StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine

Vincenzo Gulisano

To cite this version:

Vincenzo Gulisano. StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine. Distributed, Parallel, and Cluster Computing [cs.DC]. Universidad Politécnica de Madrid, 2012. English. NNT : . tel-00768281

HAL Id: tel-00768281
https://theses.hal.science/tel-00768281

Submitted on 21 Dec 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DEPARTAMENTO DE LENGUAJES Y SISTEMAS INFORMÁTICOS E INGENIERÍA DE SOFTWARE

Facultad de Informática
Universidad Politécnica de Madrid

Ph.D. Thesis

StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine

Author

Vincenzo Massimiliano Gulisano
M.S. Computer Science

Ph.D. supervisors

Ricardo Jiménez Peris
Ph.D. Computer Science

Patrick Valduriez
Ph.D. Computer Science

December 2012

Acknowledgments

I would like to thank my supervisor Ricardo Jiménez Peris, my thesis co-director Patrick Valduriez and Marta Patiño-Martínez for their help. Thanks also to all the lab colleagues (especially Mar, Damián, Paco and Claudio) and the people I had the opportunity to work with (especially Zhang, prof. Marina Papatriantafilou, Zoe and prof. Kostas Magoutis). Special thanks go to Rocío, my friends and my family.

Abstract

In recent years, applications in domains such as telecommunications, network security or large scale sensor networks showed the limits of the traditional store-then-process paradigm. In this context, Stream Processing Engines emerged as a candidate solution for all these applications demanding for high processing capacity with low processing latency guarantees. With Stream Processing Engines, data streams are not persisted but rather processed on the fly, producing results continuously.

Current Stream Processing Engines, either centralized or distributed, do not scale with the input load due to single-node bottlenecks. Moreover, they are based on static configurations that lead to either under or over-provisioning. This Ph.D. thesis discusses *StreamCloud*, an elastic parallel-distributed stream processing engine that enables for processing of large data stream volumes. *StreamCloud* minimizes the distribution and parallelization overhead introducing novel techniques that split queries into parallel subqueries and allocate them to independent sets of nodes. Moreover, *StreamCloud* elastic and dynamic load balancing protocols enable for effective adjustment of resources depending on the incoming load. Together with the parallelization and elasticity techniques, *StreamCloud* defines a novel fault tolerance protocol that introduces minimal overhead while providing fast recovery. *StreamCloud* has been fully implemented and evaluated using several real word applications such as fraud detection applications or network analysis applications. The evaluation, conducted using a cluster with more than 300 cores, demonstrates the large scalability, the elasticity and fault tolerance effectiveness of *StreamCloud*.

Keywords: Data Streaming, Stream Processing Engine, Scalability, Elasticity, Load Balancing, Fault Tolerance

Resumen

En los útimos años, aplicaciones en dominios tales como telecomunicaciones, seguridad de redes y redes de sensores de gran escala se han encontrado con múltiples limitaciones en el paradigma tradicional de bases de datos. En este contexto, los sistemas de procesamiento de flujos de datos han emergido como solución a estas aplicaciones que demandan una alta capacidad de procesamiento con una baja latencia. En los sistemas de procesamiento de flujos de datos, los datos no se persisten y luego se procesan, en su lugar los datos son procesados al vuelo en memoria produciendo resultados de forma continua.

Los actuales sistemas de procesamiento de flujos de datos, tanto los centralizados, como los distribuidos, no escalan respecto a la carga de entrada del sistema debido a un cuello de botella producido por la concentración de flujos de datos completos en nodos individuales. Por otra parte, éstos están basados en configuraciones estáticas lo que conducen a un sobre o bajo aprovisionamiento. Esta tesis doctoral presenta StreamCloud, un sistema elástico paralelo-distribuido para el procesamiento de flujos de datos que es capaz de procesar grandes volúmenes de datos. StreamCloud minimiza el coste de distribución y paralelización por medio de una técnica novedosa la cual particiona las queries en subqueries paralelas repartiéndolas en subconjuntos de nodos independientes. Ademas, StreamCloud posee protocolos de elasticidad y equilibrado de carga que permiten una optimización de los recursos dependiendo de la carga del sistema. Unidos a los protocolos de paralelización y elasticidad, StreamCloud define un protocolo de tolerancia a fallos que introduce un coste mínimo mientras que proporciona una rápida recuperación. StreamCloud ha sido implementado y evaluado mediante varias aplicaciones del mundo real tales como aplicaciones de detección de fraude o aplicaciones de análisis del tráfico de red. La evaluación ha sido realizada en un cluster con más de 300 núcleos, demostrando la alta escalabilidad y la efectividad tanto de la elasticidad, como de la tolerancia a fallos de StreamCloud.

Palabras clave: Data Streaming, Sistemas de procesamiento de flujos de datos, Escalabilidad, Elasticidad, Equilibrado de Carga, Tolerancia a fallos

Declaration

I declare that this Ph.D. Thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Vincenzo Massimiliano Gulisano*)

Table of Contents

Table of Contents i

List of Figures v

List of Tables ix

I INTRODUCTION 1

Chapter 1 Introduction 3

1.1 Application scenarios that motivated data streaming . . . 3

1.2 Requirements of data streaming applications . . . 5

1.3 From DBMS to SPEs . . . 9

1.3.1 Limitations of pioneer SPEs . . . 10

1.4 Contributions . . . 12

1.5 Document Organization . . . 13

II DATA STREAMING BACKGROUD 15

Chapter 2 Data Streaming Background 17

2.1 Data Streaming Model . . . 17

2.1.1 Data Streaming Operators . . . 18

2.2 Continuous Query Example . . . 26

2.3 Table Operators . . . 28

III STREAMCLOUD PARALLEL-DISTRIBUTED DATA STREAMING 33

Chapter 3 *StreamCloud* Parallel-Distributed Data Streaming 35

3.1 Stream Processing Engines Evolution . . . 35

3.2 Parallelization strategies . . . 38

3.2.1 Operators parallelization 43
3.2.1.1 Load Balancers 45
3.2.1.2 Input Mergers 48
3.3 *StreamCloud* parallelization evaluation 50
3.3.1 Evaluation Setup 50
3.3.2 Scalability of Queries 50
3.3.3 Scalability of Individual Operators 51
3.3.4 Multi-Core Deployment 57

IV STREAMCLOUD DYNAMIC LOAD BALANCING AND ELASTICITY 59

Chapter 4 StreamCloud Dynamic Load Balancing and Elasticity 61
4.1 *StreamCloud* Architecture 61
4.2 Elastic Reconfiguration Protocols 64
4.2.1 Reconfiguration Start 65
4.2.2 Window Recreation Protocol 68
4.2.3 State Recreation Protocol 71
4.3 Elasticity Protocol 72
4.4 *StreamCloud* Dynamic Load Balancing and Elasticity Evaluation 75
4.4.1 Elastic Reconfiguration Protocols 75
4.4.1.1 Dynamic Load balancing 77
4.4.1.2 Self-Provisioning 77

V STREAMCLOUD FAULT TOLERANCE 81

Chapter 5 *StreamCloud* Fault Tolerance 83
5.1 Existing Fault Tolerance solutions 83
5.2 Intuition about Fault Tolerance protocol 88
5.3 Components involved in the Fault Tolerance protocol 90
5.4 Fault Tolerance protocol 92
5.4.1 Active state 94
5.4.2 Failed state 96
5.4.3 Failed while reconfiguring state 100
5.4.4 Recovering state involved in previous reconfigurations 100
5.4.5 Multiple instance failures 101
5.5 Garbage collection 101

5.5.1 Time-based windows 102
5.5.2 Tuple-based windows 102
5.6 Evaluation 103
5.6.1 Evaluation Setup 103
5.6.2 Runtime overhead 105
5.6.3 Recovery Time 107
5.6.4 Garbage Collection 109
5.6.5 Storage System Scalability Evaluation 110

VI VISUAL INTEGRATED DEVELOPMENT ENVIRONMENT 113

Chapter 6 Visual Integrated Development Environment 115
6.1 Introduction 115
6.2 Visual Query Composer 116
6.3 Query Compiler and Deployer 119
6.4 Real Time Performance Monitoring Tool 124
6.5 Distributed Load Injector 126

VII STREAMCLOUD - USE CASES 129

Chapter 7 *StreamCloud* - Use Cases 131
7.1 Introduction 131
7.2 Fraud Detection in cellular telephony 131
7.2.1 Use Cases 132
7.3 Fraud Detection in credit card transactions 138
7.3.1 Use Cases 138
7.4 Security Information and Event Management Systems 143
7.4.1 SIEM directives 143
7.4.2 Directives translation 147
7.4.3 Directive translation example 148

VIII RELATED WORK 153

Chapter 8 Related Work 155
8.1 Introduction 155
8.2 Pioneer SPEs 155

8.2.1 The Borealis project . . . 155
8.2.2 STREAM . . . 157
8.2.3 TelegraphCQ . . . 158
8.2.4 NiagaraCQ . . . 159
8.2.5 Cougar . . . 159
8.3 State Of the Art SPEs . . . 160
8.3.1 Esper . . . 160
8.3.2 Storm . . . 160
8.3.3 StreamBase . . . 161
8.3.4 IBM InfoSphere . . . 161
8.3.5 Yahoo S4 . . . 161
8.3.6 Microsoft StreamInsight . . . 162
8.4 StreamCloud related work . . . 162
8.4.1 Load Shedding and Operators Scheduling protocols . . . 162
8.4.2 Parallelization techniques . . . 165
8.4.3 Load Balancing techniques . . . 167
8.4.4 Elasticity techniques . . . 170
8.4.5 Fault Tolerance techniques . . . 172

IX Conclusions **179**

Chapter 9 Conclusions **181**

X APPENDICES **183**

Appendix A The Borealis Project - Overview **185**
A.1 Query Algebra . . . 185
A.2 Operators extensibility . . . 191
A.3 Borealis Tuples Processing Paradigm . . . 193
A.4 Borealis Application Development Tools . . . 195

Bibliography **199**

List of Figures

1.1 DBMSs information processing. 6
1.2 SPEs information processing. 8

2.1 Examples of different windows models 20
2.2 Sample evolution of time based aggregate operator 24
2.3 Sample evolution of tuple based aggregate operator 25
2.4 High Mobility fraud detection query 27

3.1 SPE evolution 37
3.2 Fan-out overhead for a single operator instance 40
3.3 Query Parallelization Strategies 41
3.4 Query Parallelization in *StreamCloud* 43
3.5 Cartesian Product Sample Execution 47
3.6 Cartesian Product Sample Execution 48
3.7 Query used for the evaluation. 52
3.8 Parallelization strategies evaluation 53
3.9 Parallel Aggregate operator evaluation. 54
3.10 Parallel Map operator evaluation. 54
3.11 Parallel Join operator evaluation. 55
3.12 Parallel Cartesian Product operator evaluation. 56
3.13 Join maximum throughput vs. number of *StreamCloud* instances per node. 58

4.1 Elastic management architecture. 63
4.2 Example of tuple contributing to several windows 65
4.3 Example execution of reconfiguration protocols shared prefix 67
4.4 Example of which windows are managed by old owner or new owner instances during Window Recreation protocol 68
4.5 Sample reconfigurations. 70
4.6 Sample execution of the buckets assignment algorithm 74
4.7 Evaluation of the elastic reconfiguration protocols. 76

4.8 Elastic Management - Dynamic Load Balancing. 78
4.9 Elastic Management - Provisioning Strategies. 80

5.1 Active Standby Fault Tolerance 84
5.2 Passive Standby Fault Tolerance 85
5.3 Upstream Backup Fault Tolerance 85
5.4 Example of fault tolerance for aggregate operator 89
5.5 StreamCloud fault tolerance architecture 91
5.6 Bucket state machine. 93
5.7 Linear Road. 103
5.8 Input rate evolution of data used in the experiments 106
5.9 Latency measured at subcluster 0 106
5.10 Latency measured at subcluster 1 107
5.11 Deploy and State Recovery times for changing subcluster sizes 108
5.12 Deploy and State Recovery times for changing number of buckets 109
5.13 Garbage Collection evaluation 111
5.14 Storage System Scalability Evaluation 111

6.1 Abstract query definition 118
6.2 Compiling an abstract query into its parallel-distributed counterpart 121
6.3 Subquery partitioning 122
6.4 SC Statistics Monitor architecture 125
6.5 snapshot of Statistics Visualizer presenting the statistics of the Aggregate operator . . 126

7.1 Consumption Control Query 133
7.2 Overlapping Calls Query 135
7.3 Blacklist Query 136
7.4 Improper Fake Transaction 139
7.5 Restrict Usage Query 141
7.6 Rules firing and directives cloning example 144
7.7 OSSIM directive translation guidelines 147
7.8 OSSIM Directive translation to continuous query 149

8.1 Box-Splitting example 166
8.2 Box Sliding example 168

A.1 High Mobility fraud detection query 186
A.2 *StreamCloud* threads interaction 194

List of Tables

2.1 Example tuple schema 18

3.1 Cluster setup 50

4.1 Parameters used by elasticity protocols 65
4.2 Static vs. Elastic configurations overhead. 79

5.1 Sample input and output tuples for operator A_2 90
5.2 Variables used in algorithms 94
5.3 Linear Road tuple schema 104

6.1 VQC Operators legend 119

7.1 CDR Schema 133
7.2 Credit Card Fraud Detection Tuple Schema 139
7.3 OSSIM rule parameters 145
7.4 OSSIM input tuples schema 148
7.5 OSSIM output tuples schema 149

Part I

INTRODUCTION

Chapter 1

Introduction

In this chapter we provide an introduction to data streaming and to Stream Processing Engines (SPEs). We first introduce some of the application scenarios that motivated the research in the data streaming field. We discuss which are the most important requirements of applications that perform on-line data analysis and discuss why previous existing solutions like Data Base Management Systems (DBMSs) are not adequate and are being replaced by SPEs. We provide a short overview of pioneer SPEs, discussing how they overcome the limitations of previous existing solutions. Finally, we discuss which are the limitations of existing streaming applications and how they motivated *StreamCloud*, the parallel-distributed SPE presented in this thesis.

1.1 Application scenarios that motivated data streaming

In this section we present the application scenarios that, during the last decade, motivated the research in the field of data streaming. As presented in [ScZ05], [BBD$^+$02], [CCC$^+$02] and [ACC$^+$03] these scenarios include both existing and emerging applications that share the same need for high processing throughput with low latency constraints. Among others, examples of these scenarios include fraud detection, network security, financial markets, large scale sensor networks and military applications. In the following, we present each motivating scenario motivating is need for high processing capacity with low processing latency.

Fraud detection in cellular telephony. Fraud detection applications in cellular telephony require the ability of processing data whose size is in the order of tens or hundreds of thousand mes-

sages per second. As reported by the Spanish commission for the Telecommunications market [cmt], the number of mobile phones in Spain exceeds the 56 million units. Hence, a sample fraud detection application that is used to spot mobile phones appearing to maintain more than one communication at time must process loads that can go from few tens of thousand messages per second to millions of messages per second (e.g., in correspondence with mass events such as national feasts or religious events). In cellular telephony fraud detection applications, the need for low latency processing arises from the fact that the faster the detection, the lower the amount of money the fraudster costs to the company. That is, if a cloned number is detected after one week, the money spent during that week is lost by the company.

On-line trading. scenarios involving credit card transactions are another example of applications demanding for high processing capacity and very low processing latency. As reported by The Nilson Report [Nil], the projection for the year 2012 says that the number of debit cards holders in the U.S. is approximately of 191 million people, for a total of 530 million debit cards whose estimated purchase volume is 2089 billion dollars. With respect to applications that involve credit cards transactions it is even more imperative to provide low latency guarantees as such applications must comply with strict time limitations that are often smaller than one second.

Financial Markets applications. Another example of existing application demanding for processing of big volume data with low delay is related to Financial Markets. As discussed in [ScZ05], the Options Price Reporting Authority (OPRA [OPR]) estimated a required processing rate of approximately $120,000$ messages per second during year 2005. The growing rate has been so fast that the required capacity exceeded the 1 million messages per second in 2008. As for the on-line trading applications, financial market applications demand for very low processing latencies that are often below the one second threshold.

Network traffic monitoring. With respect to applications monitoring traffic network (e.g., Intrusion Detection applications), the need for high processing capacity arises from the huge volume of data moving through the Internet. As presented by CAIDA (the Cooperative Association for Internet Data Analysis [cai]), an Internet Service Provider (ISP) usually sustains traffic volumes that is at least in the range of tens of Gigabytes per second. With respect to applications that analyze the traffic to detect possible threats, the need for high processing capacity arises not only from the huge traffic volume that must be processed in real-time in order to block possibly harmful events but also from the type of computations run over the data, that usually define not trivial aggregation and comparison of on-line and historic data.

Sensor networks, RFID networks. Several emerging applications demanding for high processing capacity and low processing latency arise from sensor networks. These applications are becoming popular due to the decreasing costs of sensors that, nowadays, allow for the creation of

big networks of such devices with small deployment costs. Examples of such scenarios include military applications where soldiers (or vehicles) can be equipped with GPS devices to monitor their location. Another sample scenario is RFID tagging for animal tracking or RFID tagging of products being produced or sold in big industry.

All the presented application scenarios share the same need for high processing capacity and low processing latency. Furthermore, they share the same need for processing of data whose behavior evolves over time. This aspect is important as it implies several challenges that must be addressed. A first consideration that can be made about data behavior is that its volume can abruptly change over time. As introduced with respect to cellular telephony fraud detection applications, the number of records that must be processed can suffer abrupt changes in correspondence with national events. The traffic behavior not only involves changes in the overall volume, but also in its distribution. As en example, the number of phone calls made (or received) by a mobile phone can vary a lot over time. Another important aspect of the data processed by the presented application scenarios is the possible presence of incomplete and out-of-order data. Consider, as an example, a data streaming application used to process data collected from a sensor network. In such scenario, part of the data produced by sensor might disappear (e.g., when a sensor runs out of battery). In this case, the application should still consume the data produced by the other sensor and produce the desired result. Similarly, a sensor might experience some delay in the forwarding of the produced data. In this case, the data streaming application should address the possibility of late processing of information in order to correct imprecise results computed before.

