can be consulted in Cugola and Margara [41].

#### Deployment Model

A CEP system *deployment model* refers to its runtime architecture, or how the system components are distributed over the set of available servers at runtime. The first CEP systems used a *centralized* architecture, in which all queries and system components run in a single server. This architecture rapidly reached its limits and led to the development of *distributed* CEP systems, in which user queries and system components run in more than one server and communicate using a network.

Distributed architecture enables CEP systems to improve their scalability by running queries on different servers. In most cases, it also implies that a larger number of events can be processed by a single query because its operators are also distributed. Moreover, it can also translate to better availability because distributed CEP systems usually implement fault tolerance mechanisms, such as replication and standby components.

It is common to classify distributed CEP systems further as *clustered* or *networked* accord-

ing to the network type that connects the servers. In a *clustered* system, servers are connected to the same high-speed low-latency network and are geographically close. Conversely, in a *networked* system, part of the communication links are implemented through high-latency networks such as the Internet.

### Interaction Model

The term *interaction model* refers to the communication style used by EPN components to interact with each other. More specifically, three types of interaction are characterized: from event producers to the CEP system; between query operators; and from the CEP system to event consumers. These interactions define the system *observation model*, *forwarding model*, and *notification model* respectively. For all these models, a *push* and a *pull* style are defined. In the former, the event origin proactively sends data to their destination, whereas in the latter, the event destination pulls data from the origin.

### Query Definition Language

In most CEP systems, users use a proprietary *query definition language* to define queries. Despite standardization efforts [88], a great variety of languages are still in use today. Cugola and Margara [41] classified existing languages into three groups:

- *Declarative*: the expected results of the computation are declared, often using a language similar to SQL. The Continuous Query Language (CQL) [20] is the most prominent representative of this category. The following is a CQL query example:

```
Select IStream(*)
From S1 [Rows 40000]
     S2 [Range 600 Seconds]
Where S1.A = S2.A
```

- *Imperative*: the computations to be performed are directly specified as a sequence of operations, usually by an imperative programming language or visually as a graph of operators. The Aurora Stream Query Algebra (SQuAl) [1] inspired most languages in this category. Figure 2.2 shows an SQuAl query.

- *Pattern-based*: queries are defined by firing conditions and a set of actions that are executed whenever these conditions are met [41]. The firing conditions contain patterns of events that usually include sequence, causality, and composition operators. The

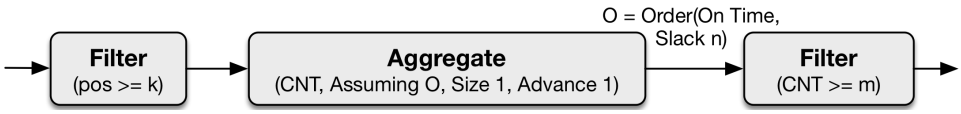



Figure 2.2: An SQuAl query example (adapted from Abadi *et al*. [1]).

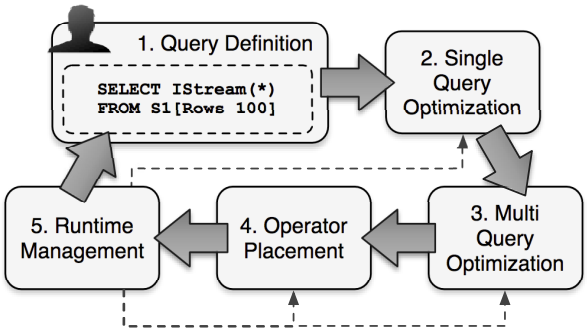



Figure 2.3: Query lifecycle.

Rapide [105] and TESLA [40] languages are examples of this category. The following is a query example written in Rapide:

```
(RFQId ?id, Time ?T1) (RFQ(?Id) at ?T1 ->
[* rel ˜] (Time ?T2) Bid(RFQId is ?Id) at ?T2
where ?T2 <?T1 + Bnd);
```

### 2.1.5 Query Lifecycle

Query lifecycle management (QLM) can be defined as the set of tasks necessary to manage a query from the time of its definition by a user up to its execution and subsequent retirement.

In this thesis, the query lifecycle is defined by the five major steps illustrated in Figure 2.3. The cycle starts when the user creates queries using a CEP query definition language. Each query is submitted to the CEP system, where it is first analyzed and optimized in isolation (*Single-Query Optimization*) and then in conjunction with other running queries (*Multi-Query Optimization*).

In the *Operator Placement* step, the query operators are mapped to a subset of the available computing resources and start executing. Following, during *Runtime Management*, the system maintains the query execution, responding to context changes such as hardware and software failures. This step is typically the most important and lasts the longest because, unlike database queries, a CEP query runs continuously for a specified period of time or until the user decides to shut it down. In addition, during runtime, the system may need to re-optimize and re-place queries when runtime conditions change. This dependency is represented in Figure 2.3 by

dashed arrows from box 5 to boxes 2, 3, and 4. Finally, based on the results obtained by the query, users can decide to refine it or to create one or more new queries, which originates a new cycle.

In the following subsections, each of the QLM steps is detailed.

### Query Definition

Query definition is the step in which users define the CEP queries they want to execute. As mentioned in Section 2.1.4, each system usually has its own query language that is used for this purpose. In addition, the way that users interact with the system to define and submit queries differs enormously from system to system. For example, commercial CEP packages such as Oracle Stream Explorer [123] and Software AG Apama [139] have full-fledged interfaces that help define queries and monitor their execution. On the other hand, many academic [128] and lower-level systems [18] provide only programming language APIs that are mostly targeted to software developers.

### Single-Query Optimization

Single-query optimization (SQO) is the action of modifying a query to improve its efficiency while keeping its functional properties unchanged. The "efficiency" of a query is usually measured with respect to some optimization criterion such as processing latency, CPU usage, or network consumption. This step is essential because it reduces the need for technical knowledge by users: non-optimized queries are corrected before they are run, reducing their impact on the system.

SQO is executed right after a new query is created and registered. Consequently, this step assumes no *a priori* knowledge about available resources or about the state of the network and servers.

### Multi-query Optimization

Multi-query optimization (MQO) consists of finding overlaps (common partial results) between queries and merging them into a single query while maintaining their logical separation. This step usually optimizes the same criterion as the single-query optimization step. MQO can be executed as a separate step when a new query is created or periodically to take into account modifications in the underlying queries. In both cases, one of the greatest difficulties is to decide which queries should be considered in the analysis.

### Operator Placement

Operator placement is the step in which a query execution is mapped into the set of available computing resources. In the context of distributed and cloud-based CEP systems, this usually translates into determining the number and types of servers required and how the queries should be split among multiple processors.

This step is executed during initial system deployment, when a new query is registered, and in general whenever a reconfiguration requires a placement decision. For instance, when an operator is duplicated to parallelize its execution, the placement routine is called to decide where the new operators should be deployed. Because of this variety of scenarios in which placement is used, it is common to use different approaches to deal with incremental and global placement decisions, e.g. placement of a new operator versus placement of all running queries. For more information about operator placement strategies, the survey by Lakshmanan *et al*. [97] can be consulted.

### Runtime Management

Runtime management refers to the self-managed evolution of a system at runtime. During this step, queries are reconfigured in response to context changes such as violations of monitored parameters, hardware and software failures, and sudden bursts of events. This step is the most commonly implemented of all the steps in query lifecycle management.

To support proper runtime management, CEP systems usually define and enforce a number of *self-management policies* aiming to improve or to maintain the quality of service for queries. For instance, this may involve duplicating a query operator to parallelize its execution and increase query throughput [38], or moving operators to underloaded servers [70].

The implementation of self-management policies in CEP systems requires two main capabilities: detecting when a reconfiguration is required, and executing reconfiguration actions. The *detection* step frequently involves monitoring system metrics, such as CPU load and operator queue size, and comparing them with some threshold.

The *execution* of reconfiguration actions, on the other hand, can have many different forms and implementations. One possible classification of these actions focuses on their *scope* and coarsely categorizes them as *behavioural* or *structural*. Behavioural actions change operator and system parameters, but do not modify the query or the system structure. Common examples are load shedding [1], buffer resizing [103], and operator prioritization. Conversely, structural actions require adapting the structure of queries and their mapping into system resources. Splitting a query to distribute its execution between two servers is an example of a structural action [38].

## 2.2 Cloud Computing

Cloud computing is at the core of two main ideas presented in this research. First, offering *CEP as a Service* (*CEPaaS*) is a prime example of *Software as a Service* (*SaaS*), which is one of the three main service types offered by cloud providers. Second, the architecture of the *CEPaaS* system proposed in this research is based on the assumption that multiple *Infrastructure as a Service* (*IaaS*) providers exist and have publicly accessible resources spread around the globe.

Because of the strong relationship with this research, the next subsections discuss cloud computing concepts and describe its main benefits. In particular, it is examined how multiple-cloud architectures can be used to improve the quality of offered services. As described later in Chapter 7, the *CEPaaS* system uses a multi-cloud architecture to improve its availability and to reduce query latency.

### 2.2.1 Cloud Computing Definition

The National Institute of Standards and Technology (NIST) provides the most commonly accepted definition for cloud computing [113]:

> "Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

Despite differences in definition, most authors point out similar characteristics for the cloud computing model [131, 146]:

- *Shared pool of resources*: cloud providers usually have a large number of computing resources (CPU, storage, network) that are pooled and shared among customers.

- *Virtualization*: cloud providers make extensive use of virtualization to enable resource sharing. Virtualization is implemented by a software layer which partitions the server hardware into many virtual servers. In practice, this usually translates to a significant increase in the resource utilization level of a datacentre.

- *Elasticity*: cloud services can dynamically change how much of a resource is consumed in response to how much is needed [131]. Therefore, elasticity enables cloud users to allocate enough resources to match their real instantaneous demand instead of overprovisioning for the worst case.

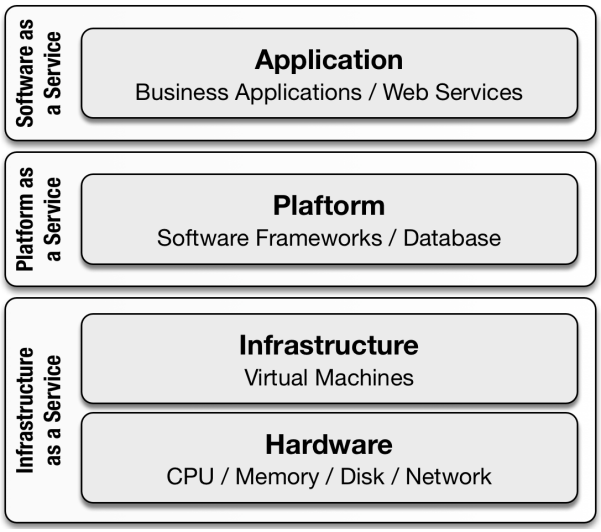



Figure 2.4: Cloud computing architecture (adapted from Zhang *et al*. [154]).

- *Measured service (pay as you go)*: resource consumption is fine-grained metered, enabling flexible billing models. Cloud customers pay according to the type and quantity of resources used, e.g., a small fee per CPU per hour or per gigabyte stored and transmitted.

- *Automation*: interaction with cloud providers occurs through automated APIs and interfaces. This means that resources can be automatically managed and easily integrated with other management software.

For many, the capacity to provision and release resources quickly (elasticity) in tandem with the "pay as you" go model are the key benefits of cloud computing. Together, they enable service providers (cloud users) essentially to eliminate capital expenditure (CAPEX) with infrastructure. Alternatively, these expenses are transformed into operational expenditure (OPEX), which facilitates experimentation with new services and fosters innovation.

### 2.2.2 Cloud Computing Architecture

Figure 2.4 depicts a common representation of cloud computing, in which each layer represents different resources that can be managed by a cloud provider:

- *Hardware*: hardware resources such as servers, storage, and network devices.

- *Infrastructure*: a pool of storage and computing resources that are created by partitioning physical resources using virtualization technologies.

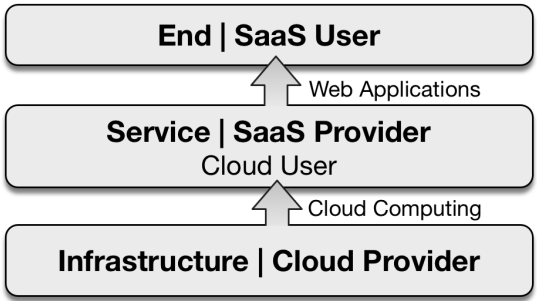



Figure 2.5: Simplified cloud computing architecture (adapted from Armbrust *et al*. [21]).

- *Platform*: frameworks, middleware solutions, and other tools to facilitate application development and deployment.

- *Application*: applications that are deployed in the cloud infrastructure and are consumed by the service's customers.

This representation directly relates to the common taxonomy of cloud services known as "*X as a Service*". According to this taxonomy, cloud services are classified based on the type of resources they offer to users. Therefore, services are typically classified in *Infrastructure as a Service* (*IaaS*), *Platform as a Service* (*PaaS*), and *Software as a Service* (*SaaS*).

A simplified view of this model is shown in Figure 2.5, in which only two roles are discerned: Infrastructure/Cloud Providers (IP) and Service/SaaS Providers (SP). In this view, infrastructure refers more generically to computing and platform resources, which are used by SPs to offer end-user applications to their customers.

Alternatively, cloud providers can also be differentiated according to their deployment and usage model. Based on this perspective, clouds are generally classified as:

- *Public clouds*: clouds in which resources are offered to the general public. This is the original concept, which has the characteristics and advantages already discussed.

- *Private clouds*: clouds in which resources are offered and consumed by a single organization. In this type of cloud, the datacentres are usually located on the company's premises, a measure that enables tighter control of security and governance aspects. The scale of private clouds is generally measured in hundreds instead of thousands of servers, and resources sharing is limited to a company and its partners.

- *Community clouds*: clouds in which resources are shared among a community of users from organizations with shared concerns [113]. For example, a cloud might be shared among an industry vertical or a research consortium.

This taxonomy normally also includes the concept of *hybrid clouds*, which is discussed in the next section.

### 2.2.3 Multiple Cloud Architectures

A recent research topic in the cloud computing field is the simultaneous use of multiple clouds to provide services. The motivation for this approach is normally related to the use of resources, which would otherwise not be available, to improve the quality of the services offered.

The use of resources from multiple clouds can be considered from the perspectives of the cloud (infrastructure) provider or of the cloud customer [67, 96].

A cloud customer may use multiple clouds with the following goals:

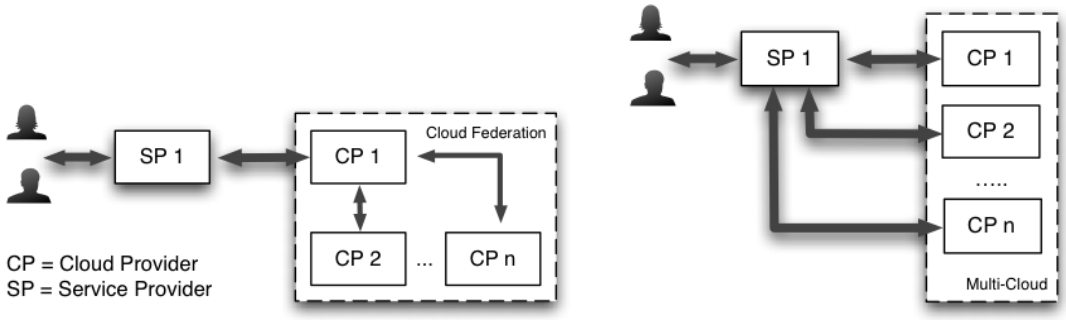
- *Increase availability*: this goal relates to the fact that no single provider can guarantee 100% availability. The recent history of outages [9, 59, 115] in some of the largest cloud infrastructure providers highlights this issue. Therefore, using more than one provider is a way to avoid a single point of failure [21] and to increase infrastructure availability.

- *Overcome providers' restrictions*: cloud providers may have restrictions on the services they offer. For instance, Amazon EC2 [10] has a provisioning limit of 20 server instances per region; if a user needs more, he or she must make an explicit request to the company's support department, and the request is subject to their approval. Other examples of restrictions are the operating systems and hardware offered by a provider, the Service Level Agreement (SLA), and others.

- *Increase application distribution*: cloud providers have datacentres located in a limited number of places. This can be problematic because some regulations and policies require storage and processing of data within national boundaries. In addition, this limited geographic range complicates the deployment of latency-sensitive applications, such as CEP systems, where geographic proximity results in lower latency to end-users. Using more than one provider naturally increases the geographical distribution of the available datacentres.

- *Reduce vendor lock-in*: this goal is related to the use of provider-specific APIs and tools by application developers. As a result of this specificity, some applications can become so intrinsically tied to a provider that the cost of porting them to another provider is greater than the benefits that portability would bring. Use of multiple clouds leads to a reduction in dependency on a single vendor.

A cloud provider, on the other hand, may use a multiple cloud approach with the intent to:




(a) Cloud federation model. (b) Multi-cloud model.

Figure 2.6: Multiple cloud models.

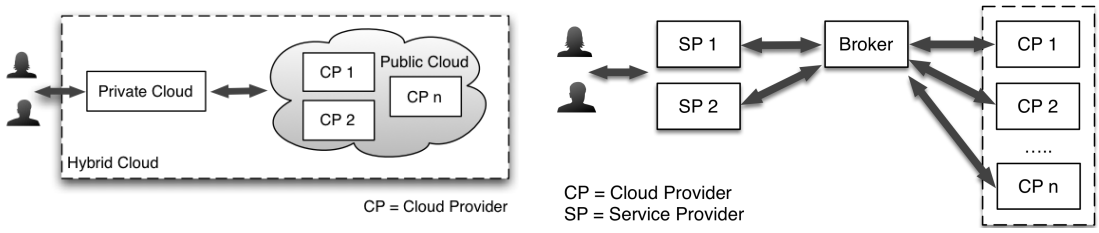
- *Expand on demand*: a provider can use resources from other providers if it reaches the limits of its own infrastructure.
- *Improve the offered SLAs*: a provider can improve the availability of its services by using external resources in case its own fail. Similarly, providers can offer resources located in places where they do not have a physical presence, or even services that they do not implement by themselves.

As multiple cloud architectures are a recent research topic, there are some inconsistencies in the nomenclature used in the field. This research uses the terminology proposed by Grozev and Buyya [67], in which two main concepts are defined:

- *Cloud Federation*: is a model in which multiple cloud providers cooperate and lease resources from each other. In this case, the cooperation initiative comes from the providers, which form alliances to offer improved services to their customers and/or to maximize their own benefit.
- *Multi-Cloud*: is a model in which the cloud customer is responsible for managing and orchestrating multiple providers to achieve some (or all) of the goals discussed above. Therefore, design decisions are made based on customer objectives, and the cloud providers are not aware of each other.

Figures 2.6a and 2.6b depict the differences between the two approaches. Note that in the cloud federation model, the service provider (SP) is not aware of the federation and interacts with a single cloud provider (CP). Conversely, in the multi-cloud model, the CPs do not maintain a mutual relationship among themselves. The establishment of individual contracts with each provider and the management of the service as a whole is the SP's responsibility.

Ferrer *et al*. [54] presented similar concepts, but also described two other models:




(a) Hybrid cloud model. (b) Brokering model.

Figure 2.7: Alternative multiple cloud models.

- *Hybrid Clouds*: in this model, a private cloud is extended to use public clouds when on-premises resources are insufficient. Similar definitions have been proposed in other studies, for example in Bittencourt *et al*. [28] and Zhang *et al*. [154]. Figure 2.7a presents this architecture.

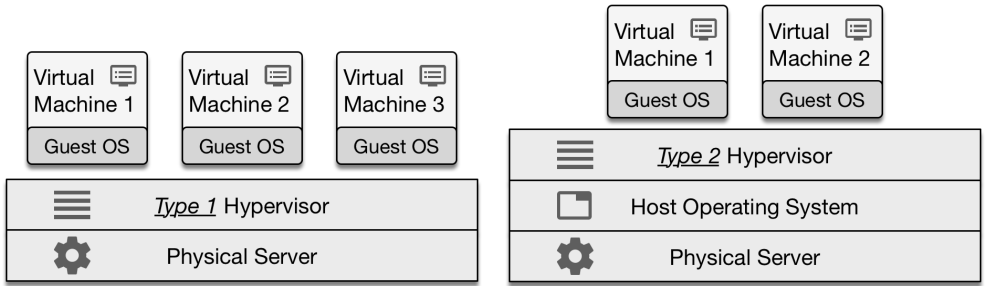
- *Brokering*: in this model, there is a broker that acts as an intermediary in interactions between CPs and SPs. The broker's role is to negotiate and aggregate resources from multiple CPs and offer them to SPs as needed. Theoretically, SP applications are simplified because they no longer need to interact with multiple CPs. At the same time, CPs do not need to manage individual contracts with customers because all interactions are performed through the brokers. A high-level overview of this case is depicted in Figure 2.7b.

In this research, hybrid clouds are considered a special instance of the multi-cloud model in which the role of SP / cloud user is played by the company that owns the private cloud. Moreover, brokering is not considered as a different model of organizing multiple cloud providers. In fact, it can be argued that the broker's role is to ease the formation of cloud federations and/or multi-clouds, and the interaction with the broker is not the goal itself. Consequently, this research considers the existence of a broker as an implementation detail of cloud federations or multi-clouds.

## 2.3 Container-Based Virtualization

As discussed in Section 2.2, the virtualization concept is at the core of cloud computing and its resource sharing model. Usually, virtualization is implemented by a software layer called *hypervisor*, which is responsible for partitioning the physical hardware and presenting these partitions to virtual machines (VMs).

The two most common hypervisor architectures are illustrated in Figure 2.8. A *Type 1* hypervisor runs directly on top of the hardware, whereas a *Type 2* runs as an application of




(a) Type 1 hypervisor. (b) Type 2 hypervisor.

Figure 2.8: Hypervisor architectures.

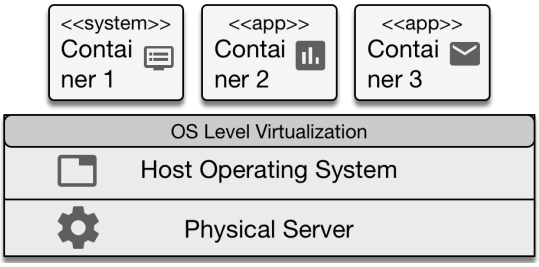



Figure 2.9: Container-based virtualization.

a *host* operating system. In general, Type 1 hypervisors are more efficient and have better performance, whereas Type 2 hypervisors are more flexible and easier to install. Regardless of hypervisor type, the virtual machines are independent units that can install their own *guest* operating system and have access to a slice of the hardware resources.

More recently, another type of virtualization started to become widespread in enterprise and research communities. *Container-based* virtualization, or *operating-system-level* virtualization, is a type of virtualization that uses facilities provided by the operating system kernel to implement isolation between *containers*. Figure 2.9 illustrates container-based virtualization.

Containers and VMs can be compared based on three main aspects [141]:

- *Functionality*: containers share the same OS kernel, whereas VMs have their own guest OS. Consequently, a container crashing an OS kernel may affect other containers running in the same host. Because of their maturity, hypervisors also tend to implement functionalities, such as live migration and reconfiguration of VMs, which are not available for containers.

- *Efficiency*: because each VM has its own operating system copy, VMs tend to use more RAM than containers.

- *Performance*: the overhead of running containers is smaller than that of running VMs because containers execute directly on top of the operating system. Therefore, the per-

formance of applications running on containers is very similar to the performance of running them on bare metal.

In Linux systems, the origin of container-based virtualization is related to projects such as VServer [140] and openVZ [122]. These projects introduced most of the concepts that were later incorporated into the kernel and into the LXC tools [100] to support native containers in Linux. Two of these main concepts are:

- *namespace*: enables creation of separate namespaces for resources that are usually global, such as filesystems, process identifiers, users and networks.
- *cgroups*: enables creation of groups of processes and association of resource consumption constraints within these groups. For instance, *cgroups* can be used to limit the amount of CPU and memory that a container can use.

### 2.3.1 Application Containers

Containers can be classified into two main categories according to the type of workload they execute:

- *System containers*, when they execute system-level processes and behave like a full operating system [52]. In this case, containers are used as virtual servers similarly to the VM approach.
- *Application containers*, when they execute user-level applications. In this context, containers are used to provide extra isolation between processes running on the same host.

Despite being originally envisioned in the context of system containers, the popularity of containers really took off with application containers. PaaS providers, such as Heroku [72], use containers to pack and execute user applications in a lightweight yet isolated environment. Moreover, companies like Google also run most of their workload in containers [148].

Today, the use of application containers is consistently associated with the Docker tool [114]. Docker adds many important features to LXC to enable portable execution of containers:

- *Bundling*: applications and all their dependencies are bundled together in a standardized *container image* format, which is independent of the software stack used to develop the application.
- *Versioning*: container images are versioned similarly to how source code is versioned in software configuration management systems. Therefore, it is possible to track, commit, and rollback changes made to an image. In addition, this mechanism also enables

incremental download (upload) of images, in which only the differences from previous versions need to be received (sent).

- *Reuse*: container images can be reused as a basis for other images. This feature facilitates development of new images and possibly reduces image transfer size because the base image needs to be transferred only once.
- *Sharing*: public repositories containing images are available to facilitate their distribution and reuse.
- *Tooling*: tools are available to facilitate creation of images and their distribution to image repositories.

The process isolation mechanisms provided by container-based virtualization in tandem with the portability of container images are important enablers of container management systems, which are described in the next subsection.

### 2.3.2 Container Management Systems

The use of clusters to run computational intensive tasks has been the subject of intense research in areas such as High Performance Computing (HPC) and Grid Computing. Research in these areas usually focuses on systems to manage clusters and strategies to schedule user jobs optimizing certain criteria, such as the time to complete all jobs [27, 78] and energy consumption [51]. Most of this research, however, relies on specialized hardware and infrastructure software that are not available to everyone.

Recently, clusters of commodity servers have emerged as an important computing platform for both Internet services and scientific applications [80]. These clusters are cheap and fast to build, and today they power most cloud computing providers, which use them not only to run their own workloads, but also to execute their users' jobs and services. It is in this context that the first *Container Management Systems* (CMS) have appeared.

This research defines CMS as software systems which control clusters of commodity servers and use application containers as the basic unit of management. The container-based approach brings two main advantages to these systems:

- By using container-based virtualization, processes running on the same server are controlled and isolated with very low overhead. This significantly improves system utilization, yet maintains application performance.
- By encapsulating all dependencies in a hermetic container image, application containers abstract away details about the operating system and hardware on which they run [32].

This facilitates distribution and deployment and also shifts the datacentre perspective from machine-centric to application-centric.

Google popularized this approach with Borg [148], a CMS that has been in production for over a decade controlling most of their workload. More recently, Google released Kubernetes [63], an open-source CMS based on Borg. The next section presents more details about Kubernetes.

### Kubernetes

The main goal of Kubernetes is to manage container-based applications across a cluster of servers [63]. It builds on top of Docker a series of functionalities needed for running applications, such as naming, interconnectivity, scheduling, scaling, and monitoring.

The main concept used by Kubernetes is a *pod*, which can be defined as a set of containers that are always scheduled together. A pod is usually composed of one main container (e.g., a Web server) plus one or more containers that provide auxiliary services (e.g., log rotation or backup services).

In Kubernetes, a pod can also have an associated *replication controller*, which is responsible for guaranteeing that a certain number of pod replicas are always running. Replication controllers monitor the health of replica pods and create new ones if they detect a failure. Finally, Kubernetes also allows the definition of *services*, which are used to implement a basic discovery mechanism. A service has a fixed name and IP address that can be accessed cluster-wide. Requests sent to this name or IP address are automatically forwarded to pods that implement that service.

Figure 2.10 shows the architecture of a Kubernetes cluster. Its main components are:

- *Distributed Storage*: maintains information about the cluster and all running applications.
- *Main Server API*: provides a REST API which is used to access cluster data. It implements basic CRUD and validation functionalities.
- *Scheduler*: communicates with the API and schedules new pods in the cluster.
- *Controller Manager*: implements the replication controller logic.
- *Kubelet*: runs in every node of the cluster. It communicates with the other components to enforce local actions and to provide monitoring information.
- *Proxy*: provides access to services from each node by forwarding requests to the appropriate pods.

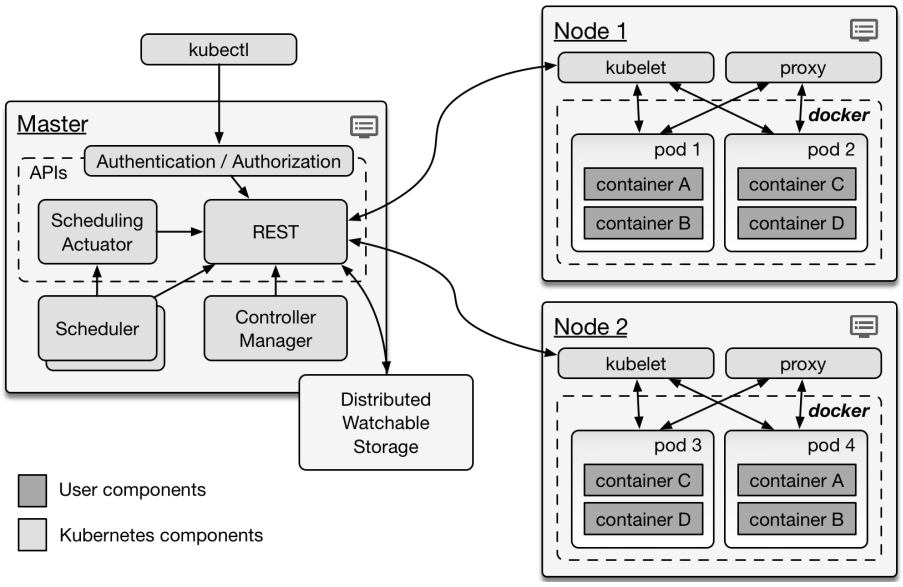



Figure 2.10: Kubernetes architecture (adapted from Google [63]).

- *kubectl*: command-line tool used to manage the cluster.

Kubernetes is a project that is still in active development. In this research, Kubernetes is used to implement the *CEPaaS* system described in Chapter 7.

## 2.4 Autonomic Computing

Autonomic computing aims to build computing systems that can manage themselves based on high-level objectives determined by system administrators [91].

The essence of autonomic computing is *self-management*. Theoretically, self-management frees system administrators from the burden of operating and maintaining complex computing systems, while keeping their performance optimal. Self-management is composed of four main aspects:

- *Self-configuration*: autonomic systems can configure themselves in dynamic environments, finding services and providers that they depend on and broadcasting their capabilities.

- *Self-optimization*: autonomic systems can monitor their performance and workload, searching for opportunities to fine-tune internal parameters. Moreover, they can update these parameters, test for improvements, and rollback unsuccessful changes.

- *Self-healing*: autonomic systems can diagnose failures and defects and isolate problematic components. They can also update these components automatically and execute tests

to ensure that the system is working properly.

- *Self-protection*: autonomic systems can protect themselves from attacks and cascading component failures.

In autonomic computing, a system is usually controlled by an *autonomic manager*, which is responsible for enforcing the system self-management capabilities. The manager implements a control loop conceptualized by the MAPE-K framework [85], which is depicted in Figure 2.11. The framework is named after the five functions composing it:

- *Monitor*: monitors events from the managed systems to infer symptoms and sends them to analysis;
- *Analyze*: analyzes symptoms and infers whether changes are required. If needed, sends request for changes to the plan function;
- *Plan*: selects the actions that must be performed based on the analysis results;
- *Execute*: executes the selected actions;
- *Knowledge base*: contains every required piece of information about the system, including actions that may be performed, their representations, and the inference rules used by the four other functions.

Note that the MAPE-K functions might not exist as separate entities, but logically all of them are always present in an autonomic manager.

This research applies autonomic computing concepts in two main parts: first, the *AGeCEP* formalism presented in Chapter 4 uses the MAPE-K framework as a standard and intuitive way to describe self-management policies. Second, the *CEPaaS* system described in Chapter 7 implements a simple autonomic manager to control runtime queries and guarantee their performance.

## 2.5 Summary

This chapter discussed background concepts needed for the remainder of this thesis. First, it presented an overview of complex event processing and stream processing and clarified how they relate with each other. Next, cloud computing was discussed with special emphasis on multiple cloud architectures and how they can be used to improve the quality of services offered. Following, container-based virtualization and container management systems were also

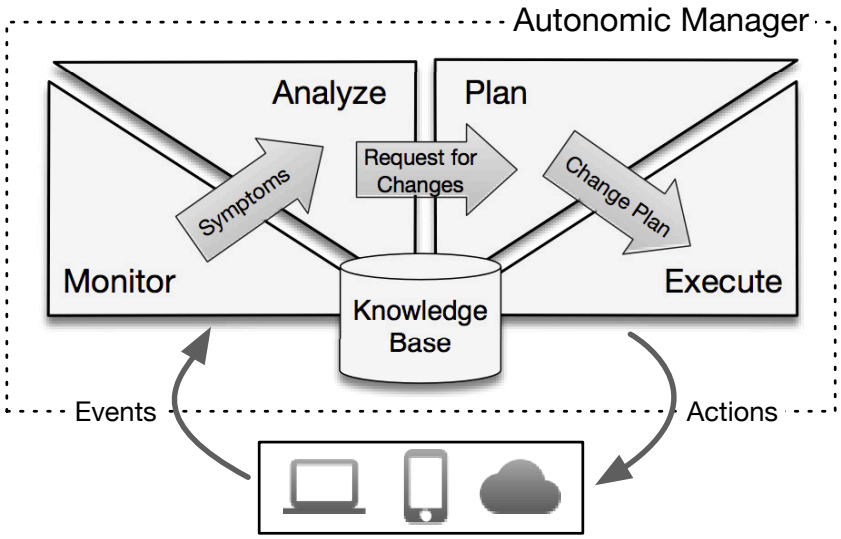



Figure 2.11: MAPE-K autonomic loop.

examined. Finally, autonomic computing and the MAPE-K loop were presented and discussed in the context of the *CEPaaS* system.

The next chapter presents an extensive review of studies related to the contributions developed in this research, including other CEP and SP systems, CEP formal models, and cloud computing simulators.

# Chapter 3

# Literature Review

This chapter presents research related to this thesis divided into three categories: CEP systems, CEP formal models, and cloud computing simulators. Note that this chapter still differentiates between CEP and SP to respect the terminology used by the original authors of each study. Clarifications are provided whenever necessary.

## 3.1 Complex Event Processing

This section presents a comprehensive review of the most important CEP and SP systems and research projects related to the *CEPaaS* system developed in this thesis. It starts with traditional research projects which, despite being developed over a decade ago, established current terminology and introduced many techniques still in use today. Following, Section 3.1.2 presents recent projects that leverage modern architectures to improve the quality of service offered to users. Section 3.1.3 discusses projects that aim to offer CEP or SP in the services model, and Section 3.1.4 reviews systems based on multi-cloud architectures. These two ideas are relatively unexplored, yet are at the core of this thesis. Next, Section 3.1.5 compares cloud-based CEP systems with *CEPaaS*. Finally, Section 3.1.6 discusses all these studies in the context of the contributions presented in this research.

### 3.1.1 Traditional Systems

This section presents important historical projects that have shaped modern CEP and SP research. These are highly influential studies that serve as a basis for most modern systems.

### Stream Processing

NiagaraCQ [37] is one of the earliest SP projects. The system aimed to enable continuous queries over XML datasets distributed over the Internet. The queries are written in a language specific to XML files and are recurrently executed based on a timer interval or whenever the source datasets change. The main NiagaraCQ contribution is an incremental group optimization algorithm that can group similar queries to reuse common computations.

The Aurora [1] project is probably the most influential development in this field. Queries in Aurora are specified using a graphical language composed of boxes and arrows, in which the former represent query operators and the latter connection points between them. Aurora is a centralized engine in which user-specified QoS constraints drive the behaviour of two components: the engine scheduler, which decides the operators to run at any given time and the number of events they consume; and the load shedding mechanism, which decides whether discarding events can improve the system processing rate. Later on, Aurora was extended by the Aurora* [38] and Medusa [24] systems.

Aurora* [38] extended Aurora to support distributed execution in clustered environments. It introduced server load management using two main techniques: box sliding, which migrates an operator to one of its immediate neighbours, and box splitting, which splits an operator across many servers and parallelizes its execution. The Medusa system, on the other hand, introduced a load management mechanism aimed at federated distributed systems [24]. In this scenario, multiple participants in different administrative domains establish contracts between each other to define a price range for processing a unit of work. Generally, a unit of work is migrated to another participant if the local execution cost is higher than the remote cost.

Later, both Aurora* and Medusa were merged into Borealis [2]. In addition to all functionalities just mentioned, Borealis also added features for revision of query results, dynamic query modification and a QoS-driven load management architecture that acts at local, neighbourhood, and global levels.

Developed at about the same time as Aurora, TelegraphCQ [36] is also an important project in the SP field. TelegraphCQ is a general purpose continuous query system that focuses on adapting behaviour in the face of changes in data sources, network conditions, server availability, and user needs. One of its main contributions is the Flux operator [137], which is used to provide parallel processing through data partitioning and to implement load balancing and fault tolerance.

Finally, the STREAM [19] project, developed at Stanford University, should also be highlighted. Its main contribution is the Continuous Query Language (CQL) [20], a declarative query definition language. CQL is remarkably similar to SQL and has influenced many subsequent languages. Indeed, Oracle Stream Explorer [123], which is a commercial solution, uses

CQL as one of their query definition languages.

#### Complex Event Processing

The origin of Complex Event Processing is attributed to Luckham [106], who explored the need for a new technology as a response to new challenges that enterprises had been facing with increasing automation and interconnection of IT systems. Nevertheless, the technological underpinnings of CEP had been in development for a long time. Indeed, Luckham's CEP work was based on Rapide [105], a simulator that used the concepts of event causality to model interactions among various components of a distributed system.

The GEM language [110] is another precursor of CEP, even though the field terminology was not established at the time GEM was published. GEM is used to define queries to monitor distributed systems, including constructs to specify composite events, temporal constraints, and event windows. These constructs are now part of most modern CEP query languages.

The advancement of CEP is also often related to publish / subscribe (pub-sub) systems. For instance, Pietzuch *et al*. [125] published an important study describing an architecture that can augment existing pub-sub systems with composite event detection capabilities. In their work, the authors presented a language for specifying composite events and a distributed detection engine based on finite-state automata. Similarly, the PADRES project [99] is also a pub-sub system for composite event detection with its own query language. As important contributions, the authors presented distributed event-detection algorithms that can share computation among multiple queries and reduce network traffic.

The Cayuga [44] project, on the other hand, was developed as a general purpose event processing system. Cayuga uses the Cayuga Event Language (CEL) to express queries, which are executed as non-deterministic finite-state automata. The project focused on developing efficient data structures to provide scalability regarding both the number of queries and the volume of events. However, Cayuga is a centralized system that is limited to the processing capacity of a single server.

Finally, NextCEP [134] is another general purpose CEP system with an automata-based execution model, but with an SQL-like query language. Its main results are related to query rewriting mechanisms and distributed execution in cluster environments.

### 3.1.2 Modern Systems

This section reviews recent systems that propose new approaches aimed at real-time continuous processing of datasets. These projects are coarsely classified into three major groups. The first group is composed of systems inspired by the MapReduce [43] computing platform.

The second group is composed of open-source platforms that provide generic CEP / SP distributed computing capabilities. All these are readily available for download and have been maintained by open-source communities for many years. Finally, the third group contains CEP / SP systems that leverage cloud resources to provide additional scalability and availability.

### MapReduce-Based

MapReduce is a computing paradigm aimed at processing and generating large data sets [43]. The paradigm is based on a two-step computation: the first step is implemented by a *map* function, which receives key-value pairs as input and generates a set of intermediate key-value pairs as output. In the second step, a *reduce* function receives a set of keys and all values generated for each of these keys in the map phase, and performs a final computation aggregating the values received.

In the original article, Dean and Ghemawat [43] presented a platform to run programs written using this paradigm in which the map and reduce tasks are automatically parallelized and executed on a server cluster. The platform also provides fault tolerance based on re-execution of failed tasks. Since its introduction, MapReduce has been used in diverse scenarios and has originated different implementations, among which Apache Hadoop [16] is the most popular. Nevertheless, despite its success, the MapReduce paradigm is not an appropriate solution for CEP because:

- MapReduce computations are batch processes that start and finish, whereas computations over event streams are continuous tasks that finish only upon user request.
- The inputs of MapReduce computations are snapshots of data stored in files, and the contents of these files do not change during processing. Conversely, event streams are continuously generated and unbounded inputs [98].
- To provide fault tolerance, most MapReduce implementations, such as Google MapReduce [43] and Hadoop [16], write the results of the map phase to local files before sending them to reducers. In addition, these implementations store the output files in distributed and high-overhead file systems (Google File System [57] or HDFS [16] respectively). This extensive file manipulation adds significant latency to the processing pipelines.
- Not every computation can be efficiently expressed using the MapReduce programming paradigm, and the paradigm does not natively support composition of jobs.

Despite these limitations, the prevalence and success of MapReduce have motivated many researchers to work on systems that leverage some of its advantages while trying to overcome its limitations when applied to low-latency processing.

One of the first projects in this direction was developed by Condie *et al*. [39]. In this work, the authors proposed an online MapReduce implementation with the goal of supporting online aggregation and continuous queries. To reduce processing latency, the map and reduce steps are pipelined by having the map tasks send intermediate results to the reduce tasks. The authors also introduced the idea of executing reducers on snapshots of the data received from the mappers. This mechanism enables generation of partial approximate results, which is particularly useful for interactive analytics scenarios. All these changes were implemented on Apache Hadoop and demonstrated in a monitoring system prototype.

Despite these modifications, the work by Condie *et al*. [39] still has limitations that hinder its use in CEP scenarios. For instance, if the reducers are not scheduled simultaneously with the mappers, the mappers cannot push intermediate results to them. In addition, the platform does not support elasticity, which is a very important requirement for scenarios where the event input rate is subject to wide fluctuations and burst behavior.

Other researchers have also leveraged the familiar MapReduce programming model, but focused on providing alternative runtime platforms. For instance, Logothetis and Yocum [102] proposed a continuous MapReduce in the context of a data processing platform running over a wide-area network. In their research, the execution of map and reduce functions is managed by a stream processing platform. To improve processing latency, the mappers are continuously fed with batches of tuples (instead of input files), and they push their results to reducers as soon as they are available. This approach is similar to that adopted by the StreamMapReduce [31] system, which send events (key-value pairs) directly from one processing stage to another without the persistence of intermediate results.

Similarly, the M3 [8] project aimed to provide a memory-based MapReduce implementation. In this project, the execution engine transforms user queries into a sequence of MapReduce jobs and executes them. The general idea of this transformation is to map each query operator to a pair of map / reduce functions; the article presented examples of transformations of filters, joins, and aggregates, but did not detail or formalize this process. The authors also claimed that the system could provide fault tolerance and adapt to dynamic workloads, but no implementation details or experimental results were shown.

Alternatively, the difficulty of expressing online computations using MapReduce has also motivated the creation of alternative programming models. For instance, the Muppet project [98] presented a new paradigm called MapUpdate, which mimics MapReduce by specifying computations as two functions (Map and Update). The main difference is that the update phase has access to *slates*, data structures that contain persistent state related to each update key. In theory, these slates enable easier implementation of iterative and stateful algorithms.

### Open-Source Platforms

Recently, many open-source CEP / SP distributed computing platforms have emerged to handle applications with low latency requirements and large data volume. The most prominent examples of this category are Storm [18], Yahoo's S4 [117], Spark Stream [153], and Samza [17].

Storm [18] was created at Backtype, a company later acquired by Twitter. At Twitter, Storm was used for a long time as the *de facto* SP platform [144] and was only recently superseded by Heron [95]. Today, Storm is an open-source project managed by the Apache group.

Storm is based on user-defined *topologies*, which are directed graphs in which the vertices define computations to be performed and the edges specify event streams flowing from one vertex to another. Events, also known as *tuples*, are produced by special vertices named *spouts* and processed by vertices named *bolts*. The platform provides spout implementations for connecting with different event sources such as message queues, but it is also possible to create new ones. In addition to the processing graph, a topology also defines the number of threads (tasks) for each bolt and how the input streams are partitioned among the available tasks. For example, it is possible to partition an input stream randomly or based on the hash values of specific attributes.

A Storm cluster is managed by a single master node called *Nimbus*, which is responsible for scheduling topologies into worker servers. Each worker runs a *supervisor*, which monitors local tasks and restarts them if needed. The supervisor also communicates the local state to Nimbus. If a worker node fails, its local tasks are rescheduled into other nodes. Storm guarantees that each tuple is processed at least once by tracking tuples throughout the processing pipeline. Recently, a new API called Trident was created over Storm. This API provides ready-to-use CEP operators such as joins, aggregates, and filters. Moreover, it changes the processing model from tuple-by-tuple to small batches and adds new abstractions that enable *exactly-once* processing semantics.

Yahoo's S4 [117] is based on similar concepts. A computation receives one or more event streams as input and processes them using a sequence of *processing elements* (PE). In S4, each PE is defined by a type, the type of events that it consumes, and the attributes that are part of the event key. An important difference from Storm is that a new PE instance is created for each distinct key value on its input. Therefore, S4 also implements a garbage collector mechanism to remove old / unused PE instances from memory and to avoid memory overflow.

An S4 cluster is formed by a set of *processing nodes* (PNs) that use Zookeeper [83] to coordinate among themselves. Each PN has a PE container that manages local PEs; events are distributed among the PNs based on a hash value derived from each event's key attributes. A communication layer is used by the PEs to forward events to the appropriate PN. Furthermore, to provide fault tolerance, the S4 cluster has a (configurable) number of standby instances,

which are used if active nodes fail. In addition, an uncoordinated checkpoint mechanism saves the PEs state based on time intervals or event counts. This checkpoint enables faster recovery of failed processing nodes.

Spark Stream [153] is a stream processing platform built on top of Apache Spark [152] and based on the *discretized streams* (D-Stream) model. In this model, events are grouped into short-duration batches and stored in special abstractions called *resilient distributed datasets* (RDD) [152]. RDDs are in-memory partitioned data structures that can be processed only by a set of *transformations*. A transformation, in turn, can generate output RDDs and maintain auxiliary state in others. A user query is defined as a sequence of such transformations over the input events. This processing model is significantly different from that used in other CEP systems because, at each stage, tasks are scheduled to execute transformations and can be discarded afterwards. This is possible because all necessary input is available as RDDs in the cluster nodes memory. Conversely, in most other systems, the computations are continuous processes that constantly receive and generate events.

Spark Stream's model has three major advantages over continuous processing: first, it unifies stream and batch programming models by expressing both as RDD transformations. Second, it enables implementation of fault tolerance. When a node fails, the lost RDDs are reconstructed (in parallel) by reapplying the sequence of transformations that originated them, starting at the input events. Finally, it enables speculative execution of tasks to avoid delays caused by *stragglers*. However, because events must be batched before processing, Spark Stream's processing latency tends to be higher compared to similar platforms.

Finally, Samza [17] is a stream processing platform created by LinkedIn and the most recent project discussed in this section. Samza differs from other platforms by defining a strong model for event streams. In Samza's model, event streams are seen as append-only partitioned logs, where each partition contains a totally ordered sequence of events. Partitions provide a natural way of consuming an event stream in parallel, yet there is no ordering guarantee between events from different partitions. In addition, Samza's model requires event streams to be multi-subscriber and replayable so that an event can be repeated if a subscriber fails. In practice, event streams are implemented by Apache Kafka [94], a message broker also created by LinkedIn. Kafka provides partitioned *topics* that can be directly used as Samza event streams.

In Samza, users create *jobs* that perform transformations on a set of input streams and write their results to an output stream. Because each event stream is a persistent Kafka topic, jobs are totally decoupled from each other. This architecture brings two main advantages to the system. First, the performance of slow consumers does not affect any upstream job because events are buffered in the intermediary topics. Second, the results of any job can be easily read from the job's output stream, which facilitates reuse and composition. As disadvantages, this model

adds delays to system latency and complicates the creation of multi-step processing because there is no notion of query in the system.

Samza also has unique support for implementation of stateful jobs by providing a fast key-value store at each node of the cluster. With this approach, jobs manipulate state data without the overhead of accessing persistent storage via the network. However, because they are local, these data can be lost in case of server failure. To solve this problem, Samza replicates every operation in the local store to a special Kafka topic. This way, a job can rebuild the local store state by replaying the operations from this topic.

### Cloud-Based

Recent research projects have been leveraging cloud resources to provide robust and elastic CEP systems. This section reviews relevant projects available in the literature.

ElasticStream was developed by Ishii and Suzumura [87] and was one of the first to use cloud resources to improve the quality of SP systems. It uses IaaS servers to provide additional computing capabilities in the case that local resources are insufficient to handle the input event streams. The authors formulated infrastructure allocation as an optimization problem that aims to minimize expenses on cloud resources while maintaining system capacity. An optimizer is executed periodically, and its output guides the allocation of public cloud resources and the split of input streams between local and remote servers.

Esc [133] is another distributed SP system designed to be elastic. Queries are defined using a directed acyclic graph (DAG) of processing elements, which are executed on a cluster of servers. Esc uses an autonomic manager to implement self-management policies that attach new servers to the resources pool and create more instances of processing elements. Esc has a very simple load balancing mechanism, based on killing a worker process and expecting it to be re-allocated on a less loaded machine.

Similarly, the goal of the StreamCloud [69] project was to create a SP engine that can handle very large input streams. To achieve this objective, the authors presented a parallelization strategy that splits user queries into subqueries and runs each subquery on a subset of the available machines. In addition, logical input streams are split into many physical streams and processed in parallel. This approach was contrasted with two other common parallelization strategies: *query-cluster* and *operator-cluster*. According to the article results, the subquery strategy achieved better throughput and greater scalability than the other approaches. This research also discussed two protocols to move processing load from one server to another and developed basic elasticity capabilities based on CPU monitoring. The system assumes a private cloud environment in which idle servers are always available and ready to use.

The StreamHub [25] project aimed to create a high-throughput, low-latency, and scalable

publish/subscribe system. The StreamHub design was inspired by the distributed SP frameworks described in Section 3.1.2 - "Open Source Platforms". The core functionalities are implemented by a sequence of three operators that scale independently: access point (subscription partitioning), matching, and exit point (dispatching). One of StreamHub's unique characteristics is that it can use different libraries to match publications to subscriptions, making the system adaptable to many situations, including CEP.

The TimeStream [128] system is another SP engine designed for deployment in clusters and cloud environments. Its main goal is to provide reliable event processing even in the presence of failures and reconfigurations. A rule in TimeStream is specified using a declarative language and is converted by a compiler into a DAG. To provide fault tolerance and enable reconfiguration actions, the system uses a concept called *resilient substitution*, which depends on tracking the output and state dependency of each operator. Assuming that an operator $o$ is in state $\tau$, the state dependency is the subset of the operator's inputs that led $o$ to this state. Accordingly, the output dependency of a result $e$ is the operator's previous state dependency plus the input $i$ that caused the generation of $e$. If an operator $o$ fails, a new instance $o'$ is created and the $o$ state dependency is repeated, leading $o'$ to the same state $\tau$. Similarly, a subgraph of the DAG can be replaced by another (equivalent) by replaying state dependencies of the original subgraph.

Nephele [103] is an SP system that can self-adapt to satisfy user-defined QoS constraints. In Nephele, queries are also characterized by DAGs and distributed over many worker servers. A distributed monitoring infrastructure detects QoS violations at runtime and implements two reconfiguration mechanisms: output buffer resizing and task chaining. The former changes the number of events that are buffered before being sent to the next operator, whereas the latter combines multiple operators into a single one that is logically equivalent.

Murray *et al*. [116], on the other hand, proposed a new computational model called *timely dataflow* that can be used for data parallel cyclic dataflow programs, including but not limited to CEP. In this model, the computations are directed graphs that can include loops, making them especially suitable for iterative processing. Vertices of the dataflow graph can be notified when all messages before a (logical) timestamp $t$ have been processed. This feature enables vertices to check for end-of-loop conditions and to implement window-based operators. The authors presented a runtime platform for this model called Naiad, which parallelizes and distributes dataflow graphs over a server cluster. The graphs are executed without central coordination, yet the platform has limited elasticity and fault tolerance mechanisms.

Fernandez *et al*. [53] focused on an integrated approach for scaling out and fault tolerance in cloud-based SP systems. In their approach, stateful operators implement callback functions that explicitly convert their internal state into a set of key-value pairs. This state, along with

the operator's output buffer events and routing state, are backed up into upstream operators. This backup is used whenever a new operator instance needs to be created, which can occur in two different situations: an operator has failed and needs to be recreated; or an operator has become a processing bottleneck, and new instances are needed to increase its throughput. Note that both situations might require new VMs to be allocated. To avoid long delays in this allocation, the system maintains a pool of pre-allocated VMs that are transferred to the application when needed.

Google's MillWheel [4] is the SP system used internally at Google. It is based on the familiar graph-based model, in which vertices called *computations* encapsulate user logic. MillWheel provides an *exactly-once* delivery guarantee, which differentiates it from most other SP systems. This guarantee is implemented as follows: upon receiving an event, the computation checks for duplication by consulting a database. If the event is not a duplicate, the computation is executed, which can result in updates to timers, internal state changes, and generation of output events. Following the execution, the processed event id, the new computation's state, and all produced events are checkpointed to a backup database in a single atomic write. Finally, the senders are acknowledged, and the produced events are sent downstream. By writing all state to persistent storage, MillWheel guarantees that events are not processed more than once and also implements transparent failure recovery. As another important contribution, MillWheel also provides low watermarks for input streams. These low watermarks indicate a timestamp up to which all events have been processed and can be used to implement window-based operators and timers in computations.

The FUGU [70] system extended a commercial SP system based on Borealis [2] to provide load balancing and elasticity capabilities. FUGU has a centralized monitoring infrastructure which collects metrics from system nodes and moves operators when an overloaded node is detected. The operators to be moved are selected so as to minimize the latency spikes caused by their movement. To calculate this spike, the system considers all operators that are affected by the movement and the extra delay caused by enqueued events.

StreamMine3G [111, 112] is an evolution of the StreamMapReduce system [31]. In StreamMine3G, operators are defined using a MapReduce-like programming interface and assembled into a DAG. At runtime, operators are replicated and input data partitioned among these replicas. The most distinctive features of StreamMine3G are the support for multiple fault-tolerance schemes and a mechanism that automatically selects the most appropriate scheme based on user-defined recovery time constraints. For instance, if long recovery is unacceptable, then the system applies an active replication scheme in which two operator instances located in different servers process all events (in duplication). On the other hand, if a longer recovery can be tolerated, then passive replication is employed. In this scheme, a new instance is created only

when the first one fails. This support to multiple schemes enables users to trade-off between recovery time and resource usage.

Finally, Heron [95] is an SP engine created at Twitter to overcome Storm [18] limitations when used at very large scale. For instance, Storm runs many tasks in the same worker process, which complicates debugging and resource isolation. In addition, Storm assumes a homogeneous cluster, has a single point of failure (Nimbus), and does not support backpressure. Heron, on the other hand, has a different architecture, yet is API-compatible with Storm. Heron runs on a shared infrastructure controlled by an Aurora scheduler[1] [15] on top of Mesos [80]. In this architecture, each topology is scheduled as a set of application containers controlled by Linux cgroups [100]. One container runs a *Topology Master*, which controls the topology lifecycle, and every other container runs a single *Stream Manager* instance and a set of *Heron Instances*. The Stream Manager manages communication between containers and implements the backpressure protocol. Each Heron Instance, in turn, maps to a topology task. By leveraging this new architecture, Twitter improved Storm's throughput by 14 times and latency by 15 times. In addition, it also increased cluster utilization and system debuggability.

### 3.1.3 CEP Services

Loesing *et al.* [101] were among the first authors to propose SP as a service. They introduced Stormy, an SP engine that uses techniques from cloud storage systems [64] to provide scalability and availability. Stormy distributes queries and input events using consistent hashing and implements a gossip protocol to disseminate the mapping of queries to nodes. To provide availability, queries are replicated, and events are processed by all replicas. Nevertheless, the article presented no results about system performance or scalability.

Currently, the main cloud providers also offer managed services that support CEP / SP functionalities, but these are mostly targeted at application developers. For instance, Amazon offers Amazon Kinesis [12] and AWS Lambda [13], whereas Google offers Cloud Pub/Sub [62] and Cloud Dataflow [61].

AWS Lambda [13] enables users to run generic code, encapsulated in functions, in response to triggers from other Amazon services, HTTP endpoints, or activities from mobile applications. There is no infrastructure to be managed, and the service provides automatic scaling. Nevertheless, the service has no notion of queries, buffering, or many other functionalities needed for CEP systems.

Amazon Kinesis [12], on the other hand, is a suite of services aimed at real-time stream processing. The main service of this suite is Amazon Kinesis Streams, which provides as a

[1] Not to be confused with the Aurora SP engine [1]

managed service publish-subscribe functionalities similar to Apache Kafka [94]. To consume data from Kinesis Streams, a developer can write programs using the Amazon Kinesis Client Library (KCL). KCL provides a platform for running SP applications, including automatic load balancing and input stream re-sharding. However, KCL is not a managed service, which implies that developers must control the infrastructure in which KCL programs run.

Similarly to Amazon Kinesis Stream, Google Cloud Pub/Sub [62] is also a publish-subscribe messaging service. Cloud Pub/Sub is globally deployed, which ensures low latency for data sources and consumers distributed around the world. To consume data from Cloud Pub/Sub, developers can use Cloud Dataflow [61], which provides a fully managed service for batch and stream processing. In fact, the Cloud Dataflow service is an implementation of the Dataflow programming model presented by Akidau *et al*. [5].

The Dataflow model has been created to express programs that process unbounded unordered data sources and generate event-time ordered results. This model provides capabilities that enable users to fine-tune computations and trade off correctness, latency, and cost. In a dataflow, data are represented by parallel collections of key-value pairs that can be processed by two core primitives: *ParDo* for generic parallel processing, and *GroupByKey* for processing data grouped by key. For unbounded streams, the *GroupByKey* construct is extended with a generic window concept that can group data in fixed, sliding, and session windows. In this case, *triggers* determine when window results are produced and how multiple results from the same window relate to each other. Because it is generic, the model can be implemented by either batch or stream processing systems. For instance, at Google, a batch and a streaming version have been implemented in FlumeJava [35] and MillWheel [4] respectively.

### 3.1.4 Multi-Cloud CEP

The idea of exploring multiple cloud environments to improve the quality of service provided by a CEP service is almost unexplored in the literature.

Photon [14] is a system in production at Google that is solely focused on joining two event streams. The system operates under very specific circumstances, in which a primary stream of events needs to be joined with a secondary stream that happens within seconds of the primary. Both streams can be unordered because the events come from distributed datacentres. In addition, the primary stream can be delayed relative to the secondary. Photon maintains replicas of the processing pipeline in multiple cloud datacentres to provide fault tolerance. These pipelines synchronize by means of a replicated, strongly consistent global database.

JetStream [145], on the other hand, proposed a generic framework for transferring events between geographically distributed datacentres. The proposed framework is generic and can

integrate with different CEP systems. The main idea is to decide on a batch transfer size autonomically based on system metrics such as network latency, input event rate, and number of destinations. The authors also introduced the idea of using multiple routes across nodes located in different datacentres to increase the aggregated bandwidth between them.

### 3.1.5 Comparison

Table 3.1 summarizes the characteristics of modern cloud-based CEP systems and of the *CEPaaS* system developed in this research. The systems are compared based on the following criteria:

- *Paradigm* adopted by the system: (a) *SP*, (b) *CEP*, (c) *batch*, (d) *publish-subscribe*, or (e) *function*.
- *Availability* to be downloaded and used: (a) *open source*, (b) *research prototype*, (c) *proprietary software*, or (d) *managed service*.
- *Cluster types* used by the system: (a) *specific*, if the system uses its own mechanism to define a cluster; (b) *shared*, if the system runs in a cluster shared with other applications and is controlled by a specialized cluster management software; or (c) *container*, if the system runs in a cluster controlled by a container management system.
- *Elasticity* implementation: (a) *no*, if the system does not implement elasticity; (b) *not applicable*, if the system runs in a shared cluster (in this case elasticity is implemented at the cluster manager level); (c) *bursting*, if the system uses public cloud resources when private resources are not sufficient; (d) *pool*, if the system can attach pre-allocated servers to the cluster; or (e) *regular*, if the system can attach and de-attach servers to the cluster on-demand.
- *Query definition* language: (a) *DAG-based*, (b) *declarative*, (c) *not available*, or (d) system-specific categories, such as transformations for Spark Stream [153] and timely dataflow for Naiad [116].
- *Management* implementation: (a) *centralized*, with or without standby replicas; (b) *hierarchical*, if the management processes are organized in a hierarchical fashion; (c) *per-query*, if each query has its own management process; (d) *distributed*, if management is implemented by all participant nodes and there is no single point of failure.
- *Distinctive features* that differentiate a system from the others.

| | Paradigm | Availability | Cluster type | Elastic | Query definition | Management | Distinctive features |
|---|---|---|---|---|---|---|---|
| **Storm [18]** | SP | Open Source | Specific | No | DAG | Centralized | Production-ready / Stability |
| **S4 [117]** | SP | Open Source | Specific | No | DAG | Distributed | Decentralized management |
| **Spark Stream [153]** | SP / Batch | Open Source | Specific<br>Shared<br>Container | No<br>Not applicable<br>Not applicable | Transformations | Per query | RDDs |
| **Samza [17]** | SP | Open Source | Shared | No | Not available | Per query | Stateful recovery |
| **ElasticStream [87]** | SP | Research | Specific | Bursting | DAG (fixed) | Centralized | Cloud bursting |
| **Esc [133]** | SP | Research | Specific | Regular | DAG | Centralized | Autonomic manager |
| **StreamCloud [69]** | SP | Research | Specific | Pool | DAG | Centralized | Operator set parallelization |
| **StreamHub [25]** | Pub-Sub | Research | Specific | No | Pluggable | Unknown | Tiered architecture |
| **TimeStream [128]** | CEP | Research | Shared | Not applicable | Declarative | Per query | Resilient Substitution |
| **Nephele [103]** | SP | Research | Specific | No | DAG | Hierarchical | QoS monitoring |
| **Naiad [116]** | SP / Batch | Proprietary | Specific | No | Timely dataflow | Distributed | Programming model |
| **Fernandez et al. [53]** | SP | Research | Specific | Regular | DAG | Centralized | Fault-tolerance / Scale-out |
| **MillWheel [4]** | SP | Proprietary | Container | Not applicable | DAG | Centralized (w/ standby) | Exactly-Once semantics |
| **FUGU [70]** | SP | Research | Specific | Regular | DAG | Centralized | Operator migration |
| **StreamMine3G [111, 112]** | SP | Research | Specific | Regular | DAG | Centralized (w/ standby) | Adaptive fault-tolerance |
| **Heron [95]** | SP | Proprietary | Container | Not applicable | DAG | Per query | Containers / backpressure |
| **AWS Lambda [13]** | Function | Service | Unknown | Unknown | Not available | Unknown | Managed Service |
| **Amazon Kinesis [12]** | Pub-Sub | Service | Unknown | Unknown | Not available | Unknown | Managed Service |
| **Google Pub/Sub [62]** | Pub-Sub | Service | Unknown | Unknown | Not available | Unknown | Simplicity |
| **Google Dataflow [61]** | SP / Batch | Service | Unknown | Unknown | Dataflow | Unknown | Unified model |
| **Photon [14]** | SP | Proprietary | Container | Not applicable | Stream Joining | Unknown | Multi-Cloud |
| **CEPaaS** | CEP | Research | Container | Not applicable | DAG | Centralized (w/ standby) | Multi-Cloud / Multi-tenancy |

Table 3.1: Comparison of *CEPaaS* and modern CEP systems.

Most current systems are based on the simpler SP paradigm rather than on CEP, and DAG-based query languages have been almost universally adopted[2]. Moreover, it is possible to note a recent trend towards shared clusters runtime environments, both traditional and container-based ones. These environments have been leveraged to increase resource utilization and to reduce costs of datacentres. In addition, they greatly simplify CEP system management implementations because many features, such as server monitoring and elasticity, are now provided by the cluster manager.

### 3.1.6 Discussion

The *CEPaaS* system proposed in this thesis is obviously influenced by the CEP / SP pioneers discussed in Section 3.1.1. Specifically, the "box-and-arrows" query definition language proposed by Aurora [1] is a natural model for specifying event processing queries, and therefore a similar language has been adopted by *CEPaaS*. In addition, Aurora's strategy of parallelizing an operator by splitting input event streams is also used in this thesis.

The proposed *CEPaaS* system, however, offers many unique contributions which have not been explored in the literature. Compared with existing systems offering CEP / SP services (Section 3.1.3), *CEPaaS* differentiates itself by operating at a higher level of abstraction. By offering pre-defined query templates, *CEPaaS* can be used by end users, yet it can still be extended by developers whenever needed. *CEPaaS* also provides a complete solution and does not need to integrate with other systems and services.

*CEPaaS* leverages multi-cloud environments to improve the system QoS, whereas most other systems do not. Similarly to Photon [14], user queries run on multiple datacentres to provide datacentre-level fault tolerance. In addition, *CEPaaS* uses multiple clouds to increase application distribution and stay (geographically) closer to event producers and consumers. As shown in Chapter 7, this feature considerably reduces query latency.

In terms of architecture, *CEPaaS* is most similar to Twitter's Heron [95], as both are based on application containers that are scheduled into a shared infrastructure. Nevertheless, Heron is used to run Twitter workloads, whereas *CEPaaS* is a multi-tenancy service. This difference is reflected in the granularity of what resides in a container: Heron containers run many tasks belonging to a single topology, whereas *CEPaaS* runs all tasks from a particular user query. This decision has been made because it facilitates resource allocation and control according to the tenant subscription level. Apache Samza [17] also uses containers to schedule jobs in a YARN cluster [147], but as already mentioned, Samza does not include the concept of a query and is not designed to be multi-tenant.

[2]Naiad [116] and Google Dataflow [5] languages can also be considered DAG-based.

*CEPaaS* also shares similarities with the Esc system [133] because both use an autonomic manager to control queries and the execution environment. *CEPaaS*, however, is based on the *AGeCEP* formal model (Chapter 4) of reconfiguration actions that guarantee correctness of reconfigured queries. Moreover, the proposed *AGeCEP* formalism can be extended with new query operators that are seamlessly integrated into the autonomic manager.

Furthermore, it is important to emphasize how *CEPaaS* relates with other open-source SP / CEP platforms such as Storm [18] and S4 [117]. In fact, *CEPaaS* provides abstractions on top of these frameworks, and it is even possible to use them as a query execution engine. Nevertheless, *CEPaaS* opted for an implementation based on the Akka toolkit [6] because it does not require a central manager and is lower-level, which enables more flexibility.

Finally, many of the fault-tolerance, scalability, and elasticity techniques used by modern cloud-based systems (Section 3.1.2) can be integrated into *CEPaaS* and are orthogonal to this thesis. Indeed, the authors plan to explore query parallelization and fault tolerance of stateful operators as future work.

## 3.2 CEP Formal Models

The *AGeCEP* formalism presented in Chapter 4 was created to provide a technology- and language-agnostic representation of queries, and to enable creation of generic and reusable procedures for CEP query management. On the contrary, most previous research into CEP formal models was developed in the context of specific query languages [20, 30]. These models attach semantics to queries written using these languages yet they generally cannot be applied to other contexts without significant adaptation.

More recent research has targeted the development of language-independent formalisms for CEP [71, 93]. These authors recognized the importance of a generic model to enable formal analysis of user-defined queries, including procedures such as correctness checking and query equivalence determination. Nevertheless, these models differ significantly from *AGeCEP* because they focus only on defining query semantics and do not include reconfiguration actions.

Sharon and Etzion [138] proposed the *event processing network* (EPN) formalism as a way to specify event-based applications independently of the underlying implementation. More recently, Rabinovich *et al.* [129] and Weidlich *et al.* [149] built upon Sharon and Etzion's research by implementing simulation, static, and dynamic analysis of EPNs. EPNs share similarities with *AGeCEP* because they are also language-agnostic and use directed graphs as their basic representation. However, the main goal of EPNs is to represent applications that can be translated into system-specific queries, whereas the proposed *AGeCEP* aims to provide a

generic query representation.

Cugola *et al*. [42], on the other hand, proposed an approach and an accompanying tool called CAVE that can be used to prove generic properties about user-defined queries. They convert queries written in different languages into a pattern-based model, which is transformed along with the properties to be proved into a constraint satisfiability problem. If a solution to this problem exists, then the properties are proven to be true.

The REX tool [49] is similar to CAVE, as it also aims to prove generic properties about application queries. REX, however, uses a formalism based on timed automata. Both CAVE and REX are very specific to their purposes and are not as appropriate to represent queries and generic management procedures as *AGeCEP*.

Finally, Hong *et al*. [82] presented the work that most closely approximates *AGeCEP* objectives. In their research, queries written in both declarative and pattern-based languages are converted to a graph-based query execution plan, and a set of transformation rules is applied to optimize them. Note, however, that their focus is solely on multi-query optimization, whereas *AGeCEP* targets procedures covering the entire query lifecycle.

## 3.3 Cloud Computing Simulators

Simulators have been used in different fields, such as grid computing [33], to overcome difficulties related to the execution of repeatable and reproducible experiments. More recently, the usage of simulators in cloud computing has become widespread and a number of simulators has been developed such as CloudSim [34], GreenCloud [92], and iCanCloud [119]. In the context of CEP systems, however, there are no such simulators available. Because of this limitation, this section reviews cloud computing simulators instead. As it will be described in Chapter 6, one of these cloud simulators (CloudSim) is used as a basis for *CEPSim*.

CloudSim [34] is a well-known cloud computing simulator that can represent various types of clouds, including private, public, hybrid, and multi-cloud environments. In CloudSim, users define workloads by creating instances of *cloudlets*, which are submitted and processed by virtual machines (VMs) deployed in the cloud. Among the most interesting CloudSim features is the customizability of its resource management policies, such as:

- *VM allocation (provisioning)*: determines how to map a user-requested VM to one of the physical hosts available in a datacentre. Cloud providers normally use strategies that try to maximize the utilization of their servers without violating existing service level agreements (SLA).

- *VM scheduling*: determines how the VMs deployed on a physical host share the available

processing elements (PEs). Currently, CloudSim provides two VM scheduling policies: space-shared and time-shared. In the former, each VM has exclusive access to the PEs to which it is allocated, whereas in the latter, VMs share the host PEs by executing on slices of the available processing time.

- *Cloudlet scheduling*: determines how the cloudlets running in a VM share the available VM PEs. Similarly to *VM scheduling*, both space-shared and time-shared strategies are available.

The major drawback of CloudSim to simulate CEP is its simple application model, which is more appropriate for simulation of batch jobs. Normally, a *cloudlet* represents an independent finite computation with a length defined by a fixed number of instructions. Moreover, the *cloudlet*'s internal state other than its expected finish time is invisible. CEP queries, on the other hand, are continuous computations that run indefinitely or for a specific period of time. In addition, tracking queries' internal state during simulation is essential to the analysis of any given CEP system. For example, by monitoring the query operators' queue size, one can determine whether the operators are keeping up with the incoming event rate. The work discussed in this research circumvents the limited CloudSim application model with a new model based on *AGeCEP*, as discussed in Chapter 6.

Because of its limitations, CloudSim has already originated many extensions in the literature [56, 66, 68]. Garg and Buyya [56] created NetworkCloudSim, which extends CloudSim with a three-tier network model and an application model that can represent communicating processes. Grozev and Buyya [66], on the other hand, presented a model for three-tier Web applications and incorporated it into CloudSim. Finally, Guérout *et al*. [68] focused on implementing the Dynamic Voltage and Frequency Scaling (DVFS) model on CloudSim. These extensions are orthogonal to *CEPSim* because they do not focus on CEP.

Conversely, GreenCloud [92] is a cloud simulator developed as an extension of the network simulator NS-2 [118]. Differently from CloudSim, GreenCloud focuses on packet-level simulation and energy consumption of network equipment, but not on modelling of complex applications.

The iCanCloud simulator [119], on the other hand, provides functionalities that are more similar to CloudSim. In addition, it can also parallelize simulations and has a GUI for user interaction. Its application model, however, is based on low-level primitives and needs to be significantly customized to represent CEP applications. The choice of CloudSim over iCanCloud in this research was motivated by CloudSim's more mature codebase, the authors' previous experience, and the larger number of extensions available for CloudSim.

## 3.4 Summary

This chapter presented a comprehensive review of research related to the contributions developed in this thesis. It started reviewing the most important CEP / SP systems, from traditional research projects that established current terminology to modern systems that leverage cloud-based architectures to improve the quality of service offered to users. Following, this chapter also described projects that aim to offer CEP / SP in the services model and CEP systems based on multi-cloud architectures. Finally, this chapter discussed CEP formal models and cloud computing simulators.

In the next chapter, the first contribution of this research is presented: the **A**ttributed **G**raph R**e**writing for **C**omplex **E**vent **P**rocessing Management (*AGeCEP*) formalism.

# Chapter 4

# Attributed Graph Rewriting for CEP Management - Concepts

This chapter[1] introduces the **A**ttribute **G**raph R**e**writing for **CEP** Management (*AGeCEP*) formalism. It starts with a discussion about *AGeCEP* motivation and benefits (Section 4.1) and with an introduction of the basic ideas on which the formalism is based (Section 4.2).

As it will be detailed further, *AGeCEP* requires the characterization of CEP operators according to their reconfiguration capabilities. To enable this characterization, a generic classification of CEP operators is presented in Section 4.3. Finally, Section 4.4 details how the formalism represents queries and their reconfigurations.

## 4.1 Motivation

Despite the recent surge of interest in CEP systems, the current CEP research landscape is still young and fragmented. As mentioned in Section 2.1, a large variety of solutions exist and they often use inconsistent terminology and different query definition languages. Consequently, most ongoing research and development is performed in the context of specific systems and languages.

Of particular interest for this research, algorithms and techniques aimed at query lifecycle management (QLM) have often been developed in such a system-specific fashion. For instance, Aurora* can dynamically move processing load to neighbouring servers [38], and Nephele can dynamically resize the output buffers of query operators [103]. Both these examples illustrate important query management techniques, in which the system self-adapts to changing conditions. However, they were developed in the context of their respective systems and cannot be

[1] A conference paper containing preliminary results from this chapter has been published [75]. The contents of this and the next chapter have also been submitted as a journal paper which has been accepted for publication [79].

easily generalized.

In the context of the *CEPaaS* system, this fragmentation is even more critical because:

- *CEPaaS* is a user-facing service and, therefore, it must be flexible regarding the interface and language used to define queries. For instance, some users may prefer to create queries using a visual language whereas others prefer to write SQL-like statements.
- *CEPaaS* provides high availability, low-latency, and elasticity by leveraging cloud environments. The management of large cloud deployments leads to complex algorithms and reinforces the need to reuse results from related research.
- *CEPaaS* accepts user definition of new operators. The ability to integrate them to the query management loop and to treat them as first-class citizens is essential to the system.

To overcome these challenges, this research introduces **A**ttributed **G**raph R**e**writing for **C**omplex **E**vent **P**rocessing Management (*AGeCEP*), a formalism that provides a technology- and language-agnostic representation of queries and of reconfiguration actions that act on the queries.

In *AGeCEP*, queries are represented as directed acyclic graphs whose vertices and edges are augmented with a standardized set of attributes. These attributes characterize operators according to their reconfiguration capabilities and can be used for decision making in management procedures. Reconfiguration actions, in turn, are defined with graph rewriting rules based on the Single-Pushout approach [104]. *AGeCEP* rules consider the vertices' characteristics, as expressed by their attributes, to decide whether a rule can be applied. By doing so, the formalism can establish correctness guarantees for reconfigurations: they are never applied to incompatible operators and queries. In addition, *AGeCEP* rewriting rules are also associated to mutators that are executed as side-effect of rule application. This mechanism guarantees that changes performed in the query models are correctly reflected in the real system.

*AGeCEP* query model provides a common representation to which different query definition languages can be converted, including languages that accept user-defined operators. In addition, by also providing a model for reconfigurations, *AGeCEP* establishes a common ground through which most management procedures can be expressed. These procedures, in turn, can be applied to control any CEP system that uses *AGeCEP* as its underlying formalism, including the *CEPaaS* system. Therefore, *AGeCEP* facilitates not only the *understanding* of procedures from existing systems, but also their *reuse* and application to other contexts.

Because it is a formal model, *AGeCEP* also enables formal analysis of user queries and their transformations, including procedures such as structure validation, correctness checking, and equivalence determination. These procedures, however, are out of scope of this research.

The following sections discuss in details the *AGeCEP* formalism. A complete evaluation of *AGeCEP* is presented in Chapter 5.

## 4.2 Attributed Graph Rewriting for CEP Management

The **A**ttributed **G**raph R**e**writing for **C**omplex **E**vent **P**rocessing Management (*AGeCEP*) formalism has been developed to enable specification of management procedures and, in particular, self-management policies that can be applied to QLM in CEP systems.

To achieve this goal, two main challenges have to be overcome: the first was to find a query representation that is language agnostic, yet can encode all information required by the management procedures. The second was to find a way to specify unambiguous reconfiguration actions that act on the represented queries. The following subsections discuss these challenges further.

### 4.2.1 Modelling Queries

*AGeCEP* represents CEP queries as Attributed Directed Acyclic Graphs (ADAGs). Given a query graph, each vertex represents a query element, and each edge represents an event stream flowing from one element to another. In such a graph, query elements are further classified as:

- *event producers*: sources of events processed by the query;
- *event consumers*: consumers of query results;
- *query operators*: any processing logic that can be applied to one or more input streams and generates one or more output streams as a result.

Because the graphs used are attributed, it is possible to represent properties that qualify the vertices and edges and enrich their representation. Here, the attributes considered should include all pieces of information required by management procedures.

To identify these properties, a novel classification of query operators focusing on their reconfiguration capabilities was elaborated. Integrating a new operator into *AGeCEP*, therefore, is simply equivalent to classifying it properly. Details of this classification are presented in Section 4.3.

Use of ADAG as a language agnostic representation of CEP queries is a natural choice corroborated by many studies in the literature. For instance, most CEP systems based on imperative languages use (non-attributed) DAGs to represent user queries [1, 18, 117]. Systems that use declarative languages, on the other hand, transform user queries into query plans to

make them "executable", which often leads to DAG-like structures (e.g. the STREAM system and the CQL language [20]). Finally, systems based on pattern-based query languages may use alternative representations that cannot be directly converted to DAGs. However, even in this case, previous research [82] has shown that these queries can be converted to DAG structures and consequently, to the *AGeCEP* representation. To demonstrate the generality of *AGeCEP*, conversions from all three query language groups are further discussed in Section 4.4.1.

### 4.2.2 Modelling Reconfiguration Actions

In CEP systems, management procedures may act on different steps of the query lifecycle and have various goals. In a broad sense, however, they all follow a similar structure in which: (i) potential problems are detected; (ii) appropriate reconfiguration actions are selected; and (iii) the selected actions are applied as a response.

In this structure, problem detection and action selection are mostly independent of the chosen query representation. On the other hand, the representation and enforcement of reconfiguration actions is heavily influenced by this choice. *AGeCEP*, therefore, also focuses on the definition and representation of reconfiguration actions. These actions can be applied to reconfigure queries and can be used by any procedure, including but not limited to self-management policies. More precisely, because this research focuses on reconfiguration of queries modelled by ADAGs, it is natural to represent the actions under consideration using a graph transformation formalism.

Such reconfigurations can be modelled formally, yet visually and intuitively by graph rewriting rules. Graph rewriting is a well-studied technique [132] with multiple applications [124, 135], including self-management [47, 130]. In particular, a graph rewriting rule formally specifies both a reconfiguration (i.e., its effect) and the context in which it can be applied (i.e., its applicability), enabling the study and establishment of guarantees of reconfiguration correctness [48, 81].

### 4.2.3 Discussion

In the context of self-management policies and autonomic computing, *AGeCEP* queries and reconfiguration actions are part of the *knowledge base* (KB). Specifically, *AGeCEP* focuses on representing *"what and how it can be done"* and not on the decision making process that determines *"what should be done"*. The MAPE-K modules of an autonomic manager are expected to use *AGeCEP* to implement their functions in conjunction with other information available in the KB such as monitored events and inference rules.

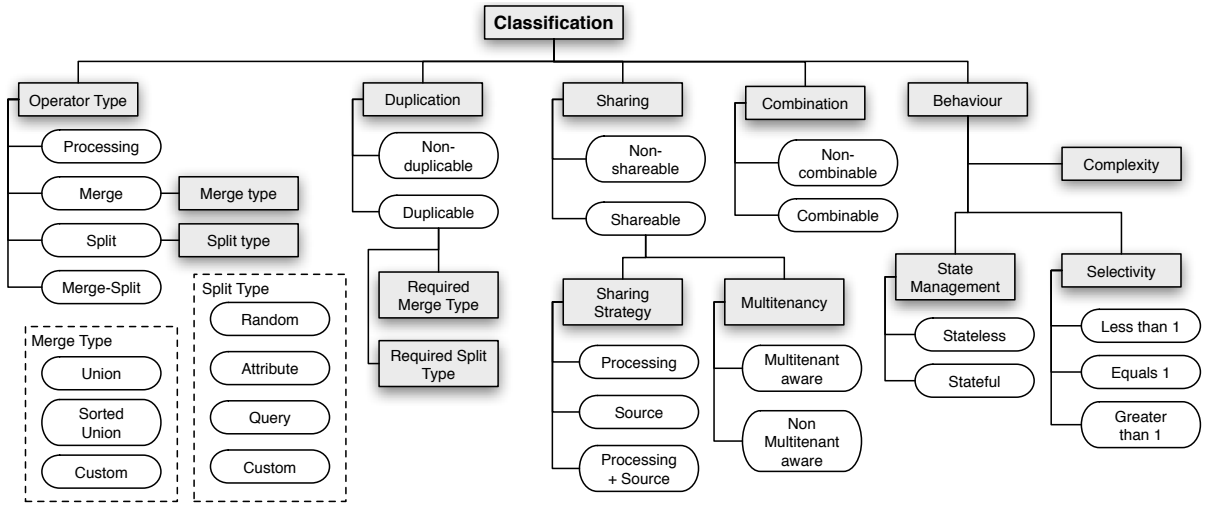



Figure 4.1: *AGeCEP* classification of operators.

In particular, it is expected that additional information will be present in the KB to model the CEP system runtime environment. This information is also essential for QLM policies, especially for the operator placement and runtime management steps.

Note that by limiting *AGeCEP* scope to queries and reconfiguration actions, it is possible to integrate *AGeCEP* with existing models and techniques rather than forcing the adoption of particular ones. By doing so, *AGeCEP* can be applied to a broader range of scenarios.

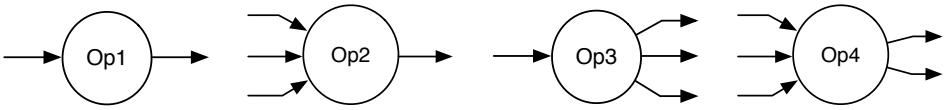
Section 5.1 discusses how existing representations and meta-models can be coupled with *AGeCEP* to cover the whole MAPE-K loop and thereby implement a complete autonomic manager.

## 4.3 Classification of CEP Operators

One underlying purpose of *AGeCEP* is to abstract queries and operators while expressing any information relevant to their management. To achieve this goal, this section identifies a set of criteria related to operator management and presents a novel classification of CEP query operators focused on their reconfiguration capabilities.

This classification is at the core of *AGeCEP* approach to query reconfigurations. As it will be discussed in Section 4.4, *AGeCEP* rewriting rules are applicable to virtually any set of properly classified CEP operators.

Figure 4.1 presents an overview of the criteria on which the operators are classified. Each criterion is detailed in the subsequent subsections.




(a) Processing. (b) Merge. (c) Split. (d) Merge-Split.

Figure 4.2: Operator types - examples.

### 4.3.1 Operator Type

The *type* criterion classifies operators according to the number of input and output streams. There are four different categories in this criterion, illustrated in Figure 4.2:

- *Processing*: the operator has one input and one output stream only. These operators can filter events from the input stream, transform them, or both.
- *Merge*: the operator has two or more input streams, which are processed together and merged into one output stream.
- *Split*: the operator has one input stream, which is processed and split into two or more output streams.
- *Merge-Split*: the operator has more than one input stream and more than one output stream.

Merge operators are sub-classified according to the type of merge they execute:

- *Union*: input events are output as they arrive, with no ordering guarantees.
- *Sorted union*: input events are output sorted based on a specified set of attributes.
- *Custom*: a customized function defines how the input streams are merged.

Finally, split operators are also characterized based on the type of split they perform:

- *Random*: input events are sent to a randomly selected output stream.
- *Attribute*: the output stream is selected based on the values of a specified set of attributes.
- *Query*: input events are split according to the query from which they come. This can be considered as a special type of *attribute* split in which the attribute under consideration is the query id of the incoming event.
- *Custom*: a customized function defines how the input events are split.

### 4.3.2 Sharing

The *sharing* criterion refers to the ability of a single runtime instance to be shared by two or more occurrences of an operator. This characteristic is especially important for multi-query optimization, in which the results of common query subgraphs are reused among queries.

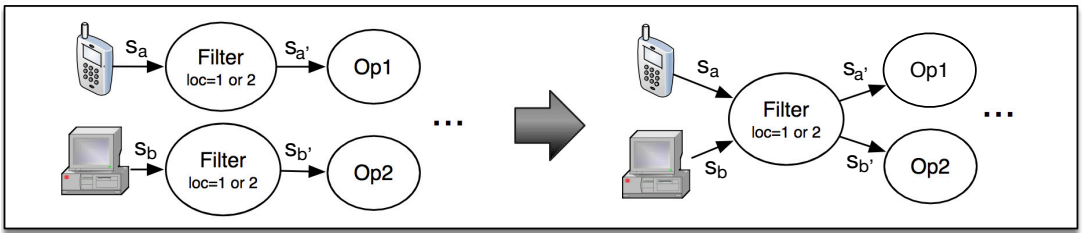
This criterion is essentially determined by the operator implementation. An operator is *non-shareable* if one runtime instance must be created for each operator occurrence. On the other hand, an operator is *shareable* if a single runtime instance can implement more than one occurrence. In this case, three *sharing strategies* are identified:

- *Processing*: one operator instance is shared among occurrences that execute the exact same processing, but using different input streams as sources.
- *Source*: one operator instance is shared among occurrences that execute similar processing using the same input streams as sources.
- *Processing+Source*: one operator instance is shared among occurrences that execute the same processing on the exact same input streams.

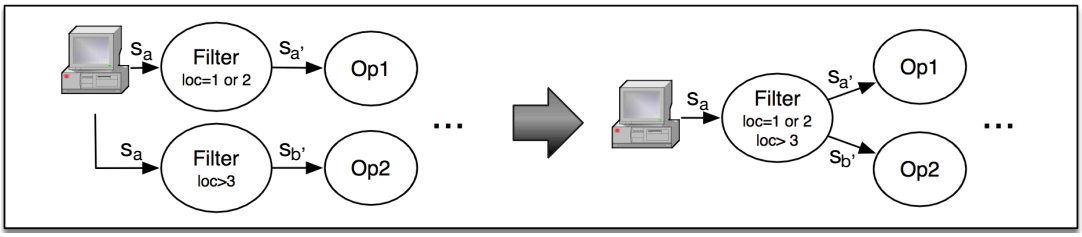
Figure 4.3a informally illustrates an example of a *processing* shareable operator. In this case, a single instance can be used to process both input streams $s_a$ and $s_b$, as represented in the right-hand part of the figure. This sharing is possible because the same filter ($loc = 1$ or 2) is applied to both streams. Moreover, note that the operator implementation must keep track of the event sources to send the results to the correct output stream. This type of sharing is usually applied when an operator instance consumes a lot of memory, and it is therefore important to create as few instances as possible.

In the example from Figure 4.3b, the filter operator is *source* shareable. In this case, both filter occurrences process the same input stream $s_a$ and have predicates over the attribute *loc*. The resulting filter instance implements both predicates and maintains the outputs of each original operator. This type of sharing is applied when it is more efficient to implement multiple processing logics as a single operation than it is to implement these logics independently. For instance, the *predicate indexing* technique presented by Madden *et al*. [109] enables source shareable filters.

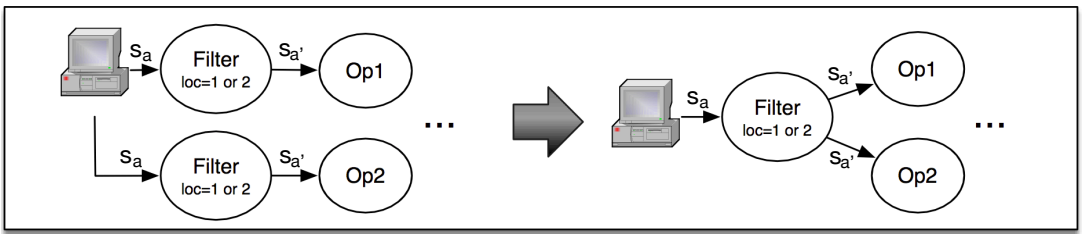
Finally, in Figure 4.3c the filter operator is assumed to be *processing+source* shareable. In this example, the exact same data processing is executed on the same input stream, and therefore only a single instance is necessary. However, the operator must duplicate all output events and send them to all original output streams. This type of sharing enables savings in both memory and CPU consumption and is the most commonly used by CEP systems.




(a) Processing sharing.




(b) Source sharing.




(c) Processing+Source sharing.

Figure 4.3: Sharing strategies.

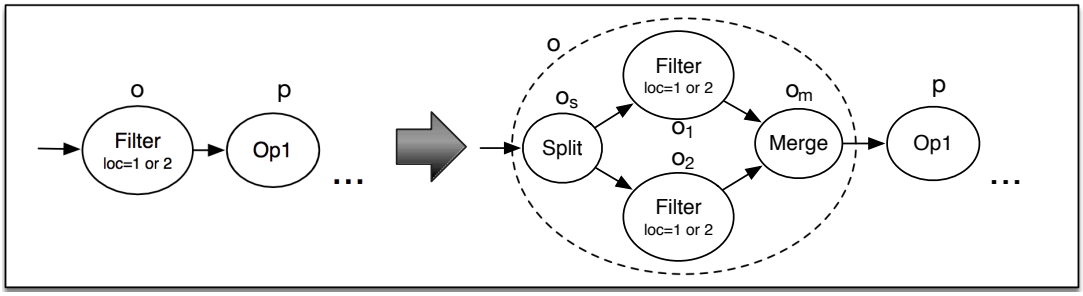



Figure 4.4: Duplication strategy.

In addition to their sharing strategy, shareable operators can also be categorized according to their *multi-tenancy* support:

- *Multitenant-aware*: the operator can be shared by queries from different tenants.
- *Non-multitenant-aware*: the operator can be shared only among queries from the same tenant.

This criterion is important in the *CEPaaS* scenario because the system is used by many customers at the same time. In this case, a *multitenant-aware* operator needs to guarantee that customer-related state is kept isolated and that its implementation is independent of customer-specific data. If these conditions are satisfied, the operator can be shared among queries from different tenants according to the sharing strategy criterion.

### 4.3.3 Duplication

A common strategy used to increase operator throughput is to create more than one instance of the operator, assign them to different servers (or cores), and split the input events among these instances. This strategy is illustrated in Figure 4.4.

Because of the prevalence of this strategy, the proposed classification contains a *duplicable* criterion, which is true when the operator can be duplicated and the processing load distributed according to the described strategy. Moreover, when an operator is *duplicable*, two other aspects must be considered: the *required split type*, and the *required merge type*. These two criteria determine the type of split (merge) operator that precedes (succeeds) the duplicated operator. The possible split (merge) types are the ones defined in Section 4.3.1.

The *required split* and *required merge* types are ultimately defined by the duplicated operator implementation. Generally speaking, stateless operators can be duplicated and accept *random* splits because each event is processed in isolation. Conversely, stateful operators usually require *attribute* splits because they process together events with similar characteristics

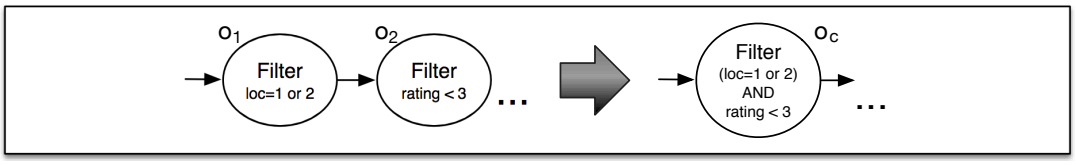



Figure 4.5: Combination strategy.

(same attribute values). The *query* split is a less common strategy that is used mostly by *processing shareable* operators that are also *duplicable*.

Moreover, note that operators requiring *random* splits can actually be preceded by any type of split. In this sense, a random split is considered weaker than the others because it imposes fewer constraints on how the split should be done.

Finally, note that a *sorted union* merge type is needed in scenarios in which the output stream must be kept ordered after duplication. For example, in Figure 4.4, there is no guarantee that the events will reach the merge operator $o_m$ in the same order they reached the split $o_s$. If order needs to be maintained, then the operator $o_m$ must be a *sorted union*.

### 4.3.4 Combination

The *combinable* criterion is true when two or more consecutive occurrences of an operator $o$ can be combined into a single operator $o_c$, whose effect is equivalent to applying all the combined operators in any order. Figure 4.5 illustrates this criterion applied to a filter operator. It is clear that two consecutive filters using different predicates can be combined into a single filter with a new predicate defined as the conjunction of the original predicates.

In most cases, the combined operators $o_i$ and the equivalent one $o_c$ have the same implementation. In other cases, $o_c$ is different. For example, two binary joins may be combined as a multiple join operator, which usually has a different implementation than the binary operator. Hence, the implementer of a combinable operator $o$ has the responsibility to provide:

- the implementation of operator $o_c$ resulting from the combination of $o$ instances.

- a function that, given the parameters of successive instances of $o$, returns the parameters of the equivalent combined operator $o_c$.

This criterion is especially useful for SQO, in which operators can be combined to reduce the number of operators in a query.

### 4.3.5 Behaviour

This category groups the characteristics of an operator related to its functional behaviour. More specifically, operators are classified according to three criteria:

- *Complexity*: refers to the computational complexity of an operator as a function of the size of the input streams.

- *State Management*: indicates whether the operator is *stateless* or *stateful*. A *stateless* operator processes each event in isolation, whereas a *stateful* one maintains internal state that is regularly updated with the arrival of new events.

- *Selectivity*: refers to the relation between the number of output and input events. An operator selectivity less than one means that the number of output events is less than the number of input events, whereas a selectivity greater than one implies that the number of output events is larger than the number of input events.

### 4.3.6 Discussion

The classification presented in this section has been created based on an extensive literature review of query lifecycle management research. It focuses on intrinsic reconfiguration capabilities of query operators that are crucial to establish how they can be reconfigured. As demonstrated in Section 5.3, these properties enable the expression of a myriad of different procedures in the context of CEP systems.

Nevertheless, it is expected that not all properties required by current and future systems are expressed in this classification. For this reason, the classification can be easily extended with other criteria as needed. In addition, extrinsic operator properties, such as runtime information, are not part of the classification because they are too numerous and tightly coupled to the management procedures that use them. Section 4.4.1 discusses how new criteria and attributes are handled in *AGeCEP*.

## 4.4 Representation of Queries and Reconfigurations

*AGeCEP* provides graph-based models to represent two fundamental aspects of dynamic CEP systems: the system state, which is primarily defined by the running queries; and possible reconfigurations of this state, given by a set of reconfiguration actions. The following subsections detail both models.

### 4.4.1 Query Representation Using ADAGs

In *AGeCEP*, each user-defined query $q$ is represented by an *attributed directed acyclic graph* $G$. Because the graph is attributed, the vertices and edges are augmented with a set of attributes that qualify them. Formally, such an attributed graph can be specified by a triple $(V, E, ATT)$, where:

- the vertices $V$ represent the query elements,
- the edges $E$ represent event streams flowing from one element to another,
- and $ATT$ is a family of attribute sets indexed by $V \cup E$.