1.2 Requirements of data streaming applications

In this section, we discuss why pioneer SPEs do not address adequately the requirements of the application scenarios that motivated the research in the data streaming field (i.e., high processing capacity and low processing latency).

For decades, DBMSs have been used to store and query data. Modern DBMSs provide distributed and parallel processing of data; furthermore, they are designed to tolerate faults. One of the reasons why DBMS are so popular comes from their powerful language (e.g., SQL), that allows for complex manipulation of data by means of a set of well known basic commands. Nevertheless, DBMSs are not adequate for the application scenarios introduced in the previous section. The reason is that the primarily goal of DBMSs is data persistence rather than data querying. That is, they are designed to maintain efficiently collections of data that is accessed and aggregated only when a query is issued by a user (or accessed by queries triggered periodically). This data manipulation paradigm incurs in high overheads when applied to applications that demand for high processing capacity with low processing latency.

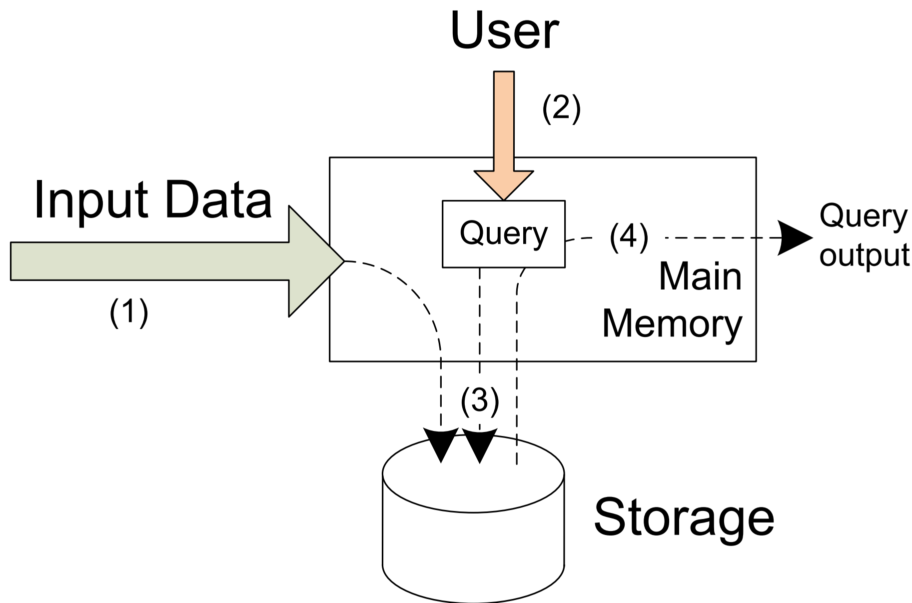


Figure 1.1: DBMSs information processing.

Figure 1.1 presents a high-level overview of how data is stored and processed by a DBMS. Input data is continuously stored while being received (1). The input messages (we refer to them as *tuples*) contain information that is persisted into *relations*. As an example, in a scenario where a DBMS is being used to maintain the information related to what is being sold in a shop, a relation might contain records like *Product ID*,*Quantity* and *Price*. Each time something is purchased by someone, a new record is stored in the relation for each item that has been bought (*Product ID*), specifying also how many units (*Quantity*) and the overall price (*Price*). When a request to process a query provided by a user is received (2), the query is installed and data is read from the storage (3). Finally, data is processed and the query result is outputted (4). With respect to the previous example, a possible query might be issued to know how many units of a given product have been sold during the last six months.

As discussed in [ScZ05], this approach does not match the requirements of the applications introduced in the previous section. With respect to the presented data streaming requirements, we discuss now the ones that are clearly not addressed by DBMSs.

The most important consideration is that, in a data streaming application, new data is useful if it is used to update the computation being run over it. That is, data is not sent to a processing node in order to be stored; rather, data is sent in order to produce new results as soon as possible. This requirement implies that a query is no longer something that is sporadically executed by a user (or periodically triggered). On the contrary, a query is running continuously and its computation is updated any time new information is available. In data streaming, this type of query is referred to as *continuous query*.

As an example, consider a scenario where a sensor network in used to monitor the temperature of the rooms of a building in order to generate an alarm in case of a fire (i.e., whenever the temperature of a room exceeds a given threshold). It is easy to see that the presence of a fire should be detected as fast as possible. That is, each temperature measurement report generated by any sensor in the building should be checked as soon as possible in order to detect a possible fire. The requirement of processing continuously the incoming data by means of a *continuous query* implies that the operations run for each incoming tuple should be kept as low as possible in order to reach for a higher processing capacity.

If we consider how the architecture of traditional DBMSs can be modified in order to reduce the per-tuple processing latency, the first idea is to remove the persistence of each incoming message. The removal of the persistence reduces the per-tuple processing latency significantly as writes to and reads from persistent storage take significantly longer than accesses to main memory. This modification introduces several new challenges about how data is processed. The first challenge relies in the fact that the available memory is smaller than the available storage space; hence, not all the information can be maintained. There is need for some mechanism to maintain only part of the information relevant to the continuous queries running in the system. It should be noticed that the requirement of maintaining only part of the information is an intrinsic need consequence of the nature of the problem. With respect to the example of the sensor network monitoring the temperature of a building, temperature reports will be generated continuously; hence, if all the temperature reports are maintained, they will always eventually saturate the available memory, no matter how big the latter is. As we will discuss in the following chapter, different models exists to maintain only portion of the incoming information. All of them share the same idea that recent information is usually more relevant than old one. When only a portion of the information is maintained, a new challenge arises with respect to how to compute blocking operations with partial data (e.g., in order to compute the average of a measured value, the query first needs to known all the measurements from which to compute it). The idea is to compute functions over portions of the incoming data (referred to as *windows*). With respect to the example, the user can be interested in monitoring the average temperature of each room during the last 30 seconds.

Figure 1.2 presents how information should be processed by an SPE. Input information is processed directly by a continuous query (1). With respect to the example of the temperature monitoring, each measurement generated by a sensor is sent to the continuous query in charge of detecting fires. For each incoming message, the query updates its internal state (2). In the example, each time a measurement from a given room is received, the value of the average temperature is update removing the contribution of the measurements older than 30 seconds and adding the contribution of the last received tuple. Finally, if it is the case, an output is generated by the continuous query (3). In the example, the updated average value of a room temperature is checked against a given threshold and,

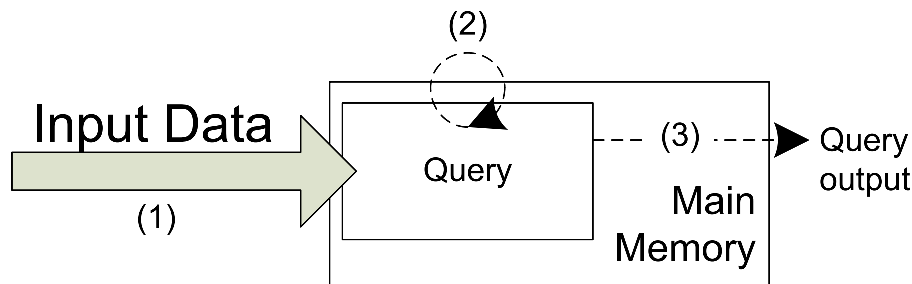


Figure 1.2: SPEs information processing.

if it exceeds it, an alarm is generated.

Scenarios such the one present in the example, where an application checks for specific conditions happening while new data is being processed have been addressed in the past by hand coded solutions designed to specifically compute the desired computation. Even if such ad hoc solutions provide high processing capacity, they are hard to port between platforms and hard to maintain. This observation is very important with respect to SPEs. In fact, one of the requirements in data streaming is that the language used to express queries should be powerful and simple, to ease the porting of existing DBMS applications to SPEs and to avoid to recur to hand coded solutions. As we will present in the following section, one of the data streaming research topics has been the definition of a powerful yet simple language to specify how data should be processed. Two main ways of defining continuous queries have been introduced, one defines them as graphs of processing units (specifying directly how information should flow among the units that manipulate it) while the other extends the SQL language enriching it with data streaming capabilities.

The last requirement we discuss in this section is related to the need of some kind of persistence for part of the information processed by an SPE. Even if we discussed why the data streaming processing pattern excludes the persistence of each incoming tuple before its processing, it is clear that it is always desirable to maintain part of the information. For instance, in the example of the temperature monitoring, the user might be interested in maintaining information related to all the alarms that are generated by the system. This requirement is even clearer when defining continuous queries where real-time information is being compared with historic information. For example, let us consider a continuous query defined to compute the highest temperature experienced during the last 200 days. In this case, it is easy to see that the application will maintain a record for each day and that these records will be loaded by the continuous query to compare it with the real-time information. This requirement should be addressed defining some combined use of SPEs and lightweight DBs (e.g., in memory DBs) or relaying in traditional DBs when write and read requests have very low frequency (i.e., when the introduced overhead is negligible).

1.3 From DBMS to SPEs

In this section we give a short overview of some of the pioneer SPEs prototypes. We present how they marked an evolution with respect to previous existing solutions overcoming their limitations. We also provide an overview of the evolution of these pioneer SPEs. Finally, we discuss which are their limitations and introduce the challenges that motivated our work.

Starting approximately from the year 2000, several SPE prototypes have emerged. These prototypes have been designed either as improvements of DB based solutions (i.e., solutions that still rely on DBs) either as “fresh" solutions that do not rely on DBs.

With respect to the prototype applications that were designed to improve DB based solutions to fit with the requirements of data streaming, we introduce *Cougar* [BGS01], *TelegraphCQ* [CCD+03] [Des04] and *NiagaraCQ* [NDM+01] [CDTW00].

The *Cougar* research project focused on sensor databases, where long running queries are issued to combine live data from a sensor network with some stored data. Even if *Cougar* does not mark a step in the evolution of SPEs as it still heavily relies on DBs, the authors introduce an important aspect of data streaming applications: the need for distributed and local processing of data. That is, they discuss why an approach where all the sensor data are initially collected by a single node and then processed by it performs worse than a solution where sensor data is processed locally continuously (i.e., discarding unneeded information or aggregating it) and routed up to a central node computing the result of a given query.

NiagaraCQ is one of the first research projects that extends a query language to adapt it to continuous queries. The project focused on how to efficiently run continuous queries over XML data files. The XML-QL query language has been extended in order to define long standing queries that update their computation periodically. Furthermore, the project focused on how to define incremental evaluation of the different functions being computed and incremental grouping of the overlapping computations of different queries.

The *TelegraphCQ* research project is one the first projects that marks a significant step in the evolution of SPEs as it relies on DBs to persist information but introduces separate data processing units that manipulate the data. That is, relations are used as a glue between operators that are designed to manipulate records and that communicate with a dedicated API (named *Fjord*). Different modules are introduced in *TelegraphCQ*: *Ingress* and *Caching* modules have been introduced to collect records from external applications, *Adaptive Routing* modules are used to distribute tuples among modules and *Processing* modules have been designed to manipulate tuples flowing through the system.

In parallel with *Cougar*, *NiagaraCQ* and *TelegraphCQ*, two research projects have focused on SPEs that do no rely on DBs to store or manipulate data: *STREAM* [ABB+04] [ABW06] and *Aurora* [CCC+02] [ACC+03] [BBC+04] (evolved later to *Borealis* [AAB+05b] [ABC+05]).

With *STREAM*, research has focused on how to ease the migration from DBMS to SPEs. One of the most important aspect of STREAM is the introduction of the CQL query language, an evolution of the SQL language, enriched with data streaming related operations. As introduced in Section 1.2, one of the requirements of data streaming applications is to define operations over portions of the incoming information (referred to as *windows*). The idea is to represent streams by means of relations so that data manipulation can be defined using SQL-like commands that operate on them. That is, the relevant portion of incoming information that is queried is maintained like a relation and its records are queried by means of traditional DBMSs queries. Finally, the resulting relation is converted to a stream of tuples outputted to the end user.

The last pioneer SPE we introduce is the centralized SPE *Aurora* and its distributed SPE evolution *Borealis*. The project proposed a new query language where queries are defined as Directed Acyclic Graphs (DAG) of operators, where each node in the graph represents a processing units that consumes input tuples and produces output tuples while edges define how tuples flow among processing units. Being centralized, Aurora only allows for the definition of queries that are entirely executed at the node where Aurora is running. The initial centralized SPE Aurora has been the base of the *Borealis* Project (intermediate steps in the evolution of *Aurora* SPEs include *Aurora** and *Medusa* [CBB+03] [SZS+03]). The *Borealis* project is one of the first distributed SPEs. With a distributed SPE, a single query can be executed using multiple nodes. That is, a graph of operators that define how input data should be processed in order to produce the desired results can be run at multiple nodes, assigning different operators at different nodes.

Being an open source project and one of the first distributed SPE, and due to its flexibility in defining new operators, the *Borealis* project has been used as the base of *StreamCloud*, the SPE being presented in this thesis.

1.3.1 Limitations of pioneer SPEs

In this section we focus on the limitations of the most important pioneer SPEs and we introduce the challenges that motivated our work.

The main limitation of both centralized and distributed SPEs is that they both concentrate data streams to single nodes. A query running at a centralized SPE receives and processes all the data at the same node. In the case of a distributed SPE, even if different operators of a single query are deployed at different nodes, each data stream is processed by a single machine (i.e., all the input data stream is routed to the machine running the first operator of the query, the stream generated by the first operator is concentrated to the node running the second operator, and so on). This single-node bottleneck implies that centralized and distributed SPEs do not scale, their processing capacity is bound to the capacity of a single node. In order to overcome this limitation, data streams should not be processed by a single node. Sources providing the information for data streaming applications are

often distributed (e.g., antennas generating reports about mobile phone calls, sensor measuring the temperature of a building, and so on). In order to attain a scalable SPE, the information from these distributed data sources needs must always to be processed in parallel.

The parallel processing of the information provided by distributed data sources gives room to several challenges. First of all, there is need for a parallelization technique for queries (and its operators). The parallelization technique must guarantee *semantic transparency*, meaning that results produced by a parallel query are equivalent to the ones produced by its centralized (or distributed) counterpart.

The second challenge arises from the fact that previous solutions have been designed as static solutions. That is, how queries (and operators) are assigned to the available nodes is decided statically at the time when queries are deployed. This static configuration is not appropriate due to the highly variable nature of data streams. As introduced in Section 1.1, the valley and peak loads of data streams can be orders of magnitude distant. These problems might affect centralized, distributed and parallel SPEs. That is, a given deployment for a parallel query can be inadequate with respect to the current load. As an example, the number of available nodes can be lower than the required one due to a peak in the system input load (*under-provisioning*). On the other hand, the number of assigned nodes can be higher than the needed one due to a decrease in the system load (*over-provisioning*). It should be noticed that both under-provisioning and over-provisioning can affect the same configuration at different times, depending on the fluctuations of the system input load. To overcome this limitation, an SPE should be elastic. That is, nodes should be provisioned or decommissioned depending on the current system load to minimize the used computational resources while guaranteeing the Quality of Service (QoS). It is important to notice that elasticity must be combined with dynamic load balancing in order to be effective. That is, new nodes should be allocated only if the current load cannot be processed by the assigned nodes as a whole. On the contrary, nodes could be provisioned simply due to uneven distribution of the overall load.

The third challenge is related to the tolerance of possible failures. Fault tolerance is a common requirement for distributed applications as the increase in the number of nodes usually translates in a higher probability of faults. Fault tolerance protocols have been already introduced for distributed SPEs. However, existing protocols do not cope with parallel-distributed SPEs with elastic and dynamic load balancing capabilities. The first limitation is that existing protocols are designed for static configurations and cannot cope with continuously evolving configurations. The second limitation is due to the fact that they do not cover failures that might happen while the system is being reconfigured (i.e., a failure happening while a node is offloading part of its processing to a less loaded node).

The last challenge that motivated our work is the need for a development environment that eases the definition, implementation and execution of parallel queries. That is, since our goal is transparent parallelization (i.e., the execution of a parallel query is equivalent to the execution of its centralized counterpart), the user should only define an *abstract* centralized query and asked to provide minimum

information about how to parallelize it (e.g., provide information about which are the available nodes that can be used to run the parallel query).

All the challenges that motivated our work have been studied and addressed in *StreamCloud*: a parallel-distributed SPE that provides dynamic load balancing, elasticity and fault tolerance. We summarize in the following section which are the main contributions of the proposed work.

1.4 Contributions

This thesis presents *StreamCloud*: a parallel-distributed SPE with dynamic load balancing, elastic and fault tolerance capabilities. The contribution of the presented work can be summarized as follows:

- A new parallelization technique for parallel data streaming queries and operators running in a shared nothing cluster. The technique provides both syntactic and semantic transparent parallelization. Semantic transparency guarantees that the results produced by a parallel-distributed query are equivalent to the ones produced by the original centralized query. Syntactic transparency allows the user to define queries providing the same information provided to define centralized queries with only additional information about the nodes at which to deploy it. The parallelization technique has been designed, implemented and evaluated for the basic data streaming operators and also for complex queries.

- Dynamic load balancing and elastic capabilities for data streaming operators. Load balancing is triggered dynamically as a consequence of a uneven distribution of the processing among the running node. When the assigned nodes as a whole cannot cope with the incoming load, *StreamCloud* automatically provisions new nodes. We define load-aware provisioning so that the number of provisioned nodes is computed based on the current load and number of assigned nodes. Decommissioning of nodes is triggered each time the load can be managed by less nodes than the assigned ones.