Formally, each set of the family $ATT$ is defined as a sequence of triples $(N, L, T)$, where $N$, $L$, and $T$ are the attribute name, value, and type (i.e., domain) respectively.

To represent the types of elements and interactions that may be involved in a CEP system, *AGeCEP* also defines stereotypes for the vertices and edges of a query graph. Each stereotype specifies a set of attributes that are common to elements of that specific stereotype.

#### Vertex Attributes

The vertices from a query graph $G = (V, E, ATT)$ can represent event producers, event consumers, or query operators, denoted as $V_p$, $V_c$ and $V_o$ respectively. $V_p$, $V_c$ and $V_o$ specify a partition of $V$, i.e., all sets are disjoint subsets of $V$, and their union is $V$.

*Query operators* all belong to the same stereotype and therefore share the same list of attributes depicted in Table 4.1. The nature of these attributes is directly related to the properties considered relevant for defining self-management policies, which were identified in the classification presented in Section 4.3. As mentioned, this classification is extensible and new criteria can be added as needed. In this case, the new criteria translate directly to new attributes, and the possible values for the criteria correspond to the attributes domain.

*Event producers and consumers* also define their own stereotypes, which contain the first five attributes of the operator stereotype: *id*, *impl*, *params*, *inDegree*, and *outDegree*. Event producers (consumers) necessarily have an *inDegree* (*outDegree*) equal to 0.

#### Edge Attributes

*AGeCEP* uses a single stereotype for edges. The attributes of this stereotype are described in Table 4.2. Note that except for *id*, all edge attributes can be inferred from the graph structure and vertex attributes. Similarly, the *inDegree* and *outDegree* of a vertex can also be inferred

| Name | Type | Description |
|---|---|---|
| *id* | String | a unique identifier |
| *impl* | String | the operator implementation name |
| *params* | List of Strings | the operator parameters |
| *inDegree* | $\mathbb{N}$ | the number of incoming edges |
| *outDegree* | $\mathbb{N}$ | the number of outgoing edges |
| *type* | {“processing”, “merge”, “split”, “merge-split”} | operator type |
| *mergeType* | {“union”, “sorted”, “custom”, “N/A”} | if *type* =“merge”, the merge type |
| *splitType* | {“random”, “attribute”, “query”, “custom”, “N/A”} | if *type* =“split”, the split type |
| *shareable* | Boolean | Is the operator shareable? |
| *shStrategy* | {“processing”, “source”, “proc+source”, “N/A”} | if *shareable*, the sharing strategy |
| *multitenant* | Boolean | and the multitenant awareness |
| *combinable* | Boolean | Is the operator combinable? |
| *combImpl* | String | if *combinable*, the combined operator $o_c$’s impl. name |
| *combParam* | fun: List of List of strings → List of strings | and $o_c$’s parameters function |
| *duplicable* | Boolean | Is the operator duplicable? |
| *reqMerge* | {“union”, “sorted”, “custom”, “N/A”} | if *duplicable*, the succeeding *mergeType* |
| *reqSplit* | {“random”, “attribute”, “query”, “custom”, “N/A” } | and the preceding *splitType* |
| *stateful* | Boolean | Is the operator stateful? |
| *selectivity* | {“1”, “< 1”, “> 1” } | the operator selectivity |
| *complexity* | {“logn”, “n_logn”, “n”, “n2”, “exp” } | the operator complexity |

Table 4.1: Attributes of the vertex stereotype “operator”.

| Name | Type | Description |
|---|---|---|
| *id* | String | a unique identifier |
| *sources* | the power set of $V_p$ | producers of events flowing through the edge |
| *queries* | List of Strings | the set of queries that share the edge |
| *attrs* | List of Strings | name of attributes according to which the events in the edge are grouped |

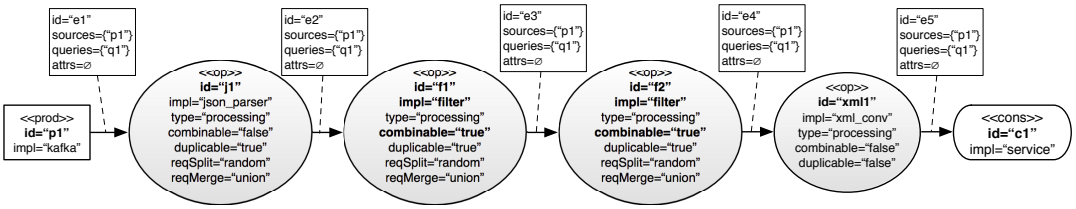
Table 4.2: Edge attributes.

from the graph. Nevertheless, they are maintained as attributes to simplify the definition and implementation of reconfiguration actions.

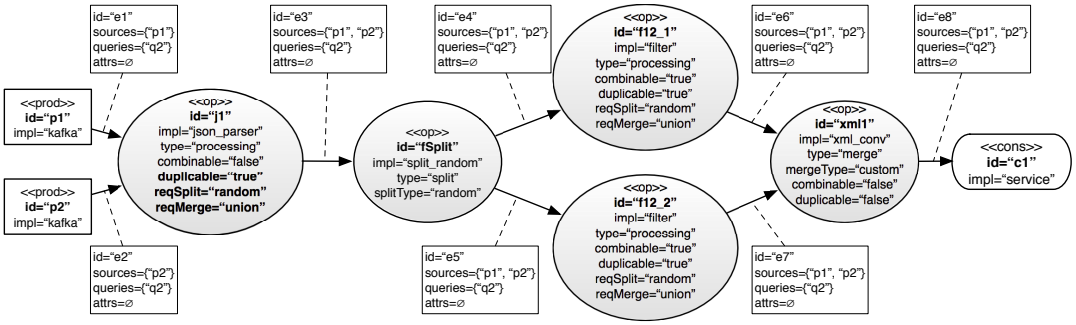
Furthermore, it should be emphasized that neither vertex nor edge attributes are closed sets and can be extended whenever necessary. In particular, extrinsic properties such as operator placement and runtime information can also be modelled as vertex and edge attributes.

### Example: *AGeCEP* Query Representation

Figure 4.6 shows two queries $q_1$ and $q_2$ using the *AGeCEP* representation. To simplify the figure, some attributes have been omitted; only attributes relevant to the experiments in Section 5.4 are included. The notation $\langle\langle op \rangle\rangle$ specifies that the vertex is of the operator stereotype,




(a) Query $q_1$ - Sequential version.




(b) Query $q_2$ - Duplicated filtering.

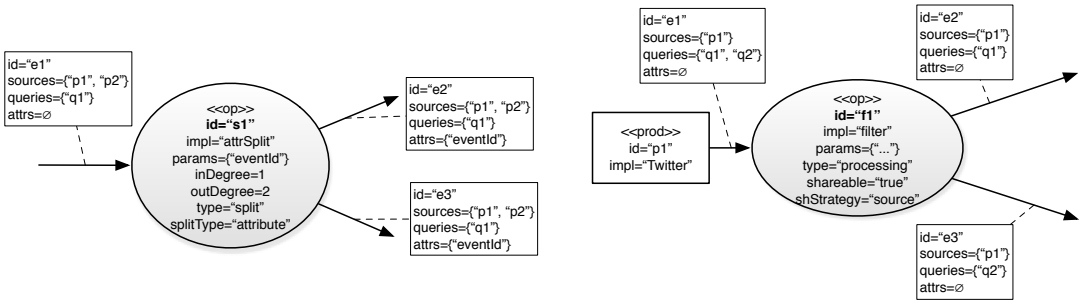
Figure 4.6: JSON to XML conversion - Storm queries.

whereas $\langle\langle prod \rangle\rangle$ and $\langle\langle cons \rangle\rangle$ qualify an event producer or an event consumer. These queries have been extracted from the Powersmiths' WOW system [127], a sustainability management platform that uses live measurements of buildings to support energy management and education. In WOW, queries are implemented in Apache Storm [18] and are used to process readings coming from building sensors managed by the platform. The conversion from Storm queries to *AGeCEP* is straightforward because Storm also represents queries using DAGs.

Query $q_1$ in Figure 4.6a is used to convert sensor readings from the JSON format to the native WOW format (XML). The query is implemented as a sequence of four operators: first, operator $j_1$ converts the JSON reading to a Java object. Next, filters $f_1$ and $f_2$ remove invalid readings from the event stream. Finally, operator $xml_1$ converts the reading to an XML document and forwards it to the appropriate service.

Query $q_2$ in Figure 4.6b is used for the same purpose, but has a different structure. First, two producers are attached to the JSON parser operator $j_1$. Following, $j_1$ connects to a split operator, which distributes the incoming events randomly between two instances of operator $f_{12}$. Each instance of $f_{12}$ executes a processing (filtering) logic equivalent to the sequential application of $f_1$ and $f_2$. In particular, for this graph, the fact that the operators process events coming from both producers $p_1$ and $p_2$ is reflected in the *sources* attributes of edges $e_3$ to $e_8$.

To provide further illustration of edge attributes, additional examples are shown in Figure 4.7. In Figure 4.7a, the attribute *sources* of edge $e_1$ indicates that $s_1$ processes events coming from producers $p_1$ and $p_2$. The attribute *attrs* of edges $e_2$ and $e_3$, on the other hand,




(a) *attrs* attribute. (b) *queries* attribute.

Figure 4.7: Additional examples of edge attributes.

shows that operator $s_1$ splits the input events according to the *eventId* attribute.

Moreover, in the scenario depicted in Figure 4.7b, the *queries* attribute of edge $e_1$ indicates that $p_1$ is an event producer shared between queries $q_1$ and $q_2$. Operator $f_1$, in its turn, is a *shareable* operator that produces a separate output stream for each query.

### Example: Converting from Declarative Languages

Figure 4.8 exemplifies how queries written in the CQL language [20] are converted to the ADAG format used in *AGeCEP*. The original queries $q_1$ and $q_2$ are shown in Figure 4.8a. As it is common in declarative query languages, CQL queries are transformed into a graph-based execution plan before being actually run. Figure 4.8b depicts the resulting plan for $q_1$ and $q_2$. Both queries were processed together and transformed into a single plan.

From this plan, the conversion to the *AGeCEP* representation is direct: operators and queues are mapped to vertices and edges respectively. The resulting ADAG is shown in Figure 4.8c. Note that the graph expresses most information presented in the query plan, including the fact that the *seq_window* operator can be shared among queries that process the same input sources.

### Example: Converting from Pattern-Based Languages

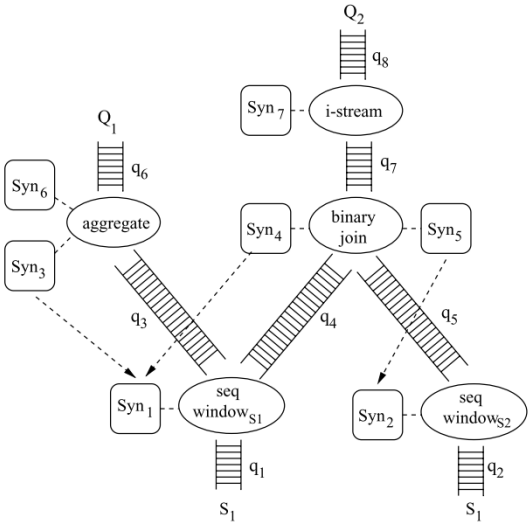
Figure 4.9 shows the conversion from a Cayuga Event Language (CEL) [44] query to *AGeCEP*. CEL is a pattern-based language, even though it uses keywords that are similar to SQL. For instance, CEL applies the operator $NEXT$ to search for a sequence of two events that satisfy a stated condition. This construct is characteristic of this language group. Other pattern-based operators, such as iteration (operator $FOLD$) and parameterization, are also part of CEL.

In Cayuga, queries are transformed into a non-deterministic finite state automaton to be executed. Figure 4.9a shows a query $q$ and its corresponding automaton. Transitions between

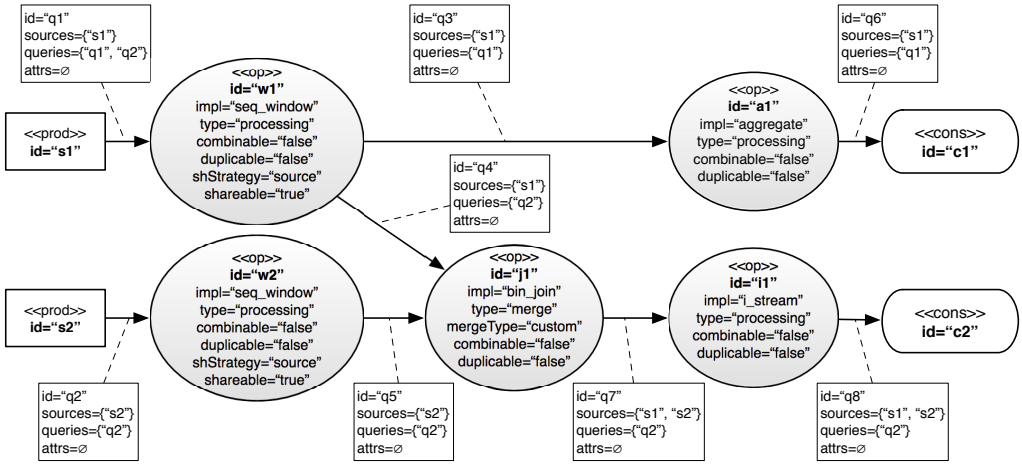
```
Q1: Select B, max(A)
    From S1 [Rows 50,000]
    Group By B
```

```
Q2: Select Istream(*)
    From S1 [Rows 40,000],
         S2 [Range 600 Seconds]
    Where S1.A = S2.A
```

(a) CQL queries [20].

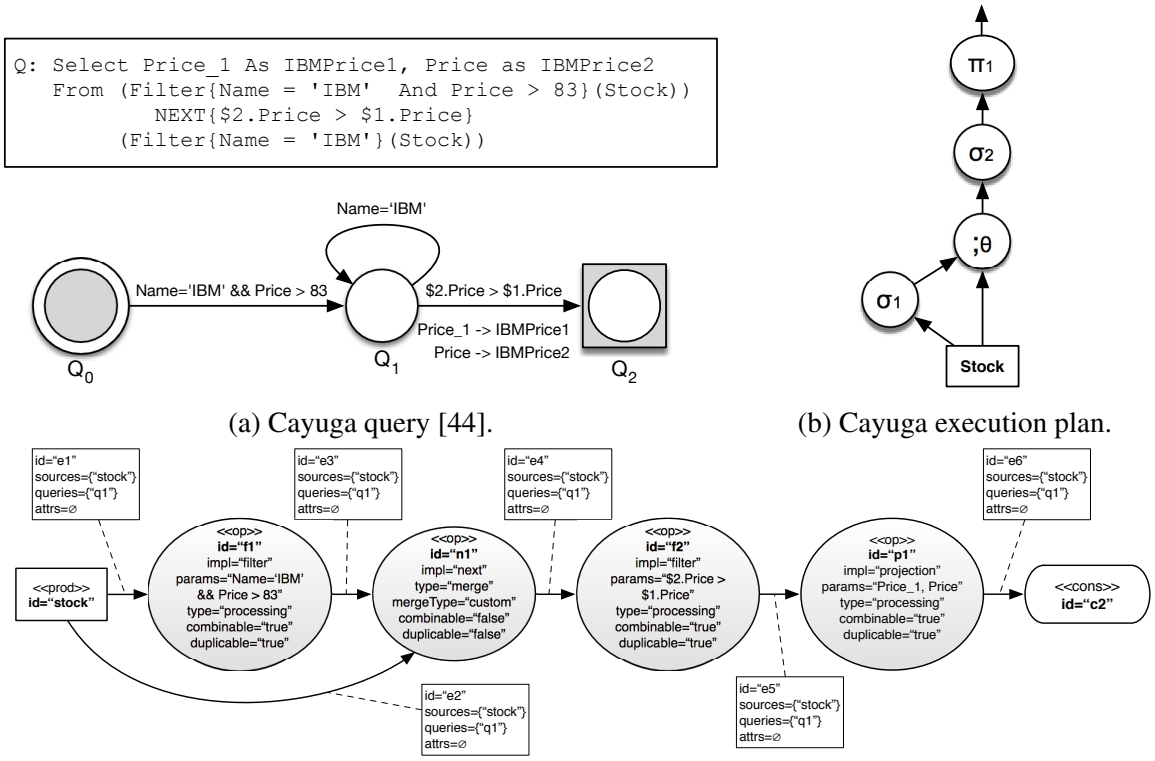



(b) Generated execution plan [20].




(c) *AGeCEP* representation.

Figure 4.8: Conversion from a CQL query to *AGeCEP*.




(a) Cayuga query [44]. (b) Cayuga execution plan.

(c) *AGeCEP* representation.

Figure 4.9: Conversion from a Cayuga query to *AGeCEP*.

states of this automaton are triggered when the conditions in the edges are satisfied. Moreover, when a transition is triggered, a function is executed to map events from one schema to another. Even though this automaton can be represented as a graph, its semantic is different from *AGeCEP* queries. For instance, the automaton states (vertices) are associated with input streams, whereas in *AGeCEP* vertices represent operators.

Hong *et al*. [82] presented a procedure to convert Cayuga automata to graph-based execution plans. Basically, they introduced two new query operators that implement the *NEXT* and *FOLD* logics and a procedure to convert edge transitions to a sequence of a filter followed by a projection. The execution plan for the example query is depicted in Figure 4.9b. Once transformed to a graph-based execution plan, the conversion to *AGeCEP* is direct and results in the ADAG shown in Figure 4.9c.

### Discussion

The *AGeCEP* query representation has been designed to be as generic as possible. Most queries written in imperative and declarative languages can be converted directly to an *AGeCEP* ADAG. Pattern-based languages, on the other hand, require additional procedures for

conversion, such as the one presented by Hong *et al*. [82] and demonstrated in the previous example. These additional procedures are needed because most pattern-based languages are executed as automata that do not follow *AGeCEP* graph-based model. In other words, there is a semantic mismatch between the models that must be solved before using *AGeCEP* to represent pattern-based queries .

Nevertheless, such mismatch should be solvable in most cases. For instance, Hong *et al*. [82] mentioned that the Sase language [150] could be transformed to a graph-based execution plan using a procedure similar to that used to transform CEL queries. Similar procedures could also be applied to TESLA [40]. The development of such procedures, however, is outside the scope of this research and may need to be analyzed case by case.

### 4.4.2 Query Reconfiguration Using Graph Rewriting

In *AGeCEP*, query reconfigurations are formally expressed in a rule-oriented fashion using graph rewriting rules.

Various ways of specifying graph rewriting rules have been developed in the past [132]. This research uses the graphical representation and underlying formalism of the AGG[2] tool [143], a well-established graph transformation environment [136]. AGG is based on the Single Push-Out (SPO) approach [45, 104].

#### Graph Rewriting Rules

The SPO approach is an algebraic technique for graph rewriting based on the category theory [22], where a rule $r$ is specified by $L \xrightarrow{m} R$, where:

- $L$ and $R$ are attributed graphs called the left-hand and right-hand sides of $r$.
- $m$ is a partial morphism from $L$ to $R$, i.e., a morphism from a sub-graph $L_m$ of $L$ to $R$. This morphism is not necessarily injective.

A rule $r : L \xrightarrow{m} R$ is applicable to a graph $G$ if $G$ contains an image of $L$, i.e., if there is a homomorphism $h$ from $L$ to $G$. Such homomorphism is denoted as $h : L \rightarrow G$. Also, the notation $h(G_s)$ is used to denote the image of some subgraph $G_s$ of $G$ by the morphism $h$. The application of $r$ to $G$ with regard to $h$ consists of constructing the push-out [22] of $m$ and $h$, as illustrated in Figure 4.10. The result of this application is the graph $m_h(G)$.

Informally, the application of $r$ to $G$ with regard to $h$ consists of replacing the image of $L$ in $G$ by an image of $R$. This can be understood as a three step process:

[2] http://user.cs.tu-berlin.de/~gragra/agg/index.html

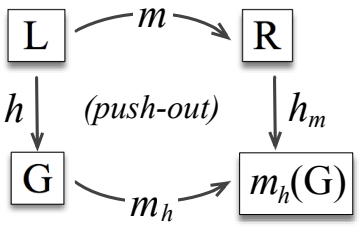



Figure 4.10: Construction of a push-out: application of a graph rewriting rule.

1. erasing the image by $h$ of the part of $L$ that is not in $m$'s domain, $h(L \backslash L_m)$.

2. adding an isomorphic copy of the part of $R$ that is not in the image of $m$ (a copy of $R \backslash m(L_m)$).

3. if $m$ is not injective, i.e., if some vertices $v_i$ of $L$ have the same image by $m$, then the images of these $v_i$ by $h$ are merged.

For the rest of this thesis, morphisms $m$ of the introduced rules may not be explicitly shown. Such morphisms are implicitly defined as the identity mapping between the largest common sub-graphs of $L$ and $R$, where vertices and edges are uniquely identified by their id.

The application of a rule $r$ to a graph $G$ is illustrated in Figure 4.11. The rule $r$ and its corresponding left- and right-hand sides $(L, R)$ are depicted in Figure 4.11a. In this rule, the morphism $m$ from $L$ to $R$ is implicit and defined by the identity mapping, as described in the previous paragraph. The highlighted nodes in $L$ and $R$ correspond to $L_m$ and $m(L_m)$ respectively.

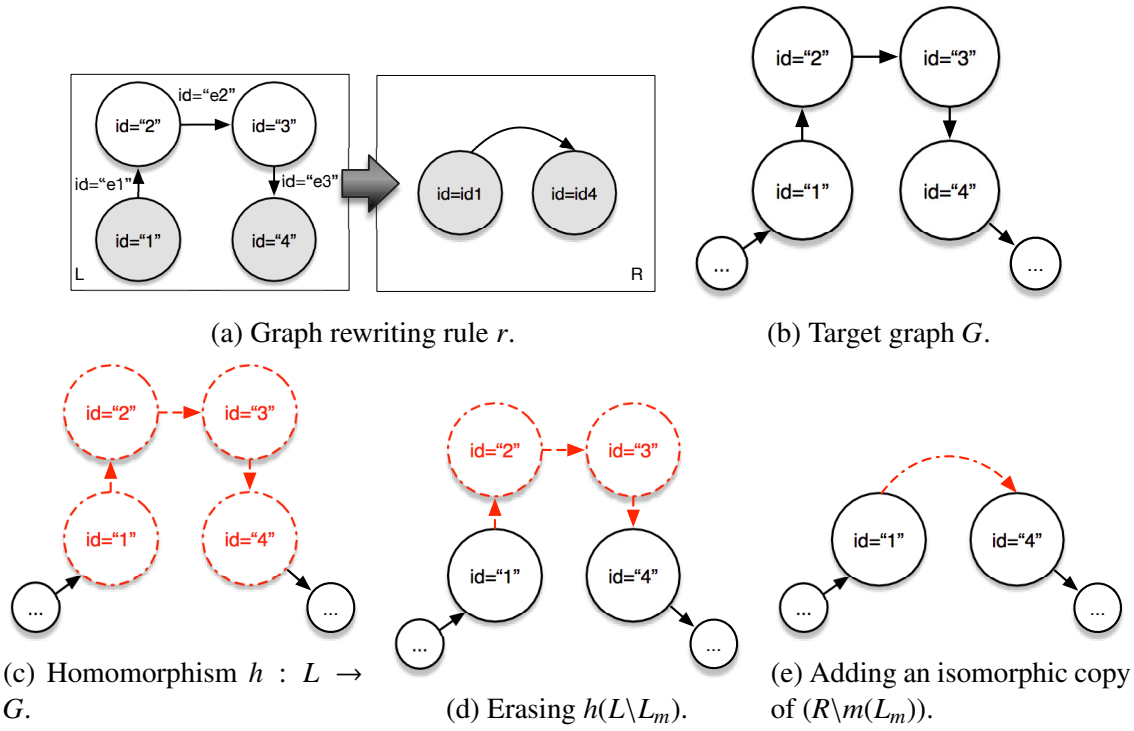
The target graph is presented in Figure 4.11b, and the steps required to apply the rule are shown in Figures 4.11c to 4.11e. First, a homomorphism $h : L \rightarrow G$ is found. Next, $h(L \backslash L_m)$ is removed from $G$, followed by the addition of an isomorphic copy of $R \backslash m(L_m)$. The rule has the effect of suppressing the nodes with *id* equal to 2 and 3 and connecting directly the nodes with *id* 1 and 4.

### Rewriting Rules and Attributes in *AGeCEP*

Vertices and edges appearing on the left- and right-hand side of *AGeCEP* rules are analogous to those appearing in queries: operators, event producers or consumers, and event streams. Hence, they can also be classified according to the stereotypes described in Section 4.4.1.

One of the main differences is that attributes appearing in a rule may be defined as:

- *fixed value*. Fixing an attribute value in $L$ means that the corresponding attribute in the image by $h$ should have the same value. A fixed value is either a parameter of the rule or a constant written between quotes.




(a) Graph rewriting rule $r$.

(b) Target graph $G$.

(c) Homomorphism $h : L \rightarrow G$.

(d) Erasing $h(L \backslash L_m)$.

(e) Adding an isomorphic copy of $(R \backslash m(L_m))$.

Figure 4.11: Illustration of a graph rewriting rule $r$ and its application.

- *non-fixed value*. If an attribute value is not fixed in $L$, the corresponding attribute in the image by $h$ can have any value. Non-fixed valued attributes are omitted in the rule definition.

- *variable*. If an attribute is associated with a variable in $L$, the variable is bound to the value of the corresponding attribute in the image by $h$. If the variable appears more than once in $L$, all its occurrences must bind to the same value; otherwise, the rule is not applicable. A variable that appears in $L$ can also appear in $R$. In this case, the variable in $R$ is replaced with its bound value.

- *operations*. Attributes may be associated with simple operations in $R$ (typically increment or decrement of values). These operations are applied along with the rule.

### Mutators: Extending Rewriting Rules with Actions on the Real System

Mutators were first introduced as a lightweight method for handling attribute changes [48]. They were described as *arbitrary algorithms updating the value(s) of none, one or some attributes*. Any rewriting rule could be enriched with a set of mutators which were executed at the end of its application phase.

Later on, a new kind of mutator was introduced [46] to describe actions on real systems,

typically through method or API calls. Such mutators are called external as opposed to internal mutators that act only on the model.

In *AGeCEP*, graph rewriting rules are specified as a couple $(L \xrightarrow{m} R, ACTS)$, each rule being enriched with a set $ACTS$ of external mutators $\mu$ that enforce model changes on the real system through API calls.

### Correctness of Rewriting Rules in *AGeCEP*

In dynamic systems, a crucial undesirable implication is a potential loss of correctness resulting from system adaptations.

In *AGeCEP*, the correctness of a reconfiguration is linked to the reconfiguration capabilities of the affected operators: a rule describing a reconfiguration should be applied only to operators with the proper capabilities (e.g., duplication should be applied to duplicable operators). This is guaranteed by fixing the value of the corresponding attributes on the left-hand side of a rule. Therefore, a properly classified operator can be safely reconfigured using the defined rules.

### Examples

Figure 4.12 illustrates a graph-rewriting rule $P_{comb}$ whose goal is to combine a sequence of two query operators into a single new operator. This rule is part of the operator combination policy, which will be detailed in Section 5.3.1.

The left-hand side of the rule encodes all necessary conditions that operators must satisfy to enable the combination:

1. the output of $o_1$ is exactly the input of $o_2$ i.e.:
   (a) they are directly connected, as represented by the edge $(o_1, o_2)$, and
   (b) $o_1(outDegree)$ = "1" and $o_2(inDegree)$ = "1";
2. they are combinable with each other, i.e.:
   (a) they are combinable, that is, $o_1(combinable)$ ="*true*" and $o_2(combinable)$ ="*true*", and
   (b) they have the same implementation, as represented by the attribute *impl* of both operators $o_1$ and $o_2$ in $L$ being associated to the same variable $op\_impl$.

The right-hand side of the rule describes the result of a combination. It consists of deploying a new operator whose *impl* is determined by the *combImpl* attribute of the combined operators, and whose parameters are calculated using the function *combParam* applied to

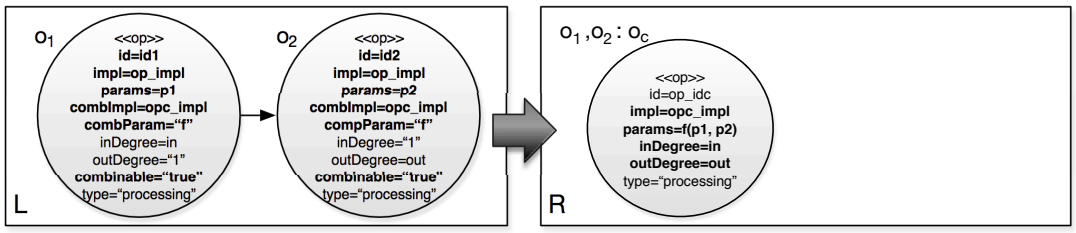



Figure 4.12: Combination of two combinable successive operators $P_{comb}$.

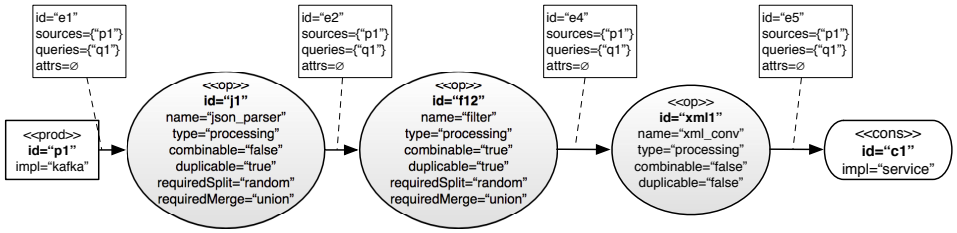



Figure 4.13: Query $q_1$ - optimized version.

$o_1(params)$ and $o_2(params)$. The rule morphism is not injective and associates both $o_1$ and $o_2$ with $o_c$. This means that $o_c$ is not, strictly speaking, a new operator, but rather the result of merging $o_1$ and $o_2$. As a result, $o_c$ has the inputs of $o_1$ and outputs of $o_2$.

The result of applying this rule to query $q_1$ from Figure 4.6a is shown in Figure 4.13.

## 4.5 Summary

This chapter has introduced the concepts of the **A**ttributed **G**raph **Re**writing for **C**omplex **E**vent **P**rocessing Management (*AGeCEP*) formalism. This formalism was developed to overcome the fragmentation of current CEP research and development landscape. *AGeCEP* proposes a language-agnostic abstraction of CEP queries and a formalism to manipulate them, enabling definition of self-management policies that can be integrated into potentially any CEP system.

*AGeCEP* represents CEP queries using attributed directed acyclic graphs (ADAG), a powerful abstract representation to which different query definition languages can be converted. In *AGeCEP*, query vertices and edges have a standardized set of attributes that encode information relevant to self-management. These standard attributes are based on a novel classification of CEP operators that focuses on their reconfiguration capabilities and also constitutes a major contribution of this research.

Self-management policies may ultimately trigger the execution of system reconfigurations. *AGeCEP* formalizes reconfigurations of queries using graph rewriting rules. Notably, a graph rewriting rule formally specifies both a reconfiguration and the context in which it can be applied, enabling specification of consistent reconfigurations that guarantee internal

self-protection. Moreover, *AGeCEP* graph rewriting rules are enriched with mutators, which associate API calls with the application of a rule and guarantees that model changes are also applied in the real system.

The next chapter focuses on practical aspects of using *AGeCEP* and evaluates it regarding both its expressiveness and performance.

# Chapter 5

# Attributed Graph Rewriting for CEP Management - Evaluation

This chapter presents a thorough evaluation of the *AGeCEP* formalism. First, Section 5.1 describes the design of an autonomic manager based on *AGeCEP* representations of queries and reconfiguration actions. Next, Sections 5.2 and 5.3 assess *AGeCEP* expressiveness by using it to express generic operator placement procedures and a selected set of self-management policies. Finally, Section 5.4 evaluates the performance of rewriting rules applied to reconfigure CEP queries. By considering both expressiveness and performance, it is shown that *AGeCEP* can indeed be used as formal foundation of the *CEPaaS* system.

## 5.1 AGeCEP-Based Autonomic Manager

This section presents the design of an *AGeCEP*-based autonomic manager. The focus is not on implementation details, but on how existing approaches can be integrated with *AGeCEP* to tackle the whole MAPE-K[1] loop and thus implement a complete autonomic manager for CEP systems.

The presented design uses FRAMESELF [7], a framework that aims to enable implementation of autonomic managers that rely on the MAPE-K loop. In particular, FRAMESELF provides meta-models and mechanisms for implementing inference rules and communication between modules.

[1] Monitor, Analyze, Plan, Execute - Knowledge

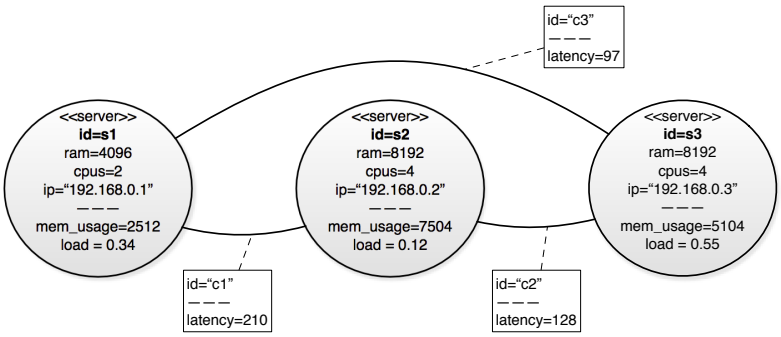



Figure 5.1: Runtime environment representation.

### 5.1.1 Runtime Environment Representation

Modelling the runtime environment is an important aspect of CEP systems that is mostly determined by the operator placement strategy used. Previous research has traditionally represented queries and the runtime environment as graphs [3, 97]. For the *AGeCEP*-based autonomic manager presented in this section, a similar approach has been used: *AGeCEP* queries are extended with attributes that are relevant for placement decisions, and the runtime environment is modelled as an (undirected, potentially cyclic) attributed graph.

Figure 5.1 shows an attributed graph that represents a runtime environment composed of three servers. In the graph, vertices and edges represent computational resources and logical connections between them respectively. The vertices contain intrinsic attributes that model server characteristics such as the number of CPUs, total RAM, and a unique IP address. Vertices and edges are also augmented with attributes representing runtime information such as actual CPU load and memory usage. These attributes are important for decision making and must be updated when the corresponding monitoring information is available.

By using this environment representation and the *AGeCEP* query model, the placement of an operator into a server can be represented using two approaches: as an operator attribute whose value contains a unique server identifier, or as an edge connecting the operator to the server. For the remainder of this thesis, the first approach is assumed. Operator placement procedures are further discussed in Section 5.2.

### 5.1.2 MAPE Modules

The following subsections discuss how the *AGeCEP* formalism is used by each module of the MAPE-K loop in the *AGeCEP*-based autonomic manager.

| Type | Event | Description |
|---|---|---|
| Runtime | $QueueSize(o, n)$ | Queue from operator $o$ has size $n$ |
| | $CPULoad(s, l)$ | CPU load from server $s$ has value $l$ |
| User | $NewQuery(q)$ | New query $q$ was created |
| | $NewQueries(Q)$ | Set of queries $Q$ was created |
| Manager | $Duplicated(o, q)$ | Operator $o$ from query $q$ was duplicated |

Table 5.1: Monitored events.

**Monitor: Receiving Events**

To implement the *monitor* module, it is assumed that the runtime environment and user queries are instrumented to publish monitoring events to the autonomic manager. As an alternative, a specialized monitoring module can poll the system for monitoring data and forward them to the manager on behalf of system components. Moreover, it is expected that events representing user interaction with the system, such as creation of new queries, will also be made available to the manager.

Once the manager receives monitoring events, it updates the query and environment models that are stored in the KB and continues to execute the MAPE-K loop. Note that more advanced architectures with multiple distributed managers can also be implemented, but are outside the scope of this research.

Table 5.1 shows common monitored events used by CEP systems. They are coarsely classified into three groups: events generated by the *runtime* environment, including user queries; events initiated by the *user*; and events generated by the autonomic *manager*.

Note that instead of trying to enumerate all possible events, the table only includes events used by the self-management policies from Section 5.3. This decision is aligned with *AGeCEP* extensible and generic nature: the monitoring data that CEP systems must provide are tightly linked to the placement procedures and self-management policies implemented by the autonomic manager. Therefore, defining a fixed set of monitoring events would restrict the scope of policies that can be implemented. Instead, *AGeCEP* allows policies to define the events they need. Analogously to how rule mutators establish a contract that CEP systems must implement, the events required by a policy define a contract of events CEP systems must provide.

### Monitor, Analyze, and Plan: Inference Rules

Inference rules are central to the *monitor*, *analyze*, and *plan* modules of the MAPE-K loop. In general, these rules are used to infer new information based on the KB and on freshly received information.

In the monitor module, the events received by the autonomic manager are processed by inference rules to infer *symptoms*. The analysis module uses these symptoms along with the *AGeCEP* query and environment models that are stored in the KB to generate *Request for Changes* (RFC). Finally, in the plan module, the RFCs and the KB models are used by another set of rules to create *Change Plans* (CP).

In the FRAMESELF framework, inference rules are implemented by inference engines such as Jess [90] and JBoss Drools [89]. This research used JBoss Drools. Symptoms, RFCs, and CPs are represented as plain Java objects that are exchanged between the MAPE-K modules.

### Execute

The *execute* module is in charge of carrying out the CPs that it receives. In the *AGeCEP*-based autonomic manager, a CP is simply a sequence of reconfiguration actions modelled by graph rewriting rules that are associated with a set of side-effect mutators. The *execute* module enforces each reconfiguration of the CP in two steps: (1) applying the graph rewriting rule to update the models; and (2) executing its associated mutators to update the real system.

In practice, a mutator is an API call that must be implemented by the CEP system being managed. Accordingly, for the feasibility studies presented in this chapter, a minimal API has been defined as follows:

- $startOperator(o, m)$: deploys and starts an operator $o$ in server $m$;
- $stopOperator(o)$: stops and deletes an operator $o$;
- $connect(o_1, o_2)$: creates a connection between operators $o_1$ and $o_2$;
- $disconnect(o_1, o_2)$: removes a connection between operators $o_1$ and $o_2$;
- $redirect(o_1, o_2)$: redirects all $o_1$ input (output) streams to input (output) $o_2$;
- $migrate(o, s_1, s_2)$: migrates an operator $o$ from server $s_1$ to server $s_2$.

## 5.2 Feasibility: Operator Placement

This section discusses placement in the *AGeCEP* context. As mentioned in Section 2.1.5, operator placement procedures are used to determine the global initial placement of all queries, to decide on the placement of a new query or of new operators, or to adjust the current placement dynamically. In particular, placement can be used by management procedures that may require placement decisions (e.g., whenever a new operator is deployed).

### 5.2.1 General Principle

Independently of the goal and of the algorithm used, operator placement procedures can usually be described by a general framework composed of three steps:

1. Metrics from the operators and servers are collected to build a snapshot of the current system status. These metrics are usually directly available, for example in the case of operator queue sizes and server CPU load, but occasionally they need to be estimated by specialized components that run concurrently with the system. For instance, in Pietzuch *et al*. [126], each server communicates with its neighbours to estimate its coordinates in a latency space. In the case of dynamic placement adjustments, these metrics are also used to trigger procedure execution. For example, in Heinze *et al*. [70], dynamic adjustment is triggered when an overloaded server is detected.

2. Using the collected data as input, an algorithm is executed to find the new placement. Because the general operator placement problem is NP-hard [97], these algorithms are usually heuristics that aim to maximize or minimize a utility function estimated from the collected metrics. For instance, Pietzuch *et al*. [126] aimed to minimize network usage, whereas Xing *et al*. [151] tried to maximize load correlation among servers.

3. The results of the algorithm are applied. If a placement has been calculated for new queries or operators, then they are created in the appropriate servers. Conversely, if a dynamic adjustment is being performed, then the operators that have changed allocation are migrated to their new servers.

It is argued here that most operator placement procedures can be expressed using the *AGeCEP* formalism and integrated into the *AGeCEP*-based autonomic manager by adapting them as follows:

1. The query and runtime environment representations are augmented with attributes corresponding to the monitored metrics. The metrics are collected using the same mechanisms as before and are sent to the autonomic manager, which updates the corresponding

attributes upon receipt. In the case of dynamic adjustment procedures, a monitoring inference rule is also created to start the placement procedure based on the collected metrics.

2. When a placement decision is required, the input data are obtained from the autonomic manager KB, and execution of the placement procedure takes place in the same way. At this point, the chosen environment representation determines how the input data is collected. For instance, if placement is represented by edges connecting operators to servers, then gathering all operators from a server requires traversing such edges starting from the server vertex. On the other hand, if placement is represented by operator attributes, then the same goal requires a search through all operator vertices.

3. Finally, with the newly calculated placement information, the operators are deployed or migrated through rewriting rules that update the KB and invoke the API calls *startOperator* or *migrate* accordingly.

### 5.2.2 Examples

This section presents how two different placement procedures can be expressed using *AGeCEP*.

#### Borealis (Xing *et al.* [151])

In their work, Xing *et al*. [151] presented heuristic procedures for global and dynamic adjustment of placements with the goal of minimizing the end-to-end latency of queries. The general idea of the presented heuristics is that, given an operator $o$ that needs to be placed, a server must be found with a current workload that is not correlated with $o$'s workload. To calculate load correlation, the heuristics build a time series of each operator's load based on monitored data. A server load, in turn, is defined as the sum of all its operators' loads.

To adapt these heuristics to *AGeCEP*, the load time series can be maintained as an attribute of operator vertices. Calculations performed by the heuristics require only this data. As a result of the algorithms, migration rewriting rules are executed for each operator that has been selected to move.

#### FUGU (Heinze *et al.* [70])

Heinze *et al*. [70] presented a dynamic adjustment placement procedure for the FUGU system. The general idea is to detect overloaded servers and to move operators from them to underloaded servers. The operators to be moved are selected based on the latency spikes that their migration will cause; operators with small spikes are moved first.

A server is detected as overloaded when its CPU utilization exceeds a threshold for $x$ consecutive measurements. This detection can be easily implemented as a monitoring inference rule. To decide which operators are moved, the latency spike estimation uses the following operator metrics: load, incoming and outgoing network traffic, state size, input rate, and processing time per tuple. Note that these metrics can all be obtained from the query execution engine and stored as attributes of the corresponding operator vertices in the KB.

After the operators to be moved are selected, their destination is determined based on a heuristic analogous to the bin-packing problem, in which the server's available CPU capacity constitutes the bins and the operators' loads are the items weight. Once again, these data are readily available in the KB. Finally, the resulting migrations are enforced with the aid of rewriting rules.

## 5.3 Feasibility: Self-Management Policies

This section introduces a selection of self-management policies defined using *AGeCEP* and the autonomic manager presented in Section 5.1. For the sake of readability, algorithms and inference rules are presented as informal descriptions or pseudocode. Appendix A contains the corresponding inference rules defined in Drools Rule Language [89].

### 5.3.1 Operator Combination

#### Description

The *operator combination* (*Comb*) policy is directly related to the "combinable" criterion of the *AGeCEP* classification. This policy is used to combine sequences of $n$ combinable operators $o_1, \ldots, o_n$ into a single operator $o_c$ which has the same effect on the event stream as the combined sequence. Figure 4.5 shows an example of such a sequence and the result of applying this policy.

This policy is mostly used in the single-query optimization step and reduces the number of operators constituting the query, which brings savings in memory consumption. It can also improve query latency and throughput because the number of operators that are traversed from event generation to event consumption is reduced.

#### Realization Using the MAPE-K Loop

**Monitor** A new query $q$ is submitted by the user, which is signalled by a $NewQuery(q)$ event. The event is simply forwarded as a $NewQuery(q)$ symptom to the analysis module.

**Algorithm 5.1:** *CombineAll*($q$) action, combination of all combinable operator sequences in $q$.

**while** *exists a homomorphism* $h : L_{comb} \rightarrow q$ **do**
  apply $P_{comb}$ rule to $q$ w.r.t. $h$;
**end**

**Analysis** When a *NewQuery*($q$) symptom is received, the analysis module checks whether at least one pair of successive operators $(o_1, o_2)$ are combinable. This is equivalent to checking whether there is a homomorphism $h : L_{comb} \rightarrow q$, where $L_{comb}$ is the left-hand side of the graph rewriting rule shown in Figure 4.12. If such a homomorphism exists, a *Combine*($q$) request for change (RFC) is sent to the plan module (Algorithm A.1).

**Plan** Upon receipt of a *Combine*($q$) RFC, the *CombineAll*($q$) action is inserted into the change plan.

**Execute** The execution of the *CombineAll*($q$) action is described by Algorithm 5.1. The $P_{comb}$ rule is specified in Figure 4.12; its applicability and effect have been described in Section 4.4.2. This rule operates at the model level only and therefore has no associated mutator.

### 5.3.2 Operator Duplication

#### Description

*Operator duplication* (*Dupl*) is a policy used to parallelize an operator execution by creating multiple instances of the operator and splitting input events among these instances. As shown in Figure 4.4, the original input stream of an operator $o$ is split by an operator $o_s$ such that $o_s(type)$ = "*split*", and the outputs are merged back into a single stream by an operator $o_m$ such that $o_m(type)$ = "*merge*". The attributes $o(reqSplit)$ and $o(reqMerge)$ determine $o_s(splitType)$ and $o_m(mergeType)$ respectively.

*Dupl* can be used to achieve load balancing by distributing the operator instances over several servers, or to improve query throughput. Generally, query throughput can be improved when the following conditions are satisfied:

1. the operator processing rate is lower than the incoming event rate.
2. additional resources exist to which extra instances of the operator can be allocated.

However, this policy may also lead to an increase in resource consumption due to the deployment of supplementary operators.

### Realization Using the MAPE-K Loop

**Monitor** The autonomic manager receives periodic events containing operators performance metrics. Based on these data, a monitoring rule might identify an operator as a bottleneck if it is not outputting events as fast as it is receiving them. Whenever a bottleneck is pinpointed, a $Bottleneck(o)$ symptom is sent to the analysis module (Algorithm A.2).

**Analysis** Whenever a $Bottleneck(o)$ symptom is received, the analysis module checks whether $o$ is duplicable. If this condition is satisfied, a $Duplicate(q, o)$ RFC is sent to the plan module (Algorithm A.3).

**Plan** When planning a $Duplicate(q, o)$ operation, two scenarios must be considered (Algorithm A.4):

- $o$ has not yet been parallelized, which implies that duplication requires deployment of a new instance of $o$ and of the appropriate split and merge operators. In this case, the placement procedure is invoked to determine the place of these new operators and an $InitialDuplication(q, o, s_s, s_o, s_m)$ action is inserted into the change plan, where $s_s$, $s_o$, and $s_m$ are the placements determined for the split, the new instance of $o$, and the merge operator.

- $o$ has already been parallelized, or more precisely, adequate split and merge operators have already been deployed. Therefore, duplication consists only of adding a new operator instance. In this case, the placement $s_o$ of this new operator is determined and the $AdditionalDuplication(q, o, s_o)$ action is added to the change plan.

**Execute** Depending on the change plan received, two different actions may be executed: an $InitialDuplication(q, o, s_s, s_o, s_m)$ or an $AdditionaDuplication(q, o, s_o)$ action.