- Both dynamic load balancing and elastic capabilities rely on a dynamic reconfiguration protocol to transfer the processing between the nodes. Two dynamic reconfiguration protocols have been proposed with different trade-offs. The first protocol has been designed to minimize the amount of information exchanged between nodes and fits well when working processing involves small state. On the other hand, the second protocols aims at completing the state transfer as soon as possible and fits better with processing that involves bigger states.

- An innovative fault tolerance protocol for parallel-distributed SPEs. The proposed protocol addresses aspects that were not previously considered by other fault tolerance protocols such how to tolerate failures happening while reconfiguring the system (i.e., failures happening while

the load is being balanced or during elasticity reconfigurations). We provide details about how the fault tolerance protocol has been designed and implemented and we provide a through evaluation for it.

- A complete Visual Integrated Development Environment (IDE) that has been designed to ease the interaction of the user with *StreamCloud*. This IDE allows the user to visually compose a query via a drag-and-drop interface (i.e., like for a centralized query) and to compile it to its parallel-distributed counterpart just specifying additional information about the available nodes for the initial deployment of the query. Furthermore, the user can specify if elasticity should be provided just defining which nodes should be assigned at deploy time and which nodes can be provisioned if necessary. Furthermore, the IDE provides a monitoring tool with a web based interface to monitor the state of the queries running at *StreamCloud* (i.e., input and output throughput, CPU consumption, deployed queries, and so on). Finally, the IDE provides a parallel data injector to ease benchmarking and adapters to read data from sources like text or binary data files.

- Finally, we present several real world use cases that have been studied and implemented as continuous queries run by *StreamCloud*. We consider fraud detection applications in the context of cellular telephony, fraud detection applications in the context of on-line credit card transactions, applications related to the detection and mitigation from Distributed Denial of Service (DDoS) attacks and applications related to Complex Event Processing (CEP) systems.

1.5 Document Organization

The rest of this document is divided into 8 chapters and it is organized as follows:

- *Chapter 2, Data Streaming Background* introduces data streaming basic concepts like streams, operators and continuous queries providing both definitions and examples.

- *Chapter 3, Stream Processing Engine Parallelization* presents how pioneer SPEs evolved and the limitations that motivated our research. The chapter also introduces *StreamCloud* parallelization technique and provides a through evaluation of it with respect to both operators and queries.

- *Chapter 4,* StreamCloud *Dynamic Load Balancing and Elasticity* presents the protocols that have been designed for state transfer, dynamic load balancing and elasticity. The chapter provides a through evaluation of all the proposed protocols.

- *Chapter 5,* StreamCloud *Fault Tolerance* presents the protocols for *StreamCloud* fault tolerance mechanism. As for the previous chapters, we present how the protocols have been designed and implemented and we include a through evaluation of them.
- *Chapter 6, Visual Integrated Development Environment* presents the IDE that has been designed and developed together with *StreamCloud* in order to ease the interaction with the user.
- StreamCloud *use cases* Chapter 7 introduces some of the use cases that have been used together with *StreamCloud*.
- *Chapter 8, Related Work* presents the related work.
- *Appendix A, The Borealis Project* provides an overview of the borealis SPE, the prototype upon which *StreamCloud* has been build.

Part II

DATA STREAMING BACKGROUD

Chapter 2

Data Streaming Background

This chapter introduces data streaming basic concepts such streams (the unbounded sequences of tuples received by the Stream Processing Engine), data streaming operators (the base units consuming and producing streams tuples) and continuous queries (graphs of interconnected operators that allow for rich, real-time analysis of data). We conclude the chapter presenting a sample query used to spot frauds in cellular telephony applications.

2.1 Data Streaming Model

A data stream S is an unbounded, append-only sequence of tuples. All tuples $t \in S$ share the same schema, composed by fields $(F_1, F_2, \ldots, F_n)$. We refer to field F_i of tuple t as $t.F_i$. Each field is defined by its name and data type. In our model, we assume that, for any schema, a field $ts \in (F_1, F_2, \ldots, F_n)$ represents the time when the tuple has been created. Field ts provides a time dimension for each tuple. Furthermore, it allows for time based ordering of tuples. We assume tuples belonging to any stream generated by a data source to have non-decreasing ts values. We also suppose data sources have clocks that are well-synchronized using a clock synchronization protocol like NTP [Mil03]. When clock synchronization is not feasible at data sources, tuples are timestamped at the entry point of the Stream Processing Engine (SPE).

Table 2.1 presents a sample schema of a Call Description Record (CDR) stream used in mobile phone networks. The schema consists of 9 fields. Fields *Caller* and *Callee* represent the mobile phones making and receiving the call, respectively. Field *Time* specifies the call starting time

Field Name	Field Type
Caller	*text*
Callee	*text*
Time	*integer*
Duration	*integer*
Price	*double*
Caller_X	*double*
Caller_Y	*double*
Callee_X	*double*
Callee_Y	*double*

Table 2.1: Example tuple schema

(expressed in seconds). In our example, $ts = Time$. Field *Duration* specifies the call duration (expressed in seconds). Field *Price* specifies the call cost (in €). Fields *Caller_X* and *Caller_Y* represent the geographic coordinates of the *Caller* phone number while fields *Callee_X* and *Callee_Y* represent the geographic coordinates of the *Callee* phone number while fields. In cellular telephony, streams defined by a schema similar to the presented one are generated by antennas which mobile phones connect to.

Continuous Queries are defined over one or more input streams and produce one or more output streams. A continuous query is defined as a directed acyclic graph (DAG) with additional input and output edges. Each vertex u is an operator that consumes tuples from one (or multiple) input streams and produces tuples for one (or multiple) output streams. An edge from operator u to v implies that tuples produced by u are processed by v. Queries are defined as “continuous" because results are computed in an on-line fashion as input tuples are processed.

2.1.1 Data Streaming Operators

This section provides an overview of the basic data streaming operators. Data streaming operators are the base unit used to process and produce output tuples. They’re defined by at least one input stream and one output stream. Feeding tuples are taken from the input stream while tuples produced by the operator are sent through its output stream. Data streaming operators are classified depending on whether they maintain any state while processing input tuples. *Stateless operators* perform a one-by-one processing of input tuples. Therefore, each tuple is processed individually and the corresponding output is (eventually) produced without maintaining any state. Examples of stateless operators include *Map* (the data streaming counterpart of the relational *projection* function), *Filter* (the data streaming counterpart of the relational *select* function) and *Union* operators. *Stateful operators* maintain state as they process multiple input tuples in order to produce one output tuple. Examples of stateful operators include *Aggregate* and *Join* operators. Due to the unbounded nature of

data streams, stateful operators produce output tuples that refer to portions of the processed input tuples. This mechanism is known as *windowing*. As presented in [PS06], windows are used to maintain only the most recent part of a stream. Time based windows (also referred to as *logical windows*) are defined over a period of time (e.g. tuples received in the last 10 minutes); tuple based windows (also referred to as *physical windows*) are defined over the number of stored tuples (e.g., last 50 received tuples).

The scope of a time-based window (i.e., the period of time it covers) is defined by 4 parameters: window start W_S, window end W_E and parameters $Size$ and $Advance$. Any incoming tuple t is added to the window maintained by a stateful operator if $W_S \leq t.ts < W_E$. If $t.ts \geq W_E$ the window is updated, either changing W_S, W_E or changing its $Size$. Depending on how the 4 parameters that define the scope of a window change, different windows models have been introduced:

- **Landmark window**: this window keeps W_S fixed, while W_E is continuously updated to the latest tuple timestamp. This is, the window $Size$ is always growing, including each new incoming tuple.

- **Sliding windows**: when using sliding windows, both W_S and W_E are updated depending on the timestamp of the input tuples being processed. Window parameter $Size$ is fixed and the constraint $W_E - W_S = Size$ is maintained when updating windows boundaries. Whenever W_S and W_E are updated, we say the window slides. W_E is updated each time the incoming tuple timestamp $t_{in}.ts >= W_E$. Parameter $Advance$ defines how W_S is updated:

  – $Advance = 0$: if $Advance$ is set to 0, being $[W_S, W_E[$ the current windows boundaries and being t_{in} an input tuple so that $t_{in}.ts > W_E$, windows boundaries are shifted to $[t_{in}.ts - Size + 1, t_{in}.ts + 1[$. That is, the window is shifted enough to include the new incoming tuple.

  – $Advance < Size$: in this case, the current window $[W_S, W_E[$ is updated to $[W'_S, W'_E[$, where $W'_S = W_S + Advance$ and $W'_E = W_E + Advance$. Shifting is repeated until $t_{in}.ts < W'_E$.

  – $Advance = Size$: this window model, known as *Tumbling Window* shifts the current window by $Size$ time units. That is, window $[W_S, W_E[$ is shifted to $[W_S + Size, W_E + Size[$.

When windows are tuple-based, a maximum fixed window of $Size$ tuples is maintained. Similarly to time-based windows, parameter $Advance$ specifies how the window evolves when it is shifted. If $Advance = 0$, then, once the window is full, each new incoming tuple causes the removal of the earliest one. If $0 < Advance \leq Size$, any time the window is full, the earliest $Advance$ tuples are discarded.

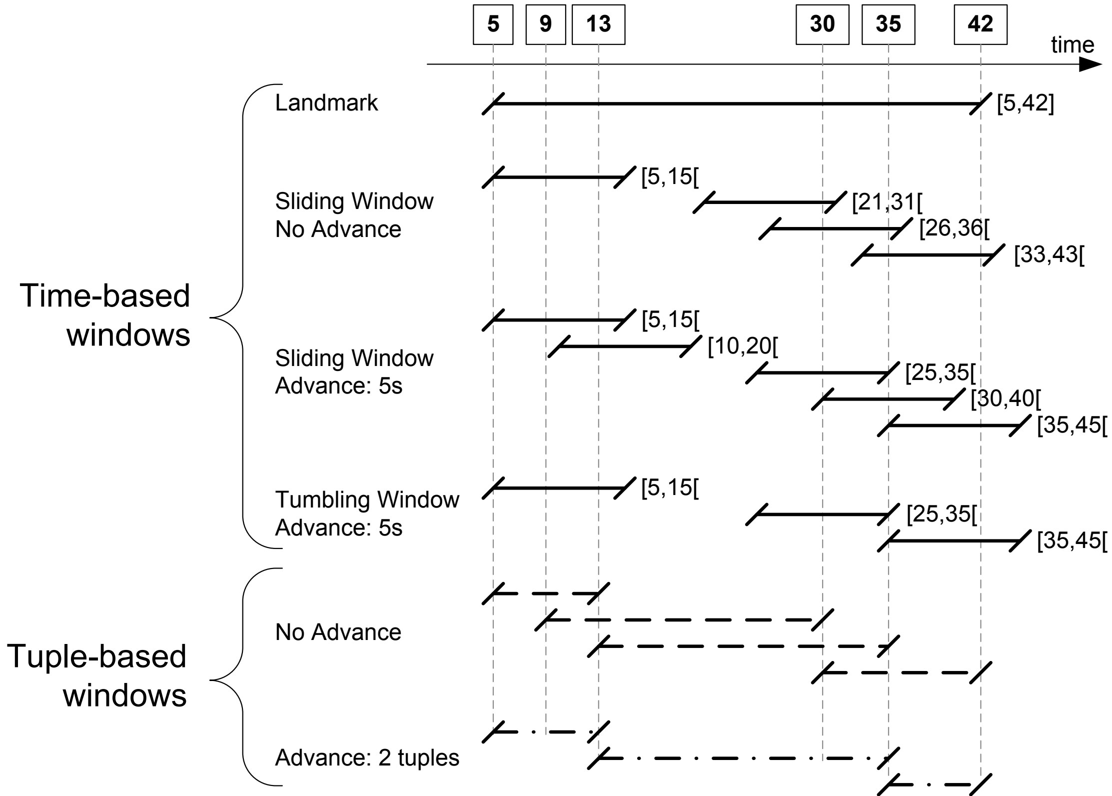


Figure 2.1: Examples of different windows models

Figure 2.1 presents how the different windows models evolves considering a sequence of 6 input tuples with timestamps $5, 9, 13, 30, 35, 42$, respectively. A landmark window will have $W_S = 5$ (as 5 is the first timestamp received) while W_E will grow while processing incoming tuples. After processing the last tuple at time 42 window boundaries will be $[5, 42]$. A sliding window with parameters $Size$ and $Advance$ set to 10 and 0, respectively, will initially cover time period $[5, 15[$. Upon reception of the tuple with timestamp 30, the window will shift to $[21, 31[$. Similarly, processing of tuples having timestamps 35 and 42 will cause the window to shift to $[26, 36[$ and $[33, 43[$. If $Size$ and $Advance$ parameters are set to 10 and 5, respectively, the first window $[5, 15[$ will shift by steps of 5 seconds (i.e. $[10, 20[$,$[15, 25[$ and so on). If $Advance$ is set to be equal to $Size$ (tumbling windows), then windows slides without covering overlapping portions of time. The first window $[5, 15[$ will shift to $[15, 25[$,$[25, 35[$ and so on. The last two examples of Fig. 2.1 presents sample evolutions of tuple based windows. If no $Advance$ parameter is set, once the window is full (i.e., upon processing of the tuple with timestamp 13), each new incoming tuple causes the removal of the earliest one. If parameter $Advance$ is set 2, each time the window is full, the 2 earliest tuples are discarded.

Orthogonally to the different windows models, different models exist with respect to when the content of a window is made available generating an output tuple. As presented in [BDD$^+$10], dif-

ferent events can trigger the creation of an output tuple carrying the current value of a window: (1) *content-change* outputs are generated each time a new incoming tuple changes the value of a function associated to the window, (2) *window-close* outputs are generated only before shifting a window, (3) *non-empty content* outputs are generated only if the corresponding window contains at least 1 tuple while (4) *periodic* outputs are generated every time λ tuples or λ time units have been processed. Existing research and commercial SPEs implement different triggering policies: STREAM, the Standford Stream Data Manager [STRa], implements all the 4 triggering conditions; StreamBase SPE [Strc] does not implement the *content-change* triggering condition. Similar to STREAM, Esper [Espa] implements all the 4 triggering conditions.

StreamCloud defines the same window semantics defined by the Borealis project [Borc]: window models defines both time based and tuple based sliding windows, where parameters $Size$ and $Advance$ are configurable by the user. With respect to the policy defining when output tuples are created by stateful operators, the implemented reporting strategies are *window close* and *non-empty content*. That is, each time a window is going to be shifted, its corresponding value is outputted if the window contains at least one tuple.

The rest of this section includes a detailed description of the basic data streaming operators. Each operator is defined as:

$$OP_N\{P_1,\ldots,P_m\}(I_1,\ldots,I_n,O_1,\ldots,O_p)$$

Where OP_N represents the operator name, $P_1,\ldots,P_m$ represent a set of parameters that specify the operator semantics (e.g., functions used to transform input tuples, predicates used to decide which information to discard or parameters related to the windowing model), $I_1,\ldots,I_n$ a set of input streams and $O_1,\ldots,O_p$ a set of output streams. Optional parameters are defined using square brackets. All the examples refer to the schema presented in Section 2.1.

Map The Map operator is a generalized projection operator used to transform the schema of the input tuples, e.g., to transform a field representing a speed expressed in Km/h to m/s. The Map is defined as:

$$M\{F'_1 \leftarrow f_1(t_{in}),\ldots,F'_n \leftarrow f_n(t_{in})\}(I,O)$$

where I and O represent the input and output streams, respectively. t_{in} is a generic input tuple and $F'_1,\ldots,F'_n$ is the schema of output tuples. Each input tuple is transformed by functions $f_1,\ldots,f_n$.

The following example considers a Map operator converting field $Price$ from euros to dollars.

$$M\{Caller \leftarrow Caller, Callee \leftarrow Callee, Duration \leftarrow Duration, Time \leftarrow Time,$$
$$Dollars \leftarrow 1.2492 * Price, Caller_X \leftarrow Caller_X, Caller_Y \leftarrow Caller_Y,$$
$$Callee_X \leftarrow Callee_X, Callee_Y \leftarrow Callee_Y\}(I, O)$$

Filter The Filter is a generalized selection operator used either to discard or to route tuples from one input stream to multiple output streams. As an example, a filter operator can be used to discard CDRs referring to phone calls whose price is lower than 3 €. The Filter is defined as:

$$F\{P_1, \ldots, P_m\}(I, O_1, \ldots, O_m[, O_{m+1}])$$

where I is the input stream, $O_1, \ldots, O_m, O_{m+1}$ is an ordered set of output streams and $P_1, \ldots, P_m$ is an ordered set of predicates. Each incoming tuple t_{in} is forwarded to O_i, being i the minimum value for which P_i holds. If no predicate is satisfied, t_{in} is either routed to O_{m+1} (if defined) or discarded. Both input and output tuples share the same schema.

The following example considers a Filter operator routing CDR tuples depending on the price associated to each call.

$$F\{Price \leq 5e, Price \leq 10e\}(I, O_1, O_2, O_3)$$

This operator routes incoming tuples to three different output streams O_1, O_2, O_3. Tuples referring to a phone call whose price is less than or equal to 5 euros are routed to O_1. Tuples referring to a phone call whose price is greater than 5 euros and less than or equal to 10 euros are routed to O_2. Finally, tuples referring to a phone call whose price is greater than 10 euros are routed to O_3.

Union The Union operator is used to merge tuples from multiple input streams into a single output stream, e.g., to merge CDRs coming from distinct antennas. All the input streams and the output stream tuples share the same schema. The Union operator is defined as:

$$U\{\}(I_1, \ldots, I_n, O)$$

where $I_1, \ldots, I_n$ is a set of input streams and O is the output stream.

Aggregate The Aggregate operator is used to compute aggregation functions such $mean$, $count$, min, max, $first_val$ and $last_val$ over windows of tuples. It is defined as:

$$Agg\{WType[, ts], Size, Advance, F'_1 \leftarrow f_1(W), \ldots, F'_n \leftarrow f_n(W)$$
$$[, Group - by = (F_{i_1}, \ldots, F_{i_m})]\}(I, O)$$

$WType$ specifies the window type, if $WType$ is set to $time$ the window W is time-based; if $WType$ is set to $tuples$ window W is based on the number of tuples. $Size$ specifies the amount of tuples to be maintained in W. If $WType = time$, being t_{in} an incoming tuple and W_S the timestamp of the left boundary of window W, the window is full any time $t_{in}.ts - W_S \geq Size$. Once W is full, an output tuple carrying the result of the aggregate functions $f_1(W), \ldots, f_n(W)$ is produced. Subsequently, W is shifted discarding tuples $t \in W : t.ts \in [W_S, W_S{+}Advance[$ and W_S is updated to $W_S + Advance$. If, after the window has been shifted, $t_{in}.ts - W_S \geq Size$, the window is shifted repeatedly until $t_{in}.ts \in [W_S, W_S + Size[$. As we introduced before, an output tuple is produced each time the window is shifted if the latter contains at least one tuple. Attribute ts is optional as it must be provided only for time-based windows.

The following example considers an aggregate operator used to compute, on a per-hour basis, the number of phone calls made by each distinct phone number and their average duration.

$$Agg\{time, Time, 3600, 600, Calls \leftarrow count(), Mean_Duration \leftarrow mean(Duration),$$
$$Group - by = (Caller)\}(I, O)$$

In the example, $WType = time$. Window $Size$ and $Advance$ parameters are set to 3600 and 600 seconds, respectively. Once the window W is full, an output tuple will be produced every 600 seconds, and will contain the aggregate information of the last 3600 seconds. Two aggregate functions are used to define the output tuples: 1) function $count()$, with no parameters, that counts the number of processed tuples, 2) function $mean(Duration)$ that computes the average value of field $Duration$ for all the tuples contained in any window W. Attribute $Group - by$ is set to input field $Caller$. Therefore, a separate window W will be maintained for each distinct value of the field $Caller$. The schema associated to the output stream consists of the fields $Caller, Time, Calls$ and $Mean_Duration$.

Figure 2.2 shows a sample execution of the previous aggregate operator. Input tuples are shown as orange boxes. Output tuples are shown as gray boxes. All input (and output) tuples refer to calls made by phone number A. For the ease of the explanation, input tuple schema does not include the geographic coordinates of caller and callee phone numbers.

Five input tuples $t_i, i = 1 \ldots 5$ are processed by the operator. Tuple t_1 refers to a phone call started at time 25, lasting 30 seconds and whose price is $5.2e$. Tuple t_2 refers to a phone call started at time 2400, lasting 55 seconds and whose price is $11e$. Both tuples are added to the window W having boundaries $[0, 3600[$. Upon reception of tuple t_3, referring to a phone call started at time 4500, lasting 10 seconds and whose price is $2e$, window W must be shifted ($t_3.Time > 3600$). Before shifting W, the output tuple T_1 is produced. Tuple T_1 states that phone number A has made 2 phone calls with average duration 42.5 seconds during the one-hour period starting at time 0. Window W is shifted to $[600, 4200[$, tuple t_1 is purged. W must be shifted again as $t_3.Time > 4200$. Before shifting

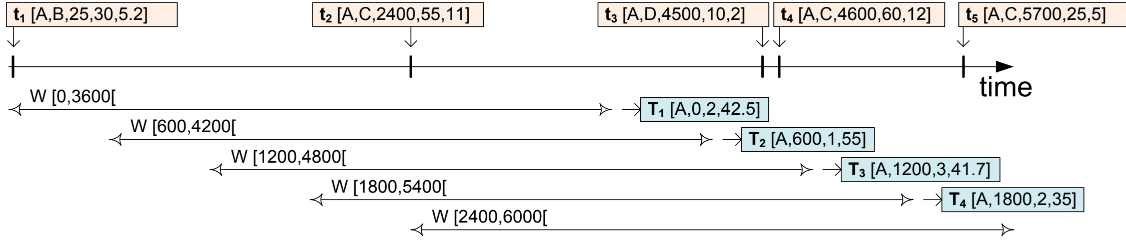


Figure 2.2: Sample evolution of time based aggregate operator

window W, output tuple T_2 is produced. Tuple T_2 states that phone number A has made 1 phone call (lasting 55 seconds) during the one-hour period starting at time 600. Finally, window W is shifted to $[1200, 4800[$; no tuple is purged and tuple t_3 is added. Tuple t_4 refers to a phone call started at time 4600, lasting 60 seconds and whose price is $12e$. Tuple t_4 causes no shifting as it falls within current W boundaries. Upon reception of tuple t_5, referring to a phone call started at time 5700, lasting 25 seconds and whose price is $5e$, window W must be shifted ($t_5.Time > 4800$). Before shifting W, output tuple T_3 is produced. Tuple T_3 specifies that phone number A has made 3 phone calls with average duration 41.7 seconds during the one-hour period starting at time 1200. Window W is shifted first to $[1800, 5400[$ (tuple t_2 is purged) and subsequently to $[2400, 6000[$ (no tuple is purged). Output tuple T_4 is produced after the first shifting. Tuple T_4 specifies that phone number A has made 2 phone calls with average duration 35 seconds during the window starting at time 1800.

If $WType = tuples$, window W is full each time $Size$ tuples have been added to it. Once the corresponding output is produced, the earlier $Advance$ tuples are purged from W. Parameter $Group - by$ is optional, it can be used to define equivalence classes over the input stream. Given $Group - by = F_{i_1}, \ldots, F_{i_n}$, a separate window W is maintained for each distinct combination of fields $F_{i_1}, \ldots, F_{i_n}$ values found processing input tuples. The schema of output tuples is composed by $Group - by$ fields $F_{i_1}, \ldots, F_{i_n}$ (if any), ts (if $WType = time$) and fields $F'_1, \ldots, F'_n$.

The following example considers an aggregate operator used to computethe minimum and maximum phone call duration of the last 3 phone calls made by each phone number. The Aggregator operator is defined as:

$$Agg\{tuples, 3, 2, Min_Duration \leftarrow min(Duration), Max_Duration \leftarrow max(Duration),$$
$$Group - by = (Caller)\}(I, O)$$

In the example, $WType = tuples$. Window $Size$ and $Advance$ parameters are set to 3 and 1 tuples, respectively. Once W is full (i.e., 3 tuples have been received), an output tuple is produced and the window is updated removing the two earliest tuples maintained by the window. Two aggregate functions are defined: 1) function $min(Duration)$, that maintains the shortest call duration, and 2)

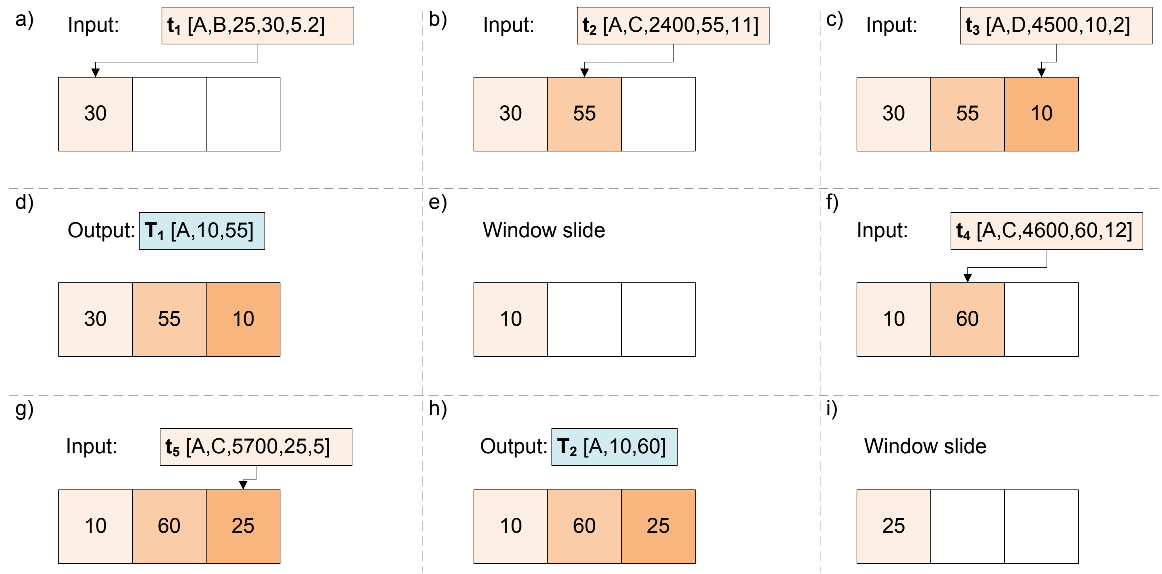


Figure 2.3: Sample evolution of tuple based aggregate operator

function $max(Duration)$ that maintains the longest call duration. Attribute $Group-by$ is set to input field $Caller$. Therefore, a separate window W is maintained for each distinct value of the field $Caller$. The output stream schema consists of the fields $Caller, Min_Duration, Max_Duration$.

Figure 2.3 presents a sample execution of the aggregate operator. As in the example of figure 2.2, input tuples are shown as orange boxes while output tuples are shown as gray boxes. The input tuples considered in this example are the same of the ones presented in the previous example.

The first two tuples t_1, t_2 are stored in the window and no output is produced as the window is still not full (Fig. 2.3.a and 2.3.b). Upon reception of the third tuple t_3, the window becomes full (Fig. 2.3.c). The output tuple T_1 carrying the shortest and longest phone calls durations (resp. 10 and 55 seconds) is produced (Fig. 2.3.d). Subsequently, the window is updated discarding the two earliest tuples (Fig. 2.3.e). After updating the window, tuple t_4 is stored without producing any result (Fig. 2.3.f). Upon reception of tuple t_5 (Fig. 2.3.g), the output tuple T_2 is produced (Fig. 2.3.h) and ,subsequently, the window is updated discarding the two earliest tuples (Fig. 2.3.i).

Join and Cartesian Product These two stateful operators are used to match tuples from two distinct input streams, a separate window is maintained for each input stream. They only differ in the complexity of their predicate. The Join operator defines a predicate that, expressed in Normal Conjunctive Form, defines at least one term referring to an equality between two fields of the different input streams (e.g., match phone calls referring to the same *Caller* number) while the Cartesian Product defines a predicate that can be arbitrarily complex (e.g., match CDRs whose spatial distance is lower than 15Km).

Both operators are defined in the same way:

$$J\{P, WType[, ts], Size\}(S_l, S_r, O)$$

$$CP\{P, WType[, ts], Size\}(S_l, S_r, O)$$

S_l and S_r are two input streams referred to as left and right input streams, respectively. They can carry tuples defined by different schemas. Both operators define a pair of windows W_l, W_r. W_l is used to maintain tuples received on left stream. W_r is used to maintain tuples received on right stream. Parameters $WType$, ts and $Size$ are similar to the aggregate parameters presented in 2.1.1.0.4. Nevertheless no *Advance* parameter is defined for the Join and the Cartesian product operators. If the windows are time-based (i.e., $WType = time$) left window W_l (resp. right window W_r) is full if, being t_{in} an incoming tuple on the right (resp. left) stream and W_S the window start of W_l (resp. W_r), $t_{in}.ts - W_S \geq Size$. That is, tuples received on the left stream are used to purge tuples from right window while tuples received on the right stream are used to purge tuples from the left window. As no *Advance* parameter is specified, whenever a window W is full, all tuples $t \in W : t.ts < t_{in}.ts - Size$ are discarded. If windows are tuple-based (i.e., $WType = tuples$) left window W_l (resp. right window W_r) is full is $Size$ tuples have been added. After an incoming tuple has been used to purge the opposite window, the former is matched with all the tuples in the latter. An output tuple is produced for each pair of tuples verifying predicate P. Finally, tuple t_{in} is added to its corresponding window (left is the tuple has been received on the left stream, right otherwise). The schema of the output tuples is defined as the concatenation of the left and right input schemas. As the schema of the left stream and the right stream may define fields with the same name, the field names of the left stream are modified adding the prefix $Left_$ while the ones of the right stream are modified adding the prefix $Right_$.

2.2 Continuous Query Example

In this section we present a sample continuous query used in cellular telephony to detect frauds involving cloned cards numbers. The idea is to spot phone numbers that, between two consecutive phone calls, cover a suspicious space distance with respect to their temporal distance. As an example, consider a mobile phone involved in a call in Spain at 12:00 CET and in a second call in U.K. at 12:20 CET. The distance between Spain and U.K. cannot be traversed in 20 minutes, therefore, the phone number is considered to be cloned. Notice that in order to detect cloned phone numbers, we must consider consecutive calls involving the same phone number both as caller or callee. This query is referred to as *high mobility fraud*, figure 2.4 shows its composing operators.

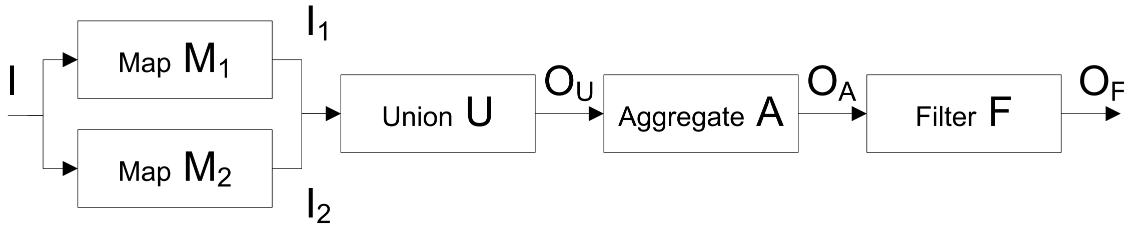


Figure 2.4: High Mobility fraud detection query

The system input is composed by CDRs generated by the antennas which mobile phones connect to. Initially, each incoming CDR is transformed into a pair of $Phone, Duration, Time, Price, X, Y$ tuples, one for the *Caller* number and one for the *Callee* number. Subsequently, for each consecutive pair of tuples referring to the same phone number (appearing as *Caller* or *Callee* in the previous CDRs), the speed at which the mobile phone has moved between the two phone calls is computed as the Euclidean distance divided the time distance. Finally, phone number appearing to move at a speed higher than a given threshold are reported in the system output as they represent possibly cloned numbers.

The query can be defined with 6 operators. Initially, the input stream I is used to feed map operators M_1 and M_2. These operators are defined as follows:

$$M_1\{Phone \leftarrow Caller, Duration \leftarrow Duration, Time \leftarrow Time,$$
$$Price \leftarrow Price, X \leftarrow Caller_X, Y \leftarrow Caller_Y\}(I, I_1)$$

$$M_2\{Phone \leftarrow Callee, Duration \leftarrow Duration, Time \leftarrow Time,$$
$$Price \leftarrow Price, X \leftarrow Callee_X, Y \leftarrow Callee_Y\}(I, I_2)$$

M_1 and M_2 are used to convert each input tuple into a pair of tuples, one for the caller number and one for the callee number. Subsequently, tuples from streams I_1 and I_2 are merged together into stream O_U. Merging is performed by union operator U

$$U\{\}(I_1, I_2, O_U)$$

Notice that streams I_1 and I_2 can be merged by the union operator U as they share the same schema.

The Aggregate operator A extracts times and X, Y coordinates for each pair of consecutive calls involving the same phone number. The Aggregate defines a tuple based window with *Size* and *Advance* of 2 and 1 tuples, respectively. *Group-by* attribute has been set to $Phone$ so that only tuples referring to the same phone number are matched together. Given a window maintained 2 tuples referring to the same phone number, function $first_val(F_i)$ is used to extract field F_i value of the earliest

tuple while function $last_val(F_i)$ is used to extract field F_i value of the latest tuple. The Aggregate operator is defined as:

$$A\{tuples, 2, 1, Time \leftarrow first_val(Time), T1 \leftarrow first_val(Time), X1 \leftarrow first_val(X),$$
$$Y1 \leftarrow first_val(Y), T2 \leftarrow last_val(Time), X2 \leftarrow last_val(X), Y2 \leftarrow last_val(Y),$$
$$[Group - by = Phone]\}(O_U, O_A)$$

The schema of output tuples consists of fields $Phone, Time, T1, X1, Y1, T2, X2, Y2$. The Map operator M_3 is used to compute the speed at which the mobile phone has moved between two consecutive phone calls. It is defined as:

$$M_3\{Phone \leftarrow Phone, Time \leftarrow Time, Speed \leftarrow \frac{\sqrt{(X2 - X1)^2 + (Y2 - Y1)^2}}{T2 - T1}\}(O_A, O_M)$$

Fields $Phone$ and $Time$ are maintained from input tuples. Field $Speed$ is computed dividing the Euclidean distance between the consecutive calls coordinates by the elapsed time. Finally, the filter operator F is used to allow to pass to the output stream only the tuples having field $Speed$ greater than the threshold T. Is it defined as:

$$F\{Speed \geq T\}(M_A, O_F)$$

2.3 Table Operators

In this section, we provide an overview of table operators, used to write information to or read information from external tables. As discussed in Section 1.2, one of the requirements of data streaming applications is the possibility of persisting the results of the computation being run over the data streams. Examples of these applications include data streaming queries where on-line stream features are compared with historic information or applications logging generated outputs for later analysis. The semantic of the table operators being presented is based on the table operators defined in the Borealis SPE [bora]. Table operators are being presented in order to give a complete overview of the data streaming operators that can be used to define continuous queries and because some of them are used later when presenting *StreamCloud* use cases (Chapter 7). Nevertheless, parallelization, dynamic load balancing, elasticity and fault tolerance for table operators is not in the scope of the presented work.