The $InitialDuplication(q, o, s_s, s_o, s_m)$ action is described by Algorithm 5.2. Rules $P^{init_1}_{dupl}(id, s_s, s_o, s_m)$, $P^{init_2}_{dupl}(id, id_s)$, and $P^{init_3}_{dupl}(id, id_m)$ used by the algorithm are depicted in Figure 5.2. Some attributes of $o$, $o_1$, and $o_2$ are not shown because of space constraints, but the copies $o_1$ and $o_2$ have the same values as $o$ for all attributes described in Table 4.1, except for *id*, *inDegree*, and *outDegree*. The algorithm consists of three steps:

1. $P^{init_1}_{dupl}(id, s_s, s_o, s_m)$ is applied to $q$ with respect to the unique possible homomorphism. Hereafter, when a single morphism is acceptable, it is omitted. Application of this rule creates the respective split $o_s$ and merge $o_m$ and two instances of operator $o$. Note that operators $o_s$, $o_2$, and $o_m$ are created in servers $s_s$, $s_o$, and $s_m$ as indicated by the "placement"

**Algorithm 5.2:** *InitialDuplication*$(q, o, s_s, s_o, s_m)$ action, execution of an initial duplication.

$id \leftarrow o(id)$;
apply $P^{init_1}_{dupl}(id, s_s, s_o, s_m)$ to $q$ ;
**while** *exists a homomorphism* $h : L^{init_2}_{dupl}(id, id_s) \rightarrow q$ **do**
  apply $P^{init_2}_{dupl}(id, id_s)$ to $q$ w.r.t. $h$;
**end**
apply $P^{init_3}_{dupl}(id, id_m)$ to $q$;

attribute value. In addition, operator $o_1$ is created on the same server as the original operator $o$. The mutators executed for this rule are API calls to $startOperator(o_s, s_s)$, $startOperator(o_1, s)$, $startOperator(o_2, s_o)$, and $startOperator(o_m, s_m)$.

2. $P^{init_2}_{dupl}(id, id_s)$ is repeatedly applied as long as possible to redirect all input edges previously connected to $o$ towards $o_s$. This rule is associated with the mutators $disconnect(v, o)$ and $connect(v, o_s)$.

3. The $P^{init_3}_{dupl}(id, id_m)$ rule merges $o$ and $o_m$. As a result, all output edges previously connected to $o$ are redirected to $o_m$, and the original operator $o$ is deleted. These changes are performed on the system by the mutators $redirect(o, o_m)$ and $stopOperator(o)$.

The *AdditionalDuplication*$(q, o, s_o)$ action, on the other hand, is accomplished by applying the $P^{add}_{dupl}(id, s_o)$ rule shown in Figure 5.3. It consists of the simple addition of a new instance of $o$ connected to the already existing split $o_s$ and merge $o_m$. This rule is associated with the mutators $startOperator(o_n, s_o)$, $connect(o_s, o_n)$, and $connect(o_n, o_m)$.

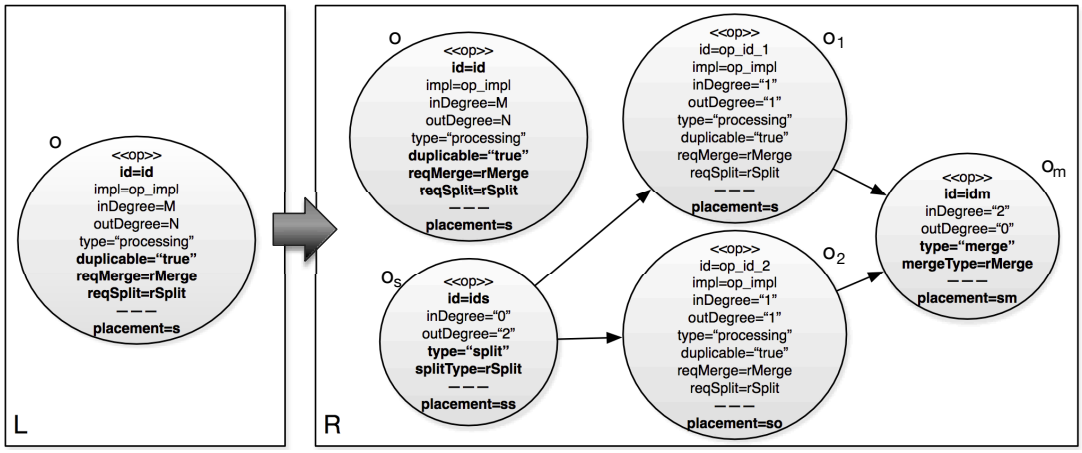
### 5.3.3 Removal of an Unnecessary Merge/Split

#### Description

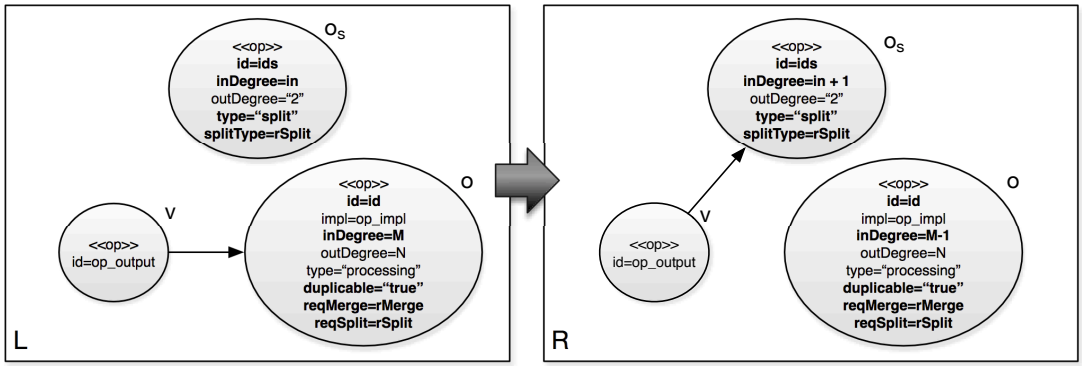
The *removal of an unnecessary merge/split* (*RemMS*) policy describes the removal of a particular pattern of a merge operator followed by a split whose impact on the event streams is null. Such a pattern has null impact if:

- The merge does not modify the streams that it processes. According to the *AGeCEP* classification, union is the only merge type satisfying this condition.

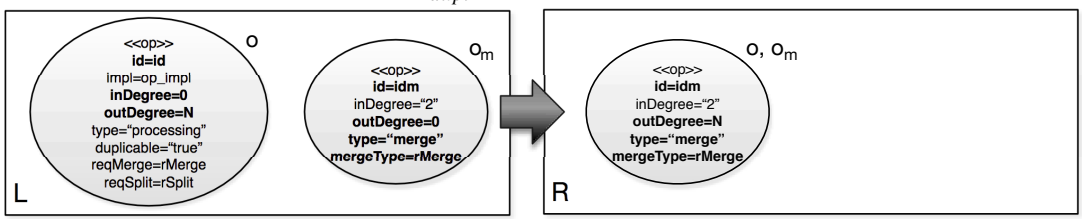
- The split operator does not strengthen the stream specificities, or in other words, the output streams of the split have the same or fewer constraints than the input streams of the merge.




(a) Step 1 - $P^{init_1}_{dupl}(id, s_s, s_o, s_m)$ - operator duplication.




(b) Step 2 - $P^{init_2}_{dupl}(id, id_s)$ - redirect input edge.




(c) Step 3 - $P^{init_3}_{dupl}(id, id_m)$ - redirect output edges.

Figure 5.2: $P^{init}_{dupl}(id)$: initial duplication.

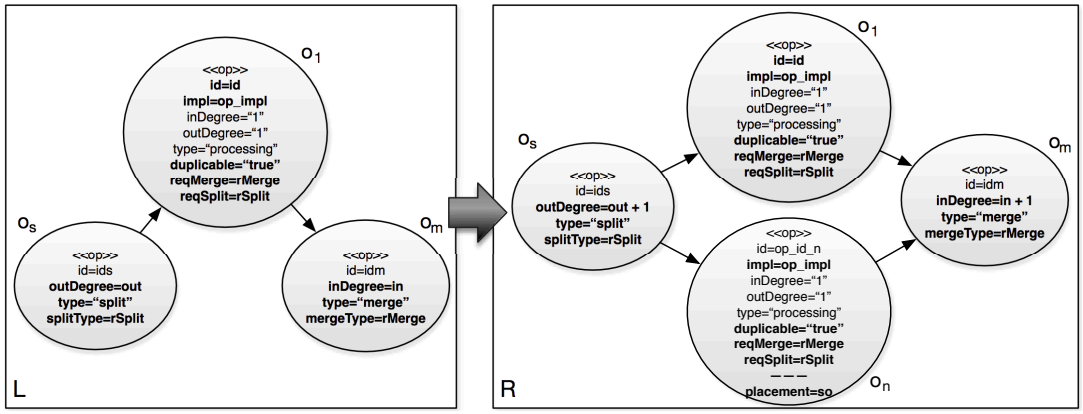



Figure 5.3: $P^{add}_{dupl}(id, s_o)$: additional duplication.

The following discussion considers only the case where the number of input streams in the merge is equal to the number of output streams in the split.

The impact of this policy is positive on both system performance and on resource consumption because unnecessary operators are suppressed. Hence, this policy is used whenever possible.

### Realization using the MAPE-K loop

**Monitor** A new query $q$ is submitted by the user, resulting in the $NewQuery(q)$ event being forwarded to the analysis module as a $NewQuery(q)$ symptom. In addition, whenever an operator $o$ from query $q$ is duplicated, a $Duplicated(q, o)$ event is also sent to analysis as a symptom.

**Analysis** Whenever a $NewQuery(q)$ or a $Duplicated(q, o)$ symptom is received, the analysis module checks for the existence of an unnecessary merge/split sequence as follows:

1. There is a homomorphism $h : L^{byp}_{rem}(id_m, id_s, e_i, e_o) \rightarrow q$, where $L^{byp}_{rem}(id_m, id_s, e_i, e_o)$ is the left-hand side of the $P^{byp}_{rem}(id_m, id_s, e_i, e_o)$ rule depicted in Figure 5.4a;

2. The split operator does not strengthen the stream specificities. This condition cannot be checked for *"custom"* splits. For the other cases, let $o_m(pred)$ be the set of all incoming edges of $o_m$ and $o_s(succ)$ be the set of all outgoing edges of $o_s$. This condition is met if there is a bijective function $f : o_m(pred) \rightarrow o_s(succ)$ such that for all $e_i \in o_m(pred)$, with $e_o = f(e_i)$, one of the following conditions is satisfied:

   - $o_s(splitType) =$ "$query$" and $e_i(queries) = e_o(queries)$;
   - $o_s(splitType) =$ "$attribute$" and $e_i(attrs) = e_o(attrs)$;

**Algorithm 5.3:** *RemoveMergeSplit*$(q, o_m, o_s, f)$ action, execution of a removal.

$id_m \leftarrow o_m(id)$;
$id_s \leftarrow o_s(id)$;
**forall the** *edges* $e_i \in o_m(pred)$ **do**
  apply $P^{byp}_{rem}(id_m, id_s, e_i, f(e_i))$ to $q$;
**end**
apply $P^{sup}_{rem}(id_m, id_s)$ to $q$;

- $o_s(splitType)$ = "*random*".

For each pair $(o_m, o_s)$, a *RemoveMergeSplit*$(q, o_m, o_s, f)$ RFC is created using an arbitrarily selected function $f$ that satisfies condition 2 (Algorithm A.5).

**Plan** Upon receiving a *RemoveMergeSplit*$(q, o_m, o_s, f)$ RFC, the action *RemoveMergeSplit*$(q, o_m, o_s, f)$ is inserted into the change plan.

**Execute** Algorithm 5.3 details how to remove an unnecessary merge/split pattern. The algorithm is executed in two parts. First, the unnecessary merge and split are bypassed using the $P^{byp}_{rem}(id_m, id_s, e_i, e_o)$ rewriting rule defined in Figure 5.4a. This rule is repeated for all pairs of edges $(e_i, e_o)$ returned by the function $f$ described in the analysis step. In the second part, the bypassed merge/split is removed using the $P^{sup}_{rem}(id_m, id_s)$ rule (Figure 5.4b).

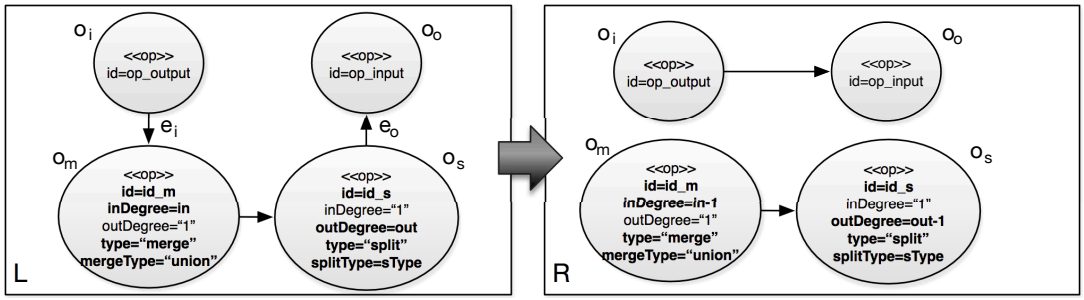
Note this policy may be executed in a running query. In such a case, each application of rule $P^{byp}_{rem}(id_m, id_s, e_i, e_o)$ triggers the mutators $disconnect(o_i, o_m)$, $disconnect(o_s, o_o)$ and $connect(o_i, o_o)$, whereas the rule $P^{sup}_{rem}(id_m, id_s)$ triggers $stopOperator(o_m)$ and $stopOperator(o_s)$.

### 5.3.4 Processing Sub-Streams (ProcSubS)

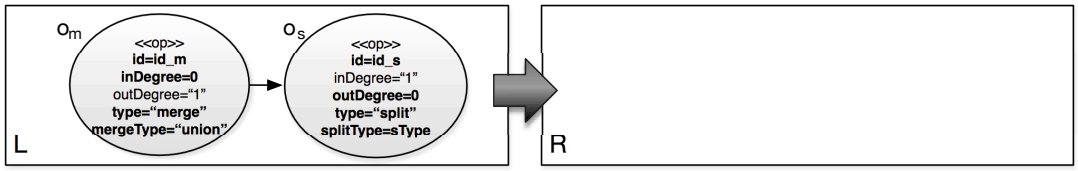
#### Description

The *processing sub-streams* (*ProcSubS*) policy transposes to CEP the strategy of dividing a problem into the solution of several sub-problems. The policy considers an operator $o$ processing the result of a merge $o_m$, as illustrated on the left-hand side of Figure 5.5a. Ideally, the operation performed by $o$ should be parallelized and conducted on each of the merged streams. In rough terms, $o$ and $o_m$ should be "swapped", as shown on the right-hand side of Figure 5.5a.

This transformation is equivalent to multiple duplications of $o$ followed by removal of the initial merge $o_m$ and the new split introduced by the duplication. Figure 5.5b illustrates the query after $n$ duplication steps, where $n$ is the number of $o_m$ input streams. The merge and




(a) Step 1 - Bypassing the merge/split $P^{byp}_{rem}(id_m, id_s, e_i, e_o)$.




(b) Step 2 - Suppressing the merge/split $P^{sup}_{rem}(id_m, id_s)$.

Figure 5.4: $P_{rem}(id_m, id_s)$: removal of an unnecessary merge/split.

split sequence highlighted in the figure can be removed by the *RemMS* policy, resulting in the desired final situation. The policy realization described in this section leverages this fact and reuses the *Dupl* and *RemMS* policies described in Sections 5.3.2 and 5.3.3.

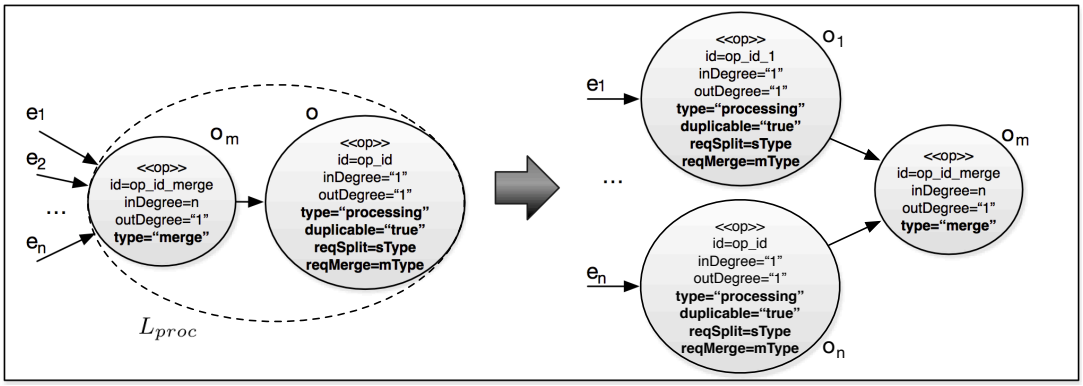
This policy can be applied under various circumstances:

- If there are enough resources to process $o$ instances in parallel, then this policy can be used to improve query throughput and latency. This effect is even more pronounced when $o(selectivity) < 1$. In this case, the policy can also be applied in the SQO step because the total number of events processed by the merge $o_m$ may be significantly reduced.
- If $o$ processes groups of events and $o(complexity)$ is greater than linear, then this policy reduces the query total CPU consumption;
- In general, the policy can be used to split the load of processing $o$ with other servers and cores.

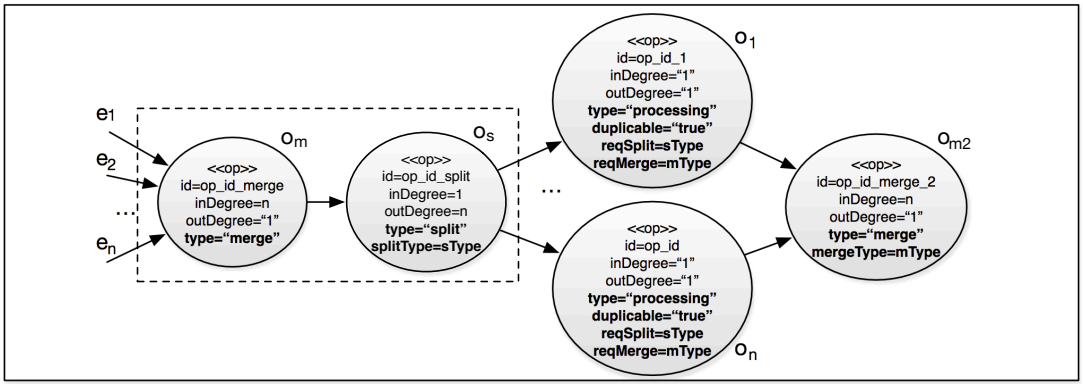
### Realization Using the MAPE-K Loop

**Monitor** This policy can be triggered at runtime whenever a bottleneck of operator $o$ is detected, which results in the $Bottleneck(o)$ symptom being sent to analysis.

**Analysis** When a $Bottleneck(o)$ is received, the analysis rule first checks if the policy can indeed improve query throughput. For instance, it can verify whether the selectivity of operator




(a) Initial and final situation.




(b) Intermediate step.

Figure 5.5: Processing sub-streams policy.

$o$ is less than 1. If this is true, the rule searches for a sequence formed by a merge $o_m$ and a duplicable operator $o$ by checking for a homomorphism $h : L_{proc} \rightarrow q$, where the graph $L_{proc}$ is indicated in Figure 5.5a. Finally, the rule verifies whether the found sequence $o_m$ and $o$ satisfies the following conditions:

1. The merge has more than one input stream,

$$o_m(inDegree) > 1;$$

2. The merge $o_m$ followed by the split introduced by the duplication of $o$ produces a removable pattern, which translates to:

   (a) The merge type of $o_m$ is "union",

$$o_m(mergeType) = \text{“}union\text{”};$$

   (b) The split $o_s$ introduced during duplication of $o$ does not strengthen stream specificities. In the *RemMS* policy, it was shown that a "random" split is always valid, whereas a "custom" split cannot be considered. In the other cases:

   - if $o(reqSplit) = \text{“}query\text{”}$, then for each stream $e$ entering $o_m$, $|e(queries)| = 1$;
   - if $o(reqSplit) = \text{“}attribute\text{”}$, then for each stream $e$ entering $o_m$, $e(attrs) = o_s(param)$, meaning the streams are already grouped with respect to the same attributes that the split discriminates.

If these conditions are satisfied, then the processing sub-stream policy can be applied. To achieve this, the rule inserts $n$ requests for duplication of operator $o$ (Algorithm A.6). Because the *RemMS* policy is already executed after each duplication, there is no need to request it explicitly. In addition, note that even though *RemMS* is triggered $n$ times, only the last time succeeds because the others cannot find the mapping function $f$ required by the *RemMS* policy.

**Plan** There is no specific plan for this policy.

**Execute** There is no specific execution for this policy.

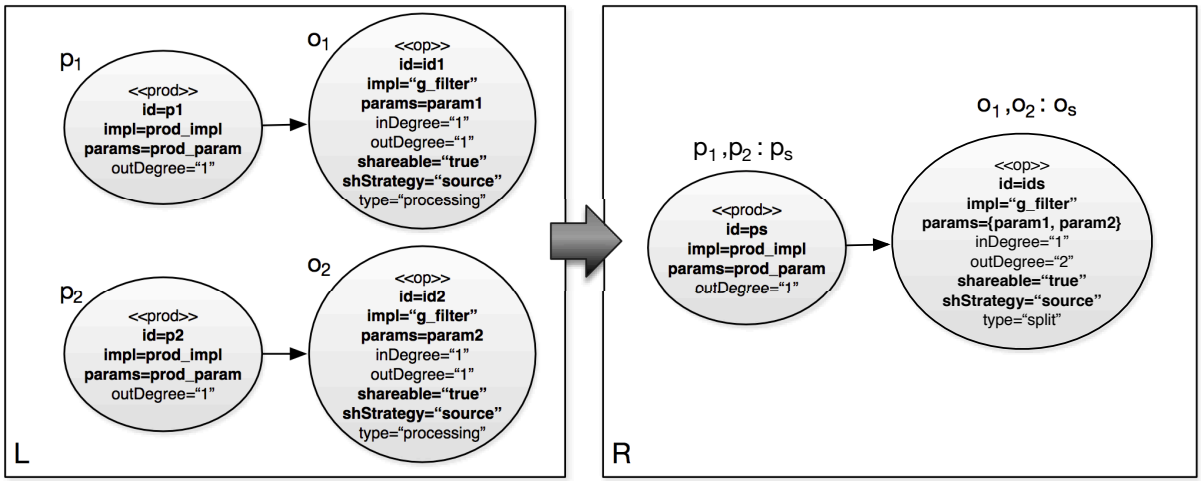



Figure 5.6: $P_{pred}$: predicate indexing.

### 5.3.5 Predicate Indexing

#### Description

This policy (*PredIndex*) implements the *predicate indexing* MQO technique introduced by Madden *et al.* [109]. The technique detects when two or more filters process the same input stream and have predicates over the same attributes and replaces both occurrences with a single filter. This is an example of source sharing, as explained in Section 4.3.2. In this case, the filter under consideration has special data structures that enable it to evaluate multiple (range) predicates more efficiently than evaluating each predicate independently.

#### Realization Using the MAPE-K Loop

**Monitor** A set of queries $Q$ is submitted by the user, resulting in a *NewQueries*($Q$) event being forwarded to the analysis module as a symptom.

**Analysis** The analysis module checks whether a pair of filters can be shared by searching for a homomorphism $h : L_{pred} \rightarrow Q$, where $L_{pred}$ is the left-hand side of the graph rewriting rule $P_{pred}$ in Figure 5.6. If a homomorphism exists, the module also checks whether the predicates range over the same attributes. If so, a *PredicateIndex*($Q$) RFC is sent to the plan module (Algorithm A.7).

**Plan** Upon receipt of the *PredicateIndex*($Q$) RFC, the *PredicateIndexAll*($Q$) action is inserted into the change plan.

**Execute** Execution of this policy is equivalent to repeatedly applying rule $P_{pred}$, described in Figure 5.6. Note that the two producer vertices $p_1$ and $p_2$ shown in $L$ actually represent the same input source, as they are associated with the same implementation and parameters. The resulting grouped filter is logically equivalent to the execution of both predicates. This policy is applied only at the query model level, and therefore there is no associated mutator.

## 5.4 Viability: Performance Evaluation

This section discusses the viability of *AGeCEP* as a formal foundation for developing generic CEP algorithms and management procedures. The analysis focuses on the time required to transform CEP queries using both simple and complex graph rewriting rules.

In the following, the AGG tool [143] was used to define and apply the graph rewriting rules. The experiments were conducted on a server with two six-core processors (Intel Xeon E5-2630, 2.6GHz) and 96GB of RAM. The server was running Ubuntu Linux 14.04 and Java 1.7.0_75.

### 5.4.1 Simple Policy

The first experiment verified the execution time and scalability of the actions executed by the *Comb* policy (Section 5.3.1). This is a simple policy that consists of a single rewriting rule in which only two vertices are matched.

The total number of queries to which the rule was applied varied from 100 to 1000, and for each number, three query compositions were tested. In the first composition, 20% of the queries were clones of query $q_1$ (Figure 4.6a), and 80% were clones of $q_2$ (Figure 4.6b). In the second and third compositions, query $q_1$ represented 50% and 80% of the total queries respectively. Note that only query $q_1$ has a sequence of combinable filters $f_1$ and $f_2$.

The graph in Figure 5.7 shows the average execution time of 30 runs along with the 99% confidence interval. The growth in execution time is close to linear. For all three compositions, 100 queries were processed in less than one second, and 1000 queries in less than 14 seconds. For the 80% composition, this is equivalent to rewriting 800 queries according to the operator combination policy.

### 5.4.2 Complex Policy

This experiment verified the performance and scalability of complex sequences of actions. To perform this experiment, the analysis was divided into two parts. First, the execution time

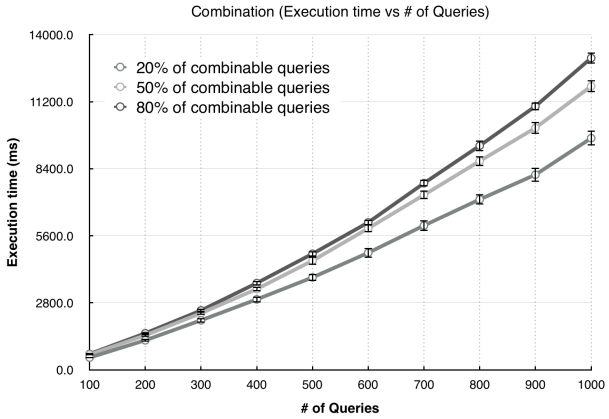



Figure 5.7: *Comb* policy execution time.

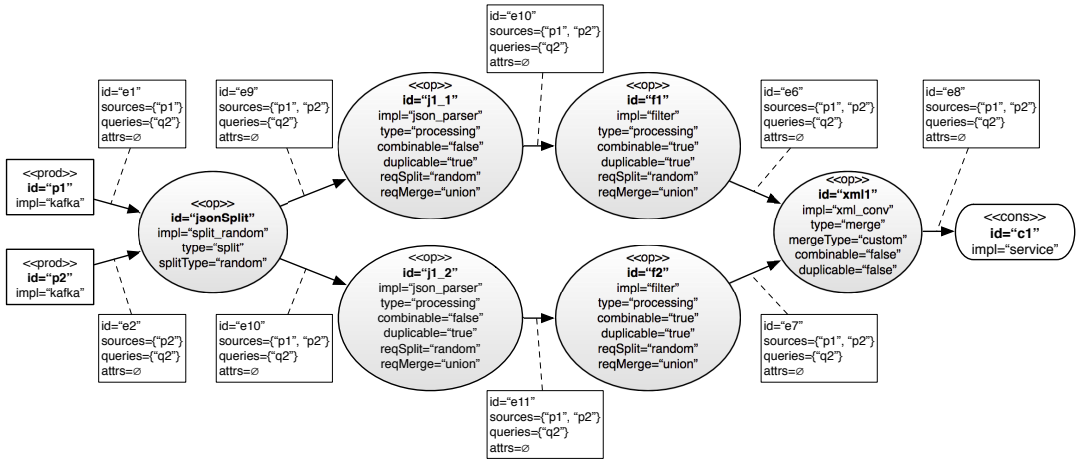
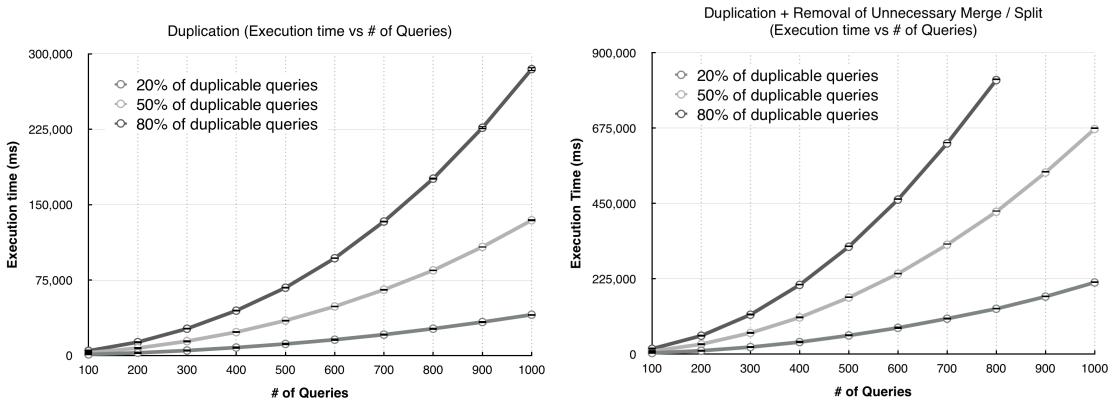



Figure 5.8: Query $q_2$ - optimized version.

for applying the *Dupl* policy (Section 5.3.2) was assessed. Following, the execution time for applying *Dupl* followed by *RemMS* (Section 5.3.3) was analyzed.

Both parts were executed using the same numbers of queries and the same query compositions as in the previous experiment. In this case, however, the duplication was applied only to the operator $j_1$ belonging to query $q_2$ clones (Figure 4.6b). Note that after $j_1$ duplication, the newly created merge forms a void sequence with the $fSplit$ operator. Applying *RemMS* therefore caused this sequence to be removed, resulting in the query depicted in Figure 5.8.

Figure 5.9a depicts the execution time of the *Dupl* policy as a function of the number of queries for all three compositions. For each duplication, four rewriting rules were applied: $P_{dupl}^{init_1}$ once to create the two instances of $j_1$ connected to a new split and merge; $P_{dupl}^{init_2}$ twice to redirect $j_1$ inputs ($p_1$ and $p_2$) to the new split; and $P_{dupl}^{init_3}$ to connect the new merge to the $j_1$ successor ($fSplit$). For the 20% composition, 1000 queries were processed in less than 40 seconds, which still is within reasonable time bounds.




(a) *Dupl* execution time. (b) *Dupl* followed by *RemMS* execution time.

Figure 5.9: *Dupl* and *RemMS* policies execution times.

The execution time to apply *Dupl* followed by *RemMS* is shown in Figure 5.9b. To execute the *RemMS* policy, three more rewriting rules were applied: $P^{byp}_{rem}$ twice to connect each instance of $j_1$ to an instance of $f_{12}$, and $P^{sup}_{rem}$ once to remove the redundant merge and split. Therefore, *Dupl* followed by *RemMS* requires the application of seven rewriting rules in total. The graph clearly shows an exponential growth that is especially pronounced in the 50% and 80% scenarios. In these scenarios, rewriting all queries may take minutes. Indeed, for the 80% scenario there are no data point for 900 and 1000 queries because the execution time exceeded the established timeout of 15 minutes.

It is important to discuss these results under proper assumptions about how these rules will be applied in practice. Finding homomorphisms in graphs is a well-known NP-complete problem [55]. Nevertheless, most of the time, these rules will be applied to a much smaller number of queries. For example, SQO policies are executed in response to new queries, and therefore only them need to be analyzed. Similarly, most runtime management rewriting rules are applied only to the small subset of running queries that need to be rewritten. For instance, as described in Section 5.3.2, duplication is performed only after a bottleneck has been pinpointed. The extreme cases described in this section were investigated for theoretical purposes and for completeness of analysis.

## 5.5 Summary

To demonstrate the feasibility of *AGeCEP* for specification and enforcement of self-management policies, this chapter introduced the design of an autonomic manager based on *AGeCEP* and a selection of five policies built on this design. Furthermore, it presented a generic procedure to adapt operator placement procedures to *AGeCEP*. Finally, this chapter

investigated the viability of *AGeCEP* by executing performance measurements of query reconfigurations. By considering both expressiveness and performance, these results suggest that *AGeCEP* can be effectively used to develop algorithms for application and integration into diverse modern CEP systems.

The next chapter discusses *CEPSim*, a simulator of cloud-based CEP system that uses *AGeCEP* as query representation model.

# Chapter 6

# Complex Event Processing Simulator

This chapter[1] introduces *CEPSim*, a simulator that has been developed to overcome the difficulties of evaluating CEP systems and of comparing query management and processing approaches. The chapter starts with a discussion about *CEPSim* motivation and benefits. Following this discussion, Sections 6.2 and 6.3 introduce *CEPSim* design principles and the foundational concepts on top of which *CEPSim* is built. Finally, the simulation algorithms and a thorough evaluation of *CEPSim* are presented in Sections 6.4 and 6.5.

## 6.1 Motivation

The resurgence of interest in CEP systems caused by the new Big Data world has been accompanied by the use of cloud environments as their runtime platform. Clouds are usually leveraged to provide the low latency and scalability needed by modern applications [25, 69, 128]. Other systems, such as the *CEPaaS* system proposed in this research, also explore cloud computing to facilitate offering CEP functionalities in the services model. In this context, the development of efficient operator placement and scheduling strategies is essential to achieve the required quality of service. However, validating these strategies at the required Big Data scale in a cloud environment is a difficult problem and constitutes a research problem *per se*.

First, cloud environments are subject to variations that make it difficult to reproduce the environment and conditions of an experiment [56]. Moreover, setting up and maintaining large cloud environments are laborious, error-prone, and may be associated with a high financial cost. Finally, there are also many challenges related to generating and storing the volume of data required by Big Data experiments.

Simulators have been used in many different fields to overcome the difficulty of execut-

[1]The content of this chapter has been published as a conference paper [76] and as a journal paper [77].

ing repeatable and reproducible experiments. Early research into distributed systems [105] and grid computing [33] used simulators, as well as the more recent field of cloud computing [34, 92, 119]. Generally, cloud computing simulators make it possible to model cloud environments and to simulate different workloads running on them. Nonetheless, these simulators are mostly based on application models and simulation algorithms that cannot represent properly the dynamics of CEP systems. To overcome these limitations, this research presents *CEPSim*, a flexible simulator of cloud-based CEP systems.

*CEPSim* extends CloudSim [34] using a query model based on *AGeCEP* and introduces simulation algorithms based on a novel abstraction called *event sets*. *CEPSim* can be used to model different types of clouds, including public, private, hybrid, and multi-cloud environments, and to simulate execution of user-defined queries on them. In addition, it can also be customized with various operator placement and scheduling strategies. These features enable system architects and researchers to analyze the scalability and performance of cloud-based CEP systems and to easily compare the effects of different query processing strategies.

## 6.2 System Overview

*CEPSim* is a simulator for cloud-based CEP systems that can be used to study the scalability and performance of CEP queries and to compare the effects of different query processing strategies. It has been developed with the following design principles as goals:

- *Generality*: it can simulate different cloud-based CEP systems independently of query definition languages and platform specificities.
- *Extensibility*: it can be extended with different operator placement and operator scheduling strategies.
- *Multi-Cloud*: it can run simulations that span multiple clouds;
- *Reuse*: it can reuse capabilities that are present in CloudSim and comparable simulators.

Because of its maturity and extensibility, CloudSim was chosen as the base cloud simulator on top of which *CEPSim* was built. Figure 6.1 shows an overview of *CEPSim* and how it is related to CloudSim.

CloudSim provides the basic *simulation framework* and two main groups of functionalities: *datacentres* and *policies*. The former group includes abstractions used to represent the physical cloud environment, whereas the latter consists of customizable strategies that control the dynamic aspects of the environment.

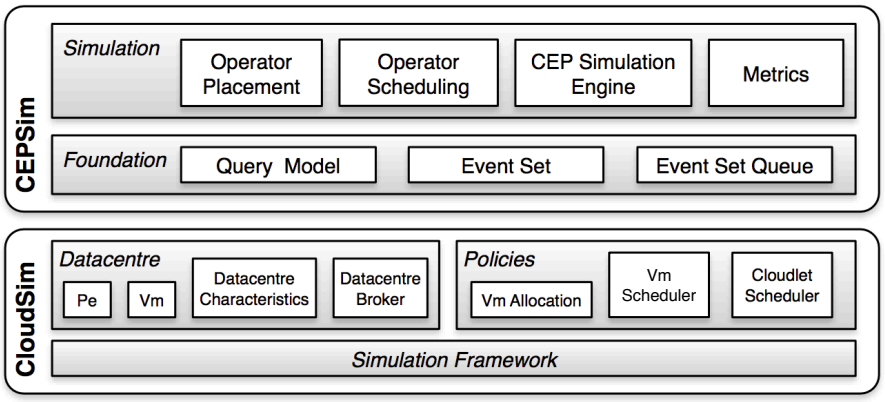



Figure 6.1: *CEPSim* overview.

*CEPSim* significantly extends these functionalities to enable simulation of CEP queries. In Figure 6.1, these extensions are also organized into two groups: *foundation* and *simulation*. The former group contains the fundamental *CEPSim* abstractions and is detailed in Section 6.3, whereas the latter implements the CEP simulation logic and is described in Section 6.4.

To achieve the *generality* goal, *CEPSim* assumes that user queries can be converted to the *AGeCEP* query model described in Section 4.4.1. As discussed previously, *AGeCEP* provides a technology- and language-agnostic representation of queries to which diverse query languages can be converted.

Once converted, *CEPSim* assumes that the queries run continuously, processing input events that are constantly pushed into the system. The input streams are expected to be unbounded, but the user must specify for how long the simulation should run.

To simulate distributed (networked) queries, *CEPSim*'s distribution model assumes that parts of the query ADAG are allocated to different VMs and that these VMs communicate with each other using a network. In addition, *CEPSim* assumes that multiple queries may be running simultaneously in the same VM and that they can belong to different users.

Finally, *CEPSim* does not execute any form of single-query or multiple-query optimization because it expects that the submitted queries have already been optimized. Nevertheless, to support these optimizations, *CEPSim* allows event sources and operators to be shared among queries according to the strategies described in Section 4.3.2.

Currently, the main limitation of *CEPSim* is the fact it only supports scenarios in which the number of simulated queries is fixed and these queries are neither reconfigured nor fail at runtime. However, most often this limitation can be circumvented by running and comparing two simulations: one of a scenario before reconfiguration, and another of a scenario after.

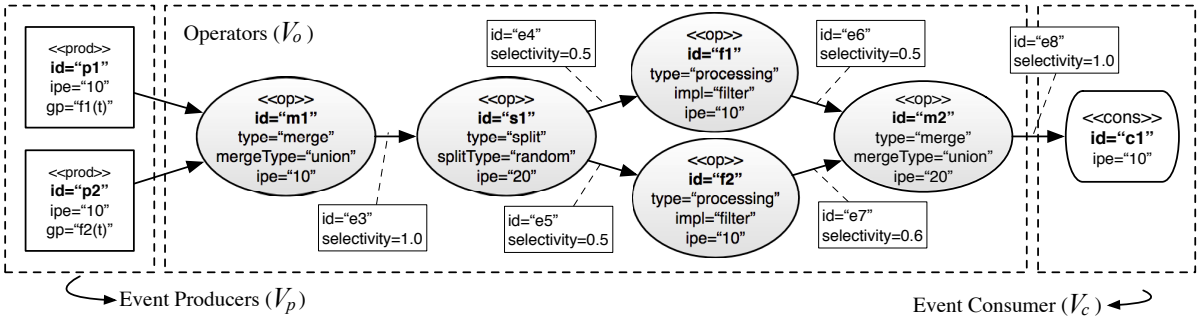



Figure 6.2: *CEPSim* query example.

## 6.3 CEPSim Foundation

This section presents *CEPSim* foundation concepts on top of which the simulation algorithms are implemented. First the *CEPSim* query model, which is used to define the simulated queries, is discussed. Following, the event set and event set queue abstractions are described.

### 6.3.1 Query Model

*CEPSim* uses *AGeCEP* as its formal foundation. Therefore, every user-defined query $q$ is represented by an attributed directed acyclic graph $G = (V, E, ATT)$, where each vertex $v \in V$ represents a query element and the edges $(u, v) \in E$ represent event streams flowing from an element $u$ to another element $v$. In addition, the set of vertices $V$ is partitioned into $V_p$, $V_c$, and $V_o$ representing *event producers*, *event consumers*, and *operators* respectively. Figure 6.2 shows an example of a query $q$. Some attributes have been omitted for the sake of clarity.

*CEPSim* overcomes CloudSim batch application model by using *AGeCEP* query model, which can represent complex data processing flows consisting of multiple interconnected steps. In addition, as discussed in Section 4.4.1, most existing CEP query languages can be converted to the *AGeCEP* model, which emphasizes the generic aspect of *CEPSim*.

Moreover, *CEPSim* extends *AGeCEP* representation in order to make it more appropriate for simulations. First, every vertex is extended with a new attribute *ipe*, which represents the number of CPU instructions needed to process a single event. This is an important piece of information required by the simulation algorithms. For event producers, this attribute estimates the number of instructions required to take an event from the system input and forward it to query execution. In other words, it does not include the effort required to generate the event because event generation does not usually occur within the CEP system.

Second, every edge $(u, v) \in E$ is extended with a *selectivity* attribute that determines how many of the events processed by $u$ are actually sent to $v$. In Figure 6.2, the query edges are

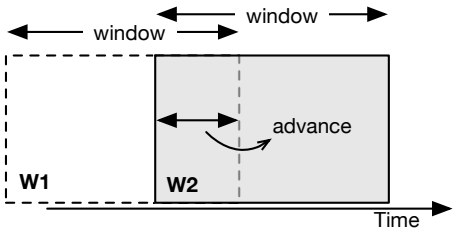



Figure 6.3: Windowed operator attributes.

annotated with their selectivity values. For instance, edges $e_4$ and $e_5$ selectivity are both 0.5. Therefore, if $s_1$ processes 100 events, 50 of them will be sent to $f_1$ and the other 50 to $f_2$. A selectivity can be greater than one in the case where the operator outputs more than one event based on a single input, e.g., creating two alarms from a single sensor reading. Note that in *AGeCEP*, *selectivity* is also a vertex attribute that refers to the total number of events that are output as a function of the number of input events. In other words, the vertex *selectivity* is the sum of all its outgoing edges *selectivity*.

Third, *CEPSim* also introduced the *"windowed"* stereotype to characterize operators that process windows of events and combine them in some manner. Typical examples are aggregation operators that count events or calculate the average value of attributes. This new stereotype is necessary because the simulation of windowed operators is implemented by a different algorithm that requires information not included in the regular *"operator"* stereotype. In particular, windowed operators have three extra attributes: a *window* size, an *advance* duration, and a *combination* function.

Figure 6.3 illustrates the *window* and *advance* concepts. The *window* specifies the period of time from which the events are taken and the *advance* duration defines how the window slides when the previous window closes. Finally, the *combination* function is defined as:

$$f : \mathbb{R}^m_{\geq 0} \rightarrow \mathbb{R}_{\geq 0} \tag{6.1}$$

where $m$ is the number of operator predecessors. This function regulates the number of events that are sent to the output given the number of events accumulated in the input. Commonly, it is defined as a constant function $f(\vec{x}) = 1$, meaning that for each window only one event is generated (e.g., for counting events).

Finally, every event producer $p$ in *CEPSim* is associated with a *generator* function $g_p$ that determines the total number of events produced by $p$ given a point in time. Formally, the generator function is defined as a monotonically increasing function from the time domain to

the set of positive integers:

$$g_p : \mathbb{R}_{\geq 0} \rightarrow \mathbb{N}, \text{ s.t. } x \leq y \text{ then } g_p(x) \leq g_p(y) \tag{6.2}$$

### 6.3.2 Event Sets

An *event set* is an abstraction that represents a batch of events and is the basic processing unit used by *CEPSim*. This abstraction has been created to improve the simulator performance and to assist in calculating the simulation metrics. Operators exchange *event sets* instead of individual events, and all system queues and temporary buffers are composed of *event sets*.

Formally, an *event set* $e$ is an instance of an *EventSet* class that contains the following attributes[2]:

- *cardinality (cn)*: number of events in the set. The notation $|e|$ is used hereinafter as a shortcut for $e.cn$.

- *timestamp (ts)*: a timestamp associated with the set, which can be used for various purposes. Most often, it contains the timestamp at which the set has been created.

- *latency (lt)*: the average of the latencies of the events in the set. Event latency is defined as the period of time elapsed from the event creation to the moment at which the event is added to the set.

- *totals (tt)*: a map that, for each producer $v_p \in V_p$, stores the number of events that must have been produced by $v_p$ to originate the events currently in the set. The goal of this attribute is to track *caused by* (or *is result of*) relationships between the events in the set and the produced events.

In addition to these attributes, four operations are also defined for event sets: *sum*, *extract*, *select*, and *update*.

- *Sum*: is applied to two event sets $e_1$ and $e_2$ and results in a new event set $e_r$ containing all events from both sets. It is defined as:

$$e_r = e_1 + e_2 \tag{6.3a}$$

[2]Hereafter, the dot notation is used to access object attributes.

such that

$$|e_r| = |e_1| + |e_2| \tag{6.3b}$$

$$e_r.ts = \frac{|e_1| \cdot e_1.ts + |e_2| \cdot e_2.ts}{|e_1| + |e_2|}, \tag{6.3c}$$

$$e_r.lt = \frac{|e_1| \cdot e_1.lt + |e_2| \cdot e_2.lt}{|e_1| + |e_2|}, \tag{6.3d}$$

$$e_r.tt : V_p \rightarrow \mathbb{R}_{\geq 0}, \text{ s.t. } e_r.tt[v_p] = e_1.tt[v_p] + e_2.tt[v_p] \tag{6.3e}$$

- *Extract*: is applied to an event set $e$ and the number of events to be extracted $n$. The results are an event set $e_r$ consisting of the extracted events, and an event set $e_m$ containing the remaining events from $e$,

$$(e_r, e_m) = e - n \tag{6.4a}$$

such that

$$|e_r| = n \tag{6.4b}$$

$$e_r.tt : V_p \rightarrow \mathbb{R}_{\geq 0}, \text{ s.t. } e_r.tt[v_p] = (n/|e|) \cdot e.tt[v_p] \tag{6.4c}$$

$$|e_m| = |e| - n \tag{6.4d}$$

$$e_m.tt : V_p \rightarrow \mathbb{R}_{\geq 0}, \text{ s.t. } e_m.tt[v_p] = e.tt[v_p] - e_r.tt[v_p] \tag{6.4e}$$

and the latency and timestamp attributes from $e_r$ and $e_m$ are the same as in $e$.

- *Select*: is applied to an event set $e$ and a selectivity $s$. It selects a subset of events from the event set:

$$e_r = e * s \tag{6.5a}$$

such that

$$|e_r| = |e| \cdot s \tag{6.5b}$$

and the remaining attributes from $e_r$ are the same as in $e$.

- *Update*: is applied to an event set $e$ and a timestamp $ts$. It simply brings the event set latency and timestamp up to date:

$$e_r = update(e, ts) \tag{6.6a}$$

**Algorithm 6.1:** Event set queue - dequeue operation.

**Data**: ▹ $Q$, Event set queue
▹ $n$, Number of events to be extracted

```
1  function dequeue(Q, n)
2      e ← empty event set
3      while n > 0 and !isEmpty(Q) do
4          h ← dequeue(Q)                  // Extract the head of the queue Q
5          if |h| > n then
6              (h, r) ← h − n
7              prepend(Q, r)               // Return r to the head of the queue Q
8          end
9          e = e + h
10         n ← n − |h|
11     end
```

such that

$$e_r.ts = ts \tag{6.6b}$$

$$e_r.lt = e.lt + (ts - e.ts) \tag{6.6c}$$

and the remaining attributes from $e_r$ are the same as in $e$.

### 6.3.3 Event Set Queues

An *event set queue* is simply a queue where the elements are event sets. As with any regular queue, it is possible to enqueue and dequeue elements in a *first-in, first-out* manner. In addition, an event set queue has an overload *dequeue* operation that receives the number of events to be extracted and returns an event set representing these events.