The Borealis SPE defines four different table operators, one for each basic SQL command: *select*, *insert*, *delete* and *update*. For all of them, the operator definition includes a parameter to specify which is the table containing the information and one parameter that expresses, by means of a SQL statement, which operation should be executed on it. As the state of a table operator is maintained by an external entity, table operators can be considered as stateless operators.

In all the following examples, we refer to a table *Top_Expensive* used to store records specifying, for each mobile phone number, the time and the price of the most expensive call being made by the user. Records of the table are composed by fields $\langle Phone, Time, Price \rangle$.

Select Operator The *select* operator defines one input stream and one output stream plus two optional output streams. The operator is used to query the given table for each incoming tuple, using the information carried by the tuple to decide which information to extract from the table. The first output stream defined by the *select* operator is used to output the tuples matching the given SQL expression. The schema of the first output stream tuples will be equal to the table schema. A second output stream can be used to forward incoming tuples (hence, its schema will be identical to the input tuples one) if the given SQL statement produces no results. The third output can be defined in order to output the amount of tuples matching the given SQL expression. In this case, the output schema is composed by the fields of the SQL *where* cause plus a field containing the counter value, referred to as $Count$. The *select* operator is defined as:

$$Select\{DB, SQL\}(I_1, O_1[, O_2, O_3])$$

Parameter DB specifies the table to be queried while parameter SQL represents the SQL statement to execute. Input stream I_1 and output stream O_1 are defined as mandatory while output streams O_2 and O_3 are defined as optional. A sample *select* operator can be used to retrieve, for each incoming tuple, the more expensive call made by the $Caller$ phone number appearing in the input tuple. This operator is be defined as:

$$Select\{Top_Expensive, \text{SELECT * FROM Top_Expensive}$$
$$\text{WHERE Top_Expensive.Phone = input.Caller}\}(I_1, O_1)$$

Insert Operator The *insert* operator is used to persist the information carried by each incoming tuple to the given table. It defines a single input stream and an optional output stream to forward tuples after they have been persisted. As for the other operators, two parameters are provided to specify the table to which information should be persisted and the SQL expression containing the operation to execute. The *insert* operators is defined as:

$$Insert\{DB, SQL\}(I_1[, O_1])$$

A sample *insert* operator to persist the information of an incoming tuple in the *Top_Expensive* table, without forwarding the incoming tuple itself, can be defined as:

$$Insert\{Top_Expensive,\text{INSERT INTO Top_Expensive}$$
$$\text{VALUES (Caller,Time,Price)}\}(I_1)$$

It can be noticed that, in the sample *insert* operator, a single input but no output stream is defined.

Delete Operator The *delete* operator is used to delete records from a given table. The operator defines one input stream and up to two optional output streams. The first optional output is used to output the tuples that have been deleted from the table. The schema for this tuples will be equal to the table schema. The second optional output is used to forward incoming tuples, its schema being equal to the one of the input tuples. The operator is defined as:

$$Delete\{DB, SQL\}(I_1[, O_1, O_2])$$

A sample *delete* operator used to remove records from the $Top_Expensive$ table can be defined as:

$$Delete\{Top_Expensive,\text{DELETE FROM Top_Expensive}$$
$$\text{WHERE Top_Expensive.Phone = input.Caller}\}(I_1, O_1, O_2)$$

In this case, each incoming tuple will be forwarded to the output stream O_2 and, in case a record is removed from the table, it will be forwarded to output stream O_1.

Update Operator The *update* operator is used to update records stored in a table. Similarly to the *delete* operator, the *update* operator defines one input stream and up to two optional output streams. The first output stream can be defined to forward incoming tuples, its schema being equal to the input tuples one. The second output stream can be defined to forward the table tuples being updated, its schema being equal to the table one. The operator is defined as:

$$Update\{DB, SQL\}(I_1[, O_1, O_2])$$

A sample operator used to update the $Price$ field to the value carried by an input tuple can be defined as:

$$Update\{Top_Expensive,\text{UPDATE Top_Expensive}$$
$$\text{SET Top_Expensive.Price = input.Price}$$
$$\text{WHERE Top_Expensive.Phone = input.Caller }\}(I_1, O_1, O_2)$$

In this case, as two output streams are defined, each incoming tuple will be forwarded to output stream O_2 while each record being updated (if any) will be forwarded to output stream O_1.

Part III

STREAMCLOUD PARALLEL-DISTRIBUTED DATA STREAMING

Chapter 3

StreamCloud Parallel-Distributed Data Streaming

In this chapter, we present *StreamCloud* technique to parallelize the execution of data streaming operators. We first discuss how Stream Processing Engines (SPEs) have evolved from centralized solutions to parallel-distributed solutions to reach for higher processing throughputs. Subsequently, we explore the challenges that motivated our work and discuss how they have been addressed. Finally, we present a thorough evaluation of the proposed parallelization technique.

3.1 Stream Processing Engines Evolution

Earliest SPE prototypes were designed as centralized applications. Some of these first centralized prototypes include Aurora [CCC^{+}02] [ACC^{+}03], STREAM (the STandford stREam datA Manager) [ABB^{+}04] [ABW06] [BBC^{+}04], and TelegraphCQ [CCD^{+}03] [Des04]. In this scenario, queries are entirely deployed at the same SPE instance. The main shortcoming of a centralized SPE is that, upon saturation of available resources (e.g., CPU), processing of tuples is delayed, leading to high latencies in system output. A first evolution step was to move from centralized to distributed SPEs. One of the first prototype of distributed SPEs is Borealis [AAB^{+}05b] [ABC^{+}05], an evolution of Aurora and Medusa [CBB^{+}03] [SZS^{+}03]. As discussed in [ÖV11], two different kinds of query execution parallelism exist: *inter-query* parallelism allows for parallel execution of different queries at different SPE instances while *intra-query* parallelism allows for execution of a single query at multiple SPE instances. *Intra-query* parallelism is further divided into *inter-operator* and *intra-operator* parallelism.

Inter-operator parallelism allows for execution of different operators belonging to the same query at different SPE instances while *intra-operator* parallelism refers to the possibility of deploying a single operator at multiple SPE instances. Centralized SPEs allow deploying distinct queries at different data SPE instances (inter-query parallelism). Distributed SPEs allow the deployment of distinct operators belonging to the same query at different SPE instances (inter-operator parallelism), but do not provide intra-operator parallelism, which is provided by parallel-distributed SPEs.

It should be notice that distributed SPEs might incur in the same problem of their centralized counterpart. Sudden spikes in the load might lead to high latencies in system output. Both centralized and distributed SPEs suffer from single-node bottlenecks: each stream goes through a single SPE instance. Any time the stream volume exceeds the SPE instance capacity (e.g., due to a increase in the system load that overcomes the instance processing capacity), the system starts delaying tuple processing, resulting in a growing processing latency. Distinct techniques have been studied to face this problem, considering both techniques that reduce time or space complexity. For instance, load shedding [TcZ+03, BDM04] was introduced as a solution to discard part of the incoming tuples if the processing SPE instance cannot cope with the system load. Decisions about which tuples to discard are taken depending on how each tuple contribute to the system output. As an example, if a Quality-of-Service (QoS) metric is defined, load shedding will minimize the QoS degradation discarding the tuples that contribute less to the overall QoS. Different other techniques have been defined to reduce the amount of information kept by an SPE instance. As presented in [BBD+02], these techniques include Sketches, Histograms and Wavelets. For instance, Exponential Histograms can be used to approximate the result of a given query aggregating together information belonging to multiple tuples, as presented in [DM07].

We look for a solution that avoids discarding any information due to sudden load variations. We see as a natural step in SPE evolution the introduction of parallel-distributed SPEs. In such model, any operator belonging to a query can be deployed in an arbitrary number of nodes. The idea is to avoid concentrating any stream into a single SPE instance, avoid thus any single-node bottleneck. It should be noticed that, in real world applications, multiple physical sources define a logical stream. As an example, imagine a scenario where information related to calls being made by mobile phone is gathered in order to be processed. All the Call Description Records (CDRs) containing information like calling number, callee number, call start time, and so on (an example of CDRs information is presented in Section 2.1) is usually composed by several physical streams generated by several computing systems. In order to avoid concentrating any stream into a single SPE instance, all the physical streams composing a logical stream should be always processed in parallel using either several physical computers or multi-core facilities provided by today off-the-shelf computers.

Figure 3.1 presents the evolution of SPEs showing how a sample query composed by 4 operators can managed by a centralized, a distributed and a parallel-distributed SPE.

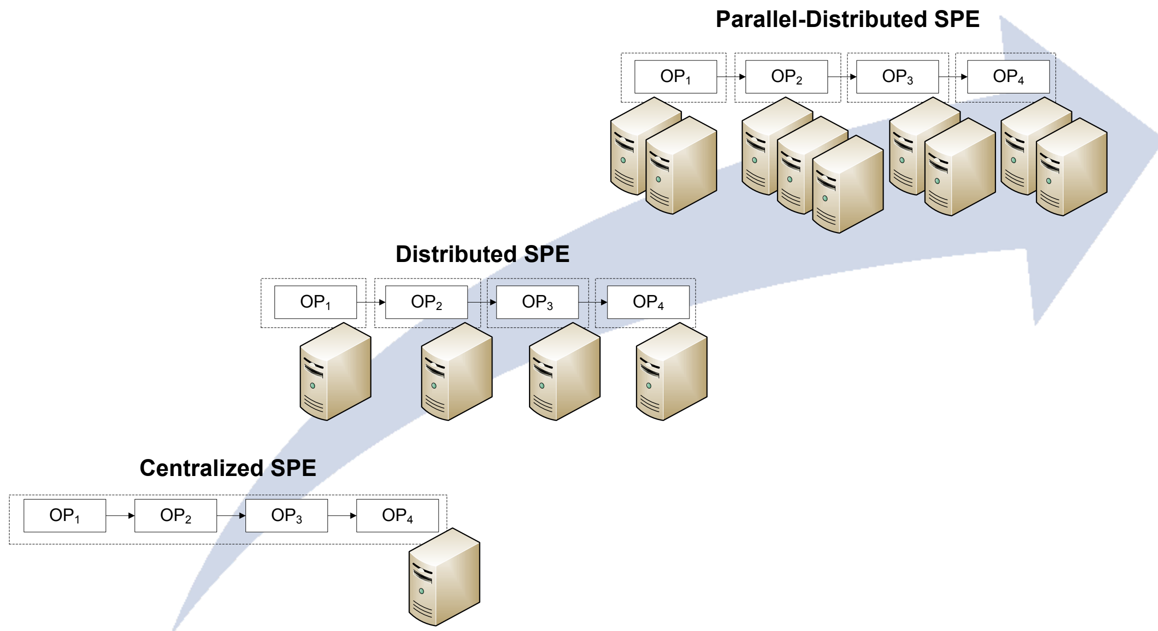


Figure 3.1: SPE evolution

When designing a parallel-distributed SPE, four challenges should be considered:

1. **Scalability**: the system must be able to process high stream volumes by aggregating the power of an increasing number of nodes.

2. **Semantic Transparency**: the results produced by a parallel-distributed query should be equivalent to the ones produced by its centralized or distributed counterpart.

3. **Syntactic Transparency**: queries should be written as for a centralized SPE, without taking into consideration any parallelization issue.

4. **Elasticity and Dynamic Load Balancing**: in a parallel-distributed SPE, the number of nodes required to process a query (or an operator) might change depending on the incoming stream volume. A static configuration might lead to either under-provisioning (i.e., allocated SPE instances cannot cope with the system load) or over-provisioning (i.e., allocated SPE instances are running below their full capacity). The system should be designed considering reconfiguration actions that can change the number of SPE instances assigned to each operator. Moreover, elasticity should be combined with dynamic load balancing. In a parallel-distributed SPE that provides dynamic load balancing, new nodes are not provisioned due to a uneven load distribution, but only if the system as a whole cannot cope with the incoming load. It should be noticed that dynamic load balancing is crucial for SPEs that rely on public cloud infrastructures (like Amazon EC2 [Ama]) where provisioning of extra computational resource usually implies a

higher cost to pay for the service.

In this chapter and in the following ones we present how *StreamCloud*, a parallel-distributed SPE with elastic and dynamic load balancing capabilities, has been designed in order to address these challenges. The main results were published on ICDCS 2012 [GJPPMV10] and TPDS 2012 [GJPPM+12]. In this chapter, we focus primarily on scalability and semantic transparency. Elasticity and dynamic load balancing will be discussed in Chapter 4 while syntactic transparency will be discussed in Chapter 6.

3.2 Parallelization strategies

In this section we discuss the different alternatives we considered to parallelize queries in *StreamCloud* and the overhead associated to each of them.

In order to parallelize an operator, some routing policy must be adopted to distribute its input tuples to the different SPE instances where the operator is deployed. Due to their one-by-one processing, routing of tuples is trivial for stateless operators. Consider a query that is used to process incoming CDRs discarding the ones referring to phone calls whose price is lower than 5 €. Such query can be parallelized using an arbitrary number of SPE instance as each single SPE instance will discard the right tuples independently on how tuples are routed. On the contrary, particular attention must be given to stateful operators. In order to produce the correct output, we must ensure that all tuples that must be aggregated/joined together are processed by the same SPE instance. For instance, with respect to the Aggregate operator presented in Section 2.1.1 computing the number of phone calls made by each distinct mobile phone and their average duration on a per-hour basis, we must ensure that all tuples belonging to the same mobile phone are routed to the same instance to produce the correct result (that is, if tuples are routed without considering the semantic of the operator, multiple output tuples referring to the same phone number can be generated, non of them being the correct answer to the query).

In order to study the different parallelization strategies, we must first consider how the operators of a query can be distributed (and parallelized) assigning them to a set of available SPE instances. We characterize the possible different strategies along a spectrum. The first aspect to consider is the granularity of the parallelization unit: on one extreme, we can chose to distribute the query operators so that all the operators are deployed at each available SPE instance. On the other extreme, we can decide to assign operators to SPE instances so that each SPE instance does not contains more than one operator. Intermediate approaches will be defined by parallelization units that contain at least two operators.

We illustrate three alternative parallelization strategies by means of the abstract query in Figure 3.3 and its deployment on a cluster of 90 SPE instances. The query is used to compute the number

of mobile phones that, on a per-hour basis, make N phone calls whose price is greater than P, for each $N \in [N_{min}, N_{max}]$. E.g., we might be interested in spotting how many mobile phones make either 5,6,7,8,9 or 10 calls costing more than 5 € on a per-hour basis. The input tuples schema is the one presented in 2.1, composed by fields *Caller*, *Callee*, *Time*, *Duration*, *Price*, *Caller_X*, *Caller_Y* and *Callee_X*,*Callee_Y*. Field *Caller* specifies the phone number making the call while field *Callee* represents the one receiving it. Fields *Time* and *Duration* represent the call starting time and duration, respectively. Filed *Price* specifies the call cost (in €). Finally, fields *Caller_XCaller_Y* and *Callee_XCallee_Y* represent the geographic coordinates.

The query is composed by two stateful operators (Aggregate operators $A1$,$A2$) and three stateless ones (Map M and Filters $F1$,$F2$). Query operators are defined as follows:

$$M\{Caller \leftarrow Caller, Time \leftarrow Time, Price \leftarrow Price,\}(I, O_M)$$

$$F1\{Price > 5\}(O_M, O_{F1})$$

$$A1\{time, Time, 3600, 600, Calls \leftarrow count(), Group - by = (Caller)\}(O_{F1}, O_{A1})$$

$$F2\{Calls \geq 5 \wedge Calls \leq 10\}(O_{A1}, O_{F2})$$

$$A2\{time, Time, 3600, 600, Phones \leftarrow count(), Group - by = (Calls)\}(O_{F2}, O)$$

Map operator M is used to transform the input tuples schema discarding unnecessary fields while keeping only the ones that are needed by the following operators (i.e., keeping fields $Caller$, $Time$ and $Price$). Map operator M is connected to Filter operator $F1$ (operator M output stream O_M is the input stream of operator $F1$). Filter operator $F1$ is used to forward only tuples whose $Price$ field is greater than 5 €. Filter operator $F1$ is connected to Aggregate operator $A1$ (operator $F1$ output stream O_{F1} is the input stream of operator $A1$). Aggregate operator $A1$ is used to count the number of calls made by each mobile phone on a per-hour basis. Aggregate operator $A1$ is connected to Filter operator $F2$ (operator $A1$ output stream O_{A1} is the input stream of operator $F2$). Filter operator $F2$ is used to filter out mobile phones making less than 5 or more than 10 calls. Filter operator $F2$ is connected to Aggregate operator $A2$ (operator $F2$ output stream O_{F2} is the input stream of operator $A2$). Finally, Aggregate operator $A2$ is used to count how many mobile phones are making either 5,6,7,8,9 or 10 calls.