Algorithm 6.1 shows this operation in pseudo-code. The algorithm removes event sets from the queue $Q$ until the resulting event set $e$ reaches size $n$. When the removed event set $h$ has more events than what is required to complete $n$, the algorithm extracts the necessary events from $h$ and returns the remaining events to the queue (lines 5-8).

Finally, an event set queue $Q$ also has a *cardinality* defined as the sum of the cardinalities of all event sets in the queue:

$$|Q| = \sum_{e \in Q} |e| \tag{6.7}$$

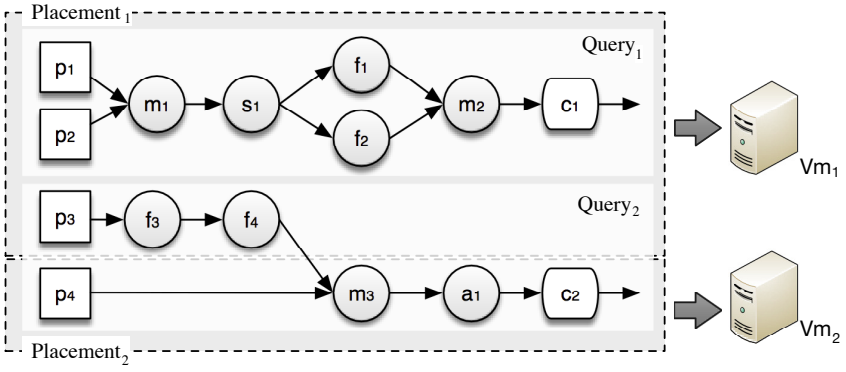



Figure 6.4: Placement definitions.

## 6.4 CEPSim Simulation

This section presents the CEP simulation logic implemented by *CEPSim*. First it is discussed the role of operator placement and scheduling strategies in the simulation. Following, the simulation procedures are presented both at operator and at placement level. Finally, it is described how *CEPSim* calculates the simulation metrics.

### 6.4.1 Operator Placement

Once the queries are modelled, the next step in any simulation is to define a set of *placements*. Each placement maps a set of query vertices to the VM where they will execute. Note that the vertices from a single query can be mapped to more than one VM, which implies distributed execution of the query. A placement can also contain vertices from more than one query. Figure 6.4 illustrates the placement concept: $Placement_1$ maps all vertices from $Query_1$ and some from $Query_2$ to $Vm_1$, whereas $Placement_2$ maps the remaining $Query_2$ vertices to $Vm_2$.

Defining placement for a query is part of its lifecycle, as discussed in Section 2.1.5. This mapping is one of the most determining factors of a CEP system performance. Because of this importance, *CEPSim* is pluggable and enables the use of different placement strategies. By default, users must manually specify the mapping of vertices to VMs when submitting a query to *CEPSim*.

### 6.4.2 Operator Scheduling

*Operator scheduling* is the procedure that, given a set of running queries and their internal state, defines which operator should run next and for how long it should run. A *scheduling strategy* can fundamentally determine the performance of a CEP system by optimizing for different aspects of the system, such as overall QoS [1] or memory consumption [23]. Because

of this significance, *CEPSim* also allows different scheduling strategies to be plugged in and used during a simulation.

*CEPSim* contains two built-in *scheduling strategies*, and both are based on an auxiliary *allocation strategy*. In this context, the allocation strategy divides the available instructions among the placement vertices, whereas the scheduling strategy determines how the vertices are traversed and how the allocated instructions are used.

The two allocation strategy implementations provided by *CEPSim* are:

- *Uniform allocation*: divides the available instructions equally among all placement vertices.
- *Weighted allocation*: divides the available instructions proportionally to the *ipe* attribute of each vertex.

These two strategies can be combined with the provided *scheduling strategies*, which work as follows:

- *Simple scheduling*: the vertices are sorted in topological order and traversed only once according to this order. Each vertex receives the number of instructions determined by the allocation strategy, independently of the number of instructions required.
- *Dynamic scheduling*: the vertices are sorted in topological order and traversed in one or more rounds. In each round, each vertex receives the minimum between the number of instructions determined by the allocation strategy and the number of instructions required to process all events in the input queues. The process is repeated until there are no more instructions left to be allocated or events to be processed. This strategy tries to redirect non-used instructions to overloaded vertices and thereby improve query throughput.

### 6.4.3 Operator Simulation

In *CEPSim*, the simulation of an operator execution is accomplished by reading event sets from the operator's input queues, processing them, and writing output event sets to its output queues. The general procedure used to simulate an operator execution is detailed in Algorithm 6.2.

The algorithm operates in three main steps:

1. *Lines 2-6*: Calculates the number of input events that can be processed. This number is the minimum between the total number of events in all input queues and the maximum number of events that can be processed given the number of allocated instructions $n$. This maximum is obtained by dividing $n$ by the operator *ipe* attribute.

**Algorithm 6.2:** Operator simulation.

**Data**: ▹ *op*, operator with attributes:
- *ipe*, instructions per second
- *pred*, operator predecessors
- *succ*, operator successors
- *input*, map of input event set queues
- *selectivity*, map of outgoing edge selectivities
- *ouput*, map of output event set queues

▹ $n$, number of instructions
▹ $ts$, start timestamp

```
1  function simulate(op, n, ts)
2      tot_in ← 0
3      forall the v_p ∈ op.pred do
4          tot_in ← tot_in + |op.input[v_p]|
5      end
6      evt ← min(tot_in, n/op.ipe)
7      e ← empty event set
8      forall the v_p ∈ op.pred do
9          no ← (|op.input[v_p]|/tot_in) * evt
10         e ← e + dequeue(op.input[v_p], no)
11     end
12     e ← update(e, ts)
13     forall the v_s ∈ op.succ do
14         en ← e * op.selectivity[v_s]
15         enqueue(op.output[v_s], en)
16     end
```

2. *Lines 7-11*: Dequeues events from the input queues and builds a new event set $e$ representing the dequeued events. The number of events dequeued from each input queue is proportional to its size. This procedure aims to balance the queues by processing more events from queues with more elements.

3. *Lines 12-16*: Enqueues the recently created event set $e$ into the operator output queues. While enqueuing, the selectivity value of the edge connecting the operator to each of its successors $v_s$ is taken into consideration.

Event producers and consumers are simulated in a similar way. Because event producers do not have predecessor vertices, the input events are read from the *generator* associated with them. Event consumers, on the other hand, do not have output queues. The processed events are accumulated into a single *output* event set that consolidates all events consumed during the simulation.

**Algorithm 6.3:** Windowed operator simulation.

**Data**: ▹ $w$, with all regular operator attributes plus:
- *window*, window size
- *advance*, advance period
- $f$, combination function
- *acc*, accumulation data structure
- *index*, current slot in the accumulation data structure
- *next*, next timestamp at which a window closes

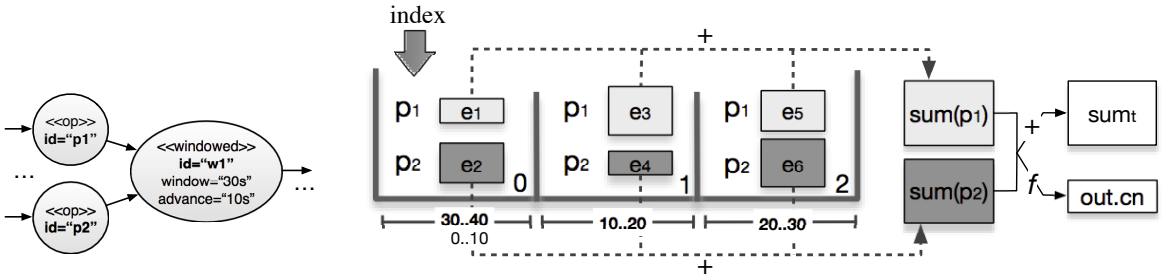
▹ $n$, number of instructions
▹ $ts$, start timestamp

```
1  function simulate(w, n, ts)
2      slots ← w.window/w.advance
3      while ts > w.next do
4          generateOutput(w, w.index, ts)
5          w.next ← w.next + w.advance
6          w.index ← (w.index + 1) mod slots
7          reset(w.acc, w.index)
8      end
9      tot_in ← 0
10     forall the v_p ∈ w.pred do
11         tot_in ← tot_in + |w.input[v_p]|
12     end
13     evt ← min(tot_in, n/w.ipe)
14     forall the v_p ∈ w.pred do
15         no ← (|w.input[v_p]|/tot_in) * evt
16         e ← dequeue(w.input[v_p], no)
17         accumulate(w.acc, w.index, v_p, e)
18     end
```

Simulating windowed operators is different because output events are generated only when a window closes. In addition, whenever a window does not close, the input events must be correctly processed and accumulated.

Algorithm 6.3 describes the simulation procedure of a windowed operator $w$. To implement the simulation, every windowed operator has an auxiliary data structure that is used to accumulate the processed events. Figure 6.5 shows an example of a windowed operator and its corresponding data structure.

The data structure works as a circular array divided into $l$ slots, on which each slot represents a timeframe equivalent to one advance period within the time window. For example, the windowed operator from Figure 6.5a has a window size of 30 seconds and an advance period of 10 seconds, resulting in an array of size 3. Initially, slots 0, 1, and 2 represent the intervals




(a) Operator example. (b) Auxiliary data structure.

Figure 6.5: Windowed operator simulation.

between 0-10, 10-20, and 20-30 seconds respectively. Each position of this array contains one event set for each operator predecessor ($p_1$ and $p_2$). These event sets accumulate events coming from the predecessors during each slot period.

To use this data structure, the windowed operator maintains two auxiliary variables, *index* and *next*. The *index* variable points to the slot where the accumulation should currently take place, whereas *next* stores the next timestamp at which the window closes.

These variables are primarily used between lines 2 and 8 of Algorithm 6.3. First, when a window closes, an auxiliary procedure *generateOutput* is invoked to generate the output event set (Algorithm 6.4). Following this invocation, the *next* and *index* variables are adjusted, and the next slot is reset. Note that this loop can be executed more than once if more than one window has closed since the last simulation.

The following lines (9 to 18) are similar to the stateless operator simulation presented in Algorithm 6.2, but instead of writing the processed event sets into the output queues, they are accumulated at the current time slot.

The last part of the windowed operator simulation is the *generateOutput* procedure shown in Algorithm 6.4. The loop between lines 4 and 11 builds an event set for each predecessor and a sum of these event sets $sum_t$. This step is also shown in Figure 6.5b, in which $sum(p_1)$ is calculated as $e_1 + e_3 + e_5$, $sum(p_2)$ as $e_2 + e_4 + e_6$, and $sum_t$ is the sum of $sum(p_1)$ and $sum(p_2)$.

From lines 12 to 16, the output event set *out* is built according to the idea that this event set is *caused by*, or *is a result of*, all events accumulated in the window:

- *cardinality* (*out.cn*) is set to the result of the combination function $f$. This function receives as argument a set of event sets, each one encapsulating all events received from a specific predecessor $v_p$ during the window timeframe, and returns the number of events that must be generated.

- *latency* (*out.lt*) is set to the average latency of all events in the window ($sum_t.lt$) plus

**Algorithm 6.4:** Windowed operator - generate output.

```
Data: ▷ w, windowed operator
▷ index, current index in w(acc)
▷ ts, start timestamp

1  function generateOutput(w, index, ts)
2      sum ← empty map
3      sum_t ← empty event set
4      forall the v_p ∈ w.pred do
5          e ← empty event set
6          for i = 0 to slots do
7              e ← e + w.acc[i][v_p]
8          end
9          sum_t ← sum_t + e
10         sum[v_p] ← e
11     end
12     out ← empty event set
13     out.cn ← f(sum)
14     out.lt ← sum_t.lt + (ts − sum_t.ts)
15     out.ts ← ts
16     out.tt ← sum(w.acc[index]).tt
17     forall the v_s ∈ w.succ do
18         en ← out ∗ w.selectivity[v_s]
19         enqueue(w.output[v_s], en)
20     end
```

their average waiting time. The waiting time is calculated as the difference between the current timestamp ($ts$) and the average timestamp of all events in the window ($sum_t.ts$).

- *timestamp* ($out.ts$) is set to the current timestamp.

- *totals* ($out.tt$) is set to the sum of all totals from the event sets in the current slot only, as events in previous slots have already been considered in past windows.

### 6.4.4 Placement Simulation

After describing how *CEPSim* simulates operators, this subsection focuses on the algorithm used to simulate queries. A pseudo-code description of this procedure is presented in Algorithm 6.5.

The first thing to note is that the basic unit of simulation is a placement, not a query, which implies that all vertices allocated to a VM are simulated at once. This approach enables

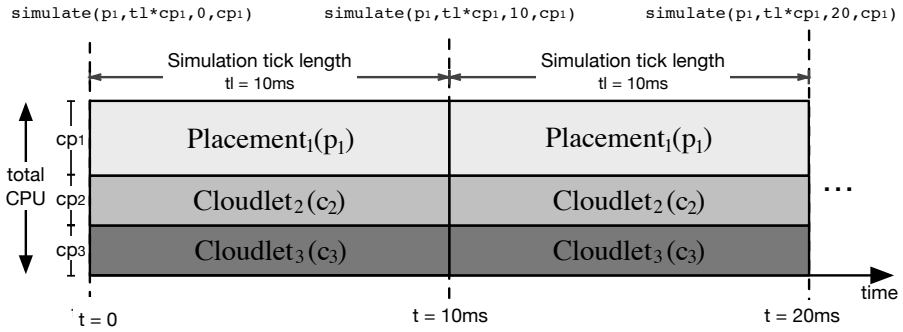



Figure 6.6: Execution of a simulation tick.

operator scheduling strategies to consider simultaneously all vertices in a VM and potentially make better decisions regarding their scheduling optimization criteria.

This procedure simulates execution of a placement for the duration of a *simulation tick*. As shown in Figure 6.6, the CloudSim simulation framework repeatedly invokes this procedure to represent the passing of time. Therefore, the *simulation tick length* is a parameter that enables users to trade off precision against computational cost. For example, if the tick is long, the procedure will be invoked fewer times, but the produced events will be grouped into relatively large event sets and processed as such. On the other hand, a shorter tick translates into smaller event sets and potentially more precise results.

The following parameters are required by the procedure: a pre-allocated number of instructions $n$, the simulation time at which the procedure has been invoked $ts$, and the CPU capacity $cp$ (measured in MIPS) available to the placement. The CloudSim simulation framework determines these arguments at each invocation: first, a *cloudlet scheduler* calculates $cp$ by distributing the total CPU processing power among all processes concurrently running on the VM. In Figure 6.6, the placement $p_1$ has only $cp_1$ MIPS available because it shares the same VM with two cloudlets $c_2$ and $c_3$. Following, the number of instructions $n$ is derived by multiplying the available capacity $cp$ by the *simulation tick length*. In Figure 6.6, the value of $n$ is equivalent to the area encompassed by each process.

In summary, there are three main steps in Algorithm 6.5:

1. All generators associated with the placement event producers are activated to determine the number of events that have been generated from the last simulation tick to the current one (lines 2-4);

2. The scheduling strategy associated with the placement is invoked to define the order in which the vertices will be simulated and the number of instructions allocated to each vertex (line 5).

**Algorithm 6.5:** Placement simulation.

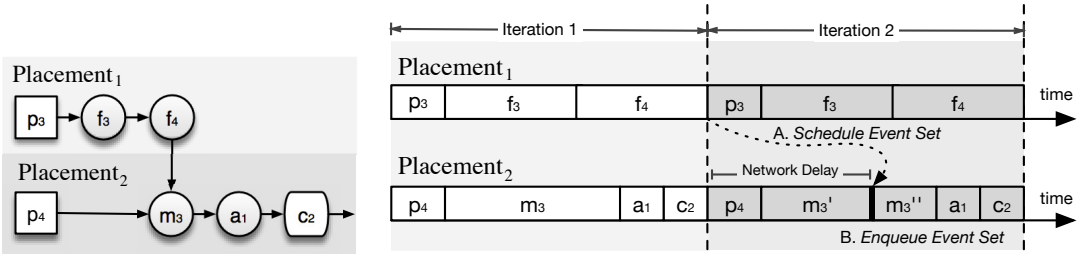
```
Data: ▹ Schedule, Operator scheduling strategy
▹ p, placement to be simulated
▹ n, number of instructions
▹ ts, start timestamp (in ms)
▹ cp, allocated CPU capacity (in MIPS)

1  function simulate(p, n, ts, cp)
2      forall the v_p ∈ p.producers do
3          generate(v_p, ts)
4      end
5      it ← schedule(p, n)
6      while next ← it.next do
7          v ← next.v
8          n_v ← next.n
9          simulate(v, n_v, ts)
10         adjustTime(ts)
11         forall the v_s ∈ v.succ do
12             en ← dequeue(v.output[v_s])
13             enqueue(v_s.input[v], en)
14         end
15     end
```

3. All vertices are traversed and simulated according to the specified order (lines 6-15). The scheduling strategy returns an iterator of pairs, each one containing a vertex pointer ($next.v$) and the number of instructions allocated to it ($next.n$). With these two parameters, the operator simulation procedure is invoked (line 9). Next, the current timestamp $ts$ is adjusted to reproduce the passing of time (line 10). Finally, the event sets in each of the vertex output queues are moved to the input queues of their respective successors (lines 11-14).

### Networked Queries

To simulate networked (distributed) queries, the *CEPSim* placement simulation from Algorithm 6.5 received two main modifications.

First, at the moment that event sets are moved from the operator output queues to the input queues of its successors (lines 11-14), the algorithm checks whether the successor vertex belongs to the same placement or not. If it does, the event set is moved to the destination queue as usual. If it does not, then the event set and the destination vertex *id* are sent to a *network interface*, which executes three main steps:




(a) Query example. (b) Simulation algorithm.

Figure 6.7: Networked query simulation.

1. Locate the placement where the successor vertex resides by consulting the *CepSimBroker* (implementation details can be examined in Appendix B).

2. Calculate the delay in transferring the event set to the destination VM. This calculation depends on the network interface implementation in use.

3. Schedule a simulation event on the destination VM signalling the arrival of the event set. This event is scheduled using the simulation framework provided by CloudSim.

The second modification is in the main loop between lines 6 and 15. Before each iteration, the algorithm checks whether any simulation event (representing the arrival of an event set) is scheduled during the operator time slice. If one is, the time slice is split in two at the event set arrival time, and the event set is enqueued into its destination queue between the two slices.

This procedure is illustrated in Fig. 6.7. In the query from Figure 6.7a, vertices $p_3$, $f_3$, and $f_4$ are placed into one VM, and the remaining vertices are placed into another. The diagram in Figure 6.7b shows the placements schedule as a function of time. At the end of the first iteration, vertex $f_4$ "sends" an event set to its successor $m_3$. This step is represented by scheduling a simulation event on the destination placement after the period of time required to transfer the event set from $f_4$ to $m_3$. In the second iteration, the simulation algorithm detects the scheduled event before starting the $m_3$ simulation. The $m_3$ time slice is split into two halves ($m_3'$ and $m_3''$), and the event set is enqueued right after $m_3'$ finishes.

### Bounded Queues

Most CEP systems limit the size of operator queues to avoid memory overflow and to maintain overall system performance. Because of this characteristic, *CEPSim* also supports the definition of bounded input operator queues. When using this feature, it is necessary to define the behaviour of the system when new events arrive at an already full queue. Currently, *CEPSim* supports the application of *backpressure* to vertex predecessors.

When using *backpressure*, at the end of the simulation procedure operators inform their predecessors about the maximum number of events accepted for the next iteration. The predecessors limit their output on the next tick if needed. Nevertheless, when an operator limits its output, it may also accumulate events in its own input queues and consequently apply *backpressure* on its predecessors. Ultimately, the backpressure arrives at the event producers, which may choose to discard extraneous events or accumulate them in their own queues.

### 6.4.5 Metrics

One of the most important parts of any simulator is the set of metrics obtained as a result of the simulation. As CEP queries performance are usually measured in terms of its *latency* and *throughput*, *CEPSim* provides built-in implementations for these two metrics.

#### Query Latency

The *query latency* metric is defined for every consumer $v_c$ as the average number of milliseconds elapsed from the moment an event arrives at the query to the moment it is consumed by $v_c$. In other words, it measures how long a query takes to process an event.

This metric can be easily obtained because event consumers have an *output* event set that accumulates all events that have been consumed during a simulation. Therefore, the value of $latency(v_c)$ is simply the latency of the $v_c$ *output* event set:

$$latency(v_c) = v_c.output.lt \tag{6.8}$$

Figure 6.8 exemplifies how the event set latencies are calculated and updated during a simulation tick. The event sets $es_1$ and $es_2$ were generated at timestamp $ts = 5$. At $ts = 10$ the producer $p_1$ sends $es_1$ to $f_1$, and at $ts = 13$ producer $p_2$ sends $es_2$ to $f_1$. Note that $es_1$ and $es_2$ latency attributes are updated to take into account the time elapsed from the event set generation to the moment they are output. When processed by $f_1$, both event sets are summed according to Equation 6.3a, resulting in a new event set $es_{12}$. At $ts = 15$, a new event set $es_3$ is created by updating $es_{12}$ timestamp and applying $(f_1, f_2)$ selectivity to it:

$$es_3 = update(es_{12}, ts) * (f_1, f_2).selectivity \tag{6.9}$$

Following, the $es_3$ event set is sent to $f_2$, where a similar procedure is executed and a new event set $es_4$ is created. Finally, $es_4$ is sent to the consumer $c_1$, where the final event set $es_5$ is created and added to the *output* event set.

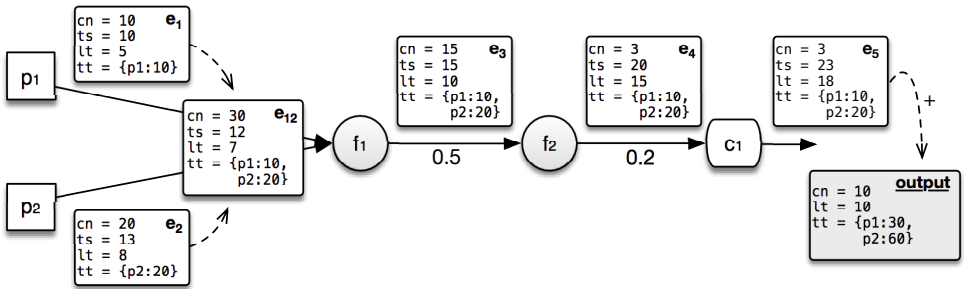



Figure 6.8: Event sets created during a simulation tick.

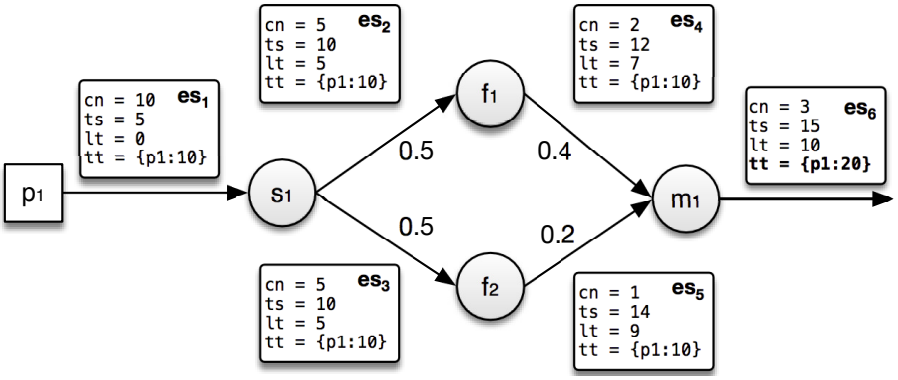


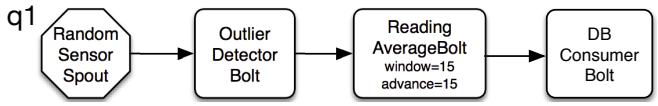

Figure 6.9: Throughput calculation - duplicated *totals* count.

### Query Throughput

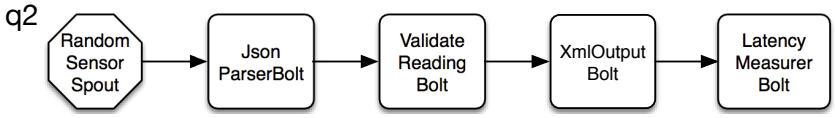
The *query throughput* metric is calculated for each consumer $v_c$ as the average number of events processed per second during its lifespan.

The throughput value of an event consumer $v_c$ is also obtained with the aid of $v_c$'s *output* event set. In this case, its *totals* attribute contains the number of events generated by each producer that have resulted in the events in the set. Therefore, the throughput can be obtained by summing the values for all producers and dividing this sum by the query simulation time (in seconds). However, if there is more than one path from a producer $v_p$ to the consumer $v_c$, then the *output* event set contains duplicates incorporated into the *totals* values for $v_p$ and needs to be fixed.

Figure 6.9 shows a scenario in which this duplication occurs. The operator $s_1$ splits $es_1$ into two event sets $es_2$ and $es_3$, which are transformed into $es_4$ and $es_5$ and combined again into $es_6$. In the $m_1$ output, $es_6.tt(p_1) = 20$, yet the correct value should be 10. The metric can be simply fixed by dividing it by 2 because there are two paths from $p_1$ to $m_1$.




(a) Query $q_1$ - Average window.




(b) Query $q_2$ - JSON converter.

Figure 6.10: Storm topologies.

Formally, the query throughput of a consumer $v_c$ is given by:

$$throughput(v_c) = \Big( \sum_{v_p \in V_p} \frac{v_c.output.tt(v_p)}{|paths(v_p, v_c)|} \Big) / q.time \tag{6.10}$$

where $|paths(v_p, v_c)|$ is the number of paths from producer $v_p$ to consumer $v_c$ and $q.time$ is the total query simulation time.

## 6.5 Evaluation

This section describes the experiments that have been performed to analyze the *CEPSim* simulator. First, *CEPSim* is validated by comparing the latency and throughput metrics obtained by running queries on a real CEP system and by simulating them on *CEPSim*. Second, the simulator performance is assessed by analyzing the execution time and memory consumption of various simulation scenarios. Finally, it is also investigated the effects of different parameters on the simulator behaviour.

### 6.5.1 Case Study

The queries used in the experiments in this section have been extracted from Powersmiths' WOW system [127], a sustainability management platform that draws on live measurements of buildings to support energy management. Powersmiths' WOW uses Apache Storm [18] to process in near real-time sensor readings coming from buildings managed by the platform.

Figure 6.10 shows the Storm queries (topologies) used in the experiments in this section. A *spout* in the Storm terminology is equivalent to an event producer, whereas a *bolt* is equivalent to an operator. There is no concept analogous to an event consumer in Storm.

There are three main steps in the query $q_1$ from Figure 6.10a: the *OutlierDetectorBolt*

Table 6.1: Storm VM cluster specification.

| VM # | CPU | Mem. | Description |
|---|---|---|---|
| 1 | 1 core - Intel Xeon E5-2630 2.6 GHz | 512 MB | zookeeper |
| 2 | 1 core - Intel Xeon E5-2630 2.6 GHz | 768 MB | nimbus |
| 3-6 | 1 core - Intel Xeon E5-2630 2.6 GHz | 2048 MB | workers |

detects and filters anomalous sensor readings, the *ReadingAverageBolt* groups readings into windows of 15 seconds and calculates their average, and the *DBConsumerBolt* stores the calculated average in a database. By aggregating the sensor data into 15-second windows, the query reduces the amount of data that is written to the database.

The query $q_2$ presented in Figure 6.10b, on the other hand, is used to convert from the JSON format to the native WOW format (XML). This query is used because some existing sensors cannot be modified to send data according to the WOW interface. The query is composed of three main steps: the $JsonParserBolt$ parses the JSON request, the $ValidateReadingBolt$ validates the request values, and the $XmlOutputBolt$ converts the request to XML format. The last bolt ($LatencyMeasurerBolt$) is used only to measure the latency and throughput of the conversion process.

### 6.5.2 Environment

Table 6.1 describes the cluster of virtual machines used in the experiments to run Storm topologies. All six VMs were deployed on the same physical server (12 cores Intel Xeon E5-2630, 2.6GHz / 96GB RAM). VMs #1 and #2 run *zookeeper* (which coordinates cluster communication) and *nimbus* (which assigns Storm tasks to workers). The workers VMs #3 to #6 are the ones which effectively execute the queries. The VM memory sizes have been dimensioned to not be a bottleneck in the experiments. A similar physical server hosted the database system and was also used to run all *CEPSim* simulations described in the experiments.

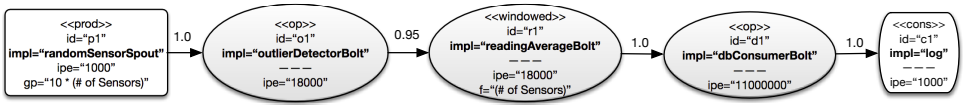
The software used in the experiments is presented in Table 6.2. All Storm topologies have been implemented using Storm's Java API and use standard Java libraries for database access and XML processing.

### 6.5.3 Set-Up

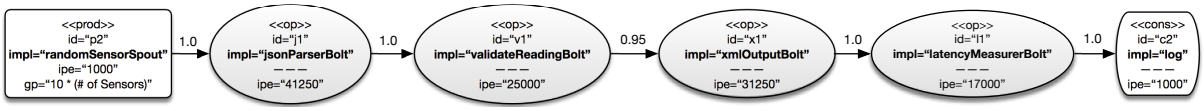
Before any simulation, the Storm queries had to be converted to *AGeCEP* formalism. This conversion was straightforward because both Storm and *AGeCEP* use DAGs as their underlying query model.

Table 6.2: Software specification.

| Name | Version | Description |
|---|---|---|
| Ubuntu | 14.04.2 | Physical server Operating System |
| CentOS | 6.5 | VM Operating System |
| VirtualBox | 4.3.24 | Virtualization Software |
| OpenJDK | 1.7.0_75 | Java Runtime Environment |
| Apache Storm | 0.9.3 | CEP system |
| MySQL | 5.5.41 | Database system |




(a) Query $q_1$.




(b) Query $q_2$.

Figure 6.11: Storm queries converted to the *AGeCEP* model.

Figure 6.11 depicts the *AGeCEP* model of both queries presented in Section 6.5.1. Attributes not used by the simulation are omitted from the figure. Each edge connecting two vertices is annotated with its corresponding selectivity. In both queries, an *event consumer* is also added to group all events consumed by the query.

To estimate the operator's *ipe* attribute, two methods have been used:

- *Latency estimation*: the operator is fed with random events at increasing rates and the average processing time (in milliseconds) is calculated for each rate value. The minimum average is assumed to be the operator latency $op_l$. The *ipe* attribute is then calculated as:

$$op.ipe = (cpu_m \cdot 10^6)/\left(\frac{1000}{op_l}\right) \tag{6.11}$$

  where $cpu_m$ is the CPU processing power estimated in MIPS.

- *Maximum throughput estimation*: the maximum throughput $op_t$ is estimated by feeding the operator process with as many events as possible. The *ipe* attribute is then estimated as:

$$op.ipe = (cpu_m \cdot 10^6)/op_t \tag{6.12}$$

Experimental results have shown that the "latency" method provides better estimation for lower throughput operators, such as the *DBConsumerBolt*, whereas the "maximum through-

Table 6.3: Simulation parameters.

| | Parameter | Value |
|---|---|---|
| *CloudSim* | VM Processor | 2 x 2500 MIPS |
| | VM Allocation Policy | Simple |
| | VM Scheduler | Time shared |
| *CEPSim* | Simulation Tick Length | 100ms |
| | Placement Strategy | User defined |
| | Allocation Strategy | Uniform |
| | Scheduling Strategy | Dynamic |
| | Generator | Uniform |
| | Queue size | 2048 |

put" method is better for higher throughput operators. This difference exists mainly because it is hard to estimate latency accurately when the time spent processing each event is very short.

For the experiments in this research, all $ipe$ values were calculated using the "maximum throughput" method, except for $DBConsumerBolt$.

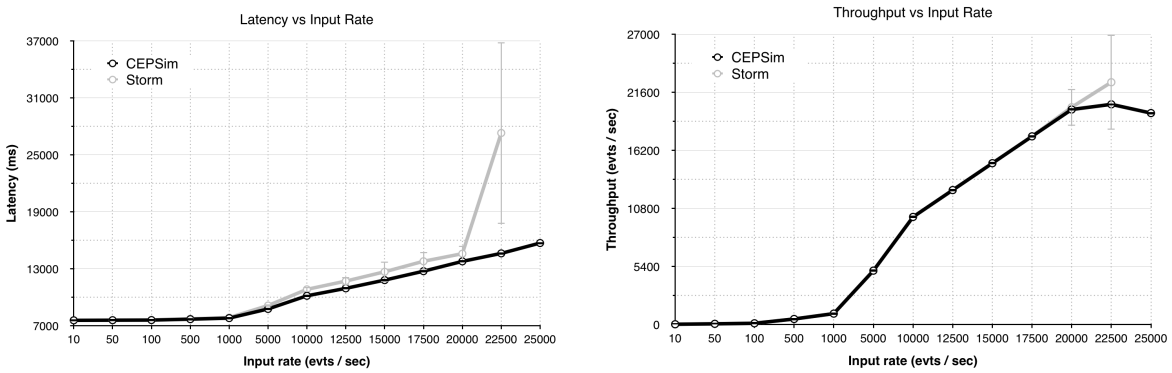
### 6.5.4 Validation

The first step in *CEPSim* validation was to unit test all components and to execute a set of sanity checks to detect programming bugs and inconsistent behaviour. After this phase, a set of experiments was executed aiming to compare the performance metrics obtained by running queries on a real CEP system (Apache Storm) and by simulating them on *CEPSim*. This validation approach is similar to the ones adopted by other simulators, such as NetworkCloudSim [56], iCanCloud [119], and Grozev and Buyya [66].

In all simulations, *CEPSim* was used to create an environment as close as possible to the Storm VM cluster. Table 6.3 summarizes the main parameters used in the simulations. VMs have been modelled as having two processors, even though only one physical processor was allocated for each. This was done because the processors used in the experiments are *hyper-threaded*, which enables a higher degree of parallelism than regular processors. The queue size was set to 2048 because by default Storm has buffers with 1024 elements at both the output and input of each operator, but in *CEPSim*, accumulation happens only at the operators' input queues.

#### Single Query

This first experiment validated *CEPSim* simulation of a single query running entirely on a single VM.




(a) Latency. (b) Throughput.

Figure 6.12: Metrics estimation results - query $q_1$.

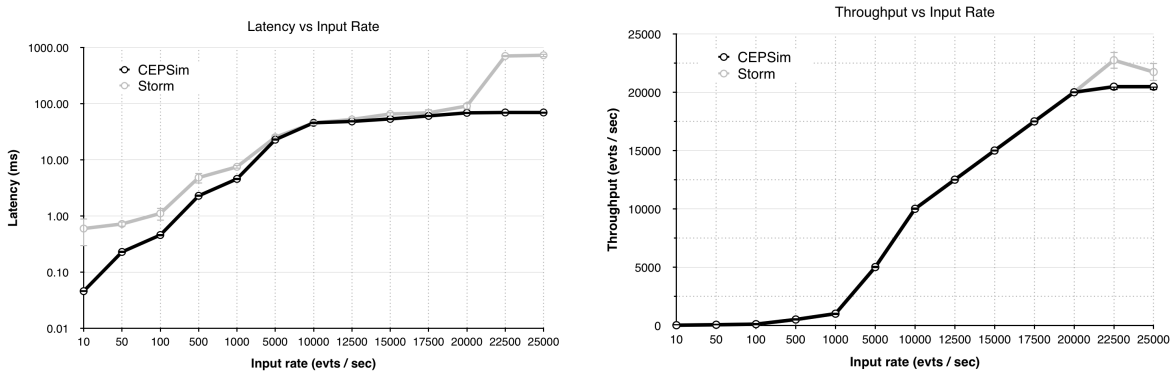
To obtain the Storm metrics, both queries from Fig. 6.10 were first instrumented to output the average throughput and latency every minute. In addition, the query *S pout*s (event producers) were modified so that the user could define the number of sensors $n$ that send data to the query. Each sensor generated 10 sensor readings per second, of which 5% were anomalies.

The graphs from the experiments were obtained by varying the number of sensors $n$, which consequently varied the number of events generated per second. For each $n$, the queries were run for 15 minutes and the average latency (throughput) for each of the last 10 minutes were collected. Note that each data point is an observation from a sampling distribution of the average query latency (throughput). *CEPSim* results were also obtained by varying $n$ and by collecting metrics of the last 10 minutes of 15 minutes simulations. The graphs show the mean value of these averages and their 99% confidence interval (in other words, the confidence interval of the sampling distribution). In most cases, the confidence interval is small and not visible in the graphs.

Figure 6.12 shows query $q_1$ latency and throughput as a function of the input rate. Generally speaking, *CEPSim* achieved very high accuracy for both metrics when compared to Storm. The latency estimation error was less than 1% up to 1000 events/second and was kept below 7.5% up to 20000 events/second. The throughput calculation was even more accurate, with almost no error up to 20000 events/second.

The major estimation error occurred at 22500 events/second, at which point the latency obtained by *CEPSim* was lower than the real value. Further analysis showed that at this point, the Storm query overloaded, and its behaviour became very unpredictable, as can be seen in the high variance of this data point. Nevertheless, *CEPSim* still correctly predicted the maximum query throughput around 21000 events/second, as shown in the throughput drop in Figure 6.12b.

Results for the latency and throughput of query $q_2$ are shown in Figure 6.13. The latency




(a) Latency. (b) Throughput.

Figure 6.13: Metrics estimation results - query $q_2$.

axis in Figure 6.13a has a log scale because the measured values encompass five orders of magnitude. Once again, the throughput calculation exhibited very small error, and the maximum query throughput was closely estimated at approximately 21000 events/second.

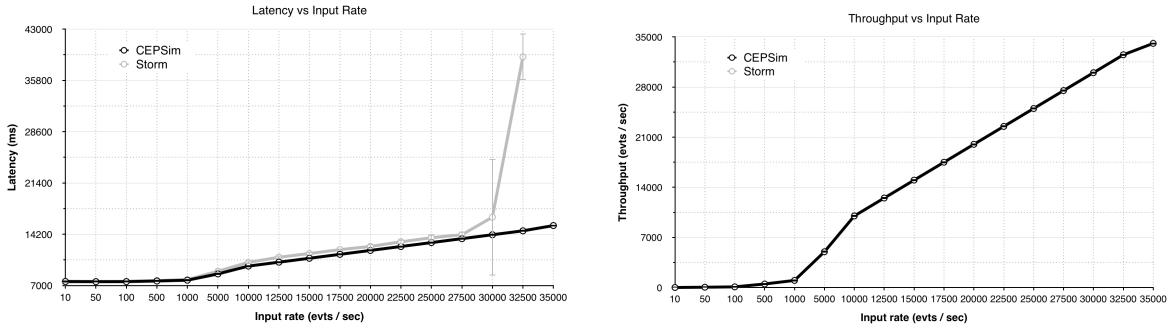
The latency estimation at slow input rates showed some error because it is extremely hard to estimate latency accurately at sub-millisecond precision. At 100 events/second, the simulation values approached those obtained with Storm and remained close up to the overload point at 22500 events/second. After this point, the simulation latency plateaued, whereas the Storm value spiked. This difference was caused mainly by the way that *CEPSim* handles full queues by using backpressure and discarding generated events. Storm, on the other hand, delays generation of events, but does not discard them.

### Networked Query

This experiment aimed to validate *CEPSim* simulation of distributed queries. To perform this experiment, the query from Figure 6.10a was distributed into two VMs, such that the *DBConsumerBolt* was placed into the *worker*2 server and all remaining vertices into *worker*1.

A constant delay network interface was used to simulate this query. In this network implementation, every event set sent through the network takes a fixed amount of time to arrive at its destination. This is a reasonable approximation because all VMs from the experiment were running on the same physical server and no real network traffic was generated. The delay was estimated as 1 ms in a separate experiment. Furthermore, a simulation tick length of 10 ms was used to improve the simulation precision (see discussion on Section 6.5.6).

Figure 6.14 shows the simulated latency and throughput were very accurate and precise. The latency error was less than 7% up to 27500 events/second. At 30000 events/second, the Storm query started to overload and the error increased, but the *CEPSim* results remained




(a) Latency. (b) Throughput.

Figure 6.14: Metrics estimation results - networked query $q_1$.

within the confidence interval. Moreover, *CEPSim* estimated the maximum throughput as approximately 34000 events/second, which is very close to Storm's overload point.

### Multiple Queries

This experiment analyzed *CEPSim*'s behaviour when simulating multiple queries running concurrently. To do so, first a Storm cluster was created at the Amazon EC2 service [10]. The setup was similar to the one presented in Table 6.1, but all VMs were configured as instances of the *m4.large* type (2 vCPUs and 8GB of RAM).

Four placements were then compared in a scenario where four copies of query $q_1$ were simultaneously run:

1. $Placement_1$: one VM, with all four queries placed on it;
2. $Placement_2$: two VMs, with two queries placed on each;
3. $Placement_3$: two VMs, with all four instances of *DBConsumerBolt* placed on one VM and the remaining bolts on the other;
4. $Placement_4$: four VMs, with one query placed on each.

To avoid possible bottlenecks in the database server, *DBConsumerBolt* was replaced by a mock implementation which does not access the database, but spins in a busy loop for 4.5 ms (the average time spent to process a single event, as measured by the methodology described in Section 6.5.3).

Table 6.4 presents the average latency of all four queries for both Apache Storm and *CEPSim* and for all four placements. The *CEPSim* column also shows the relative estimation error. Each query was set up to process 10,000 events/second. The throughput metric has been omitted from the table because it was correctly measured as 10,000 events/second in all scenarios.

Table 6.4: Multiple queries experiment - Latency measurements (in ms).

| | **Apache Storm** | **CEPSim** | |
|---|---|---|---|
| $Placement_1$ | 12921.11 | 13234.15 | +2.42% |
| $Placement_2$ | 9840.91 | 10117.70 | +2.81% |
| $Placement_3$ | 12575.42 | 12030.00 | -4.33% |
| $Placement_4$ | 9795.91 | 10061.83 | +2.71% |

The results from this experiment demonstrated that *CEPSim* can accurately simulate multiple queries running on the same VM and can be used to analyze different placement strategies. For instance, the experiment showed that running two instances of query $q_1$ on the same VM does not greatly affect their performance, as illustrated by the small latency increase from $Placement_4$ to $Placement_2$. It is also clear from $Placement_1$'s latency that placing four queries on the same VM can overload it and may not be a good option depending on the users' QoS requirements.

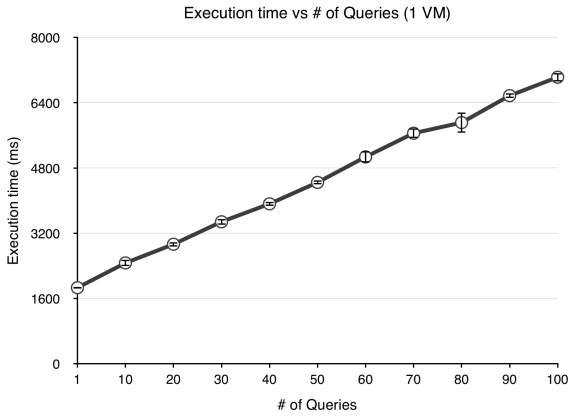
### 6.5.5 CPU and Memory Overhead

This section presents two experiments that measured the execution time and memory consumption of *CEPSim* simulations.

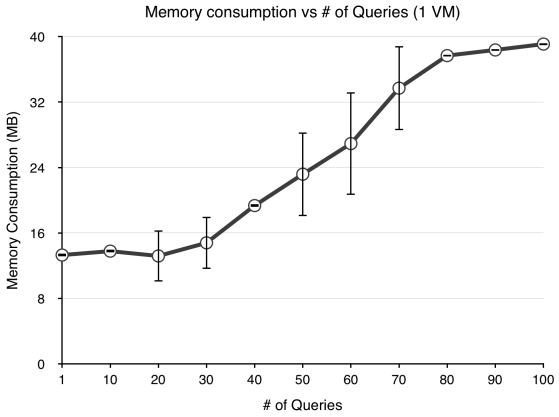
Figures 6.15a and 6.15b depict the results from the first experiment. This experiment simulated a single VM running $n$ instances of query $q_2$ from Fig. 6.10b. The simulation time was set to 5 minutes and each query processed 100 events/second. For each value of $n$, the simulation was executed 10 times and the execution time and memory consumption were recorded. The graphs show the average of these values alongside the 99% confidence interval. *CEPSim* was able to simulate 100 queries in approximately 7 seconds and using less than 40 MB of memory. Furthermore, both metrics grew sub-linearly as a function of the number of queries.

The results from the second experiment are shown in Figure 6.16. In this experiment, each VM ran a fixed number of queries, and the number of VMs in the datacentre was varied. The graphs show results for two different combinations. In the first, the number of queries per VM was set to 10 and the number of VMs varied from 10 to 1000; in the second, the number of queries per VM was set to 100 and the number of VMs varied from 1 to 100. Both combinations resulted in the same number of total queries, but enabled comparison of the effects of different query placements on *CEPSim* performance.

The results for the two combinations were very similar. The maximum simulation time was approximately 7 minutes for a total of 10000 queries, which translates to 1 million events per second. Less than one 1 GB of memory was needed to run this simulation. Once again, both execution time and memory consumption scaled sub-linearly. This same behaviour is expected

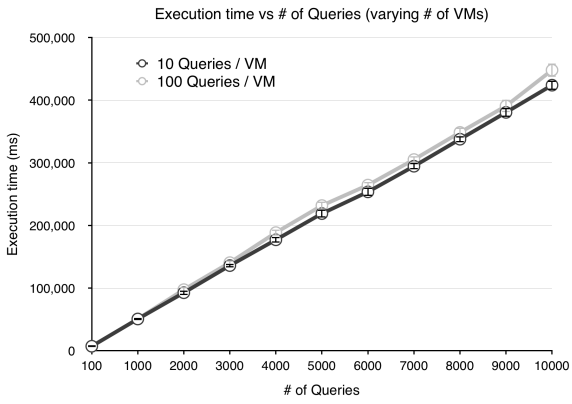



(a) Execution time.

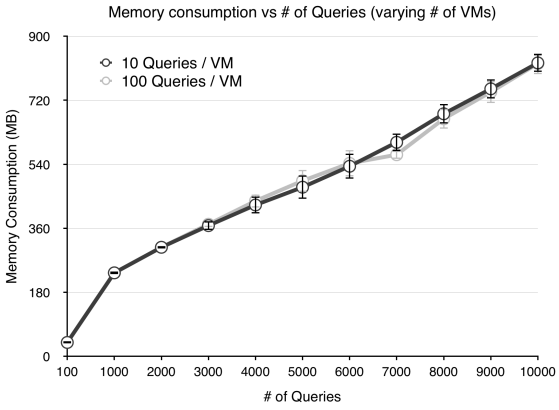



(b) Memory consumption.

Figure 6.15: Execution time and memory consumption - single VM.

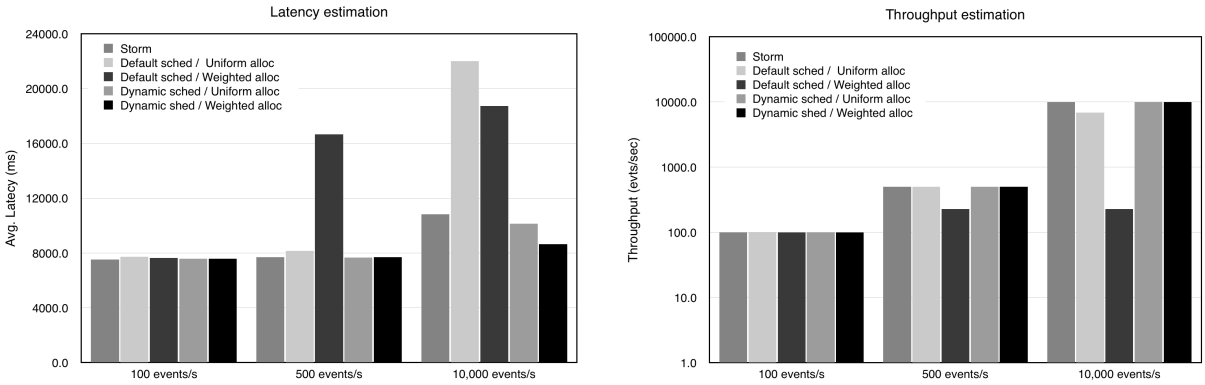



(a) Execution time.