In order to study the goodness of different parallelization strategies, we first discuss which are the important factors that define the parallelization cost. We define two factors that contribute to the parallelization cost: the *Fan-out* overhead and the *Number of hops* overhead. The *Fan-out* overhead f represents the system-wide cost related to establishing and keeping all the communication channels needed by any SPE instance to communicate with other SPE instances. Intuitively, the higher the number of communication channels with other SPE instances, the higher the overhead. Figure 3.2 presents the average CPU consumption for an SPE instance running a filter operator for an increasing number of output streams (1,10,20,30,40 and 50 output streams). In all the setups with different

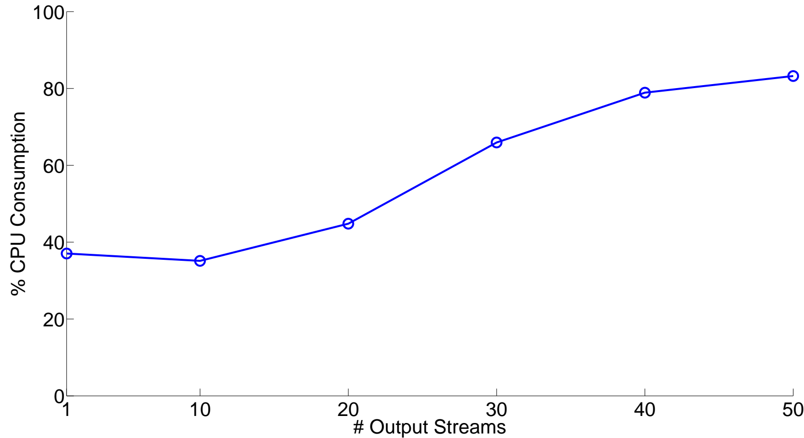


Figure 3.2: Fan-out overhead for a single operator instance

number of output streams the operator is fed with the same constant load of 5000 tuples/second. When defining 1 or 10 output streams the average CPU consumption is the same, just below 40%. Starting from 20 outputs streams, the average CPU consumption grows as the number of output streams increases. With the setup defining the higher number of output streams (50 in the example) the average CPU consumption grows over 80%, more than the double of the single-output setup.

Number of hops overhead h represents the cost payed to transfer a tuple between operators running at different SPE instances. When transferring a tuple between different SPE instances each tuple is serialized just before being sent and deserialized at the receiver site. It should be noticed that, given a query, this overhead affects all the pairs of consecutive operators running at different SPE instances. For any parallelization strategy X we define the cost function

$$c(X) = \alpha \cdot f(X) + \beta \cdot h(X)$$

where $\alpha, \beta \in [0, 1]$ are two arbitrary weights.

Query-cloud strategy - QC (Figure 3.3.b). With this strategy, the whole query is deployed at each SPE instance (90 in the example). In order to provide semantic transparency, tuples have to be redistributed just before each stateful operator. Therefore, tuples that should be aggregated/joined together are processed by the same SPE instance. In the query example, tuples are redistributed before the aggregate operator $A1$ (counting the number of calls made by each phone number) so that CDRs referring to the same $Caller$ are processed by the same SPE instance. Similarly, tuples are redistributed before operator $A2$ (counting how many times each number of phone calls appears) so that mobile phones making the same number of calls are processed by the same SPE instance. Being N the number of SPE instances where the query is deployed, each SPE instance receives one N-th of the incoming stream. Communication takes place, for every stateful operator, from each instance to

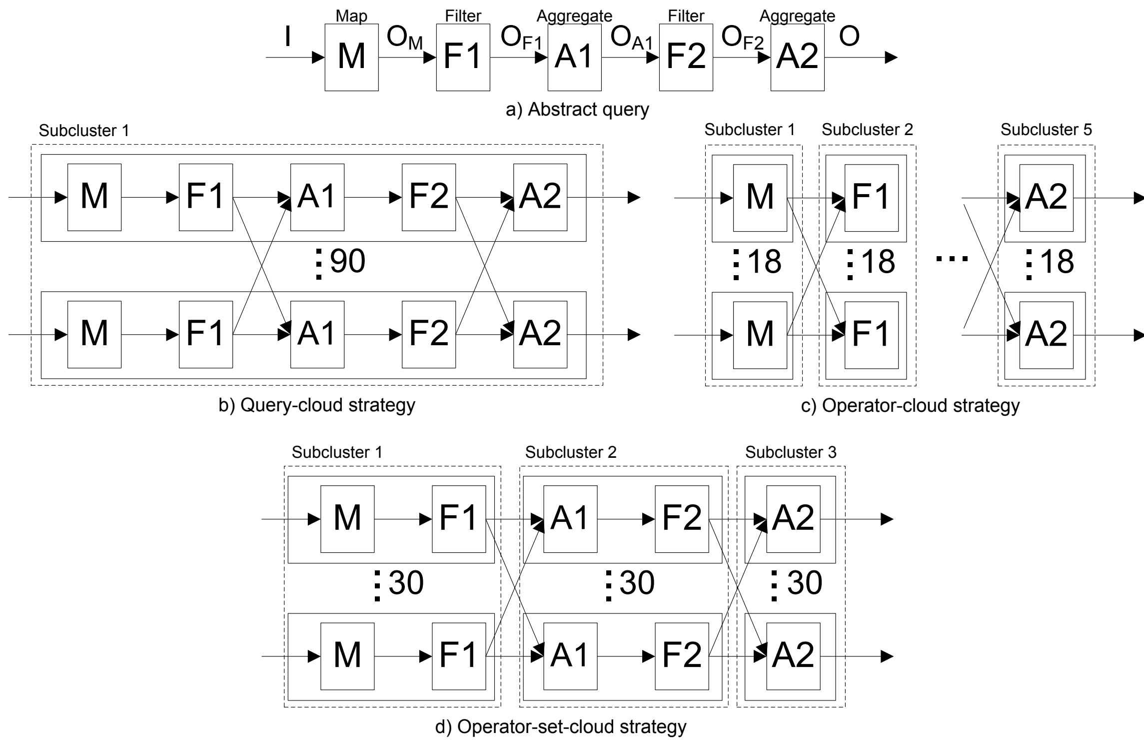


Figure 3.3: Query Parallelization Strategies

all other instances. In the example, each SPE instance receives one ninetieth of the incoming stream and communication takes place before Aggregate operators $A1$ and $A2$.

Since each of the N instances keeps a communication channel towards all the other $N-1$ instances, fan-out overhead is quadratic with respect to N. Number of hops overhead is proportional to the number of stateful operators. This is because tuples are redistributed before each stateful operator. Being s the number of stateful operators defined in a query and N the number of SPE instances, the cost of the QC strategy is:

$$c(QC) = \alpha \cdot N \cdot (N-1) + \beta \cdot s \simeq \alpha \cdot N^2 + \beta \cdot s$$

Operator-cloud strategy - OC (Figure 3.3.c). In this strategy, the parallelization unit is a single operator. Therefore, each operator is deployed over a different subset of nodes (called *subcluster*). In the example we assume available SPE instances are uniformely distributed among query operators. That is, each subcluster is deployed at 18 SPE instances and communication happens from each instance of one subcluster to all its peers in the next subcluster (downstream peers). Each SPE instance must keep a communication channel for each of the downstream subcluster instances.

Being l the number of operators composing the query and N the number of SPE instances,

$$c(OC) = \alpha \cdot \frac{N}{l} \cdot (l-1) + \beta \cdot (l-1) \simeq \alpha \cdot N + \beta \cdot l$$

Operator-set-cloud strategy - SC (Figure 3.3.d). The above parallelization strategies exhibit a trade-off between the parallelization costs (i.e., fan-out and number of hops). The QC strategy minimizes the number of hops overhead (communication happens only before stateful operators) while it maximizes the fan-out overhead (communication happens from all to all the SPE instances). On the other hand, the OC strategy maximizes the number of hops overhead (communication happens between each pair of consecutive operators) while minimizing the fan-out overhead. The Operator-set-cloud strategy aims at minimizing both at the same time, reducing the communication between instances (defining it only before stateful operators) but avoiding the deployment of the entire query at each SPE instance. The basic idea is to split a query into as many *subqueries* as stateful operators. In a query, stateful operators may be interconnected also to stateless ones, leading to different possibilities about which stateless operators to include together with the stateful operator in a subcluster. For instance, we can partition a query into subclusters that contain a stateful operator plus any of the following stateless operator separating it from another stateful operator (or the end of the query). If the query starts with stateless operators, we can also define a first subquery containing all stateless operators before the first stateful one (referred to as *stateless prefix* operators). Using this strategy, the query of Figure 3.3.a has been partitioned into three subqueries, as shown in Figure 3.3.d. As for the Operator-cloud strategy we suppose the available SPE instances are uniformly distributed among subclusters. That is, each subquery is deployed on a subcluster of 30 instances. Subquery 1 contains the map operator M and the filter operator $F1$. Subquery 2 contains the aggregate operator $A1$ and the filter operator $F2$. Finally, subquery 3 contains the aggregate operator $A2$.

The total number of hops per tuple is equal to the number of stateful operators (minus one if no additional subquery is defined for the stateless prefix operators). Communication is required from all instances of a subcluster to the instances of the next subcluster (since each subquery starts with a stateful operator it has to receive tuples from all the instances of the previous subcluster in order to produced the same result the non-distributed version produces). For simplicity, the cost function is calculated assuming that available SPE instances are uniformly distributed to query subclusters. Considering the parallelization of a query that defines a stateless prefix (i.e., a first subquery composed only by stateless operators), being s the number of stateful operators defined in a query and N the number of available SPE instances,

$$c(OS) = \alpha \cdot \frac{N}{s+1} \cdot s + \beta \cdot s \simeq \alpha \cdot N + \beta \cdot s$$

We claim that $c(QC) > c(SC)$ and $c(OC) > c(SC)$, that is, OS is the least expensive parallelization strategy. Comparing QC and SC, it can be noticed that the number of hops overhead h is the same, as they both depend on the number of stateful operators s, while fan-out overhead is higher for QC as it depends on N^2 (while it depends on N for SC). Comparing OC and SC, it can be noticed that the fan-out overhead h is the same, as they both depend on the number of SPE instances

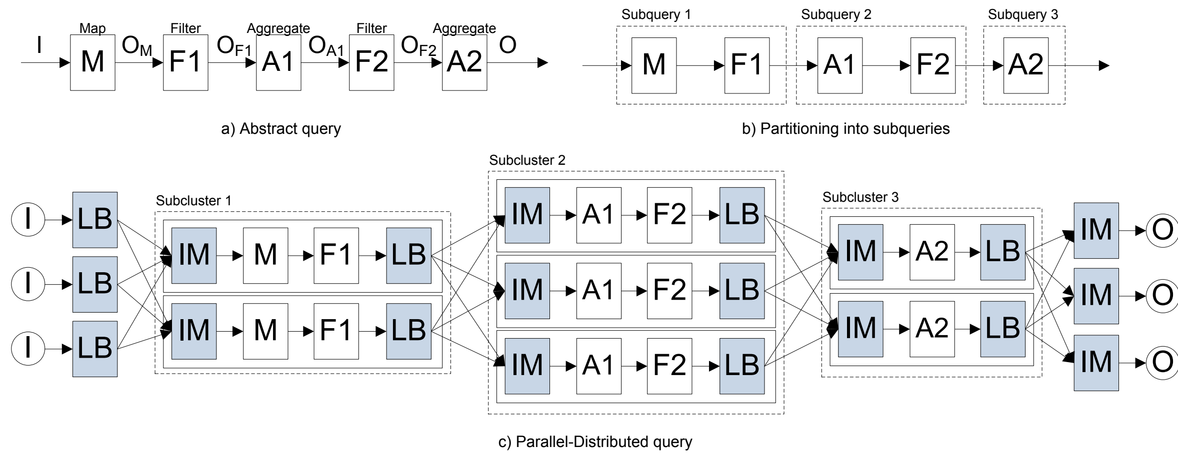


Figure 3.4: Query Parallelization in *StreamCloud*

N, while the number of hops overhead can be higher for OC as it depends on l while it depends on s for SC (by definition $l \geq s$).

StreamCloud employs the Operator-set-cloud strategy as it strikes the best balance between number of hops and fan-out overheads. According to the Operator-set-cloud strategy, queries are split into subqueries and each subquery is allocated to a set of *StreamCloud instances* grouped in a subcluster. In the rest of the document, we use *instance* to denote a *StreamCloud* processing unit (i.e., an instance of *StreamCloud* running at a given node). All instances of a subcluster run the same subquery, called *local subquery*, for a fraction of the input data stream, and produce a fraction of the output data stream. As discussed previously in this section, in order to provide *semantic transparency*, tuples must be routed making sure that tuples that must be processed together are sent to the same operator instance. We present in the following section how communication between subclusters is designed to guarantee semantic transparency.

3.2.1 Operators parallelization

In this section we present how queries are parallelized following *StreamCloud* parallelization strategy (i.e., the Operator-Set-Cloud strategy). Subsequently, we introduce the operators that encapsulate the parallelization logic, Load Balancers and Input Mergers.

Query parallelization is presented by means of the sample query in 3.1, used to compute the number of mobile phones that, on a per-hour basis, make N phone calls whose price is greater than P, for each $N \in [N_{min}, N_{max}]$.

Given a subcluster, we term as *upstream* and *downstream* its previous and next peers, respectively. Figure 3.4.a presents the query, while 3.4.b presents how it is partitioned into subqueries.

As presented in the previous Section 3.2, we must ensure that tuples that must be processed

together are sent to the same operator instance in order to provide semantic transparency. In order to do this, we must define which is the distribution unit used to route tuples from a stream to multiple downstream SPE instances. In *StreamCloud* this minimum distribution unit is referred to as *bucket*. Each stream feeding a parallel operator is partitioned into B disjoint buckets. All tuples belonging to a given bucket are forwarded to and processed by the same downstream instance. Bucket assignment is based on one (or more) fields defined by the tuple schema. Given B distinct buckets and tuple $t = (F_1, F_2, ..., F_n)$, its corresponding bucket b is computed by hashing one or more of its fields modulus B (e.g., $b = \texttt{hash}(F_i, F_j)\%B$). As explained later, the fields used to compute the hash depend on the semantics of the operator to which tuples are forwarded.

Each instance of the downstream subcluster will process tuples belonging to one (or more) buckets. Each subcluster maintains a *bucket registry* that specifies how buckets are mapped to the subcluster instance). More precisely, being BR the bucket registry and given bucket b, $BR[b].dest$ provides the instance that must receive tuples belonging to bucket b. The bucket registry associated to one subcluster is used by its upstream peers to route its incoming tuples. In the following, we say that subcluster instance A "owns" bucket b (that is, A is responsible for processing all tuples of bucket b) if, according to the BR of the upstream subcluster, $BR[b].dest = A$. The assignment of tuples to bucket is endorsed by special operators, called *Load Balancers* (LB). They are placed on the outgoing edge of each instance of a subcluster and are used to distribute the output tuples of the local subquery to the corresponding instance of the downstream subcluster.

Similarly to LBs on the outgoing edge of an instance, *StreamCloud* places another special operator, called *Input Merger* (IM), on the ingoing edge. IMs take multiple input streams from upstream LBs and feed the local subquery with a single merged stream.

Figure 3.4.c presents a sample parallel-distributed version of the considered query. In this example, input stream I is generated by the 3 different data sources. Subquery 1 is assigned to subcluster 1, composed by 2 instances. Subquery 2 is assigned to subcluster 2, composed by 3 instances. Subquery 3 is assigned to subcluster 3, composed by 2 instances. Finally, output stream O is composed by 3 distinct physical streams. Each local subquery has been enriched with an IM on the ingoing edge and a LB on the outgoing edge.

It should be notice that, if subcluster 1 instances are feed directly with the system inputs, the size of the subcluster will be fixed to 3 instances (one instance for each input stream). Similarly, if subcluster 3 instances are connected directly to the system outputs, the size of the subcluster will be fixed to 3 instances (one instance for each output stream). To overcome this limitation, tuples sent by each data source are first processed by a LB. This way, the number of LBs processing tuples from data sources is fixed (3 in the example) but the number of subcluster 1 instance can be arbitrarily chosen by the user. For the same reason, each output stream is preceded by an IM. this way, also the number of subcluster 3 instance can be arbitrarily chosen by the user.

3.2.1.1 Load Balancers

In this section we provide a detailed description of Load Balancers operators. As discussed in the previous Section 3.2.1, load balancers are used to distribute tuples from one local subquery to all the instances of its downstream subcluster. Upstream LBs of a stateful subquery are enriched with *semantic awareness* to guarantee that tuples that must be aggregated/joined together are indeed received by the same instance. That is, they must be aware of the semantics of the downstream stateful operator. In what follows, we discuss the parallelization of stateful subqueries for each of the stateful operators we have considered: Aggregate, Join and Cartesian Product.

Aggregate operator. Parallelization of the Aggregate operator requires that all tuples sharing the same values of the fields specified in the $group-by$ parameter should be processed by the same instance. In the example of Figure 3.4.a Aggregate $A1$ groups incoming tuples by their originating mobile phone while Aggregate $A2$ groups incoming tuples by their $Calls$ field. Upstream LBs partition each input stream into B buckets and use the bucket registry BR to route tuples to the N instances where the subquery instance with the Aggregate is deployed. The field specified as $group-by$ parameter is used at upstream LBs to determine the bucket and the recipient instance of a tuple. That is, let F_i be the field specified as $group-by$, then for each tuple t, $BR[\texttt{hash}(t.F_i)\%B].dest$ determines the recipient instance to which t should be sent. If the $group-by$ parameter is defined by multiple fields, the hash is computed over all of them. LBs are in charge of forwarding tuples sharing the same value of the $group-by$ field to the same subcluster instance. Algorithm 1 line 1 presents the pseudo-code used by LBs to route tuples to parallel Aggregate operators. With respect to the example of Figure 3.4.a, tuples will be routed to Aggregate operator $A1$ hashing field $Caller$ while tuples will be routed to Aggregate operator $A2$ hashing field $Calls$.

Join operator. The Join considered is an equijoin, i.e., its predicate, expressed in Conjunctive Normal Form, contains at least a term that defines an equality between two fields F_i and F_j. *StreamCloud* uses a symmetric hash join approach [ÖV11]. The protocol is similar to the one used for the aggregate operator. The attribute specified in the equality clause is used at upstream LBs (of both left and right input streams) to determine the bucket and the recipient instance of a tuple. Algorithm 1 line 1 presents the pseudo-code used by LBs to route tuples to parallel Join operators.

As an example, suppose a Join operator is used to match CDRs coming from two distinct streams sharing the same calling phone number $Caller$. Upstream LBs routing tuples of the the Join left stream and upstream LBs routing tuples of the Join right stream will both route tuples hashing field $Caller$.

Algorithm 1 Load Balancer Pseudo-Code

LB for Join & Aggregate

Upon: Arrival of t:

```
1: forward(t,BR[hash(t.F_i)%B].dest)
```

LB for Cartesian Product

Upon: Arrival of t:

```
2: for d ∈ BR[hash(t.F_i) % B].dest do
3:     forward(t,d)
4: end for
```

Cartesian Product. The Cartesian Product (CP) operator is defined by an arbitrarily complex predicate (i.e., a predicate involving multiple comparisons like $\leq$,= or $\geq$ between multiple fields of the two input streams schema). Each of the LBs used to route tuples belonging to the left and right stream partition their data into B_l and B_r buckets, respectively. As suggested by the operator name, once the two input streams have been partitioned into buckets, the cartesian product among all the buckets must be run to check all the possible matching pairs of tuples. That is, each bucket of the left stream must be checked against each bucket of the right stream (and vice versa). For this reason, each pair of buckets $b_l \in B_l, b_r \in B_r$ is assigned to an instance. Given a tuple t_l entering the upstream left LB and a predicate over fields F_i, F_j, the tuple is forwarded to all the instances owning the pair of buckets $(b_l, b_r) : b_l = \texttt{hash}(F_i, F_j)\%B_l$. Similarly, a tuple t_r entering the upstream right LB is forwarded to all the instances owning the pair of buckets $(b_l, b_r) : b_r = \texttt{hash}(F_i, F_j)\%B_r$. It should be noticed that, for each incoming tuple of the lest (resp. right) stream, the LB might forward a tuple to multiple downstream instances. From an implementation point of view, the entry BR[b].dest used to feed CP operators (i.e., the ones used at upstreams LBs) to maintain the recipient instances to which tuples of bucket b are forwarded is associated to multiple instances. Algorithm 1, lines 2-4, presents the pseudo-code used by LBs to route tuples to parallel Cartesian Product operators.

Figure 3.5.a depicts a sample query composed by a single CP operator. The query is used to find, between two streams carrying CDR records, mobile phones involved in two consecutive calls (as caller or callee) within a time window of 3 seconds. The operator is defined as:

$$CP\{L.Caller = R.Caller \vee L.Caller = R.Callee \vee$$
$$L.Callee = R.Caller \vee L.Callee = R.Callee, time, Time, 3\}(S_l, S_r, O)$$

It should be noticed that the predicate is not an equijoin because, even if it defines equalities between fields of the two streams schema, it is expressed as a concatenation of OR conditions.

Figure 3.5.a also shows a sample input sequence and the resulting output. Tuples timestamps are indicated on the top of each stream (the values to the right of the "ts" tag). For simplicity, tuples are represented as pairs $Caller, Callee$ (i.e., E, A refers to a phone call made by E to A).

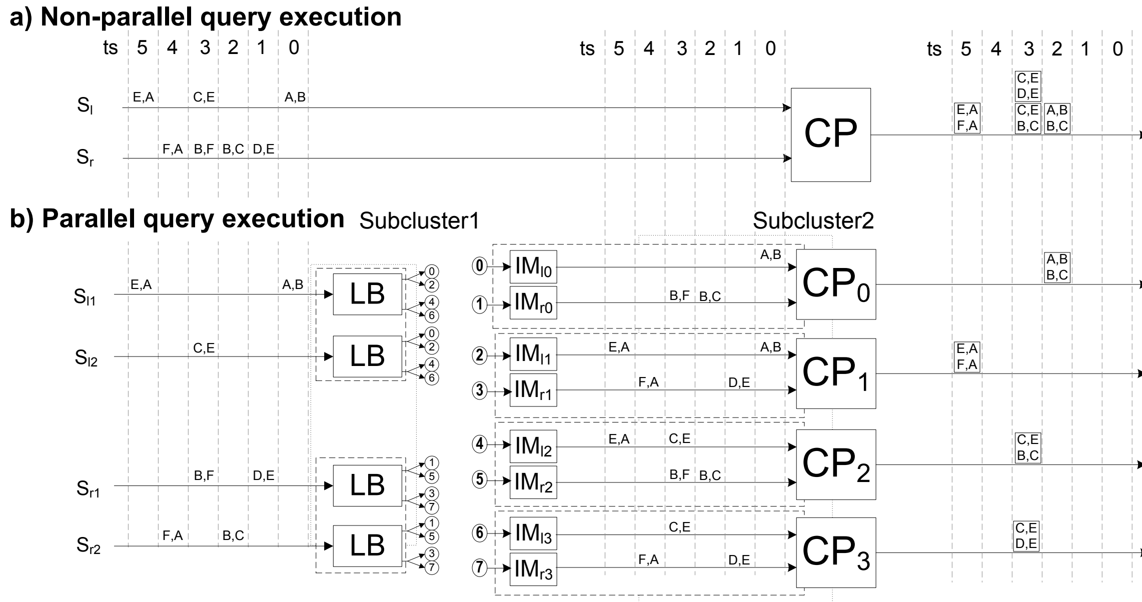


Figure 3.5: Cartesian Product Sample Execution

In the example, a CDR related to a phone call made from mobile phone A to mobile phone B is received at time 0 on stream S_l. A CDR related to a phone call made by mobile phone D to mobile phone E is received at time 1 on stream S_r, and so on. Four tuples are outputted by the CP operator. An output tuple, matching tuple A, B with tuple B, C is produced at time 2. Two output tuples, matching tuple C, E with tuple D, E and matching tuple C, E with tuple B, E, are produced at time 3. Finally, an output tuple matching tuple E, A with tuple F, A is produced at time 5.

Figure 3.5.b shows the parallel version of the query, deployed at 4 SPE instances. Both S_l and S_r are composed by two physical streams. The logical stream S_l is composed by physical streams S_{l1},S_{l2} while the logical stream S_r is composed by physical streams S_{r1},S_{r2}. Each pair of streams is forwarding tuples to one of the 4 instances of the parallel CP operator. The left stream has been partitioned into 2 buckets b_l^0 and b_l^1. In the example, tuples whose $Caller$ field is A,B or E belong to b_l^0 and are sent to the Cartesian Product instances CP_0 and CP_1 (i.e., BR[b_l^0].dest=$\{CP_0, CP_1\}$). Tuple whose $Caller$ field is C,D or F belong to b_l^1 and are sent to the Cartesian Product instances CP_2 and CP_3 (i.e., BR[b_l^1].dest=$\{CP_2, CP_3\}$). Similarly, the right stream has been partitioned into 2 buckets b_r^0 and b_r^1. Tuple whose $Caller$ field is A,B or C belong to b_r^0 and are sent to the Cartesian Product instances CP_0 and CP_2 (i.e., BR[b_r^0].dest=$\{CP_0, CP_2\}$). Tuple whose $Caller$ field is D,E or F belong to b_r^1 and are sent to the Cartesian Product instances CP_1 and CP_3 (i.e., BR[b_r^1].dest=$\{CP_1, CP_3\}$). Each of the 4 CP instances performs one fourth of the whole Cartesian Product on the incoming streams.

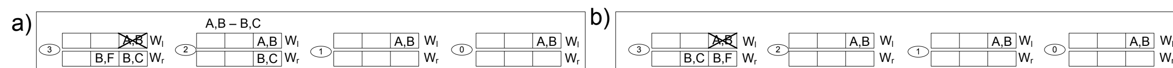


Figure 3.6: Cartesian Product Sample Execution

3.2.1.2 Input Mergers

In this section, we discuss how input mergers (IMs) (similarly to LBs) are designed to guarantee semantic transparency. As presented in Section 3.2.1, input mergers IMs are installed by *StreamCloud* on the ingoing edge of each local subquery when a query is parallelized. The goal of the IMs is to process tuples from multiple input streams (one for each upstream LB) and feed the local subquery with a single merged stream.

Due to the parallel-distributed execution, arrival order of input tuples at one operator might change with respect to a centralized scenario. That is, the tuple order of a logical stream might not be preserved when the latter is split into multiple physical streams, processed by different instances and finally merged. As an example, consider a sequence of three tuples t_1, t_2, t_3 exchanged between two operators $OP1$ and $OP2$, so that $t_1.ts < t_2.ts < t_3.ts$. If the two operators are deployed at the same SPE instance, tuples will be outputted and consumed in timestamp order. On the contrary, if the two operators are deployed at different SPE instances and tuples follow different paths to reach operator $OP2$, there's no guarantee that tuples will be consumed in timestamp order.

A naïve IM that simply forwards incoming tuples in a FIFO manner may lead to incorrect results. The following example consider two possible evolutions of the operator CP_0 presented in Figure 3.5.b. We consider a first evolution where tuples are processed in the same order of the centralized execution. In the second evolution example we consider a different processing order and we show how the corresponding output tuples differ from the ones produced execution that processed the tuples in the same order of a centralized execution. The two evolutions of operator CP_0 left and right windows are presented in Figure 3.6.(a-b). The Figure presents the tuples maintained by the left window (W_l) and the right window (W_r) at seconds 0,1,2 and 3 (the values to the left of each pair of windows).

We consider first the evolution presented in Figure 3.6.a. Tuple (A, B) is received at time 0 and buffered in W_l. Tuple (B, C) is received at time 2 and buffered in W_r. Tuples (A, B) and (B, C) are matched and an output tuple is produced. Finally, tuple (B, F) is received at time 3 and tuple (A, B) is discarded (time distance between (B, F) and (A, B) is equal to the window size, set to 3 seconds in the experiment). Figure 3.6.b presents the second possible evolution of CP_0 windows. In this execution, tuple B, C is received just after B, F. Tuple (A, B) is received at time 0 and buffered in W_l. Tuple (B, F) is received at time 3 and tuple (A, B) is discarded. Finally, tuple (B, C) is received, but no output is produced as tuple (A, B) has been already discarded.

StreamCloud IMs are designed to preserve the tuple arrival order. Hence, the execution of a parallel-distributed operator is equivalent to the one of its centralized counterpart. Multiple timestamp

ordered input streams produced by upstream LBs are merged by the IM into a single timestamp ordered output stream. This implies that the local subquery will process the tuples in the same order of its centralized counterpart, producing timestamp ordered output tuples. To guarantee correct sorting, each IM forwards an incoming tuple if at least one input tuple has been received for each of its input streams. In this case, the forwarded tuple is the one that has the earliest timestamp. To avoid blocking of the IM (if no tuple is received in one of the input streams the IM cannot forward any tuple), upstream LBs send dummy tuples for each output stream that has been idle for the last d time units. Dummy tuples are discarded by IMs and only used to unblock the processing of other streams. In the example of Figure 3.5.b, the input merger of the right input stream of operator CP_0 ensures that tuple (B, C) is forwarded before tuple (B, F), leading thus to the correct result.

Algorithm 2 presents Input Mergers and Load Balancers pseudo-code. With respect to the Input Mergers protocol, each incoming tuple t received on input stream i is buffered (Line 1). If a tuple has been received for each input i, the one having the earliest timestamp is chosen (Lines 2-3). Finally, the tuple is forwarded if it is not a dummy tuple (Lines 4-6). With respect to the Load Balancer protocol, each time a tuple t is forwarded to destination $dest$, the latest timestamp value $lastTS$ is updated to $t.ts$ while the forwarding time of the destination $lastTime[dest]$ is updated to `currentTime()` (Lines 8-9). Subsequently, a dummy tuple carrying the timestamp of the last forwarded tuple is sent to all the destination that have been idle (i.e., no tuple as been forwarded to them) during the last d time units (Lines 10-12).

Algorithm 2 Input Merger Pseudo-Code

Upon: Arrival of tuple t from stream i
1: buffer[i].`enqueue`(t)
2: **if** $\forall i$ buffer[i].`noEmpty`() **then**
3: t_0 = `earliestTuple`(buffer)
4: **if** $\neg$ `isDummy`(t_0) **then**
5: `forward`(t_0);
6: **end if**
7: **end if**

Timeout Management at LBs

Upon: `forward`(t,d):
8: lastTS:= $t.ts$
9: lastTime[d]:= `currentTime`()
Upon: $\exists dest$: `currentTime`() $\geq$ lastTime[$dest$]+d
10: dummy.ts:= lastTS
11: `forward`(dummy, nextSubcluster[$dest$])
12: lastTime[dest]:= lastTime[$dest$]+d

3.3 *StreamCloud* parallelization evaluation

In this section we present the evaluation of *StreamCloud* parallelization technique. We first introduce the evaluation setup. Subsequently, we evaluate the scalability and overhead of the parallelization strategies of Section 3.2. Two additional sets of experiments focus on individual operators and evaluate their scalability for increasing input loads. Finally, we highlight how *StreamCloud* takes full advantage of multi core architectures.

3.3.1 Evaluation Setup

The evaluation was performed in a shared-nothing cluster of 100 nodes (blades) with 320 cores. The blades composing the cluster are presented in Table 3.1. All blades are Supermicro SYS-5015M-MF+ equipped with 8GB of RAM and 1Gbit Ethernet and a directly attached 0.5TB hard disk.

Rack	1	2	3	4
# Nodes	20	20	30	30
CPU	PentiumD @2.8GHz	Xeon 3040 @1.86GHz	Xeon X3220 @2.40GHz	Xeon X3220 @2.40GHz
# Cores	2	2	4	4

Table 3.1: Cluster setup

3.3.2 Scalability of Queries

A first set of experiments was conducted in order to evaluate the performance of the different parallelization strategies introduced in Section 3.2 (i.e., query-cloud strategy QC, operator-cloud strategy OC and operator-set-cloud strategy SC). Initially, we have studied the per-tuple processing cost considering both CPU cycles devoted to tuple processing, CPU cycles devoted to tuple distribution and idle CPU cycles. The distribution of CPU cycles among the ones spent processing tuples and the ones spent forwarding them can be computed comparing the CPU utilization with the cost of each operator (representing the percentage of time the operator is active). E.g., a query with a single operator showing 80% CPU utilization and cost 0.5 results in 40% of CPU cycles devoted to tuple processing, 40% of CPU cycles devoted to tuples distribution processing and 20% of idle CPU cycles.

The query of Figure 3.7 was deployed in a cluster of 30 instances, according to the three parallelization strategies of Section 3.2. For each of the three approaches, Figure 3.8(a) shows how the CPU usage is split among tuples processing, tuples distribution and idle cycles. In the Figure, processing cost is referred to as *Operator*, distribution cost is referred to as *Distribution* while idle cycles are referred to as *Idle*.

The query-cloud approach requires communication from each of the 30 instances to all other

peers, for each of the three stateful operators (roughly $3 \cdot 30^2$ communication channels). Figure 3.8(a) shows that the overall distribution overhead is around 40%. The remaining 60% is used for tuple processing.

The operator-cloud strategy shows a distribution overhead of more than 30%, while CPU ratio used to process tuples is roughly 35%. The unused CPU cycles are due to the difference between the nominal subcluster sizes and the actual ones. For instance, suppose that the optimal distribution plan (i.e., the plan computing how to assign available SPE instances to the query subclusters in order to achieve the highest throughput) requires 4.3 instances for a given subcluster. In this case, the request translates into an assignment of 5 instances, thus leading to one CPU that is not fully utilized.

The operator-set-cloud approach exhibits the lowest communication overhead (roughly 10%). As with the previous approach, the difference between nominal and actual subcluster sizes lead to unused resources. However, since the number of subclusters defined by the operator-set-cloud approach is generally lower than the one defined by the operator-cloud approach, the "idle" percentage of the former is lower than that of the latter.

After studying how CPU cycles are devoted to both tuple processing and distribution, we have focused on the maximum throughput each strategy can achieve. The upper part Figure 3.8(b) shows the scalability of the three approaches, using up to 60 instances. For the query-cloud approach, we could only use half of the instances because the fan-out overhead with more than 30 instances was already exceeding the available resources at deployment time. For each strategy, different subcluster sizes have been evaluated and we only report the configurations that achieved the highest throughput. The *StreamCloud* approach (operator-set-cloud) attains a performance of approximately 40000 tuples/second, that is, a performance from 2 to 4 times better than operator-cloud (reaching approximately 10000 tuples/second) and query-cloud (reaching approximately 20000 tuples/second), respectively.

The bottom part of Figure 3.8(b) shows the evolution of the CPU usage for increasing loads. The query-cloud and operator-set-cloud approaches reach 100% CPU utilization. However, the former hits the maximal CPU usage with low input loads ($\leq 10,000$ tuples/second) while the operator-set-cloud approach sustains up to $40,000$ t/s. The operator-cloud approach shows a maximal CPU utilization of 60% for an input rate of approximately 20000 tuples/second (i.e., roughly 40% of the CPU is not used). As previously discussed, this is due to the difference between the nominal size of each subcluster and the actual one.