(b) Memory consumption.

Figure 6.16: Execution time and memory consumption - multiple VMs.




(a) Latency. (b) Throughput.

Figure 6.17: Parameters experiments - Scheduling and allocation strategies.

as long as the available RAM is larger than the memory required by the simulation.

### 6.5.6 Simulation Parameters

The two experiments described in this section aimed to evaluate the effect of different parameters in the simulations. First, it was analyzed how scheduling and allocation strategies affect the simulation metrics estimation. In addition, the effects of simulation tick length on *CEPSim* was assessed.

#### Operator Scheduling

To analyze the effects of operator scheduling strategies, query $q_1$ latency and throughput were estimated using the default and dynamic scheduling strategies combined with the uniform and weighted allocation strategies. Figure 6.17 summarizes the results obtained when query input rate was configured to 100, 500, and 10000 events/second.

When the default scheduling strategy was used in high input rate scenarios, the throughput was considerably underestimated and the latency overestimated. This occurred mainly because *DBConsumerBolt* was scheduled at every simulation tick, even though it receives events only when its predecessor *ReadingAverageBolt* window closes. This problem was even more pronounced when weighted allocation was used. In this case, the number of instructions that *DBConsumerBolt* received was proportional to its *ipe*, which is much higher than the other operators' *ipe*s.

When using dynamic scheduling strategy, *CEPSim* better approximated Apache Storm's results in all scenarios. Nevertheless, when used with weighted allocation, dynamic scheduling underestimated the average latency in the 10000 events/second case. In this combination, the

Table 6.5: Parameters experiments - Simulation tick length.

| Query | Tick length (ms) | Latency (ms) | | Execution time (ms) |
|---|---|---|---|---|
| Local | 10 | 10078.43 | - | 63400.43 |
| | 100 | 10140.53 | 0.62% | 9645.23 |
| | 1000 | 10415.02 | 3.34% | 2811.64 |
| Networked | 10 | 9730.00 | - | 66385.91 |
| | 100 | 9865.01 | 2.78% | 10058.58 |
| | 1000 | 12636.47 | 29.87% | 3106.90 |

dynamic strategy prioritized *DBConsumerBolt* whenever there are events on its input queues, resulting in lower latency at the cost of lower maximum throughput.

#### Simulation Tick Length

To evaluate the effects of simulation tick length on *CEPSim*, query $q1$ was simulated using different simulation tick lengths in both local and networked cases. The results are summarized in Table 6.5. The latency column shows the metric values estimated by the simulation. The execution time column displays the average of 10 simulations, each one including 100 instances of the query running for 5 minutes.

The results show that the simulation tick length enables users to adjust the trade-off between precision and computational cost. A longer tick introduced estimation error for both scenarios, but the execution time was significantly reduced. The error was more pronounced in the networked query case because of the way network communication is implemented in *CEPSim*: if a message is sent to a placement that has already been scheduled, then the message will be processed on the next simulation tick only.

### 6.5.7 Discussion

The experimental results described in this section showed that *CEPSim* can effectively model real CEP queries and simulate them in a cloud environment. Execution time measurements also demonstrated that *CEPSim* has excellent performance, being able to simulate 100 queries running for 5 minutes in 7 seconds only.

One of the main *CEPSim* use cases is to understand query behaviour at various input event rates. The experiments described in Section 6.5.4 showed that this study can be performed using *CEPSim* with relatively good accuracy and precision for both distributed and non-distributed queries and for both high and low input rate scenarios.

As another important use case, the experiments described in Section 6.5.4 - “Multiple

queries" showed that *CEPSim* can also be used to simulate multiple queries running on the same VM. The latency estimation error was kept fairly low during the experiment and enabled comparison of different operator placement strategies.

The limitations showed by *CEPSim* to simulate query $q_1$ at the maximum input rate highlighted the difficulty of simulating a system in an overloaded state. Further analysis concluded that, at this point, most of the query latency consisted of I/O waiting time, as the *DBConsumerBolt* writes to the database every event it receives. In this situation, the operating system continues to schedule other threads and processes, which can continue to process events on their turns. *CEPSim* uses a simplified model in which operator latency is caused by processing time spent on CPU only. In addition, the metric calculation errors at high input rates were also caused by differences in the strategy adopted to control the query load: while *CEPSim* uses *backpressure*, Storm follows a *pull* strategy on which events are requested from the producer only when there is available space at the operator queues.

As a final observation, it is claimed that *CEPSim* can be efficiently used for Big Data simulations. Results from the experiments in Section 6.5.5 demonstrated that the simulator scales well and handles large numbers of queries with a small memory footprint. In addition, *CEPSim* customizability also enables the user to fine control the simulation by changing parameters such as the simulation tick length and scheduling strategy. Moreover, even though Storm has not been stressed at a larger scale, most experimental results are also applicable to these scenarios. This is true because, in practice, the distribution of Storm (and other CEP systems) queries is limited to a few nodes. In other words, distinct VMs usually run independent pieces of computation that can be simulated in isolation from others.

## 6.6 Summary

This chapter presented *CEPSim*, a simulator for cloud-based CEP systems. *CEPSim* can model different CEP systems by converting user queries to the *AGeCEP* representation. The modelled queries can be simulated on different environments, including private, public, hybrid, and multi-clouds. In addition, *CEPSim* also allows customization of operator placement and scheduling strategies, as well as the queue size and data generation functions used during simulation.

Experimental results have shown that *CEPSim* can simulate a large number of queries running on a large number of virtual machines within a reasonable time and with a very small memory footprint. Furthermore, the experiments also demonstrated that *CEPSim* can model a real CEP system (Apache Storm) with good accuracy and precision. Together, these results validated *CEPSim* as an effective tool for simulation of cloud-based CEP systems in Big Data

scenarios.

By using *CEPSim*, architects and researchers can quickly experiment with different configurations and query processing strategies and analyze the performance and scalability of CEP systems. Hopefully, the availability of a simulator may also encourage research in this field.

The next chapter presents the last contribution of this research: the design and prototype implementation of the *CEPaaS* system.

# Chapter 7

# Complex Event Processing as a Service

This chapter presents the design and implementation of a Complex Event Processing as a Service (*CEPaaS*) system. First, the motivation and goals of this system are discussed in Section 7.1, followed by an overview of its architectural features in Section 7.2. Next, the system architecture, design and implementation are detailed in Sections 7.3, 7.4 and 7.5. Finally, the system is evaluated regarding its processing latency and fault tolerance in Section 7.6.

## 7.1 Motivation

Despite a recent surge of interest in CEP motivated by its use in Big Data scenarios, today the CEP market is still dominated by a few proprietary solutions [86, 123, 139] that require large investments for their acquisition, but are still not as flexible as desired. Alternatively, on the other side of the spectrum, many companies adopt open-source, low-level systems [17, 18, 153] whose deployment demands intense technical training and high operating costs.

To address these problems, this research proposes the creation of a *CEP as a Service* (*CEPaaS*) system to enable the offering of CEP functionalities in the cloud services model. This model brings many advantages to the system users, such as no up-front investment, low maintenance cost, and ubiquitous access via the Internet.

Nevertheless, offering such a service involves many challenges, which is reflected in the limited number of similar services today. First, low latency is essential to many CEP use cases, but is difficult to achieve in a service environment because there is no control over the locations of event sources and consumers. In addition, some use cases impose an unpredictable and variable load over the system, requiring the implementation of elasticity capabilities in the system.

Moreover, *CEPaaS* is inherently multi-tenancy, which also brings many implications to the system architecture and design. For instance, a multi-tenancy system has to have high

availability because an outage affects many customers and can seriously damage the service provider reputation. It is also necessary to control the resource usage of user queries and guarantee their isolation so that they do not interfere with each other.

By offering it to anyone with Internet access, the system is expected to scale primarily in the number of queries rather than in the input event rate of a small number of queries. Finally, by targeting such a wide spectrum of users, the system must be usable by non-specialists, but at the same time should not prohibit the definition of custom processing logic by advanced users.

The next sections discuss in detail the architecture, design and implementation of a *CEPaaS* system that aim to solve the mentioned challenges.

## 7.2 System Overview

To handle the challenges associated with offering CEP as a managed service, the *CEPaaS* system is built on three main pillars: a *multi-cloud architecture*, *container management systems* (CMS), and an *extensible multi-tenant design*. The first two are leveraged at the architectural level to provide a scalable and fault-tolerant runtime environment for queries. The third provides a novel design in which the system applicability increases with the number of users.

Figure 7.1 shows an overview of the system architecture. The figure depicts one primary and two secondary *deployments* of the system, each one running in a different cloud. In this context, *cloud* is loosely defined as a cluster of servers offered by a cloud provider that are connected via a high speed network and are geographically close to each other. In terms of Amazon's and Google's nomenclature, this definition implies that the servers from a cloud are running on the same *region* or *zone*.

This architecture is not strictly compliant with the multi-cloud definition provided in Section 2.2.3, which requires clouds managed by different providers. The *CEPaaS* system, on the other hand, only demands clouds that are physically apart. Note, however, that this less architecture already brings the two most important advantages of multi-cloud to the *CEPaaS* system. First, it increases system availability, as it is possible to continue to process user queries even if an entire cloud goes off-line. Second, it enables exploration of the geographical diversity of clouds, creating the possibility of a strategic deployment in which system resources are positioned close to event sources and consumers.

It is important to emphasize the architecture does not need to be modified whether the clouds are managed by different providers or not. In both cases, all three deployments from the figure contain a set of *system components* that are required for running user queries. The primary deployment also hosts components used for user interaction. Note that the number of secondary deployments is not fixed and depends on the quality of the service that the provider

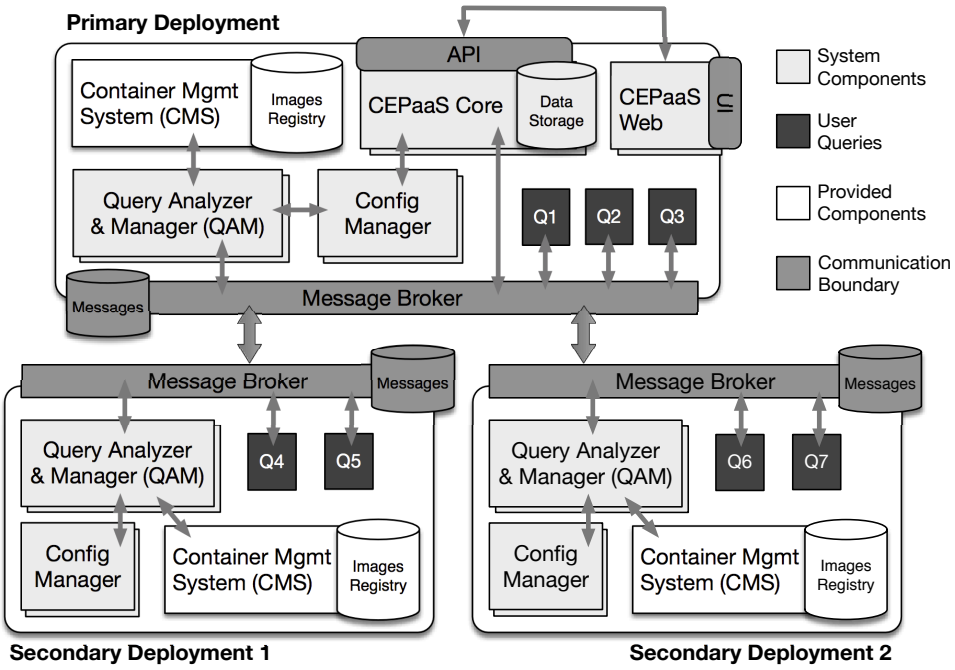



Figure 7.1: *CEPaaS* system architecture.

wants to offer.

Another important aspect of the *CEPaaS* architecture is that every deployment is managed by a CMS, which is either provided as a managed service, such as Amazon Container Service [11] and Google Container Engine [60], or is pre-installed in the cloud servers. By encapsulating every system component as an *application container* it is possible to isolate and control their resource usage. This encapsulation also facilitates and encourages independent upgrade of system functionalities. These benefits are similar to the ones brought by VMs, yet with less execution overhead and more efficient usage of resources (Section 2.3). Moreover, the infrastructure provided by a CMS guarantees that all containers are constantly running, which drastically simplifies the implementation of fault-tolerance in the system.

It is important to note that even user queries are executed as application containers in *CEPaaS*. This is a very important design decision that brings two additional benefits to the system. First, scalability in the number of queries is naturally handled as new query containers are created and scheduled by the CMS. Second, because queries have different resource requirements and workload profiles, an intelligent scheduling strategy can significantly increase the utilization level of the cloud servers.

On top of this architecture, the *CEPaaS* system adopts an extensible multi-tenant design based on a *query template* mechanism that relieves users from learning query definition languages. In the *CEPaaS* system, queries are created by simply instantiating query templates.

In addition, advanced users can still create new query templates based on a library of *operator templates* or create new operator templates based on a Java API. Finally, because query and vertex templates can be shared among customers, this design promotes a strong library of operators and queries that is maintained and reinforced by the users themselves.

## 7.3 System Architecture

The *CEPaaS* system architecture is based on one or more deployments, which run in clusters of servers in multiple cloud environments. All *CEPaaS* deployments are managed by a CMS and contain three system components necessary for running user queries: the *Config Manager*, which maintains the system configuration; the *Message Broker*, which functions as the system communication hub; and the *Query Analyzer and Manager* (QAM), which is responsible for managing the lifecycle of user queries. The primary deployment also hosts the *CEPaaS Core*, *CEPaaS Web* and *Data Storage* components, which together provide an API and a Web Application that can be used to interact with the *CEPaaS* system. In the following subsections, these components and their interactions are discussed in detail.

### 7.3.1 Container Management System

*CEPaaS* architecture assumes that every cloud used by the system is controlled by a CMS. From the architectural standpoint no specific CMS implementation is required as long as it provides all functionalities expected. In theory, even different implementations could be used simultaneously, only requiring the adaptation of the pieces of code and scripts that interact directly with the CMS.

The following list summarizes the features that must be provided by the CMS:

- *Docker support*: every *CEPaaS* component has an associated Docker [114] image, and the CMS must support the creation of multiple containers based on this image.

- *Container scheduling*: once a container is created, it must be automatically scheduled in one of the cloud servers.

- *Container fault-tolerance*: a system administrator specifies the number of replicas for each container type, and the CMS must guarantee that this number is respected. In other words, the CMS must automatically detect failed containers and restart them if needed. In *CEPaaS*, every system component is replicated at least twice to guarantee high-availability of the system.

- *Dynamic attachment of volumes*: the CMS must be capable of associating a container to a data volume and of automatically attaching it to the server in which the container is scheduled. This attachment must be dynamic because the server in which a container runs is unknown beforehand. In *CEPaaS*, the *Config Manager*, *Data Storage*, and *Message Broker* components require persistent data and, therefore, use this feature.

- *Automatic load balancing*: when multiple replicas of the same container are created, the CMS must support the creation of a *service* associated with an external IP and must automatically forward requests sent to this IP to the container replicas. This is necessary to guarantee that the *CEPaaS Core* and *Web* components are externally accessible and have a load balancing mechanism in place.

- *API access*: the CMS must provide an API that other pieces of software can use to control and monitor the runtime environment. In particular, the *QAM* component requests the creation and removal of query containers, and both *QAM* and *CEPaaS Core* examine the runtime status of existing query containers.

These other features are also expected even though they are not essential:

- *Image registry*: ideally, the CMS should have a local registry that can be used to store the container images used by the system. If a registry is not available, public registries can be used. In this case, however, the CMS must have Internet access as a way to contact them.

- *Advanced scheduling capabilities*: a good scheduling strategy should guarantee that the servers have enough resources to run all containers allocated to them and the load of all servers is relatively balanced. At the same time, the scheduler should maintain the utilization level of all servers as high as possible to reduce the cost of running the system. Finally, the scheduler should allocate replicas from the same container in different servers in order to improve the availability of each service.

Note that the interaction with the CMS takes place at two distinct moments. The first moment is when a new *CEPaaS* deployment is created and all system components must be set up. This step is usually performed by a system administrator and results in the creation of the system component containers. After a deployment is created, the *QAM* and *CEPaaS Core* components constantly interact with the CMS to create new queries and poll for their status.

Currently, the Kubernetes [63] system described in Section 2.3.2 is the only CMS which provides all functionalities required by *CEPaaS*, and, therefore, it was adopted in this research.

### 7.3.2 Message Broker

The *Message Broker* component is the communication hub of the *CEPaaS* system. Its main purpose is to decouple producers and consumers of messages, and to guarantee that messages are delivered to their intended destinations even in the case of failures. Currently, the *Message Broker* is used for the following:

- *QAM monitored events*: the QAM component uses the *Message Broker* as the source of its input events. These events include user requests to create and remove queries, and also monitoring information about the status of running queries and servers.
- *Queries input events*: all events processed by *CEPaaS* queries are sent from the event producers to the *Message Broker*, and then processed from there. Most event producers are external to *CEPaaS* and, therefore, it is fundamental to decouple them from system internals.

Every *CEPaaS* deployment has a cluster of message brokers that serve as destination of all messages addressed to its local components. Similarly to CMS, *CEPaaS* can use different implementations of message brokers as long as they provide all guarantees required by *CEPaaS*. More specifically, the message broker is expected to be fault-tolerant, to handle a high volume of events, and to provide some mechanism that enables parallel consumption of events from the same topic. Currently, the *CEPaaS* system uses Apache Kafka [94] as the *Message Broker*.

Figure 7.2 shows an overview of Apache Kafka architecture. In Kafka, messages are grouped into *topics* and each topic is divided into a number of *partitions*. Each partition, in turn, contains a sequence of messages and is internally structured as an append-only log file. To guarantee durability and fault-tolerance, Kafka replicates every partition to other nodes of the cluster. The number of replicas is configurable, and in *CEPaaS* this number is currently set to three. In Figure 7.2, both topics are illustrated with three partitions ($P_0$, $P_1$ and $P_2$) and two replicas for each partition.

When messages are sent to a topic, the client application determines a target partition based on a hash value of key attributes or some customized logic. To read messages from the topic, a consumer application can subscribe to a specific topic and set of partitions to receive the respective messages. For each partition, consumers read messages from the *partition leaders* only, which are shown in bold in Figure 7.2. Although it is guaranteed that messages from a single partition are delivered in an ordered fashion, the same does not apply between messages from different partitions and topics.

Alternatively, one or more consumers can form a *consumer group* and subscribe to the entire topic. In this case, Kafka automatically divides the partitions and assigns them to consumers belonging to the same group. Furthermore, Kafka also monitors this assignment and

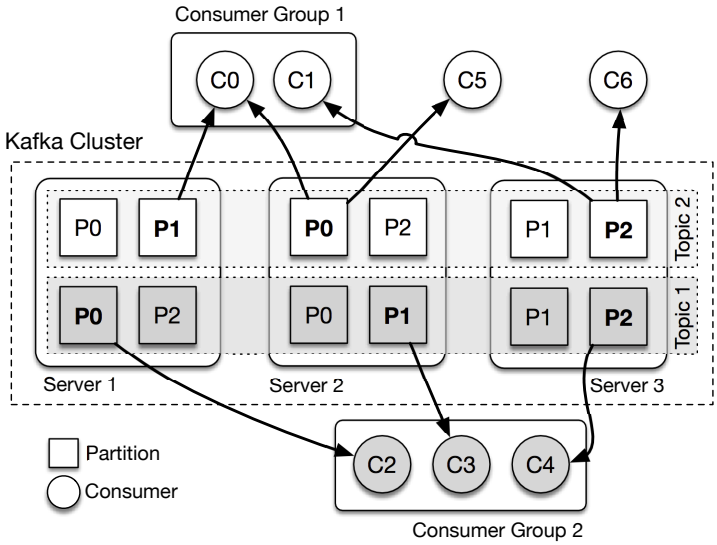



Figure 7.2: Apache Kafka architecture.

re-evaluates it in case new consumers are added or removed. This situation is illustrated in Figure 7.2 by consumer groups 1 and 2. The topic partitions are evenly divided between the consumers from the group. This is an important feature because it facilitates parallel consumption of events from the same topic.

### 7.3.3 CEPaaS Core / Web / Data Storage

The *CEPaaS Core* component implements the main API used for interaction with the *CEPaaS* system. For example, this API is used to create new event producers and queries and to remove running queries.

The API is provided as a REST interface and the data exchange format is JSON. Internally, *CEPaaS Core* is implemented using Play! Framework[1], and most of the application state is kept in the *Data Storage* component. For instance, this storage contains information about the system users, available operator templates, and the list of created queries. Runtime information, such as the status of a query, is directly obtained from the CMS and from the query containers. A detailed description of the API is provided in Appendix C.

The *Core* and *Data Storage* components are only available at the *primary deployment*. Therefore, if this deployment is off-line, all services provided by the API are not accessible. This is a current limitation of the *CEPaaS* architecture that exists because it is difficult to maintain *Data Storage* replicas consistently synchronized, especially when they are separated by a high-latency WAN network and can be concurrently modified. *CEPaaS* currently uses

[1]https://www.playframework.com

MongoDB[2] as storage, which does not support this scenario. In future work, this limitation will be lifted.

The *CEPaaS Web* component, on the other hand, is a traditional Web application that acts as the *Core* frontend and complement of the API. It is implemented using traditional Web technologies HTML 5, CSS 3, Bootstrap[3] for layout, and jQuery[4] library to facilitate JavaScript programming. All communication between *Web* and *Core* is performed via the REST interface.

### 7.3.4 Config Manager

The *Config Manager* is responsible for all configuration information needed by the system, such as the IP address of messages brokers, the name of topics used to receive data from external agents, and others.

The decision to store this information logically separated from the *Data Storage* has been taken because it is important to have configuration data always available and strongly consistent. Because these data rarely change and their size are much smaller than the size of application data, it is possible to use a storage solution that has trade-offs and consistency guarantees distinct from the *Data Storage*. In addition, systems that are commonly used to store configuration data, such as Zookeeper [83] and etcd[5] also provide extra functionalities that can be applied to coordinate distributed components.

The current *CEPaaS* prototype maintains the configuration data in a MongoDB database, which is also used as the *Data Storage* component in primary deployments. The configuration data is manually synchronized between the MongoDB instances whenever they change. The CMS monitors these instances and restarts them as soon as a failure is detected, which reduces the down time in case of failures. In the future, MongoDB will be replaced by a specialized configuration manager with better support for data synchronization and distributed coordination.

### 7.3.5 Query Analyzer and Manager

QAM is the component responsible for managing the lifecycle of user queries and of the runtime environment. Every *CEPaaS* deployment has a set of QAM replicas that process events from the local deployment.

QAM implements an autonomic manager similar to the one described in Section 5.1. Its main responsibilities include:

[2]https://www.mongodb.com
[3]http://getbootstrap.com
[4]https://jquery.com
[5]https://coreos.com/etcd/

- *Create query*: requests to create new queries are sent by the *Core* component on behalf of the system users. When receiving such requests, QAM optimizes the query graph and initializes the *Message Broker* to receive events intended for the query.
- *Start query*: requests to start a query are also sent by the *Core* component on behalf of the system users. In this case, QAM resolves all query configurations, converts the queries to the *AGeCEP* format, and invokes the CMS to create new query containers.
- *Stop query*: when QAM receives a request to stop a query, it simply invokes the CMS to remove the corresponding query container.
- *Query monitoring*: monitoring information is sent periodically by the query containers to QAM containing metrics about query performance, such as the size of operator queues, processing latency, and throughput. This data is processed by self-management policies, which can react to the monitored data and execute reconfiguration actions to improve the query quality of service.

Details of these procedures are provided in Section 7.5.

## 7.4 System Design

The *CEPaaS* system is designed to be multi-tenant, flexible, and accessible for non-technical users. To enable this, it leverages the idea of query templates as pre-defined event processing recipes that can be customized and instantiated. A class diagram showing *CEPaaS* core concepts is shown in Figure 7.3.

### 7.4.1 Tenant

A *tenant* is a company, group, or a single individual that uses the system. In a scenario in which the *CEPaaS* system is offered as a paid service, the tenant is the entity that establishes a contract with the service provider.

Associated with each tenant there is a set of *users*, who are the ones who operate the system. A user has an email and password that are used to log into the *Web* application and to authorize API calls sent to the *Core* component. In addition, every tenant also registers a set of *event sources*, representing systems, devices, or any other source that produces events consumed by the tenant queries. For instance, a tenant that uses the system to monitor the resource consumption of buildings can register as event sources electricity meters, temperature sensors, or any other device that produces data and needs to be integrated into the system.

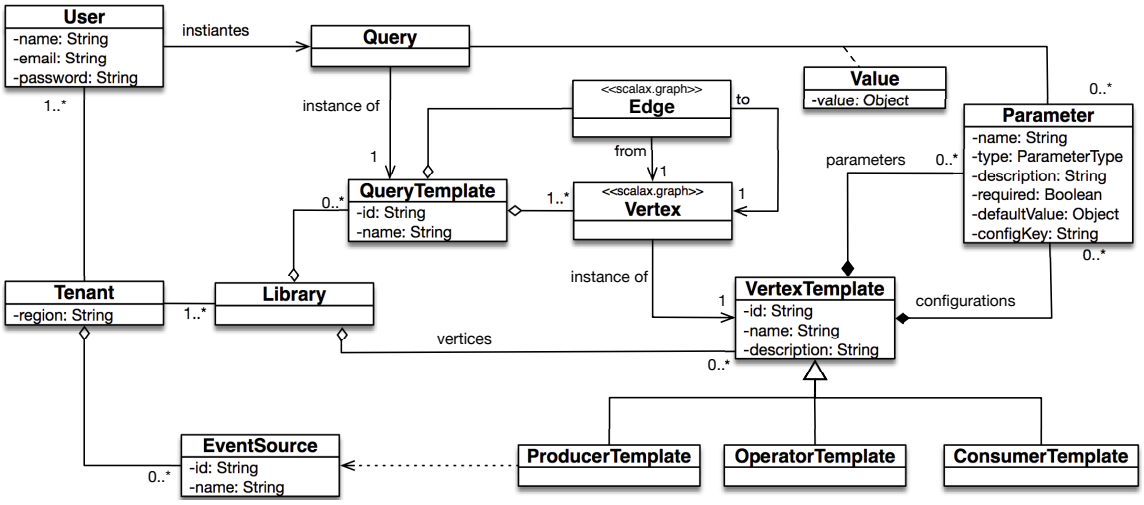



Figure 7.3: Class diagram of the *CEPaaS* system.

Finally, each tenant also has an associated *library* that contains the *vertex* and *query templates* that its users are allowed to use. Details about templates are discussed next.

### 7.4.2 Vertex Templates

*Vertex templates* constitute the *query template* building blocks. According to the role they have in a query, vertex templates are classified as *producer*, *operator*, or *consumer* templates.

A *producer template* represents a certain way to introduce events into the system. For instance, *CEPaaS* has a built-in "kafka" producer template that reads incoming events from the *Message Broker*. Nevertheless, other types of producer templates can also be used to read events from alternative sources such as Amazon Kinesis [12] and Google Pub/Sub [62].

An *operator template*, on the other hand, represents a piece of logic that can be reused in the context of event processing queries. In practice, every *operator template* instance is associated with a set of classes that implement the processing logic and are used at runtime. Details of this association are presented in Section 7.5.2. Currently, *CEPaaS* provides a set of built-in operator templates that can be used by any user with access to the system. In the future, users will also be able to create their own operator templates and (optionally) share with others.

Finally, *consumer templates* are used to represent actions executed on the query results. For instance, query results can be translated into alerts, such as email messages to system administrators, or can be published into the *CEPaaS Message Broker*. Like the other vertex template types, *CEPaaS* provides built-in implementations for the most common cases and additional ones can be added if needed.

Independently of its type, every vertex template is associated with a set of *parameters* and

*configurations*. Parameter values are used to customize template behaviour, and are set by the users. For example, the "filter" operator template has a parameter named "expression" that stores the boolean expression to be evaluated by the filter. Configuration values, on the other hand, are used to bind templates to the runtime infrastructure in which they are running. For instance, the "kafka" producer template has a configuration named "broker list" that contains the IP addresses of the local Apache Kafka brokers. Unlike parameters, configurations are not related to business logic and are automatically resolved by the *CEPaaS* infrastructure when a query is started.

### 7.4.3 Queries

Using the three types of vertex templates as building blocks, a *query template* is defined by connecting them into a coherent graph structure that implements some event processing logic. Similarly to vertex templates, users can also share query templates they created with other users. In this way, the system aims to build a strong collection of query templates that are sourced from its own user base.

There are only a few constraints about what constitutes a valid query template. First, the graph must be acyclic. Second, a query template is not allowed to have more than one producer template associated with the same event source. Finally, each query template can have only one consumer template. Note this constraint does not restrict the number of event consumers of a query: the consumer template simply represents an action that is executed on the query results. For instance, the "kafka" consumer template forwards the query results to the system *Message Broker* and, from there, they can be read from any number of other queries and external entities.

Moreover, it is important to note that a *query template* is not a runtime entity, but simply a description of a potentially reusable event processing logic. An actual running *query* is only created when a query template is *instantiated*. At this moment, the user must provide values for all vertex template parameters that do not have default values.

Figure 7.4 illustrates these concepts further. The far left of the figure shows a set of vertex templates belonging to the tenant library. In the centre, a selected set of these vertex templates are connected together to create a new query template in which a window operator groups events to calculate the average value of a certain attribute and a filter is executed over this average. For example, this query template can be used to detect sensor readings that are above or below a threshold.

The right part of the figure depicts two queries created from this query template. Note that for each query a different set of parameters is provided, which results in the same logic being executed in two different contexts. For instance, one query can be used to monitor the

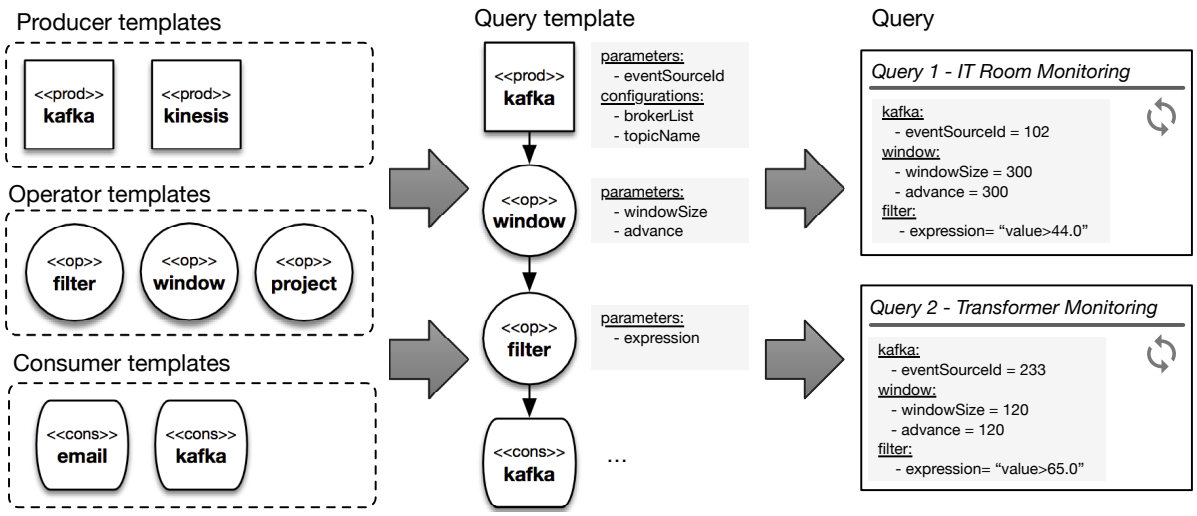



Figure 7.4: *CEPaaS* core concepts.

temperature of the IT room, whereas the other can verify a transformer temperature. Details of the query creation procedure are described in Section 7.5.4.

## 7.5 System Implementation

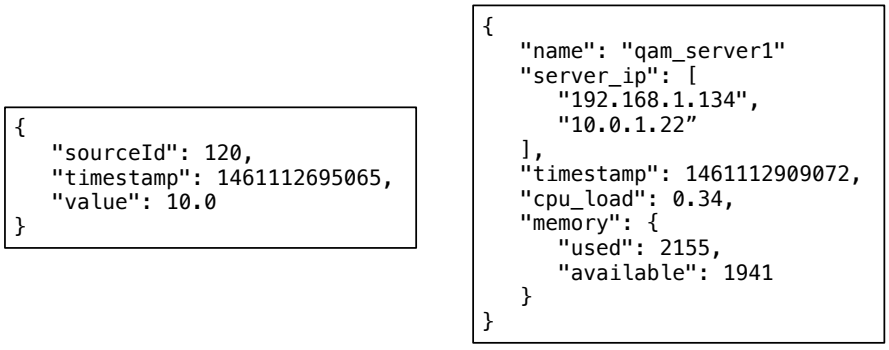
This section discusses implementation details of the *CEPaaS* system. It starts by examining how events are represented. Following that, it details how a vertex template is defined and presents the list of vertex templates currently built into the system. Finally, it describes the query execution engine including details of the transformation from query templates to containers and of the fault-tolerance guarantees provided by the system.

### 7.5.1 Events

In the *CEPaaS* system, events are defined as the computational representation of something that happened in the context of interest. Because *CEPaaS* is a multi-tenant system not tied to any specific domain, this definition is intentionally generic. For instance, depending on the domain to which the system is applied, an event can represent a sensor reading, the CPU consumption of a server, or the creation of a new user on a website.

Independently of their semantics, all events are represented as JSON documents [29]. JSON is a lightweight, structured representation of data that is used in most APIs available today, including in the *CEPaaS Core*.

Figure 7.5 depicts two events represented in JSON. The event in Figure 7.5a represents a sensor reading and contains three simple attributes. The event in Figure 7.5b also contains a




(a) Sensor reading. (b) Server monitoring information.

Figure 7.5: Events - JSON representation.

list attribute (*server_ip*) and a nested document (*memory*).

Because of the chosen representation, events in *CEPaaS* are also schema less. In other words, events are not associated with a formal description of its structure (attributes and datatypes). Therefore, event streams exchanged between operators are not "strongly typed" and the system does not check for output and input compatibility between operators when a query template is defined.

Because of this limitation, the user who instantiates a query is responsible for guaranteeing that the input events contain all data needed by the query template and the data has the correct datatypes. The disadvantage of this approach is that some errors are only detected at runtime, which complicates initial configuration and tests. Nevertheless, this also brings two advantages to the system. First, it reduces the runtime overhead because events are processed as they come without further checking. Second, it facilitates the creation of generic operators that can be used in a variety of query templates.

During processing, the processed events are encapsulated into objects called *tuples* and exchanged between query operators. Tuples keep additional information about events, such as the timestamp at which they entered the system and the sequence of operators that they have been through. This extra information is used to calculate operator performance attributes and the query quality of service.

### 7.5.2 Vertex Template Logic

In *CEPaaS*, every vertex template is defined by two main parts: the *metadata* and the *implementation*. Figure 7.6 shows the definition of the "filter" operator template.

The metadata is defined by a JSON document and includes basic information about the template, such as its name, description, and the set of parameters and configurations it uses. Moreover, the metadata also contains the template classification according to the *AGeCEP*

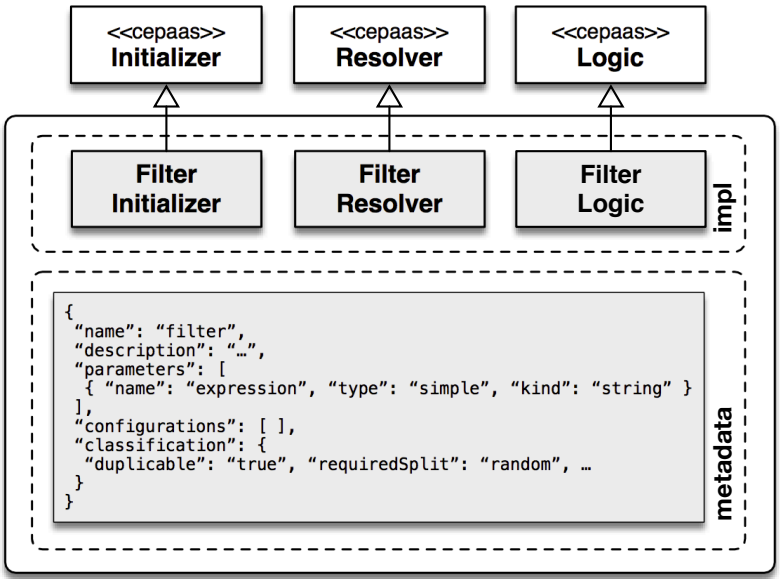



Figure 7.6: Vertex template definition.

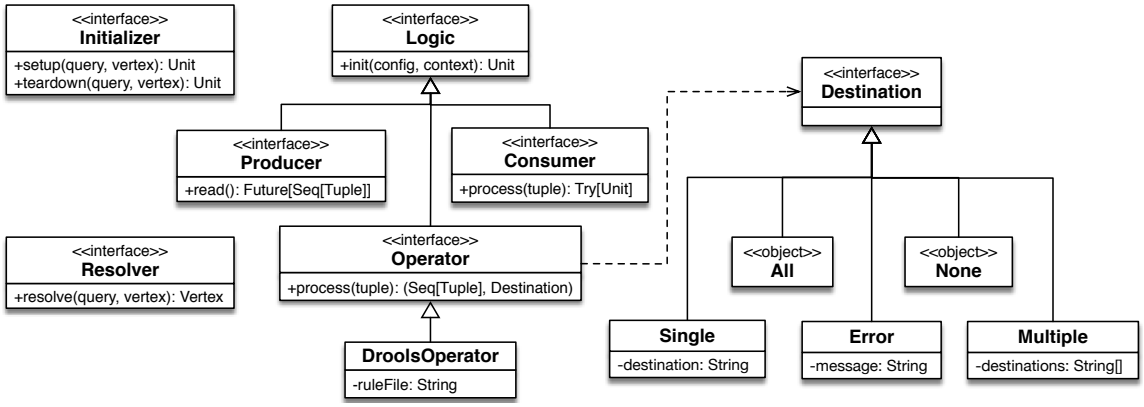



Figure 7.7: Vertex template definition interfaces.

criteria presented in Section 4.3. This classification is essential because it tells how an operator can be managed by the *CEPaaS* system. For instance, the duplicable criterion indicates if an operator can be duplicated to parallelize its execution. All these metadata are provided by the user who created the template and are stored in the *Data Storage* component.

The implementation part of a vertex template, on the other hand, is defined by a set of classes that implement three distinct interfaces: *Initializer*, *Resolver*, and *Logic*. The definitions of these interfaces are shown in Figure 7.7. Note that a vertex template does not implement *Logic* directly, but one of its sub-interfaces according to its type.

To show how each of the vertex template classes are employed, Figure 7.8 illustrates the typical states of a *CEPaaS* query and how the template classes are linked to query state transitions. When a query template is instantiated, a query is created and the *setup* method from

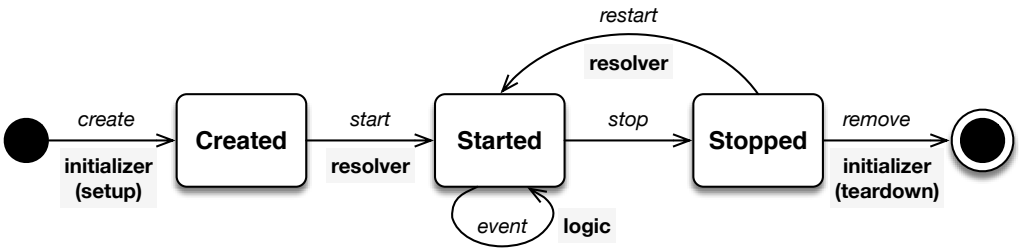



Figure 7.8: Query state machine diagram.

all vertex *initializers* used in the query is executed. The vertex initializers are responsible for preparing the environment for query execution by setting up any resource needed by the templates. For instance, during setup the "kafka" producer initializer checks in the *Config Manager* to learn whether the event source associated with it already has an allocated partition in the *Message Broker*. If it does not, then an allocation is created. Conversely, the initializers also have a *teardown* method which is run when a query is removed from the system.

The template *resolver*, on the other hand, is executed every time the query is started or restarted. Its main goal is to resolve all the vertex configurations by interacting with the *Config Manager*. For instance, in the "kafka" producer case, the resolver reads configurations about the brokers IP addresses, the tenant's topic name, and the event source partition.

Finally, the *logic* part is the one that defines the event processing logic itself. Note there is one specific interface for each vertex template type: *Producer*, *Operator*, and *Consumer*. They all inherit from a common interface *Logic* and therefore share the method *init*, which acts as the constructor of the *Logic* object. For example, in this method the "kafka" producer establishes connections with all Kafka brokers.

The main logic of a *Producer* is defined in the method *read*, which returns a future of a sequence of tuples. By using a future as the return type, *CEPaaS* enables the producer implementation to control when the method should return and, therefore, to regulate how data is transferred from the producer to the query. At runtime, every time a future is fulfilled the method *read* is invoked again to request the production of more tuples.

Analogously, an *Operator* main logic is implemented in the *process* method, which receives a tuple as input and returns a sequence of tuples and a destination to which these tuples must be sent. Note that different implementations of the *Destination* interface enable the operator to fine control the target of the tuples: to an *Error* queue, to a *Single* successor, to *Multiple* successors, to *All* successors, or to *None* of them. More details can be consulted in Appendix D, which contains a complete definition of the "filter" operator template.

The *CEPaaS* system also provides an alternative way to define operator templates by using the Drools Rule Language [89] (DRL) instead of Java code. All operator templates defined this

```
rule "too-high"
    when
        $n:  Number ($avg: doubleValue, $avg > ${upperLimit}) from accumulate(
           SensorReading(sourceId == ${sourceId}, $value : value) over window:time(${windowSize}s),
           average($value))
    then
        insert(new Alarm(${sourceId}, new Date(), "MEDIUM", "Value higher than threshold"))
end
```

Figure 7.9: A DRL rule definition.

way are associated with the same *DroolsOperator* class, which is configured to read a DRL file with the rule content.

Figure 7.9 shows a DRL example. Values specified in curly brackets are parameters of the operator template that must be specified by the user when instantiating a query template. DRL rules are declarative in nature and enable the specification of pattern-based conditions, including the use of quantifier operators and event windows. In this example, an alarm is created whenever the average value of a certain sensor over the last $windowSize$ seconds is higher than an $upperLimit$.

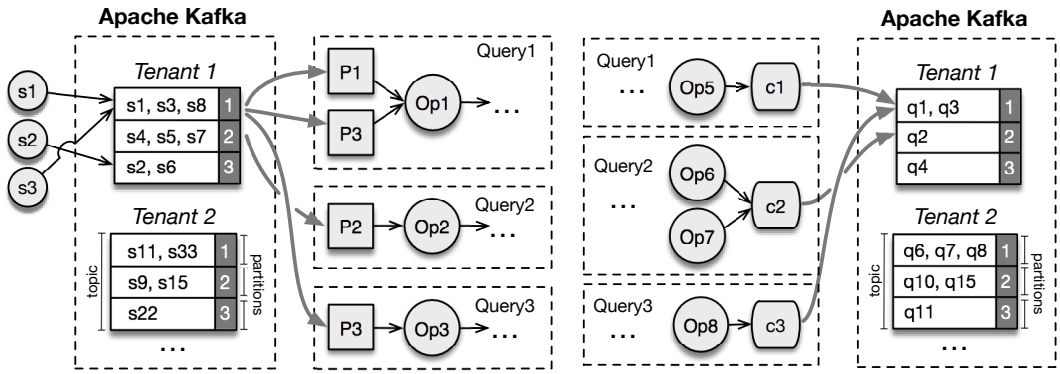
Finally, the *process* method contains the event processing logic for *Consumer*s. The method is similar to the *Operator*'s *read* method, but it simply returns a $Try$ object signaling the success or failure of the processing.

### 7.5.3 Built-In Templates

The *CEPaaS* system provides a set of built-in vertex templates that can be used by any tenant to define query templates. These built-in vertex templates provide common event processing logic that are reused and integrated with user-defined operators to implement the tenant's business requirements. The list of currently available built-in vertex templates are:

- *Kafka Producer*: reads incoming events of a specified event source from the built-in *Message Broker*. This producer assumes that each tenant has its own topic with a configurable number of partitions (by default 20), and the tenant event sources are evenly distributed among these partitions. Figure 7.10a shows a schematic of how topics and partitions are organized. Each partition of the tenants' topics receives data from many event sources. For instance, partition 1 from tenant 1 receives data from sources $s1$, $s3$, and $s8$.

  This design has been chosen over a design having one topic per event source to avoid the creation of too many topics in the broker. However, this design implies that a producer




(a) Producer. (b) Consumer.

Figure 7.10: Kafka Producer and Consumer.

may receive events from more than one event source when reading from a single partition. In this case, events that are not addressed to the producer are simply discarded. For instance, the producer $p3$ from Figure 7.10a receives data from $s1$, $s3$, and $s8$ but only forwards $s3$ events.

Moreover, this producer template can be configured to periodically *checkpoint* the offset of the last processed event into the *Messsage Broker*. By doing so, it can continue to process events from the last checkpoint after a restart. More details about this recovery mechanism are provided in Section 7.5.4.

- *Filter*: removes events that do not satisfy a boolean expression from the event stream. It supports expressions consisting of boolean, relational, and simple arithmetic operators.

- *Projection*: extracts a subset of attributes from the input events and creates new events in the output that contain only this subset.

- *Augmentation*: augments input events with new attributes that are derived from existing attribute values. It also supports boolean, relational, and simple arithmetic operators, in addition to string concatenation.

- *Windowing*: groups input events into windows, and emits tuples containing the grouped events. The current implementation is based on the window *size* and *advance* parameters, and is similar to the windowed operator described in Section 6.3.1.

- *Kafka Consumer*: sends the query results to the *Message Broker*. As in the producer template, each tenant also has a topic to store its resulting events. In this case, however, the events are partitioned per kafka consumer instance (and consequently, per query).

Figure 7.10b shows an schematic of how topics and partitions are organized for the consumers case. Each consumer sends data to a single partition of the tenant topic.

- *Email Consumer*: sends the query results to a configurable email address.

For all these built-in templates, the *Data Storage* is pre-populated with their corresponding metadata, and the operator logic classes are included in query execution engine binaries.

### 7.5.4 Query Execution Engine

*CEPaaS* query execution engine is built on top of Akka [6], a programming toolkit created to facilitate the implementation of distributed, scalable, and fault-tolerant applications. To achieve these goals, Akka applies the actor model of concurrent computation presented by Hewitt *et al*. [73].

The actor model is based on the principle that a program can be modelled as a set of *actors* that only communicate with each other through a set of asynchronous messages. When an actor receives a message, it can react by sending other messages, by designating the behaviour to be used in the next message, or by creating new actors. This relatively simple model relieves the programmer from dealing with thread management and locking issues because the actor state is only accessible by itself. In other words, if one actor needs to access another actor's state, it needs to explicitly request it with a message and the response is received later in another message. The concurrency in such a program primarily happens between actors processing their own set of messages.

The Akka toolkit adds more functionalities to this model and adapts it to distributed and fault-tolerant scenarios. In Akka, distribution is primarily achieved by making actors location-transparent. An actor communicates with others using *actor reference* objects, which can point either to local or to remote actors. From the sender point of view, communication with remote actors is indistinguishable from local communication. Fault tolerance, on the other hand, is based on a supervision hierarchy model in which an actor monitors its children actors for failures and decides what to do when such a failure is detected.

In runtime, every Akka actor is part of an *actor system*, which hosts a series of services that are shared among its actors (e.g., logging and configuration). Every actor system also has a *dispatcher* that manages actor scheduling and execution. In practice, a dispatcher is usually implemented via a thread pool. Finally, to exchange messages, every actor is associated with a mailbox that is used to hold messages destined for it. By default the mailbox is a simple unbounded queue, but other implementations are available and can be used to adapt the actor to different scenarios.

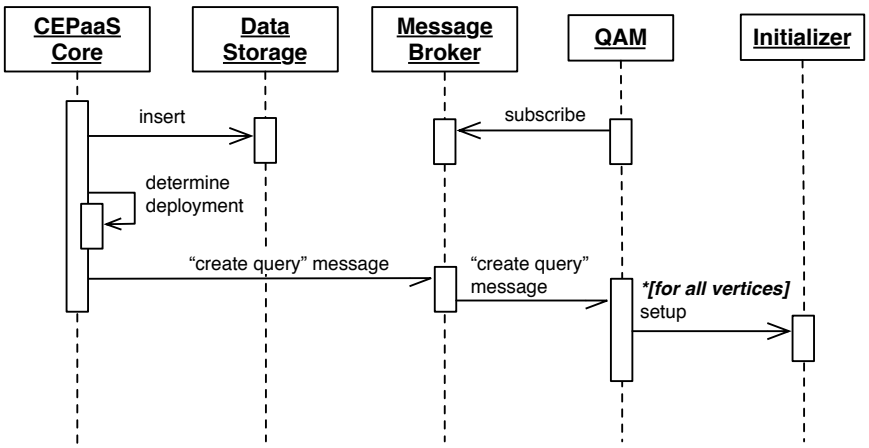



Figure 7.11: Sequence diagram - query creation.

The *CEPaaS* system leverages Akka functionalities to provide a robust and fault-tolerant environment for query execution. The following subsections discuss how a user query is created and transformed into Akka actors for execution.

### Query Creation

Figure 7.11 depicts a sequence diagram that details query creation in *CEPaaS*. When the *CEPaaS Core* component receives a creation request, it first inserts the new query into the *Data Storage*. Next, it determines in which *CEPaaS* deployment the query should run. Currently, each tenant is associated with a single deployment to which all its queries are allocated. Once determined, a "create query" message is sent to the *Message Broker* from that deployment. This message is eventually consumed by *QAM*, which invokes the *Initializer*s for all vertices and prepares the environment for the query execution.

### Query Start

Starting a query is a complex process that involves most *CEPaaS* components. To facilitate its understanding, this process is depicted in two sequence diagrams. The first, in Figure 7.12, shows it from the moment a request is received up to the query container creation. The second, depicted in Figure 7.13, illustrates how the Akka actors are created and mapped to a query.

When *CEPaaS Core* first receives a request to start a query, it determines in which deployment the query has been created and forwards the request to the corresponding *Message Broker*. This message is eventually consumed by QAM, which then executes the following steps:

1. Invokes the *Resolver*s for all vertices in the query. Here, most resolvers will interact with

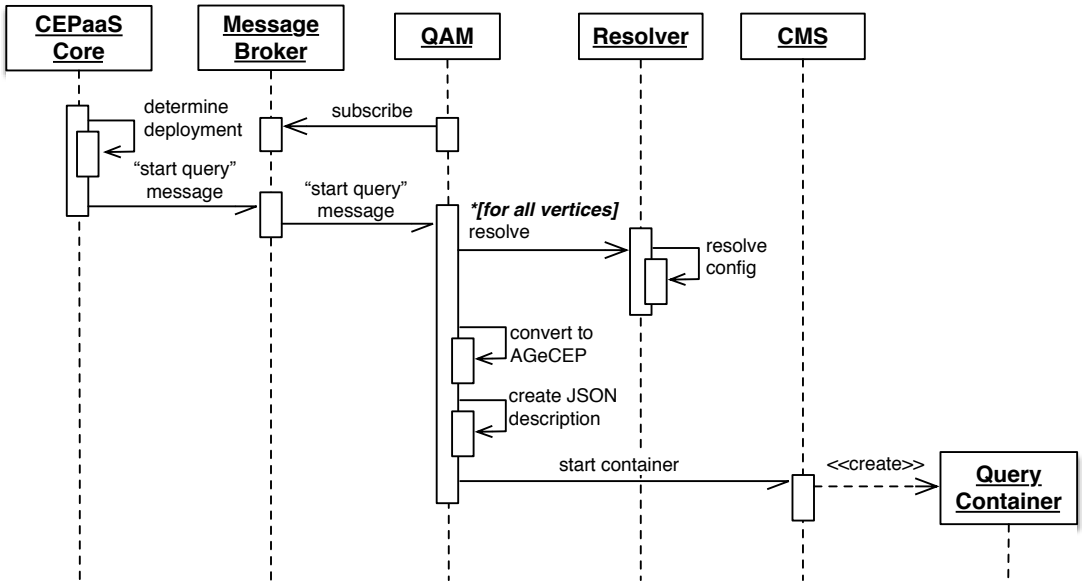



Figure 7.12: Sequence diagram - query start - part 1.

the *Config Manager* to determine its configuration.

2. Converts the query to the *AGeCEP* model. Note that for queries based on query templates the conversion to *AGeCEP* model is direct because the query template is a DAG and every vertex template is, by definition, already classified according to *AGeCEP* criteria. In the future, queries will be defined in other languages, and this step will normalize them to the universal *AGeCEP* representation.

3. Creates a JSON description of the *AGeCEP* query. This description contains all information needed to execute the query, including configuration values (determined in the previous step), name of vertex template implementation classes, and tenant details.

4. Requests the CMS to create a *query container*.

The *query container* receives as parameter the JSON document created in step 3 and passes it to the *main* method of the *query application*. In the *main* method, the application creates an actor system and a *query actor* to manage the execution of the *AGeCEP* query specified in the JSON. The query actor, in turn, iterates through all query vertices in backwards topological sort order and for each vertex creates the corresponding *Logic* class, invokes the *init* method, and creates a *vertex actor* to encapsulate the logic. Finally, a start message is sent to the *query actor* to signal the execution start.

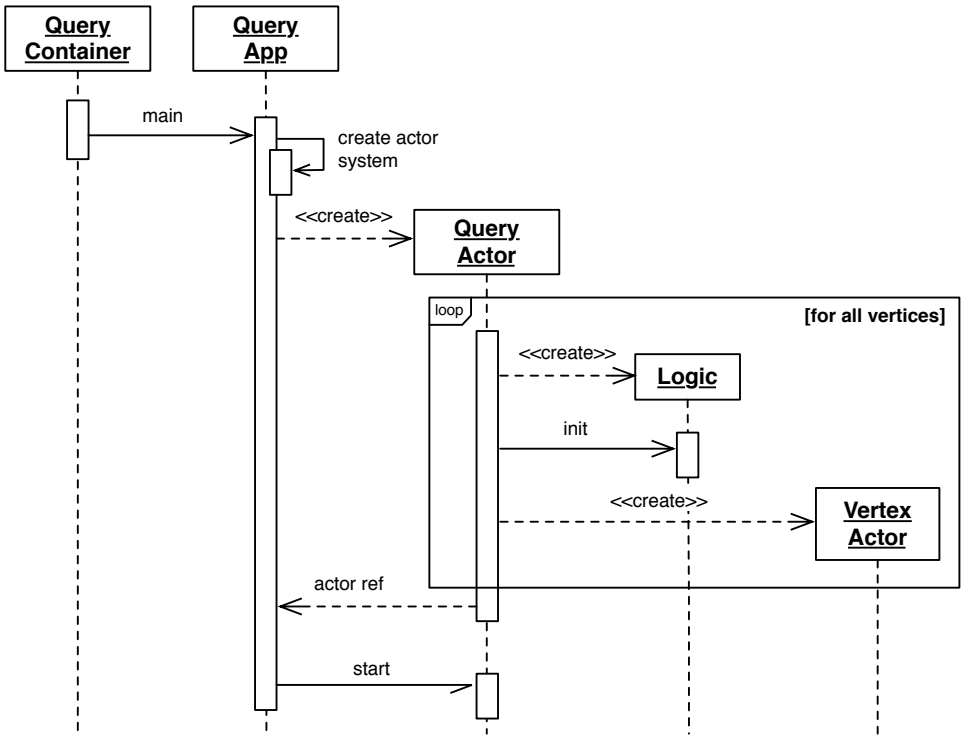



Figure 7.13: Sequence diagram - query start - part 2.

### Query Execution

Figure 7.14 illustrates how application containers and the Akka framework are used to run *AGeCEP* queries. Each query is encapsulated in a *Docker container* and is scheduled by the CMS. Inside the container, an Akka actor system runs on top of a Java Virtual Machine (JVM) and hosts all actors that are part of the query.

As explained in the previous section, every query has at least one corresponding *query actor* that is responsible for creating the vertex actors and for supervising them. The vertex actors, in turn, execute the event processing logic according to the encapsulated vertex:

1. *Producer actors*: on initialization, the actor invokes the *read* method from the encapsulated *Producer*. When the returned future completes, the actor forwards the produced events to its successor and invokes the *read* method again.

2. *Operator actors*: when an operator actor receives events from its predecessors, it forwards the events to the encapsulated operator logic and processes the results of this invocation. As explained in Section 7.5.2, the resulting events can be forwarded to one, multiple, or all successors. Alternatively, they can be discarded or sent to an error queue in case of problems.

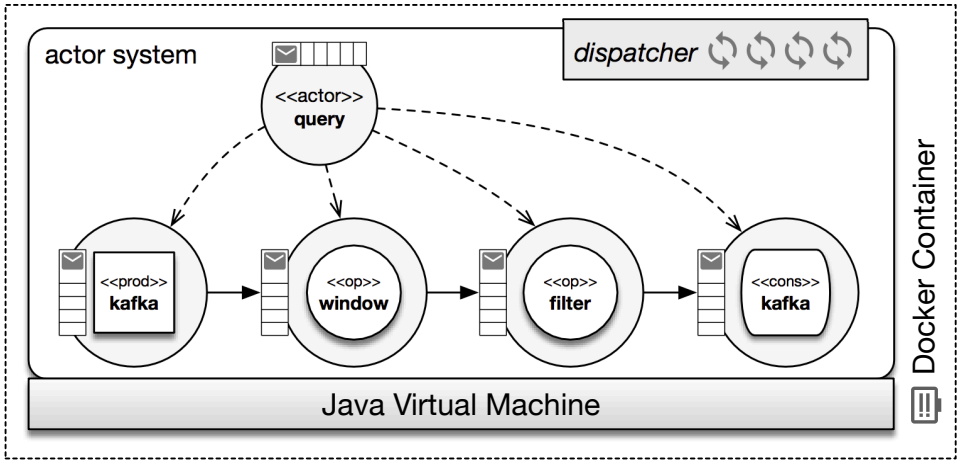



Figure 7.14: Runtime view of a query.

3. *Consumer actors*: the consumer actor works similarly to the operator actor: the events received are forwarded to the encapsulated consumer logic and the results are processed. Because consumer logic never produces new events, the only two possible outcomes are success, in which case the events are assumed to be successfully consumed, and failure, in which case the input events are sent to an error queue.

### Fault Tolerance

Fault tolerance of the *CEPaaS* queries is implemented by a two-level supervision hierarchy.

The first level occurs internally to the query container and is managed by the Akka toolkit. As mentioned in the "Query Execution" section, every query container has a query actor that creates the vertex actors and, as such, is responsible for supervising them. If a vertex actor fails, the error is detected by the supervisor, which restarts the failed actor. To avoid continuous restarts, the supervisor implements an exponential back-off algorithm that increases the time delay between restarts every time a child actor fails. In addition, if the restart delay reaches a configurable maximum (by default one minute) then the query actor itself (and the container) will also fail.

The second level of supervision is managed by the CMS. Every query container scheduled by the CMS is associated with a replication controller, which is responsible for maintaining one replica of the query container running. Therefore, if a server crashes, or a query container stops working, the replication controller detects it and requests the creation of a new container to replace it.

Because events sent by the external producers are persisted in the *Message Broker* before delivery, these producers are unaware of possible problems with their queries and no event is

lost in case of a query failure. Upon restart, the "kafka" producer retrieves from the *Message Broker* the last committed checkpoint and restarts processing events from this position. This mechanism guarantees that all events are *delivered* at-least-once to the query, but does not guarantee they are all processed. For instance, an already checkpointed event that is part of an aggregation window may be lost if a failure occurs. In addition, the calculated state of this window is also lost upon a failure and needs to be rebuilt starting from the next consumed message.

*CEPaaS* recovery mechanism, therefore, implements *gap recovery* and is similar to the *amnesia* approach presented by Hwang *et al*. [84]. Mechanisms that implement recovery with stronger guarantees have been thoroughly studied in the literature and will be incorporated to the *CEPaaS* system as future work [17, 53, 84].

#### Monitoring

In addition to the basic query creation and removal functionalities, the QAM component also monitors running queries to adjust them to changing conditions and to provide runtime information for the system users.

Monitoring information is collected periodically by the query actors and is sent to a special *Message Broker* topic. This information includes the number of events in the operator queues, total events processed, average query throughput, and average query latency. The monitoring topic is partitioned by the tenant ID and is read by a set of QAM replicas belonging to the same consumer group. As described previously, the *Message Broker* automatically distributes these partitions among consumers, enabling parallel consumption of events and significantly improving QAM scalability.

Upon receiving monitoring information, QAM executes the MAPE-K loop described in Section 5.1. First, query representations stored in an in-memory knowledge base are updated with the received information. This information is then fed to the *monitoring module* and to the set of self-management policies present in the KB. The execution of inference rules is managed by Jboss Drools [89]. Currently, only simple rules that detect whether a query is overloaded are executed. As future work, the policies described in Section 5.3 will be incorporated into QAM.

### 7.5.5 Limitations

The *CEPaaS* implementation presented in this chapter does not explore its multi-cloud architecture to the full extent. In particular, there are two main limitations that will be lifted in the future. First, the *CEPaaS* system currently does not tolerate the failure of an entire cloud

(deployment). Fault-tolerance at the cloud level, however, is essential for a cloud-based high-available system because no single provider can guarantee 100% availability. To provide such capabilities, the system will implement mechanisms to detect deployment failures and to move the execution of queries between deployments.

Second, the deployment at which queries execute is selected based on pre-defined configurations that associate each tenant with a deployment. Ideally, this selection should be dynamic based on the location of event producers and consumers. Even further, a query can be split into more than one container that are scheduled into different deployments. This dynamic scheduling can enable further reduction of the end-to-end query latency and provide an even better experience for the end user.

Another limitation of the *CEPaaS* system is the lack of native support for *enrichment* use cases, in which events are enriched with historical data previously stored in a database. Currently, the *Data Storage* is accessible only by *CEPaaS* components and cannot be read nor written by user queries. Therefore, it is the user responsibility to maintain a database in his or her own environment to be accessed by his or her queries. In the future, the *CEPaaS* system will provide a managed database that can be used for this purpose.

## 7.6 Evaluation

This section presents an evaluation of the *CEPaaS* system focused on two main aspects. First, a set of experiments were executed to measure the latency of *CEPaaS* queries and to assess the effects of the multi-cloud architecture in this latency. Second, another set of experiments validated the fault-tolerance mechanism provided by the CMS and estimated the recovery time of failed queries. The results presented here confirm that the proposed *CEPaaS* architecture can indeed be used to handle the challenges associated with offering CEP as a managed service.

### 7.6.1 Set-Up

For all experiments presented in this section, the Google Container Engine (GKE) [60] service was used to create two deployments of the *CEPaaS* system. The first deployment was created in the *us-central1* zone, located in Council Bluffs, Iowa, USA, whereas the second deployment was created in the *asia-east1* zone, located in Changhua County, Taiwan. GKE enables the creation of clusters of servers controlled by Kubernetes managed instances, which eliminates the need for installing and configuring it.

Each deployment was set up with five servers of type *n1-standard-4*, which has four virtual

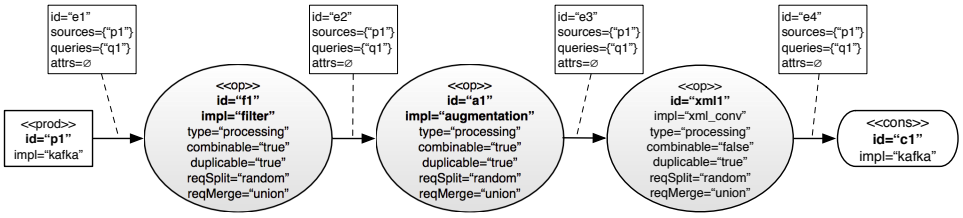



Figure 7.15: Query used in the *CEPaaS* experiments.

CPUs[6], 15 GB of memory and a 100 GB hard drive. From the five servers, three were reserved to run system components and the remaining two were used to run user queries. This separation was artificially implemented to isolate the queries from interference from other system components. Note that, in practice, this separation is not necessary and the CMS may place queries in any of the servers available in the cluster.

Figure 7.15 shows the *AGeCEP* representation of the query used in the experiments. This query has also been adapted from the Powersmiths WOW [127] system and aims to convert JSON requests, which are sent by temperature sensors, to the native WOW format (XML document). The query is composed of three main steps. First, a "filter" removes invalid temperature readings from the event stream. Second, an "augmentation" operator is used to include a new attribute in the event containing the temperature reading converted to Fahrenheit. Finally, the last operator creates a XML document based on the attributes included in the JSON. This is a user-defined operator created specifically for this experiment.

To measure the end-to-end query latency, the following procedure was executed: for each query, both a producer client and a consumer client were deployed in a single server. The producer sent events to the *CEPaaS* system at a specified rate, and every event sent was timestamped with the server local time. The consumer, in turn, read the query results from the *CEPaaS Message Broker*. For every event read, the consumer registered the receiving timestamp and compared it with the timestamp that was sent in order to obtain the latency. Note that because both producers and consumers were deployed in the same server, it was possible to avoid clock synchronization issues and to obtain more precise measurements.

Finally, all client servers were deployed in the Amazon EC2 [10] service and not in Google's cloud. This setup enabled measurements that are closer to a real user using the system via Internet because it avoids high-speed links that exist between datacentres from the same provider.

[6]A virtual CPU is implemented as a single hardware hyper-thread on a 2.6 GHz Intel Xeon E5 (Sandy Bridge), 2.5 GHz Intel Xeon E5 v2 (Ivy Bridge), or 2.3 GHz Intel Xeon E5 v3 (Haswell) [60]

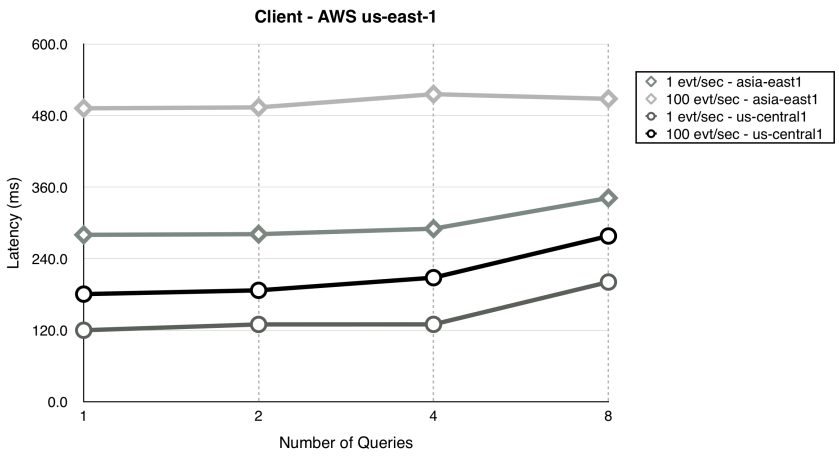



Figure 7.16: Query latency - 95% percentile - client in *us-east-1* region.

### 7.6.2 Latency Evaluation

To evaluate the end-to-end query latency and to measure the effects of the multi-cloud architecture in this latency, two experiments were executed.

In the first experiment, a client located in the *us-east-1* (Northern Virginia, USA) AWS region was used to access queries running in the *us-central1* and *asia-east1 CEPaaS* deployments. Figure 7.16 shows the results of this experiment. For each deployment, clients for 1, 2, 4, and 8 queries were run at 1 event/sec and 100 events/sec generation rate. Each measurement ran for 11 minutes, and the graphic shows the 95% percentile latency value for the last 10 minutes.

The results show that for both 1 and 100 events/sec the latency of the *asia-east1* deployment is larger than of the *us-central1*. The increase in latency ranges from 70% for the 8 queries and 1 event/sec case to 172% for the 1 query and 100 events/sec case. The results also demonstrate that increasing the number of queries up to 8 (4 queries in each server) has little effect on the latency (40% increase in the worst case). Finally, it is also possible to note that increasing the event generation rate naturally increases the 95% percentile latency value, because more events are generated and they spend more time in the system queues before being processed.

The second experiment executed is similar, but the client was located in the *asia-northeast-2* (Seoul, South Korea) AWS zone. The results for this experiment are shown in Figure 7.17. Once again, there is significant improvement for clients when they access nearby deployments (ranging from 60% to 70% in this experiment). An increase in latency at higher generation rates was also noticed.

The results obtained by these experiments conform to the hypothesis that geographical proximity translates to lower query latency. The experiments also validated the *CEPaaS* approach of leveraging multiple clouds to position system components closer to the clients.

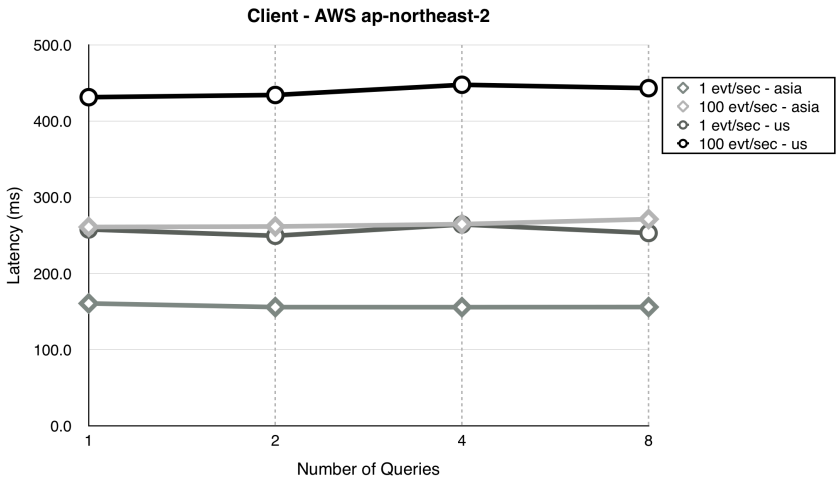



Figure 7.17: Query latency - 95% percentile - client in *asia-northeast-2* region.

### 7.6.3 Fault Tolerance

To test the fault tolerance mechanism provided by the *CEPaaS* system, two experiments were executed.

The goal of the first experiment was to assess the time required to recover from failures. To achieve this goal, an instance of the query shown in Figure 7.15 was created in the *us-central1* deployment. The query was configured to checkpoint the offset every five seconds. In addition, a client located in the *us-east-1* AWS region was created and configured to send events at 1 event/sec rate. At time $t = 30$, the query container was killed, which activated the fault tolerance mechanism. Because offset checkpointing was turned on, the query restarted its execution from the last committed offset.

Figure 7.18 shows the latency measured for each event sent during the experiment. The $y$ (latency) axis is in log scale. In general, the latency oscillates between 50 and 100 ms, and for only the four events between $t = 35$ to 39 the measured latency was higher. The maximum value of 4 seconds was measured at $t = 35$. This result shows that the query could quickly recover from the failure and return to its normal behaviour.

To assess the overhead of offset checkpointing, Figure 7.18 also shows the same experiment executed for a query without this mechanism activated. The measured latencies are similar in both cases, which shows that the overhead is small. In addition, the result also shows that the recovery time in both cases are similar, even though some messages are lost when offset checkpointing is disabled.

In the second experiment, a more complex fault tolerance scenario was also analyzed. In this scenario, four instances of the query in Figure 7.15 were created in the *us-central1* deployment and distributed between two servers. As in the first experiment, four clients were created

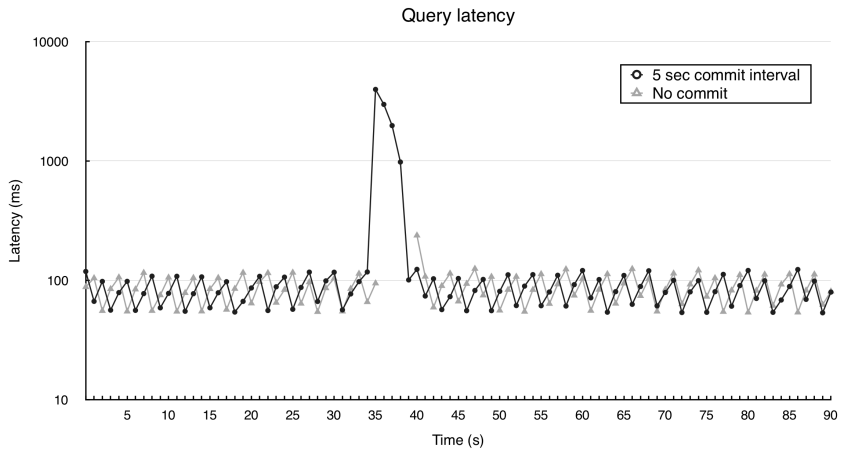



Figure 7.18: Query latency - fault tolerance experiment.

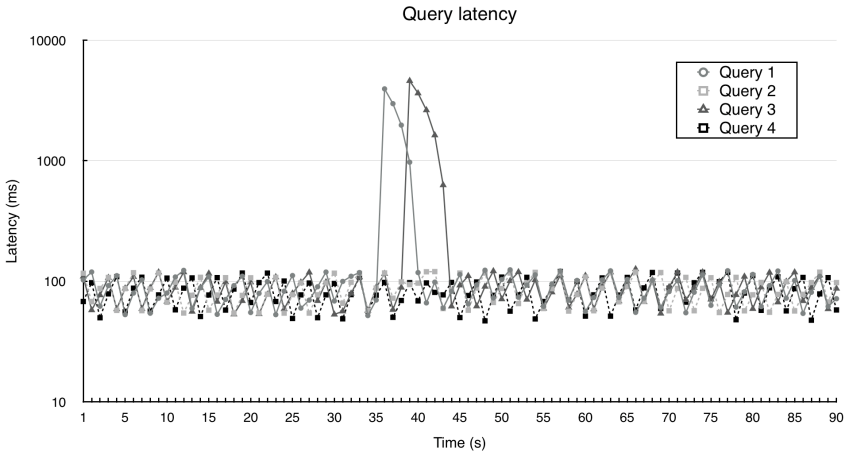



Figure 7.19: Query latency - fault tolerance experiment in a complex scenario.

in the *us-east-1* AWS region and configured to send events to the queries at 1 event/sec rate. At time $t = 30$, two queries from the same server were killed and forced to be rescheduled into the other server.

Figure 7.19 shows the results of this experiment. Queries 1 and 3 are the ones that were killed and rescheduled. First thing to note is that the latency for queries 2 and 4 were not affected during the whole experiment, which shows that queries are properly isolated from concurrent activities happening in the system. Moreover, the time needed to recover from failure and relocate the queries to another server is similar to the recovery time in the first experiment, in which the failed query was rescheduled in the same server.

These experiments show the fault-tolerance mechanism implemented by the *CEPaaS* system is effective and can quickly recover failed queries without losing messages.

## 7.7 Summary

This section presented the design and implementation of a CEP as a Service (*CEPaaS*) system and evaluated it in a series of experiments.

The proposed *CEPaaS* system is based on a novel architecture that uses multiple clouds to improve the system fault-tolerance and to explore the geographical diversity of public cloud datacentres. The architecture also explores application containers as a way to encapsulate system components, and uses CMS to schedule and to manage containers execution. Finally, the system proposes query templates as an extensible mechanism to define new queries without the need of learning query definition languages.

The experiments demonstrated that multi-cloud architecture can be explored as a way to reduce the query latency. In addition, they also showed that the fault-tolerance mechanism provided by the system is efficient and can quickly recover queries from failures.

The next chapter concludes this thesis by reviewing its main contributions and by discussing possible directions for future work.

# Chapter 8

# Conclusion

This research has presented a series of contributions towards the development of a Complex Event Processing (CEP) system that can be offered as a service and used over the Internet. The development of this CEP as a Service (*CEPaaS*) system aims to bring the advantages of the services model to CEP, but it involves many challenges that encompass the whole research and development cycle. In particular, this work identified and proposed solutions for three open problems:

- The problem of *understanding* and *reusing* existing CEP procedures and algorithms that is caused by the large variety of current solutions, the use of inconsistent terminology, and the lack of a standard query definition language.

- The problem of *evaluating* CEP systems and *comparing* them with existing approaches that is caused by difficulties in executing repeatable Big Data experiments in cloud environments and the lack of proper tools.

- The problem of *designing* and *implementing* a *CEPaaS* system, caused by functional and non-functional requirements that must be satisfied by the implementation, such as multi-tenancy, fault-tolerance, and low-latency query execution.

More importantly, even though they were identified and discussed in the context of the *CEPaaS* system, these problems are general and often found in the context of similar research. Therefore, either by considering the contributions presented in isolation or together, this work significantly advances the CEP state-of-the-art and provides novel tools and methodologies that can be applied to CEP research and development.

In the next section, the contributions of this research are reviewed. Opportunities for future work are identified in Section 8.2.

## 8.1 Contributions

To solve the problem of *understanding* and *reusing* existing CEP procedures and algorithms, this research proposed the **A**ttributed **G**raph R**e**writing for **C**omplex **E**vent **P**rocessing Management (*AGeCEP*) formalism. Its main goal is to provide a formal way to express queries and query reconfigurations independently of query definition language and technology. By doing so, it provides a common ground through which management procedures can be expressed. Moreover, it also enables procedures expressed in *AGeCEP* to be applied to systems that adopt it as the underlying formalism.

Queries in *AGeCEP* are directed acyclic graphs whose vertices and edges are augmented with a standardized set of attributes used for decision making in runtime management procedures. These attributes are defined based on a novel classification of CEP operators, which has also been developed in this research and focuses on reconfiguration capabilities of CEP operators.

Query reconfigurations, on the other hand, are represented as graph rewriting rules based on the Single-Pushout approach [104]. In *AGeCEP*, the rewriting rules consider the vertices' characteristics, as determined by their attributes, to determine whether a rule can be applied. Therefore, the formalism can establish correctness guarantees for the reconfigurations: rules are never applied to incompatible operators and queries. In addition, *AGeCEP* rewriting rules are also associated with a set of mutators, which are executed as a side-effect of rule application. This mechanism guarantees that changes performed in the query models are correctly reflected in the real system.

The applicability of *AGeCEP* has been demonstrated in this research at many levels. First, it was shown that queries written in diverse language paradigms, such as Storm topologies [18], CQL [20], and CEL [44] queries, can be converted to *AGeCEP* ADAG format. In addition, a design for an autonomic manager based on *AGeCEP* was proposed along with a specially selected set of self-management policies, including procedures introduced by other researches such as operator duplication [38] and predicate indexing [109]. Furthermore, this work presented a generic methodology to adapt operator placement procedures to *AGeCEP* and exemplified it by showing how the procedures by Xing *et al*. [151] and Heinze *et al*. [70] can be expressed using this approach. Finally, *AGeCEP* has been used in the other contributions of this work. Both *CEPSim* and the *CEPaaS* system use *AGeCEP* query representation, and the *CEPaaS* QAM component is based on the *AGeCEP* autonomic manager.

The second major contribution of this research is *CEPSim*, a simulator of cloud-based CEP systems. This tool has been developed to overcome the challenges of *evaluating* CEP systems in Big Data scenarios and *comparing* them with existing approaches. *CEPSim* can model

different types of cloud environments and can simulate the execution of *AGeCEP* queries on them. Furthermore, it can be extended with new operator scheduling and operator placement strategies. In conjunction, these features enable quick comparison of different query processing and management approaches without the hassle of setting up and maintaining large cloud environments and data sources. Moreover, *CEPSim* can also streamline the execution of long running and dynamic tests.

To implement these features, *CEPSim* extended CloudSim [34] and adapted it to CEP. At its core, *CEPSim* provides algorithms to simulate operators and queries running in single or multiple servers. These algorithms are based on a novel concept called *event sets*, which represents a small batch of events and constitutes the smallest unit of data exchanged by simulated operators. *Event sets* are used by *CEPSim* both to improve the simulation performance and to assist in the calculation of query metrics. In addition, *CEPSim* algorithms also contain extension points to which new operator placement and scheduling strategies can be attached. By doing so, the user can customize the behaviour of the simulator and analyze the effects of different strategies on query performance.

Experiments were executed to assess the simulation of queries running on single and multiple servers in both private and public clouds. The results demonstrated that *CEPSim* can effectively simulate CEP queries and estimate query performance metrics with good accuracy. Moreover, they also showed that even very large simulations can be executed by *CEPSim* in a reasonable amount of time and without excessive consumption of resources.

As the final major contribution, this research also tackled the challenge of *designing* and *implementing* a *CEPaaS* system. The resulting prototype implementation was discussed in Chapter 7.

The proposed *CEPaaS* system adopts an architecture based on multi-cloud environments controlled by a CMS. By leveraging multiple clouds, the system can resist failures of entire datacentres and even cloud providers. In addition, it can also explore the geographical diversity of clouds to position system components closer to the event sources in order to reduce query latency. The CMS, on the other hand, manages the runtime environment and provides many functionalities needed by the system, such as scheduling of components and monitoring of containers for fault-tolerance. In addition, the CMS collects metrics and logs of the running containers and underlying infrastructure, which can be used both by system administrators and by users who are looking for more information about their queries.

The *CEPaaS* system also leverages a multi-tenant extensible design to provide CEP functionalities as services. It explores the concept of *vertex templates* as a way to encapsulate event processing logic. These vertex templates are put together in *query templates*, which specify event processing recipes that can be shared among system users. In this context, *queries* are

defined as instances of query templates. The user *instantiates* a query template and provides the required parameters to customize the template to its own needs. In runtime, queries are encapsulated in application containers and use the Akka toolkit [6] as the execution engine.

Finally, experimental results showed that the proposed architecture satisfies the requirements of a *CEPaaS* system. Query latency can be significantly reduced by deploying them closer to the event producers and consumers. Also, fault tolerance is efficiently handled by the CMS.

## 8.2 Future Work

Despite the significant contributions of this research, the CEP research area is still young and has many open challenges. Therefore, the work presented here can be extended in many directions to advance even further the state of the art. In particular, it should be noticed that the *CEPaaS* system presented here is only a minimum viable product built to validate the proposed architecture. The development of such prototype, however, enabled the identification of many research challenges that can be addressed in the future.

The following is a list of future directions to which this research can be extended:

- The *AGeCEP* formalism can be extended to include other aspects commonly found in CEP management scenarios. In particular, proposing a formal model to represent modern runtime environments is an interesting research problem. This model would need to consider complex datacentre organizations, including the relationship among servers, virtual machines, and, possibly, containers. It would also need to include modelling of the physical and logical networks, as well as of multiple types of clouds. The simple model presented in this research does not take all of these aspects into consideration, which may be needed for more complex placement procedures.

- The *AGeCEP*-based autonomic manager described in Chapter 5 needs to be thoroughly tested in real scenarios. Even though the *CEPaaS* QAM component is based on this manager, the lack of support for operator migration and distributed query execution limited the management policies that could be tested with *CEPaaS*. A more complete evaluation of the *AGeCEP* autonomic manager would include testing additional policies, and also applying the same set of policies to manage different CEP systems. Notably, this autonomic manager can be integrated with *CEPSim* to facilitate testing of new self-management policies.

- Even though it is already capable of simulating various scenarios, *CEPSim* has limited functionalities regarding simulation of dynamic scenarios. Currently, queries cannot be

added or removed during a simulation, and there is no support for operator migration. Another interesting functionality that should be added is to enable users to define their own load shedding strategies and compare their effects on query performance metrics.

- *CEPSim* also needs to be extended with better runtime environment modelling capabilities. For instance, it can include the notion of clients that interact with the queries and possibly associate them with a geographic position so as to enable modelling of access latency. Moreover, it is also necessary to study the implications of simulating application containers.

- *CEPSim* performance can be further improved, especially regarding execution time. Most computers today have multiple cores and it is essential to leverage them to speed up the simulations. User queries are often independent and therefore can be simulated in parallel.

- The *CEPaaS* system approach to query execution creates a whole new era for operator and query placement algorithms. Because the queries are encapsulated in application containers, the placement problem can be translated to dividing the query into containers and placing them into the cluster. To the best of our knowledge, there is no existing research that models the operator placement problem in this way. In this context, a good scheduling strategy can significantly increase the datacentre utilization level and improve the query quality of service. From the architectural point of view, this also creates another challenge about how to integrate the CMS scheduler with *CEPaaS* query management modules.

- Currently, the allocation of queries to *CEPaaS* deployments is fixed, but this could be dynamically decided based on the geographical position of event sources and consumers. Furthermore, the system could automatically spawn new deployments when it detects a cluster of event sources from an area that is not currently attended by current deployments.

- *CEPaaS* should be better equipped to deal with scalability in the volume and velocity of events from a single stream in addition to scalability in the number of queries. This type of scalability is traditionally achieved by splitting the query execution into sub-queries that run in distinct servers and communicate via the network. In *CEPaaS*, this problem is intimately related to the placement of query containers.

- The CEP literature is vast and, therefore, existing CEP techniques can be integrated into the *CEPaaS* system to make it more robust and resilient. For example, possible future

work could integrate checkpoint of stateful operators so that the operator state is not lost upon restarts. Other techniques, such as upstream backup and persistence of intermediate messages, can also be explored.

- Even though the system was designed to incorporate user-defined logic, this functionality still needs to be completely implemented. The idea is to dynamically download and load the operator logic classes in query containers that need them. This functionality can significantly improve the system applicability and, therefore, is an important future roadmap.

- There is a challenge related to the execution of self-management policies in very large *CEPaaS* deployments. Currently, each QAM replica consumes events from a subset of tenants only, which implies that no replica has a complete picture of the system. In this context, self-management policies can only reason based on queries managed by the same replica. Moreover, the QAM knowledge base (KB) does not contain a runtime environment representation, which also limits the type of rules that can be enforced. To solve these limitations, two approaches can be explored. First, it is possible to design a hierarchy of QAMs in which top-level managers take "global" decisions whereas low-level managers are responsible for query-local actions. Another approach is to implement a distributed memory KB, in which each QAM replica has access to the whole KB that is shared and distributed among them.

- Current *CEPaaS* implementation does not handle privacy and security. Access to send and receive data from the *Message Brokers* must be granted only to authorized users. Access to vertex and query templates should also be controlled according to the users' sharing configuration. Furthermore, user-defined vertex templates should run in a managed environment in order to avoid the execution of malicious code and potential security breaches. These features obviously need to be integrated into the system before it goes to production.

- Additional producer templates should be implemented alongside the infrastructure needed to provide them. Right now, most external communications is provided via Apache Kafka. Despite its great performance and scalability, Apache Kafka uses a binary protocol to communicate with clients, and client libraries are not available in many languages. Therefore, alternative ways of receiving data can facilitate the interaction with external agents. For instance, REST HTTP and MQTT [120] are protocols that can be used for this purpose. Similarly, additional consumer templates can also be added.

Results of queries can be written to database tables or feed dynamic monitoring dashboards.

- The *CEPaaS* system can be extended to provide a lightweight way of defining schemas for events and vertex templates. By doing so, it will be possible to guarantee consistency of query templates at design time.
- Finally, an important feature for a production *CEPaaS* system is to define different QoS levels for the paying tenants and to enforce them during query execution. To achieve this goal, a number of mechanisms can be used, such as limiting the input event consumption rate and output production rate, automatically constraining the number and size of query containers that a tenant can have, and prioritizing tenant workloads. Furthermore, the system also needs to properly audit tenant usage so they can be correctly charged.

# Bibliography


[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003. doi: 10.1007/s00778-003-0095-z.

[2] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR'05)*, pages 277–289, Asilomar, CA, USA, 2005.

[3] Yanif Ahmad and Ugur Çetintemel. Network-aware query processing for stream-based applications. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30 - VLDB '04*, pages 456–467, Toronto, ON, Canada, 2004.

[4] Tyler Akidau, Sam Whittle, Alex Balikov, Kaya Bekirolu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, and Paul Nordstrom. MillWheel: fault-tolerant stream processing at Internet scale. *Proceedings of the VLDB Endowment*, 6 (11):1033–1044, August 2013. doi: 10.14778/2536222.2536229.

[5] Tyler Akidau, Eric Schmidt, Sam Whittle, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, and Frances Perry. 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*, 8(12):1792–1803, August 2015. doi: 10.14778/2824032.2824076.

[6] Akka. Akka, 2016. URL `http://akka.io`.

[7] Mahdi Ben Alaya and Thierry Monteil. FRAMESELF: an ontology-based framework

for the self-management of machine-to-machine systems. *Concurrency and Computation: Practice and Experience*, 27(6):1412–1426, April 2015. doi: 10.1002/cpe.3168.

[8] Ahmed M. Aly, Asmaa Sallam, Bala M. Gnanasekaran, Long-Van Nguyen-Dinh, Walid G. Aref, Mourad Ouzzani, and Arif Ghafoor. M3: Stream Processing on Main-Memory MapReduce. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1253–1256, Washington, DC, USA, 2012. doi: 10.1109/ICDE.2012.120.

[9] Amazon. Summary of the Amazon DynamoDB Service Disruption and Related Impacts in the US-East Region, 2015. URL `https://aws.amazon.com/message/5467D2/`.

[10] Amazon. Amazon Elastic Compute Cloud (EC2), 2016. URL `https://aws.amazon.com/ec2/`.

[11] Amazon. Amazon EC2 Container Service (ECS), 2016. URL `https://aws.amazon.com/ecs/`.

[12] Amazon. Amazon Kinesis, 2016. URL `https://aws.amazon.com/kinesis/`.

[13] Amazon. AWS Lambda, 2016. URL `https://aws.amazon.com/lambda/`.

[14] Rajagopal Ananthanarayanan, Shivakumar Venkataraman, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, and Manpreet Singh. Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams. In *Proceedings of the 2013 International Conference on Management of Data - SIGMOD '13*, pages 577–588, New York, NY, USA, 2013. doi: 10.1145/2463676.2465272.

[15] Apache. Apache Aurora, 2016. URL `http://aurora.apache.org`.

[16] Apache. Apache Hadoop, 2016. URL `http://hadoop.apache.org/`.

[17] Apache. Samza, 2016. URL `http://samza.apache.org`.

[18] Apache. Storm, distributed and fault-tolerant realtime computation, 2016. URL `http://storm.apache.org`.

[19] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: The Stanford Data Stream Management System. Technical Report 2004-20, Stanford InfoLab, 2004. URL `http://ilpubs.stanford.edu:8090/641/`.

[20] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, July 2005. doi: 10.1007/s00778-004-0147-z.

[21] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, and Randy Katz. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report 2009-28, University of California, Berkeley, 2009. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`.

[22] Steve Awodey. *Category Theory*, volume 49 of *Oxford Logic Guides*. Oxford University Press, 2006.

[23] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. Chain: operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data - SIGMOD '03*, pages 253–264, San Diego, CA, USA, 2003.

[24] Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker. Contract-based load management in federated distributed systems. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation - Volume 1*, pages 15–28, San Francisco, CA, USA, 2004.

[25] Raphaël Barazzutti, Pascal Felber, Christof Fetzer, Emanuel Onica, Jean-François Pineau, Marcelo Pasin, Etienne Rivière, and Stefan Weigert. StreamHub: A Massively Parallel Architecture for High-Performance Content-Based Publish/Subscribe. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS '13)*, pages 63–74, Arlington, TX, USA, 2013. doi: 10.1145/2488222.2488260.

[26] Tim Bass. Mythbusters: event stream processing versus complex event processing. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems (DEBS '07)*, pages 1–1, Toronto, ON, Canada, 2007. doi: 10.1145/1266894.1266896.

[27] Luiz F. Bittencourt and Edmundo R. M. Madeira. Towards the Scheduling of Multiple Workflows on Computational Grids. *Journal of Grid Computing*, 8(3):419–441, September 2010. doi: 10.1007/s10723-009-9144-1.

[28] Luiz F. Bittencourt, Edmundo R. M. Madeira, and Nelson S. da Fonseca. Scheduling in hybrid clouds. *IEEE Communications Magazine*, 50(9):42–47, September 2012. doi: 10.1109/MCOM.2012.6295710.

[29] Tim Bray. RFC7159: The JavaScript Object Notation (JSON) Data Interchange Format, 2014. ISSN 2070-1721. URL `http://tools.ietf.org/html/rfc7159`.

[30] Christian Y. A. Brenninkmeijer, Ixent Galpin, Alvaro A. A. Fernandes, and Norman W. Paton. A Semantics for a Query Language over Sensors, Streams and Relations. In Alex Gray, Keith Jeffery, and Jianhua Shao, editors, *Sharing Data, Information and Knowledge SE - 9*, volume 5071 of *Lecture Notes in Computer Science*, pages 87–99. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-70504-8_9.

[31] Andrey Brito, Andre Martin, Thomas Knauth, Stephan Creutz, Diogo Becker, Stefan Weigert, and Christof Fetzer. Scalable and Low-Latency Data Processing with Stream MapReduce. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 48–58, Athens, Greece, 2011. doi: 10.1109/CloudCom.2011.17.

[32] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *ACM Queue*, 14(1):70–93, January 2016. doi: 10.1145/2898442.2898444.

[33] Rajkumar Buyya and Manzur Murshed. GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220, November 2002. doi: 10.1002/cpe.710.

[34] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, January 2011. doi: 10.1002/spe.995.

[35] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*, pages 363–375, Toronto, ON, Canada, 2010. doi: 10.1145/1806596.1806638.

[36] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Asilomar, CA, USA, 2003.

[37] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *ACM SIGMOD Record*, 29(2):379–390, June 2000. doi: 10.1145/335191.335432.

[38] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Çetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, pages 257–268, Asilomar, CA, USA, 2003.

[39] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation (NSDI'10)*, pages 21–35, San Jose, CA, USA, 2010.

[40] Gianpaolo Cugola and Alessandro Margara. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems - DEBS '10*, pages 50–61, Cambridge, UK, 2010. doi: 10.1145/1827418.1827427.

[41] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 44(3):1–62, June 2012. doi: 10.1145/2187671.2187677.

[42] Gianpaolo Cugola, Alessandro Margara, Mauro Pezzè, and Matteo Pradella. Efficient analysis of event processing applications. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems - DEBS '15*, pages 10–21, Oslo, Norway, 2015. doi: 10.1145/2675743.2771834.

[43] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, January 2008. doi: 10.1145/1327452.1327492.

[44] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A General Purpose Event Monitoring System. In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR'07)*, pages 412–422, Asilomar, CA, USA, 2007.

[45] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic Approaches to Graph Transformation. Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In Grzegorz

Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 247–312. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. ISBN 98-102288-48.

[46] Cédric Eichler. *Modélisation formelle de systèmes dynamiques autonomes: graphe, réécriture et grammaire*. PhD thesis, Université Toulouse III, 2015.

[47] Cédric Eichler, Ghada Gharbi, Nawal Guermouche, Thierry Monteil, and Patricia Stolf. Graph-Based Formalism for Machine-to-Machine Self-Managed Communications. In *2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 74–79, Hammamet, Tunisia, 2013. doi: 10.1109/WETICE.2013.45.

[48] Cédric Eichler, Thierry Monteil, Patricia Stolf, Luigi Alfredo Grieco, and Khalil Drira. Enhanced graph rewriting systems for complex software domains. *Software & Systems Modeling*, 15(3):685–705, July 2016. doi: 10.1007/s10270-014-0433-1.

[49] AnnMarie Ericsson and Mikael Berndtsson. Rex, the rule and event explorer. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems - DEBS '07*, pages 71–74, Toronto, ON, Canada, 2007. doi: 10.1145/1266894.1266906.

[50] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications, 1st edition, 2010. ISBN 1935182218.

[51] Izaias de Faria, Mario A. R. Dantas, Miriam A. M. Capretz, and Wilson A. Higashino. Network and Energy-Aware Resource Selection Model for Opportunistic Grids. In *2014 IEEE 23rd International WETICE Conference*, pages 167–172, Anchorage, AK, USA, 2014. doi: 10.1109/WETICE.2014.31.

[52] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and Linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, Philadelphia, PA, USA, 2015. doi: 10.1109/ISPASS.2015.7095802.

[53] Raul C. Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*, pages 725–736, New York, NY, USA, 2013. doi: 10.1145/2463676.2465282.

[54] Ana J. Ferrer, Francisco Hernández, Johan Tordsson, Erik Elmroth, Ahmed Ali-Eldin, Csilla Zsigri, Raül Sirvent, Jordi Guitart, Rosa M. Badia, Karim Djemame, Wolfgang Ziegler, Theo Dimitrakos, Srijith K. Nair, George Kousiouris, Kleopatra Konstanteli, Theodora Varvarigou, Benoit Hudzia, Alexander Kipp, Stefan Wesner, Marcelo Corrales, Nikolaus Forgó, Tabassum Sharif, and Craig Sheridan. OPTIMIS: A holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28(1): 66–77, January 2012. doi: 10.1016/j.future.2011.05.022.

[55] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN 0716710455.

[56] Saurabh K. Garg and Rajkumar Buyya. NetworkCloudSim: Modelling Parallel Applications in Cloud Simulations. In *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pages 105–113, Victoria, NSW, Australia, 2011. doi: 10.1109/UCC.2011.24.

[57] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, December 2003. doi: 10.1145/1165389.945450.

[58] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, June 2003. doi: 10.1145/776985.776986.

[59] Google. Google Compute Engine Incident #15045, 2015. URL `https://status.cloud.google.com/incident/compute/15045`.

[60] Google. Google Container Engine, 2016. URL `https://cloud.google.com/container-engine/`.

[61] Google. Cloud Dataflow, 2016. URL `https://cloud.google.com/dataflow/`.

[62] Google. Cloud Pub/Sub, 2016. URL `https://cloud.google.com/pubsub/`.

[63] Google. Kubernetes, 2016. URL `http://kubernetes.io`.

[64] Katarina Grolinger, Wilson A. Higashino, Abhinav Tiwari, and Miriam A. M. Capretz. Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(22):1–24, 2013. doi: 10.1186/2192-113X-2-22.

[65] Katarina Grolinger, Michael Hayes, Wilson A. Higashino, Alexandra L'Heureux, David S. Allison, and Miriam A. M. Capretz. Challenges for MapReduce in Big Data. In *2014 IEEE World Congress on Services*, pages 182–189, Anchorage, AK, USA, 2014. doi: 10.1109/SERVICES.2014.41.

[66] N. Grozev and R. Buyya. Performance Modelling and Simulation of Three-Tier Applications in Cloud and Multi-Cloud Environments. *The Computer Journal*, 58(1):1–22, January 2015. doi: 10.1093/comjnl/bxt107.

[67] Nikolay Grozev and Rajkumar Buyya. Inter-Cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience*, 44(3):369–390, March 2014. doi: 10.1002/spe.2168.

[68] Tom Guérout, Thierry Monteil, Georges Da Costa, Rodrigo N. Calheiros, Rajkumar Buyya, and Mihai Alexandru. Energy-aware simulation with DVFS. *Simulation Modelling Practice and Theory*, 39:76–91, December 2013. doi: 10.1016/j.simpat.2013.04.007.

[69] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, December 2012. doi: 10.1109/TPDS.2012.24.

[70] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems - DEBS '14*, pages 13–22, Mumbai, India, 2014. doi: 10.1145/2611286.2611294.

[71] Sebastian Herbst, Niko Pollner, Johannes Tenschert, Frank Lauterwald, Gregor Endler, and Klaus Meyer-Wegener. An algebra for pattern matching, time-aware aggregates and partitions on relational data streams. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems - DEBS '15*, pages 140–149, Oslo, Norway, 2015. doi: 10.1145/2675743.2771830.

[72] Heroku. Heroku, 2016. URL `http://www.heroku.com/`.

[73] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence - IJCAI'73*, pages 235–245, San Francisco, CA, USA, 1973.

[74] Wilson A. Higashino, Maria B. F. de Toledo, and Miriam A. M. Capretz. REST and Resource-Oriented Architecture. In *Proc. of First International Symposium on Services Science (ISSS'09)*, pages 161–171, Leipzig, Germany, 2009.

[75] Wilson A. Higashino, Cédric Eichler, Miriam A. M. Capretz, Thierry Monteil, Maria B. F. de Toledo, and Patricia Stolf. Query Analyzer and Manager for Complex Event Processing as a Service. In *2014 IEEE 23rd International WETICE Conference*, pages 107–109, Parma, Italy, 2014. doi: 10.1109/WETICE.2014.53.

[76] Wilson A. Higashino, Miriam A. M. Capretz, and Luiz F. Bittencourt. CEPSim: A Simulator for Cloud-Based Complex Event Processing. In *2015 IEEE International Congress on Big Data*, pages 182–190, New York, NY, USA, 2015. doi: 10.1109/BigDataCongress.2015.34.

[77] Wilson A. Higashino, Miriam A. M. Capretz, and Luiz F. Bittencourt. CEPSim: Modelling and simulation of Complex Event Processing systems in cloud environments. *Future Generation Computer Systems*, 65:122–139, December 2016. doi: 10.1016/j.future.2015.10.023.

[78] Wilson A. Higashino, Miriam A. M. Capretz, Maria B. F. de Toledo, and Luiz F. Bittencourt. A Hybrid Particle Swarm Optimization-Genetic Algorithm Applied to Grid Scheduling. *International Journal of Grid and Utility Computing*, 7(2):113–129, 2016.

[79] Wilson A. Higashino, Cédric Eichler, Miriam A. M. Capretz, Luiz F. Bittecourt, and Thierry Monteil. Attributed Graph Rewriting for Complex Event Processing Self-Management. *ACM Transactions on Autonomous and Adaptive Systems*, 2016. In press.

[80] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 295–308, Boston, MA, USA, 2011.

[81] Dan Hirsch and Ugo Montanari. Consistent transformations for software architecture styles of distributed systems. *Electronic Notes in Theoretical Computer Science*, 28:4, 2000. doi: 10.1016/S1571-0661(05)80626-7.

[82] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan Demers. Rule-based multi-query optimization. In *Proceedings of the 12th International Conference on Extending Database Technology Advances in Database Technology -*

*EDBT '09*, pages 120–131, Saint Petersburg, Russia, 2009. doi: 10.1145/1516360.1516376.

[83] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, pages 1–14, Boston, MA, USA, 2010.

[84] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *21st International Conference on Data Engineering (ICDE'05)*, pages 779–790, Tokyo, Japan, 2005. doi: 10.1109/ICDE.2005.72.

[85] IBM. An architectural blueprint for autonomic computing. Technical Report June, 2005. URL http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf.

[86] IBM. IBM Streams, 2016. URL http://www-03.ibm.com/software/products/en/ibm-streams.

[87] Atsushi Ishii and Toyotaro Suzumura. Elastic Stream Computing with Clouds. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 195–202, Washington, DC, USA, July 2011. doi: 10.1109/CLOUD.2011.11.

[88] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a Streaming SQL Standard. *Proceedings of the VLDB Endowment*, 1 (2):1379–1390, August 2008.

[89] JBoss. Drools, 2016. URL http://www.drools.org.

[90] Jess. Jess, the Rule Engine for the Java Platform, 2016. URL http://www.jessrules.com/.

[91] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003. doi: 10.1109/MC.2003.1160055.

[92] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan. GreenCloud: A packet-level simulator of energy-aware cloud computing data centers. *Journal of Supercomputing*, 62(3):1263–1283, December 2012. doi: 10.1007/s11227-010-0504-1.

[93] Jürgen Krämer and Bernhard Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Transactions on Database Systems*, 34(1): 1–49, April 2009. doi: 10.1145/1508857.1508861.

[94] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *Proceeding of the 6th International Workshop on Networking Meets Databases*, pages 1–7, Athens, Greece, June 2011.

[95] Sanjeev Kulkarni, Nikunj Bhagat, Masong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*, pages 239–250, Melbourne, Australia, 2015. doi: 10.1145/2723372.2742788.

[96] Tobias Kurze, Markus Klems, David Bermbach, Alexander Lenk, Stefan Tai, and Marcel Kunze. Cloud federation. In *Proceedings of the 2nd International Conference on Cloud Computing, GRIDs, and Virtualization (CLOUD COMPUTING 2011)*, pages 32–37, Rome, Italy, 2011.

[97] Geetika T. Lakshmanan, Ying Li, and Rob Strom. Placement Strategies for Internet-Scale Data Stream Systems. *IEEE Internet Computing*, 12(6):50–60, November 2008. doi: 10.1109/MIC.2008.129.

[98] Wang Lam, Lu Liu, Sts Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. Muppet: MapReduce-style processing of fast data. *Proceedings of the VLDB Endowment*, 5(12):1814–1825, August 2012.

[99] Guoli Li and Hans-Arno Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, pages 249–269, Grenoble, France, 2005.

[100] Linux Containers. Infrastructure for container projects, 2016. URL `https://linuxcontainers.org`.

[101] Simon Loesing, Martin Hentschel, Tim Kraska, and Donald Kossmann. Stormy: an elastic and highly available streaming service in the cloud. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pages 55–60, Berlin, Germany, 2012. doi: 10.1145/2320765.2320789.

[102] Dionysios Logothetis and Kenneth Yocum. Ad-hoc data processing in the cloud. *Proceedings of the VLDB Endowment*, 1(2):1472–1475, August 2008.

[103] Björn Lohrmann, Daniel Warneke, and Odej Kao. Nephele streaming: stream processing under QoS constraints at scale. *Cluster Computing*, 17(1):61–78, July 2013. doi: 10.1007/s10586-013-0281-8.

[104] Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1-2):181–224, March 1993. doi: 10.1016/0304-3975(93)90068-5.

[105] David Luckham. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. Technical Report CSL-TR-96-705, Stanford University, 1996.

[106] David Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Professional, 1st edition, 2002. ISBN 0201727897.

[107] David Luckham. What's the difference between ESP and CEP?, 2006. URL http://www.complexevents.com/2006/08/01/what's-the-difference-between-esp-and-cep/.

[108] David Luckham and Roy Schulte. Event Processing Glossary - Version 2.0. Technical Report July, Event Processing Technical Society, 2011. URL http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf.

[109] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data - SIGMOD '02*, volume 13, pages 49–60, Madison, WI, USA, 2002. doi: 10.1145/564691.564698.

[110] Masoud Mansouri-Samani and Morris Sloman. GEM - A Generalised Event Monitoring Language for Distributed Systems. *Distributed Systems Engineering Journal*, 4(2):96–108, June 1997.

[111] André Martin, Andrey Brito, and Christof Fetzer. Scalable and elastic realtime click stream analysis using StreamMine3G. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems - DEBS '14*, pages 198–205, Mumbai, India, 2014. doi: 10.1145/2611286.2611304.

[112] Andre Martin, Tiaraju Smaneoto, Tobias Dietze, Andrey Brito, and Christof Fetzer. User-Constraint and Self-Adaptive Fault Tolerance for Event Stream Processing Systems. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 462–473, Rio de Janeiro, Brazil, 2015. doi: 10.1109/DSN.2015.56.

[113] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology. *NIST Special Publication*, 145:1–7, 2011.

[114] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239), March 2014.

[115] Microsoft. Final Root Cause Analysis and Improvement Areas: Nov 18 Azure Storage Service Interruption, 2014. URL `https://azure.microsoft.com/en-us/blog/final-root-cause-analysis-and-improvement-areas-nov-18-azure-storage-service-`

[116] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, pages 439–455, Farminton, PA, USA, 2013. doi: 10.1145/2517349.2522738.

[117] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed Stream Computing Platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177, Sydney, NSW, Australia, 2010. doi: 10.1109/ICDMW.2010.172.

[118] ns-2. The Network Simulator. URL `http://nsnam.sourceforge.net/wiki/index.php/User_Information`.

[119] Alberto Núñez, Jose L. Vázquez-Poletti, Agustin C. Caminero, Gabriel G. Castañé, Jesus Carretero, and Ignacio M. Llorente. iCanCloud: A Flexible and Scalable Cloud Infrastructure Simulator. *Journal of Grid Computing*, 10(1):185–209, March 2012. doi: 10.1007/s10723-012-9208-5.

[120] OASIS. MQTT Version 3.1.1, 2014. URL `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`.

[121] Frank J. Ohlhorst. *Big Data Analytics: Turning Big Data into Big Money*. Wiley, Hoboken, NJ, USA, 1st. ed. edition, 2012. ISBN 1118147596.

[122] OpenVZ. OpenVZ Virtuozzo Containers, 2016. URL https://openvz.org/Main_Page.

[123] Oracle. Oracle Stream Explorer, 2015. URL http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html.

[124] Francisco de la Parra and Thomas Dean. Survey of graph rewriting applied to model transformations. In *2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014)*, pages 431–441, Lisbon, Portugal, 2014.

[125] Peter Pietzuch, Brian Shand, and Jean Bacon. A framework for event composition in distributed systems. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 62–82, Rio de Janeiro, Brazil, 2003.

[126] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 49–60, Atlanta, GA, USA, 2006. doi: 10.1109/ICDE.2006.105.

[127] Powersmiths. Powersmiths WOW - Build a more sustainable future, 2016. URL http://www.powersmithswow.com/.

[128] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, pages 1–14, Prague, Czech Republic, 2013. doi: 10.1145/2465351.2465353.

[129] Ella Rabinovich, Opher Etzion, Sitvanit Ruah, and Sarit Archushin. Analyzing the behavior of event processing applications. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems - DEBS 2010*, pages 223–234, Cambridge, UK, 2010. doi: 10.1145/1827418.1827465.

[130] Ismael Bouassida Rodriguez, Khalil Drira, Christophe Chassot, Karim Guennoun, and Mohamed Jmaiel. A rule-driven approach for architectural self adaptation in collaborative activities using graph grammars. *International Journal of Autonomic Computing*, 1 (3):226–245, 2010. doi: 10.1504/IJAC.2010.033007.

[131] Jothy Rosenberg and Arthur Mateos. *The Cloud at Your Service*. Manning Publications, 1st ed edition, 2010. ISBN 1935182528.

[132] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. ISBN 98-102288-48.

[133] Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Esc: Towards an Elastic Stream Computing Platform for the Cloud. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 348–355, Washington, DC, USA, 2011. doi: 10.1109/CLOUD.2011.27.

[134] Nicholas P. Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems (DEBS '09)*, pages 1–12, Nashville, TN, USA, 2009. doi: 10.1145/1619258.1619264.

[135] Andy Schürr, Dániel Varró, and Gergely Varró, editors. *Applications of Graph Transformations with Industrial Relevance*, volume 7233 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-34175-5. doi: 10.1007/978-3-642-34176-2.

[136] Sergio Segura, David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. Automated Merging of Feature Models Using Graph Transformations. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, pages 489–505. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi: 10.1007/978-3-540-88643-3_15.

[137] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: an adaptive partitioning operator for continuous query systems. In *Proceedings 19th International Conference on Data Engineering*, pages 25–36, Bangalore, India, 2003. doi: 10.1109/ICDE.2003.1260779.

[138] G. Sharon and O. Etzion. Event-processing network model and implementation. *IBM Systems Journal*, 47(2):321–334, March 2008. doi: 10.1147/sj.472.0321.

[139] Software AG. APAMA Streaming Analytics, 2015. URL `http://www.softwareag.com/corporate/products/apama_webmethods/analytics/overview/`.

[140] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alterna-

tive to hypervisors. *ACM SIGOPS Operating Systems Review*, 41(3):275–287, 2007. doi: 10.1145/1272998.1273025.

[141] Ashish Sonone, Anand Soni, Senthil Nathan, and Umesh Bellur. On Exploiting Page Sharing in a Virtualised Environment - An Empirical Study of Virtualization Versus Lightweight Containers. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 49–56, New York, NY, USA, 2015. doi: 10.1109/CLOUD.2015.17.

[142] Michael Stonebraker, Ugur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, December 2005. doi: 10.1145/1107499.1107504.

[143] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance: Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*, pages 446–453. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi: 10.1007/978-3-540-25959-6_35.

[144] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 147–156, Snowbird, Utah, USA, 2014. doi: 10.1145/2588555.2595641.

[145] Radu Tudoran, Alexandru Costan, Olivier Nano, Ivo Santos, Hakan Soncu, and Gabriel Antoniu. JetStream: Enabling high throughput live event streaming on multi-site clouds. *Future Generation Computer Systems*, 54:274–291, January 2016. doi: 10.1016/j.future.2015.01.016.

[146] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds: Towards a Cloud Definition. *Computer Communication Review*, 39(1):50–55, January 2009. doi: 10.1145/1496091.1496100.

[147] Vinod Kumar Vavilapalli, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, Eric Baldeschwieler, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, and Hitesh Shah. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing - SOCC '13*, pages 1–16, Santa Clara, CA, USA, 2013. doi: 10.1145/2523616.2523633.

[148] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*, pages 1–17, Bordeaux, France, 2015. doi: 10.1145/2741948.2741964.

[149] Matthias Weidlich, Jan Mendling, and Avigdor Gal. Net-Based Analysis of Event Processing Networks: The Fast Flower Delivery Case. In José-Manuel Colom and Jörg Desel, editors, *Application and Theory of Petri Nets and Concurrency SE - 15*, volume 7927 of *Lecture Notes in Computer Science*, pages 270–290. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-38697-8_15.

[150] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06*, volume 10, pages 407–418, Chicago, IL, USA, 2006. doi: 10.1145/1142473.1142520.

[151] Ying Xing, Stan Zdonik, and Jeong-hyon Hwang. Dynamic Load Distribution in the Borealis Stream Processor. In *21st International Conference on Data Engineering (ICDE'05)*, pages 791–802, Tokyo, Japan, 2005. doi: 10.1109/ICDE.2005.53.

[152] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*, pages 1–14, San Jose, CA, USA, 2012.

[153] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, pages 423–438, Farminton, PA, USA, 2013. doi: 10.1145/2517349.2522737.

[154] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, April 2010. doi: 10.1007/s13174-010-0007-6.

# Appendix A

# Self-Management Policies Inference Rules

This appendix presents the inference rules used in the MAPE loop by the self-management policies from Section 5.3. The rules are written in Drools Rule Language [89], a declarative language based on the event-condition-action paradigm.

**Algorithm A.1:** *Operator combination - analysis inference rule.*

```
rule "comb-analysis"
    when
        $s: Symptom(category == "NewQuery", $query: query)
        $rule: Rule(name == "P_comb", $lhs : left)
        eval (Util.homomorphism($lhs, $query).value)
    then
        insert(new Rfc("Combine", $query))
end
```

**Algorithm A.2:** *Operator duplication - monitoring inference rule.* A bottleneck is detected if an operator queue size is trending up considering the last 5 monitoring events.

```
rule "dupl-monitor"
    when
        $op: Operator($id: id)
        Trend(this == Trend.UP) from accumulate(
            Event(source == $id, category == "queueSize", $value: value) over window:length(5),
            trend( $value ))
    then
        insert(new Symptom("Bottleneck", $op))
end
```

**Algorithm A.3:** *Operator duplication - analysis inference rule.*

```
rule "dupl-analysis"
    when
        $s:  Symptom(category == "Bottleneck")
        $op: Operator(duplicable == true) from $s.operator
    then
        insert(new Rfc("Duplicate", $op.query, $op))
        delete($s)
end
```

**Algorithm A.4:** *Operator duplication - plan inference rule.* The left-hand side of rule $P^{add}_{dupl}(id, s_o)$ (Figure 5.3) is used to verify if the appropriate merge and split operators are already in place.

```
rule "dupl-plan"
    when
        $rfc: Rfc(category == "Duplicate", $query: operator.query, $op: operator)
        Rule(name == "P_add_dupl", $lhs: left)
    then
        params = ["id": $op.id];
        h      = Util.homomorphism($lhs, $query, params);

        if (!h.getValue()) {
            p1 = Util.placementForInitDupl($query, $op);
            insert(new Action("InitialDuplication", $query, $op, p1["split"],
                p1["op"], p1["merge"] ));
        } else {
            p2 = Util.placementForAddDupl($query, $op);
            insert(new Action("AdditionalDuplication", $query, $op, p2["op"]));
        }
        delete($rfc);
end
```

**Algorithm A.5:** *Removal of an unnecessary merge/split - analysis inference rule.* The method *hasMapping* searches for the bijective function $f$ defined in Section 5.3.3.

```
rule "unnec-ms-analysis"
    when
         $s:  Symptom(category == "NewQuery" || category == "Duplicated", $query: query)
         Rule(name == "P_rem_byp", $lhs: left)
    then
        Homomorphism h = Util.homomorphism($lhs, $query)
        if (h.value) {
            foreach (Graph g : h.results) {
                merge = g.byType("merge")[0];
                split = g.byType("split")[0];
                f     = Util.hasMapping(merge, split);
                if (f != null) {
                    insert(new Rfc("RemoveMergeSplit", $query, merge, split, f))
                }
            }
        }
end
```

**Algorithm A.6:** *Processing sub-streams - analysis inference rules*. It finds a sequence of a merge and a duplicable operator by searching for an homomorphism $L_{proc} \rightarrow q$. The method *checkSubStream* verifies the sub-stream conditions defined in Section 5.3.4.

```
declare ProcSubS
    query: Query
    merge: Operator
    op: Operator
end

rule "proc-subs-bottleneck-analysis"
    when
        $s:  Symptom(category == "Bottleneck", $query: query)
        $op: Operator(selectivity < 1.0) from $s.operator
        $rule: Rule(name == "L_proc", $lhs: left)
    then
        params = ["id": $op.id ]
        h = Util.homomorphism($lhs, $query, params)
        for (Graph g : h.results) {
            insert(new ProcSubS($query, g.byType("merge")[0], g.byId($op.id)))
        }
end

rule "proc-subs-conditions-analysis"
    when
        $ps: ProcSubS()
        $query: Query() from $ps.query
        $op: Operator(duplicable == true) from $ps.op
        $merge: Operator(mergeType == "union", inDegree > 1) from $ps.merge
        eval(Util.checkProcSubStream($merge, $op))
    then
        for (int i = 0; i < $merge.inDegree; i++) {
            insert(new Rfc("Duplicate", $query, $op))
        }
end
```

**Algorithm A.7:** *Predicate Indexing - analysis inference rule*.

```
rule "mqo-analysis-init"
    when
        Rule(name == "P_pred", $lhs: left)
        $s: NewQueries($queries: queries)
    then
        Query q = $queries[0]
        for (int i = 1; i < $queries.size(); i++) {
            q = q.add((Query) $queries[i])
        }
        if (Util.homomorphism($lhs, q).value) {
            insert(new Rfc("PredicateIndex", q))
        }
        delete($s)
end
```

# Appendix B

# CEPSim Implementation

This appendix details the *CEPSim* implementation. It starts with an overview of the simulator components, and it is followed by a description of the core classes. The integration of *CEPSim* with CloudSim is also discussed.

## B.1 Overview

Based on the design principles and goals presented in Chapter 6, *CEPSim* has been designed with three main components, as shown in Figure B.1:

- *CEPSim Core*: implements the *CEPSim* concepts shown in Figure 6.1. It provides APIs that enable the definition of queries and the creation of operator placement and scheduling strategies. In addition, it also implements the simulation logic described in Section 6.4.

- *CloudSim*: implements the CloudSim concepts shown in Figure 6.1. It provides the overall simulation framework, which controls the main simulation loop and the scheduling of simulation events. It is also used to define the cloud computing environment where the queries are simulated and to customize resource allocation policies.

- *CEPSim Integration*: implements the pieces necessary to integrate the CloudSim simulation engine with the CEP-specific logic provided by *CEPSim Core*. It guarantees loose coupling between the two and enables future integration with other simulators.

The following subsections detail the *CEPSim Core* and *Integration* components.

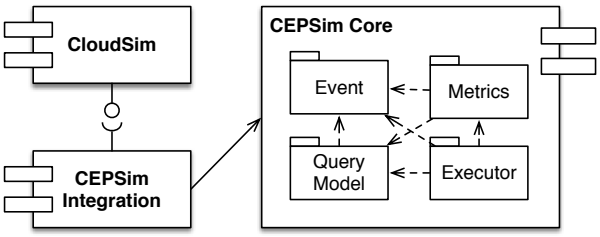



Figure B.1: *CEPSim* components.

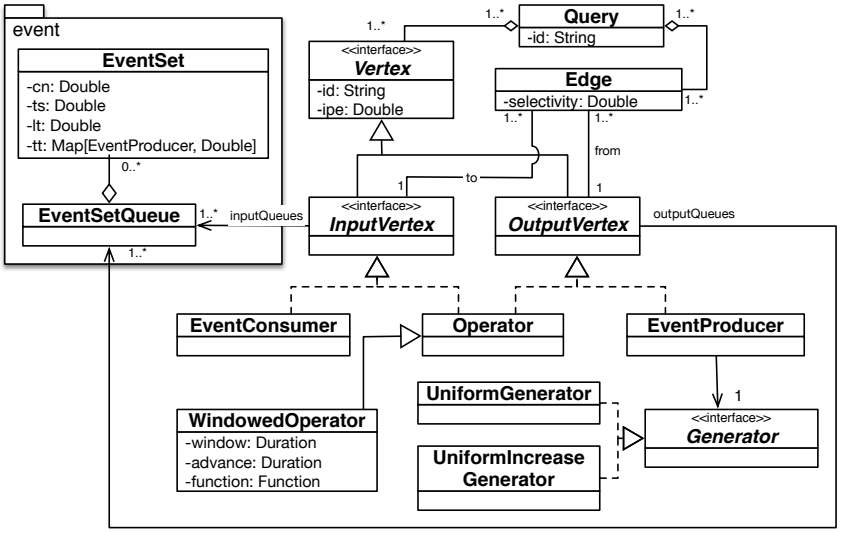



Figure B.2: Class diagram - *event* and *query model* packages.

## B.2 CEPSim Core

*CEPSim Core* classes and interfaces can be grouped into four main packages: *event*, which contains the event set and event set queue definitions; the *query model*, which contains the base classes used to describe queries; the *query executor*, which manages the query simulation; and *metrics*, which contains the metrics calculation framework.

The class diagram in Figure B.2 shows the main parts of the *event* and *query model* packages. Event sets and event set queues are implemented by classes with the same respective names in the *event* package. The *Query* class represents CEP queries and, as determined by its definition, is composed of one or more *Vertex* objects and one or more *Edge*s.

Two subclasses of *Vertex* have been identified: *OutputVertex* and *InputVertex*. The former represents vertices with outgoing edges, and the latter represents vertices with incoming edges. Note that both *OutputVertex* and *InputVertex* are associated with one or more instances of the *EventSetQueue* class representing their output and input queues respectively.

The *EventProducer* class describes event producers and therefore is a subclass of

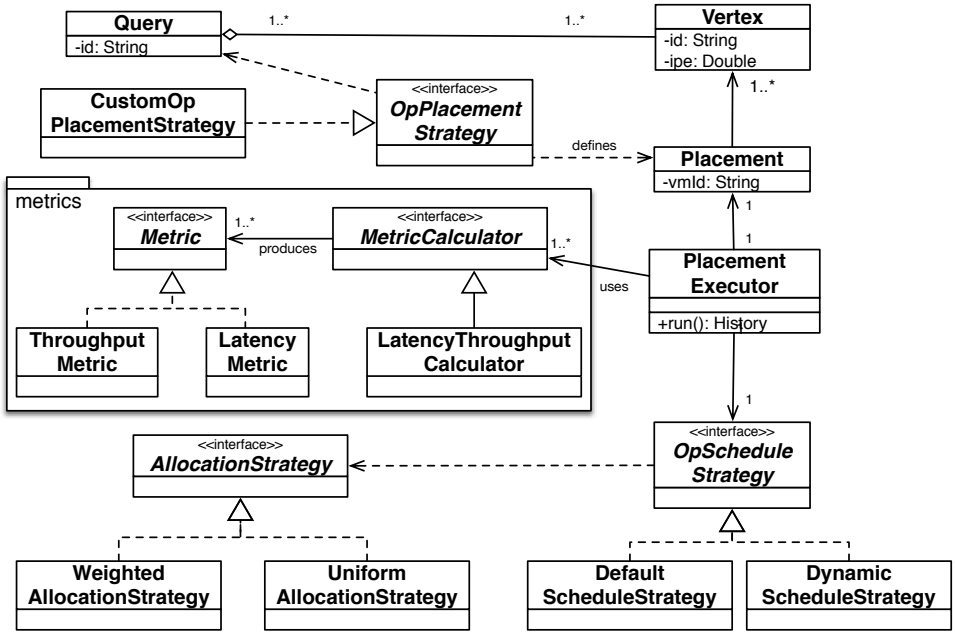



Figure B.3: Class diagram - *query executor* and *metrics* packages.

*OutputVertex* only. Similarly, *EventConsumer* characterizes event consumers and is a subclass of *InputVertex*. An *Operator* is both an *OutputVertex* and an *InputVertex* because it receives events from some vertices and sends them to others. The *Operator* class also has a *WindowedOperator* subclass that is used to represent windowed operators.

Finally, note that every *EventProducer* is associated with a *Generator* instance, which implements the generation function defined in Equation 6.2. *CEPSim* currently provides two implementations of this function:

- *UniformGenerator*: generates a constant number of events per simulation interval;
- *UniformIncreaseGenerator*: generates a uniformly increasing number of events until it reaches a maximum rate. After this point, this maximum rate is maintained until the end of the simulation.

The main classes and interfaces of the *query executor* and *metrics* packages are shown in Figure B.3. The *Placement* class is the central entity, representing the mapping of one or more vertices to the VM in which they will be executed. To create these placements, *CEPSim* users must provide an implementation of the *OpPlacementStrategy* interface, which defines an operator placement strategy. Currently, *CustomOpPlacementStrategy* is the only strategy provided by *CEPSim*, but others can be easily added. In this strategy, users must manually specify the mapping of vertices to VMs.

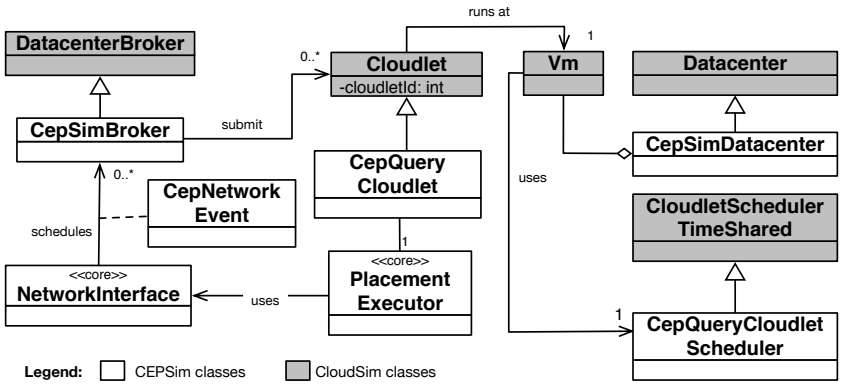



Figure B.4: *CEPSim* integration with CloudSim.

The *PlacementExecutor* class encapsulates a *Placement* and implements the placement simulation algorithm described in Section 6.4.4. This class uses an instance of the *OpScheduleStrategy* interface, which defines the operator scheduling strategy to be used during the simulation. Note that implementations for the scheduling and allocation strategies described in Section 6.4.2 are provided out-of-the-box by *CEPSim*.

In addition, the *PlacementExecutor* also interacts with one or more instances of the *MetricCalculator* interface to calculate the simulation metrics. The *LatencyThroughputCalculator* class shown in the figure is a built-in implementation that computes both metrics described in Section 6.4.5.

## B.3 CEPSim Integration

In accordance with the reuse design principle, *CEPSim* leverages many functionalities provided by CloudSim to enable the simulation of CEP queries. This section describes how CloudSim has been extended and integrated with the *CEPSim core*. The main parts of this extension are depicted in the class diagram in Figure B.4.

The main part of this extension is the *CepQueryCloudlet* class, a *Cloudlet* specialization that encapsulates the *PlacementExecutor* class described in the preceding section. During the simulation, a *CepQueryCloudlet* orchestrates a *PlacementExecutor* execution by invoking the *simulate* method at each simulation tick.

The other main classes created for the integration are:

- *CepSimBroker*: a mediator between cloud users and providers [34]. The *CepSimBroker* extends the CloudSim broker to handle *CepQueryCloudlet*s. It also maintains a mapping of all vertices to the VMs to which they have been allocated.

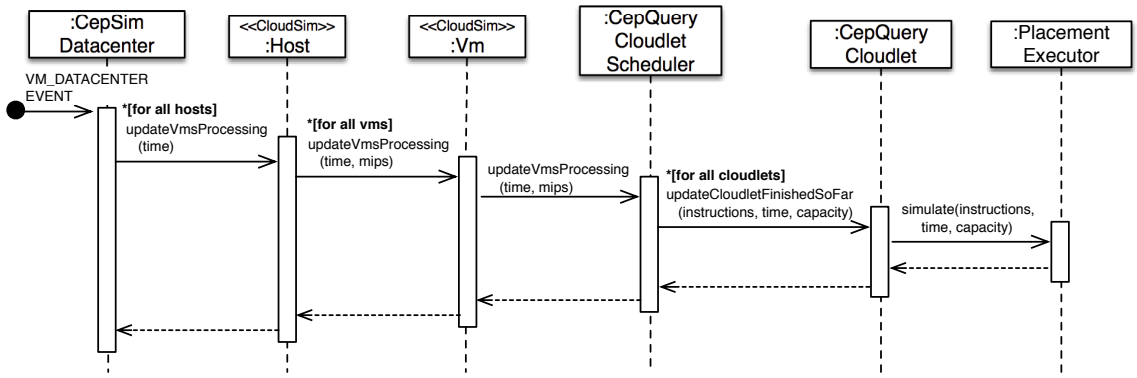



Figure B.5: Sequence diagram - simulation cycle.

- *CepSimDatacenter*: this datacentre specialization handles *CepQueryCloudlet*s and guarantees that the state of all simulated entities is updated at equally spaced intervals.

- *CepQueryCloudletScheduler*: a cloudlet scheduler defines how the processing power of a VM is shared among all cloudlets allocated to it [34]. This research extends the time-shared policy to handle infinite or duration-based cloudlets.

The sequence diagram in Figure B.5 summarizes how these classes work in tandem to implement a simulation cycle. First, the *CepSimDatacenter* receives a *Vm_Datacenter_Event* signal, which is a CloudSim simulation event used to update the state of all simulated entities in a datacentre. By default, this event is signalled when cloudlets resume or end their execution. In *CEPSim*, this behaviour has been changed so that the event is signalled at regular intervals with the length of a simulation tick. This guarantees that the CEP queries are periodically updated and renders the simulation more precise.

After receiving this event, *CepSimDatacenter* invokes the *updateVmsProcessing* method in all hosts in the datacentre. Note that the current simulation time is passed as a parameter of this method call and therefore all hosts share the same clock. Following, each host calls another *updateVmsProcessing* method in all VMs currently deployed on it. At this point, the host also informs the number of MIPS allocated to each VM, which is obtained based on the VM scheduling policy in use.

Next, the VM delegates the update task to the *cloudlet scheduler*, which determines the number of instructions available to each cloudlet running on that particular VM based on the time-shared policy. Finally, the method *updateCloudletFinishedSoFar* is invoked on every *CepQueryCloudlet*, which delegates the simulation to the encapsulated instance of *PlacementExecutor*.

# Appendix C

# CEP as a Service API

*CEPaaS Core* API is mostly composed of CRUD methods for the core entities of the system. The API has been modeled according to the REST and Resource Oriented Architecture paradigm [74], and all data is exchanged in the JSON format [29].

Table C.1 shows the main resources of the API. Most entities are namespaced by the tenant to which they belong. Query and vertex templates have their own namespace, but private templates are accessible only by the user who created them.

Table C.1: *CEPaaS Core* API.

| **Resource** | **HTTP Method** | **Description** |
|---|---|---|
| /querytemplates | GET | Obtain all query templates. |
| | PUT | Create a new query template. |
| /querytemplates/{id} | GET | Obtain details of a query template with the specified id. |
| | POST | Update the query template with the specified id. |
| | DELETE | Delete the query template with the specified id. |
| /vertextemplates | GET | Obtain all vertex templates. |
| /vertextemplates/{id} | GET | Obtain details of a vertex template with the specified id. |
| | DELETE | Delete the vertex template with the specified id. |
| /tenants/{tenantId}/libraries | GET | Obtain a list of all libraries from a tenant. |
| /tenants/{tenantId}/users | GET | Obtain a list of all users from a tenant. |
| | PUT | Create a new user associated with the tenant. |
| /tenants/{tenantId}/users/{id} | GET | Obtain details of a user with the specified id. |
| | POST | Update the user with the specified id. |
| | DELETE | Delete the user with the specified id. |
| /tenants/{tenantId}/queries | GET | Obtain a list of all queries from a tenant. |
| | PUT | Create a new query. |
| /tenants/{tenantId}/queries/{id} | GET | Obtain details of a query with the specified id. |
| | DELETE | Delete a query with the specified id. |
| /tenants/{tenantId}/queries/{id}/start | POST | Start to run a query with the specified id. |
| /tenants/{tenantId}/queries/{id}/stop | POST | Stop a query with the specified id. |
| /tenants/{tenantId}/eventsources | GET | Obtain a list of all event sources from a tenant. |
| | PUT | Create a new event source associated with the tenant. |
| /tenants/{tenantId}/eventsources/{id} | GET | Obtain details of an event source with the specified id. |
| | POST | Update the event source with the specified id. |
| | DELETE | Delete the event source with the specified id. |

# Appendix D

# CEP as a Service Operator Template Definition

This appendix contains the complete definition of the "filter" operator template. An operator template is composed of two main parts: the metadata (Figure D.1) and the implementation (Figure D.2).

The metadata is a JSON document that describes an operator, including its name, parameters, and *AGeCEP* classification. The implementation is a Scala or Java code that implements one of the vertex templates interfaces described in Section 7.5.2. In the code shown in Figure D.2, the method *configure* initializes the operator instance by parsing the "condition" parameter, and the method *process* sends to the vertex successors only events which satisfy the condition specified.

```
{
  "name": "Filter operator",
  "description": "Filter events according to specified condition",
  "parameters": [
    { "name": "condition", "type": "simple",
      "kind": "string", "required": true,
      "description": "Condition to be satisfied"
    }
  ],
  "classification": {
    "type": "processing",
    "shareable": false,
    "duplicable": true, "reqSplit": "random", "reqMerge": "union",
    "combinable": true, "combImpl": "filter", "combParam":  "conjunction",
    "stateful": false, "selectivity": "< 1", "complexity": "n"
  }
}
```

Figure D.1: Filter operator metadata.

```scala
/** FilterOperator companion object. */
object FilterOperator {

  /** Name of the condition parameter. */
  val Condition = "condition"
  def apply() = new FilterOperator()
}

/**
  * Filter operator implementation - removes events that do not satisfy a boolean
  * expression from the event stream.
  */
class FilterOperator extends Operator  {

  import FilterOperator._
  var expression: Expression[Boolean] = null

  /**
    * Initialize the operator instance.
    * @param config Map containing the parameters to be used by this instance.
    */
  override def configure(config: Map[String, Parameter]): Unit = {
    val condition = config(Condition).as[String]

    // parses the "condition" parameter and transform it into an expression
    val parser = ExpressionParser()
    parser.parseBooleanExpression(condition) match {
      case Success(result) => expression = result
      case Failure(ex) => throw new IllegalArgumentException("Invalid expression", ex)
    }
  }

  /**
    * Process an event tuple.
    * @param tuple Tuple to be processed.
    * @return Resulting tuples and destination to which these tuples must be sent.
    */
  override def process(tuple: Tuple): (Tuples, Destination) = {
    // the Operator superclass provides auxiliary methods that helps to define
    // the tuples destination
    if (expression.resolve(tuple.event)) { send(tuple) toAll }
    else { doNotSend(tuple) }
  }
}
```

Figure D.2: Filter operator implementation.

# Wilson Akio Higashino

## EDUCATION

**Dual PhD Degree, Software Engineering / Computer Science** ***2016***
*Western University, London, Canada (Software Engineering)*
*University of Campinas, Campinas, Brazil (Computer Science)*

**Master of Science, Computer Science** ***2006***
*University of Campinas, Campinas, Brazil*

**Bachelor of Science, Computer Science** ***2002***
*University of Campinas, Campinas, Brazil*

## HONORS & AWARDS

• Graduate Student Award for Excellence in Research – Department of Electrical and Computer Engineering, Western University, 2016.
• Graduate Travel Award – Department of Electrical and Computer Engineering, Western University, 2015.
• Award for Outstanding Presentation in Graduate Symposium – Department of Electrical and Computer Engineering, Western University, 2014.
• Graduate Travel Award – Department of Electrical and Computer Engineering, Western University, 2014.
• Ontario Trillium Scholarship (OTS) – Western University, 2012 – 2016.
• PhD Scholarship – CNPq (National Council for Scientific and Technological Development) Brazil, 2011-2012.
• MSc Scholarship – CNPq (National Council for Scientific and Technological Development) Brazil, 2003-2004.

## PUBLICATIONS

### Journal papers

• W. A. Higashino, C. Eichler, M. A. M. Capretz, Luiz F. Bittencourt, T. Monteil. *"Attributed Graph Rewriting for Complex Event Processing Self-Management"*. ACM Transactions on Autonomous and Adaptive Systems, *in press*.

• W. A. Higashino, M. A. M. Capretz, L. F. Bittencourt. *"CEPSim: Modelling and Simulation of Complex Event Processing Systems in Cloud Environments"*. Future Generation Computer Systems, Vol. 65, pp. 122-139, 2016.

• W. A. Higashino, M. A. M. Capretz, M. B. F. de Toledo, L. F. Bittencourt. *"A Hybrid Particle Swarm Optimisation-Genetic Algorithm Applied to Grid Scheduling"*. International Journal of Grid and Utility Computing (IJGUC), Vol. 7(2), pp. 113-129, 2016.

• K. Grolinger, W. A. Higashino, A. Tiwari, M. A. M. Capretz. *"Data Management in Cloud Environments: NoSQL and NewSQL Data Stores"*, Journal of Cloud Computing: Advances, Systems and Application, Springer Open, Vol. 2(22), 2013.

### Conference papers

• W. A. Higashino, M. A. M. Capretz, L. F. Bittencourt. *"CEPSim: A Simulator for Cloud-Based Complex Event Processing"*. Proc. of the 2015 IEEE International Congress on Big Data (BigData Congress), pp. 182-190, New York, USA, 2015.

• S. Wang, W. A. Higashino, M. Hayes, M. A. M. Capretz. *"Service Evolution Patterns"*. Proc. of the 21st IEEE International Conference on Web Services (ICWS), pp. 201-208, Anchorage, USA, 2014.

• K. Grolinger, M. Hayes, W. A. Higashino, A. L'Heureux, D. S. Allison, M. A. M. Capretz. *"Challenges for MapReduce in Big Data"*. Proc. of the 2014 IEEE World Congress on Services (SERVICES), pp. 182-189, Anchorage, USA, 2014.

• W. A. Higashino, C. Eichler, M. A. M. Capretz, T. Monteil, M. B. F. de Toledo, P. Stolf. *"Query Analyzer and Manager for Complex Event Processing as a Service"*. Proc. of the AROSA track of the 23rd IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp. 107-109, Parma, Italy, 2014.

• W. A. Higashino, M. A. M. Capretz, M. B. F. de Toledo. *"Evaluation of Particle Swarm Optimization Applied to Grid Scheduling"*. Proc. of the CDCGM track of the 23rd IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp. 173-178, Parma, Italy, 2014.

• I. Faria, M. Dantas, M. A. M. Capretz, W. A. Higashino. *"Network and Energy-Aware Resource Selection Model for Opportunistic Grids"*. Proc. of the CDCGM track of the 23rd IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp. 167-172, Parma, Italy, 2014.

• R. G. de Oliveira, P. Sigrist, W. A. Higashino, C. O. Becker, C. E. Pagani, J. Silva. *"An Application Server Approach for Number Portability in IMS Networks"*. Proc. of the 7th International Telecommunications Symposium (ITS), Manaus, Brazil, 2010.

• W. A. Higashino, M. B. F. Toledo, M. A. M. Capretz. *"REST and Resource-Oriented Architecture"*. Proc. of International Symposium on Services Science (ISSS 2009), pp. 161-171, Leipzig, Germany, 2009.

### Books

• M. B. F. de Toledo, D. Z. G. Garcia, I. M. S. Gimenes, M. Fantinato, W. A. Higashino, G. C. Silva. *"Business Process Management Systems and the Web Services Technology (Sistemas de Gestão de Processos de Negócio e a Tecnologia de Serviços Web)"*, in Portuguese, 1st ed, Editora Ciência Moderna, 248p, 2013.

### Magazines

• W. A. Higashino, P. Sigrist. *"Boost your Searches (Turbine suas buscas)"*, in Portuguese, Java Magazine, Ed. 112, pp. 44-54, Feb/2013.

• W. A. Higashino, P. Sigrist. *"Understanding the Lucene Library (Conhecendo o Lucene)"*, in Portuguese, Java Magazine, Ed. 104, pp. 14-28, Jun/2012.

• W. A. Higashino, P. Sigrist. *"Java EE and MongoDB in Practice (Java EE com MongoDB na prática)"*, in Portuguese, Java Magazine, Ed. 97, pp. 14-34, Nov/2011.

• W. A. Higashino, P. Sigrist. *"Interacting with Telecom Applications (Interagindo com Aplicações Telecom)*", in Portuguese, Java Magazine, Ed. 93, pp. 28-37, Jul/2011.

• W. A. Higashino, P. Sigrist. *"The Java Role in the Future of Telecommunications (O Futuro das Telecomunicações e o Java)"*, in Portuguese, Java Magazine, Ed. 86, pp. 8-22, Dec/2010.

### Non-referred journal papers

• E. V. Franceschinelli, H. A. Carvalho, W. A. Higashino. *"InterVoIP: Conceptualization of a VoIP Peering Architecture (InterVoIP: Concepção de uma Arquitetura para VoIP Peering)"*, in Portuguese, Cadernos CPQD Tecnologia, Vol. 8, pp. 81-88, 2012.

• W. A. Higashino, L. T. Li, A. L. N. Silva, J. A. Silva, M. F. G. Roscito. *"Evaluation of BPMS as the Orchestration Component of an Integrated Business Processes Architecture for Telecom (Avaliação de BPMS como Componente de Orquestração de uma Arquitetura para Processos Integrados de Negócios de Telecom)"*, in Portuguese, Cadernos CPQD Tecnologia, Vol. 8, pp. 89-106, 2012.

### Invited presentations

• W. A. Higashino. "Machine Learning and Computer Vision", Guest Lecturer, Anhanguera Educacional, Indaiatuba, Brazil, Apr/2010.

• W. A. Higashino. "Developing with Java: Hibernate and Struts", Senac Tour of Information Technology, Votuporanga, Brazil, Oct/2007.

## RELATED WORK EXPERIENCE

**Research Assistant** **Jun/2014 – Aug/2016**

*NSERC-CRD Project with Powersmiths International Corporation – Western University, London, Canada*
*Title: "Cloud Computing Platform for Sustainability Management"*
*Supervisor: Dr. Miriam A. M. Capretz*

**Exchange Researcher** **Jun/2014**

*LAAS-CNRS, Toulouse, France*
*Supervisors: Dr. Thierry Monteil / Dr. Ernesto Exposito*

**Teaching Assistant** **Jan/2013 – Apr/2015**

*Department of Electrical and Computer Engineering, Western University, London, Canada*

**Software Architect** ***Nov/2010 – Dec/2011***
*CPqD – Center for Research & Development in Telecom, Campinas, Brazil*

**Senior Development Analyst** ***Sep/2008 – Nov/2010***
*Venturus Innovation & Technology Center, Campinas, Brazil*

**Software Architect / Software Developer** ***Aug/2002 – Mar/2003 and Mar/2004 – Aug/2008***
*CPqD – Center for Research & Development in Telecom, Campinas, Brazil*

**Software Development Intern** **Dec/2000 – Jul/2002**
*A-HAND Laboratory, Institute of Computing, State University of Campinas, Campinas, Brazil*