3.3.3 Scalability of Individual Operators

In this section we evaluate the scalability of *StreamCloud* operators using the proposed parallelization technique. This set of experiments focuses on the scalability of subclusters with one deployed operator (i.e., Aggregate, Map, Join and Cartesian Product) and shows the associated over-

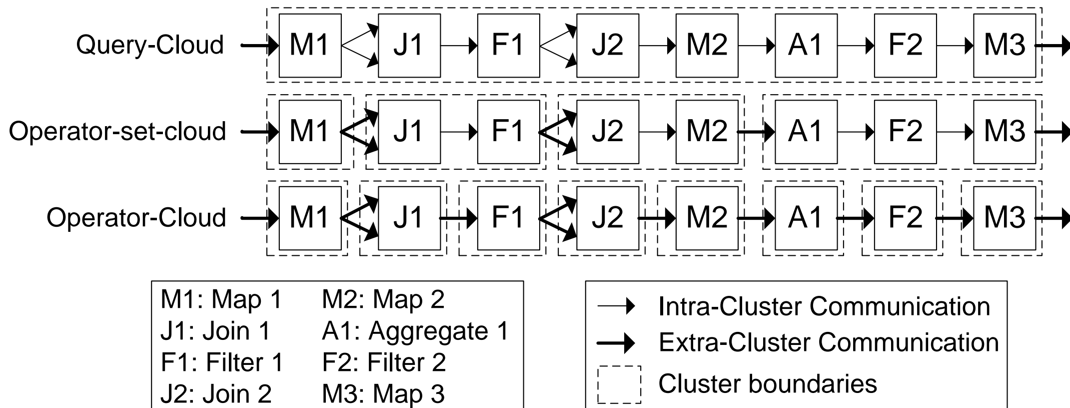


Figure 3.7: Query used for the evaluation.

head. The overhead is measured as average CPU utilization and average input queue length. The average CPU utilization is computed as the average of the CPU utilization of each SPE instance where the parallel-operator is deployed. Similarly, the average input queue length is computed as the average of the input queue length of each SPE instance. The input queue length is taken as a measure of the overhead since the input queue of an operator is usually empty when it is not overloaded while it starts growing as soon as the operator cannot cope with the current input load (i.e., when not all the tuples buffered in the input queue can be processed as soon as they are received).

Three different setups are evaluated for each parallel-operator. The first setup considers a deployment over a single SPE instance. The remaining setups consider a deployment over 20 and 40 SPE instances for the Aggregate, Map and Join operator and a deployment over 16 and 36 SPE instances for the Cartesian Product (we discuss this setup in the following section). In all the experiments, the machines used to run the parallel operators have been chosen among the ones of racks 3 and 4, as they share the same quad-core CPUs and have the same amount of main memory (see Table 3.1). In this set of experiments, each machine runs a single instance of *StreamCloud*.

All the experiments share the input schema presented in Section 2.1, referring to call description records and composed by fields $Caller$, $Callee$, $Time$, $Duration$, $Price$ and coordinates $Caller_X, Caller_Y$ and $Callee_X, Callee_Y$. Input tuples forwarded to the parallel operators are taken from a set of real anonymized CDRs.

The experiments show the throughput behavior as the injected load increases. We experienced a common pattern through all the experiments that can be summarized in three stages: (1) an initial stage with increasing throughput, CPU utilization below 100% and empty queues; (2) a second stage where throughput increases with a milder slope: instances are close to saturation and queues start growing and (3) a final stage showing 100% CPU utilization where queues reach their limits and throughput becomes stable. Each stage can be clearly seen in the bottom parts of Figures 3.3.3.0.1,

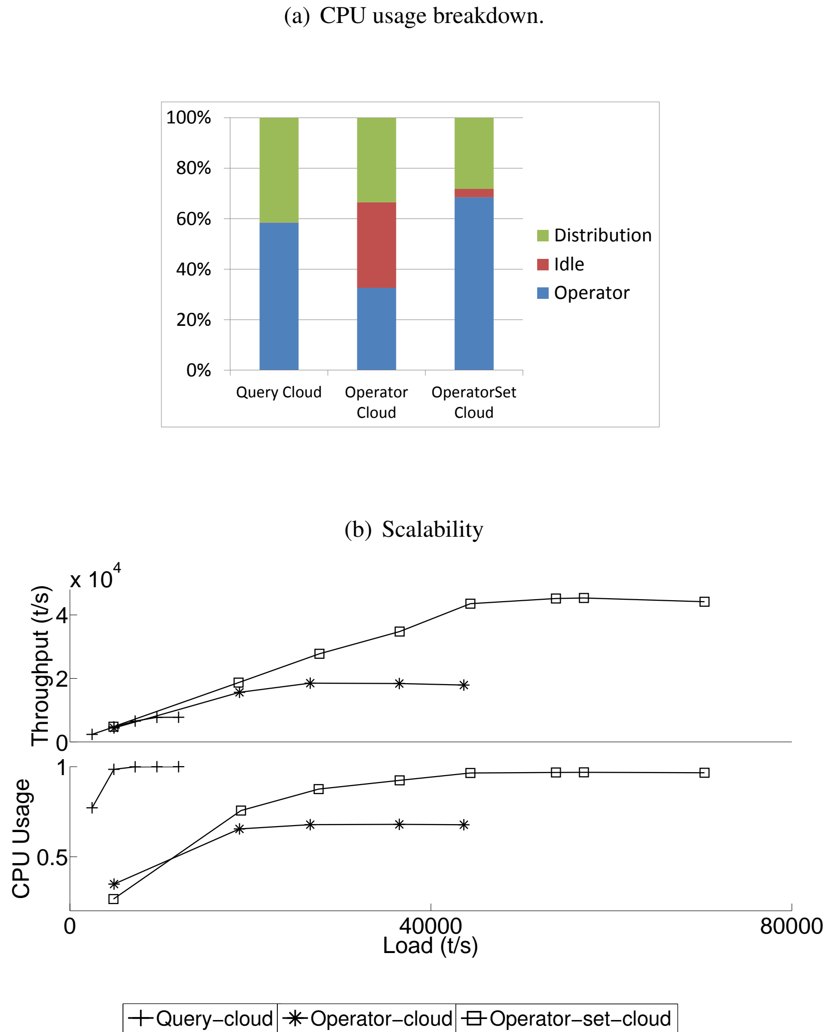


Figure 3.8: Parallelization strategies evaluation

3.3.3.0.1, 3.3.3.0.3 and 3.3.3.0.4 where solid lines show the CPU usage (left Y axis) and dotted lines show queue lengths (right Y axis).

Aggregate operator The Aggregate operator computes the average duration and the number of calls made by each mobile phone; the window size and advance are set to 60 and 10 seconds, respectively. The operator is defined as

$$Agg\{time, Time, 60, 10, Calls \leftarrow count(), Mean_Duration \leftarrow mean(Duration),$$
$$Group - by = (Caller)\}(I, O)$$

The schema of the output tuples is composed by fields $Caller$,$Calls$ and $Mean_Duration$. The Aggregate operator exhibits a linear evolution of the throughput for different input rates and number of instances (upper part of Figure 3.3.3.0.1). When deployed on a single instance, the Aggregate

manages an input rate of roughly 11000 t/s. Twenty instances manage an input rate of roughly 210000 t/s (19 times the rate of the single instance) while forty reach a throughput of roughly 430000 t/s (39 times the rate of the single instance).

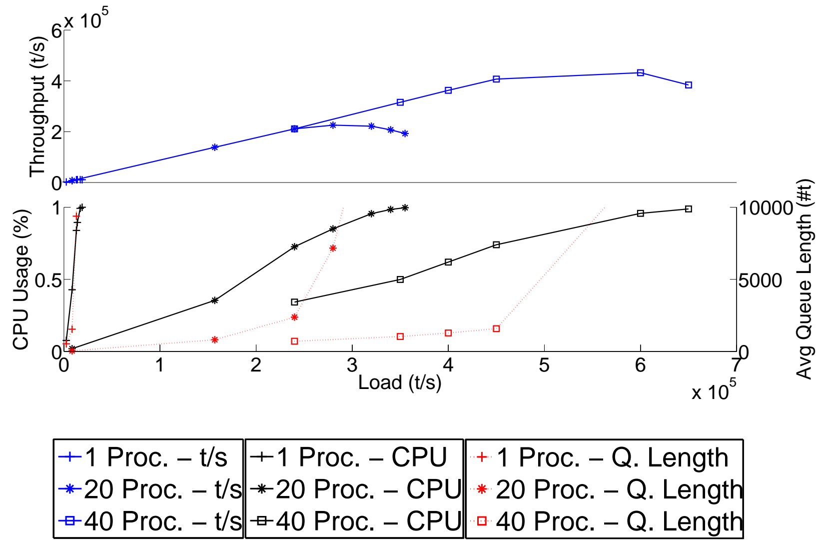


Figure 3.9: Parallel Aggregate operator evaluation.

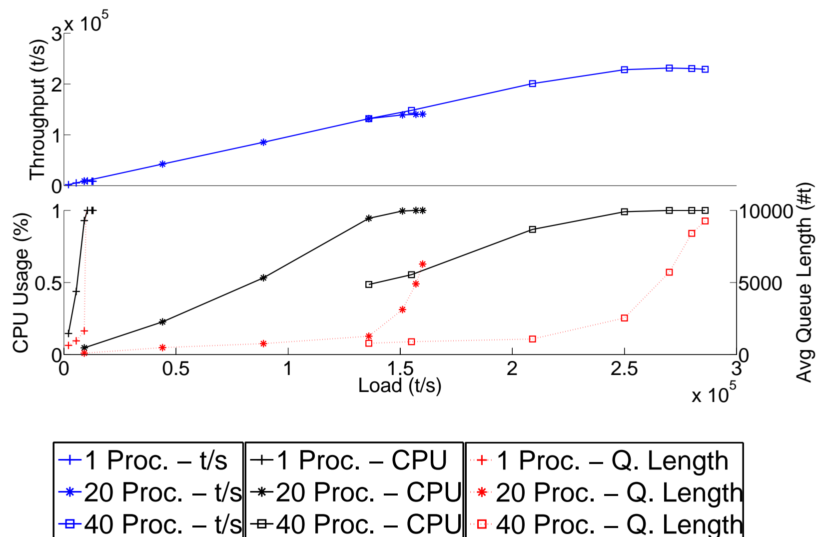


Figure 3.10: Parallel Map operator evaluation.

Map operator For each input CDR, the Map operator computes the call end time, given the start time and the duration. It is defined as

$$M\{Caller \leftarrow Caller, Callee \leftarrow Callee, Time \leftarrow Time, End_Time \leftarrow Time + Duration,$$
$$Price \leftarrow Price, Caller_X \leftarrow Caller_X, Caller_Y \leftarrow Caller_Y, Callee_X \leftarrow Callee_X,$$
$$Callee_Y \leftarrow Callee_Y\}(I, O)$$

The schema of the output tuples is composed by fields $Caller$, $Callee$, $Time$, End_Time, $Price$ and coordinates $Caller_X, Caller_Y$ and $Callee_X, Callee_Y$. As presented in the upper part of Figure 3.3.3.0.1, when deployed on a single instance, the Map manages an input rate of roughly 7000 t/s. When deployed at 20 instances the operator process 138000 t/s (19 times the rate of the single instance); doubling the number of available instances the throughput reaches 270000 t/s (38 times the rate of the single instance).

Join operator The Join operator matches phone calls made by the same user every minute. The Operator is defined as

$$J\{L.Caller = R.Caller \text{ AND } L.Time \neq R.Time, time, Time, 60\}(S_l, S_r, O)$$

The schema of the output tuples is composed by the concatenation of the fields of the left and right input schema (i.e., all the fields composing the CDRs schema).

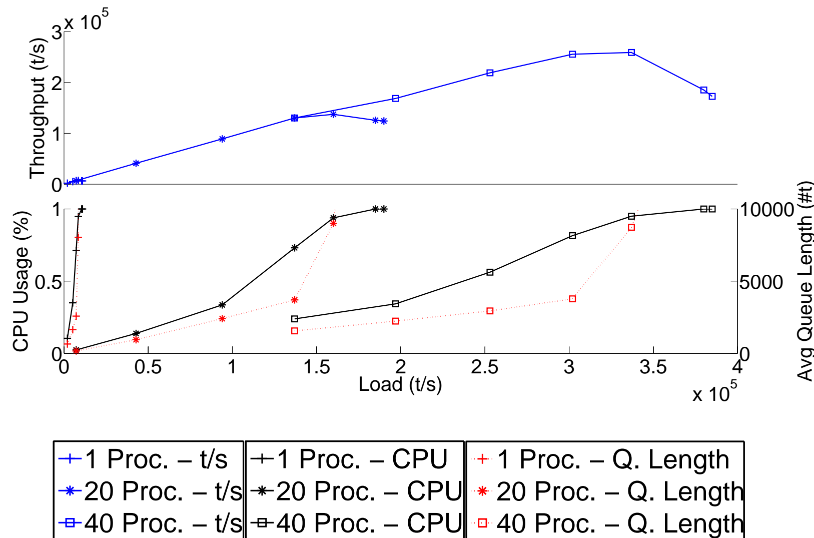


Figure 3.11: Parallel Join operator evaluation.

When deployed over 1 instance, the throughput of the Join operator is of approximately 7000 tuples/second. When deployed over 20 instances, the throughput of the Join reaches approximately 137000 tuples/second (19 times the rate of the single instance). When deployed over 40 instances the

throughput almost doubles, reaching approximately 260000 tuples/second (37 times the rate of the single instance).

Cartesian Product operator The Cartesian Product considered in the evaluation is the one presented in Section 3.2.1.1.3, used to spot, between two CDRs streams, mobile phones involved in two consecutive calls (as caller or callee). The idea is to use this query to spot mobile phones whose consecutive calls are suspiciously too close in time. A minimum interval of time in the realm of seconds is required to end a call and start a new one; for this reason, we spot suspicious phone numbers that appear in consecutive calls withing a time interval of 5 milliseconds. As discussed in Section 3.2.1.1.3, the parallelization of the Cartesian Product operator requires routing of the tuples processed by LBs to multiple instances. This requirement leads to a maximum scalability of $\sqrt{N}$ when parallelizing the operator at N nodes. When deploying the operator 16 nodes we assign one fourth of the partitions of each stream (resp. left and right) to each node. Similarly, when deploying the operator at 36 nodes, we assign one sixth of the partitions of each stream to each node.

The throughput scalability of the operator is shown in the upper part of Figure 3.3.3.0.4. When deployed over 1 instance, the throughput of the Cartesian Product operator is of approximately 14000 tuples/second. When deployed over 16 instances, the throughput of the Join reaches approximately 53000 tuples/second (3.8 times the rate of the single instance). When deployed over 36 instances the throughput reaches approximately 60000 tuples/second (5 times the rate of the single instance).

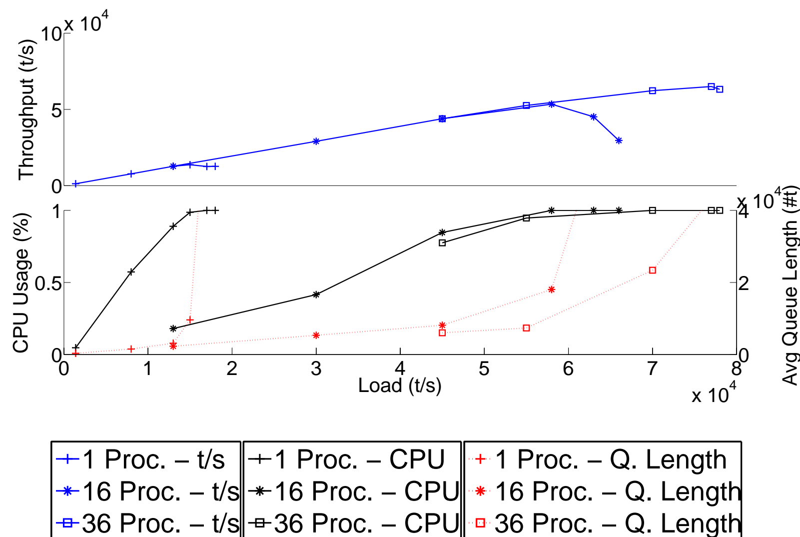


Figure 3.12: Parallel Cartesian Product operator evaluation.

3.3.4 Multi-Core Deployment

In this experiment, we aim at quantifying the scalability of *StreamCloud* with respect to the number of available cores in each node, that is, to evaluate whether *StreamCloud* is able to effectively use the available CPUs/cores/hardware threads of each node.

We focus on the Aggregate operator of Section 3.3.3, used to compute the average duration and the number of calls made by each mobile phone (window size and advance are set to 60 and 10 seconds, respectively). The evaluation has been conducted deploying the operator over 1, 10 and 20 quad-core nodes, respectively. On each node, up to 4 *StreamCloud* instances were deployed.

Figure 3.13 shows linear scalability with respect to the number of *StreamCloud* instances per node. When instantiating up to one *StreamCloud* instance per node, the single node setup achieves a throughput of approximately 11400 tuples/second, 10 nodes achieve 110000 tuples/second (9.9 times more) while 20 nodes achieve 215000 tuples/second (19 times more). When instantiating two *StreamCloud* instances per node, the throughputs of the single node, 10 nodes and 20 nodes setups grow to 22000, 218000 and 415000 tuples/second, respectively; achieving 1.9 times higher throughput on average than the single instance case. When instantiating three *StreamCloud* instances per node, the throughputs of the single node, 10 nodes and 20 nodes setups grow to 32500, 318000 and 630000 tuples/second, respectively; achieving 2.8 times higher throughput on average than the single instance case. Finally, when instantiating four *StreamCloud* instances per node, the throughputs of the single node, 10 nodes and 20 nodes setups grow to 42500, 421000 and 815000 tuples/second, respectively; achieving 3.7 times higher throughput on average than the single instance case.

The reason why *StreamCloud* scales linearly with respect to the number of available cores of each machine is due to its threads scheduling policy. *StreamCloud* defines three threads for collecting tuples received from upstream instances, processing them and for sending them to the downstream instances. As the scheduling policy enforces only one active thread at a given point in time, we can deploy as many *StreamCloud* instances as available cores and scale with the number of cores per node. We give a detailed overview of the tuples processing paradigm defined in Borealis in Appendix A.3.

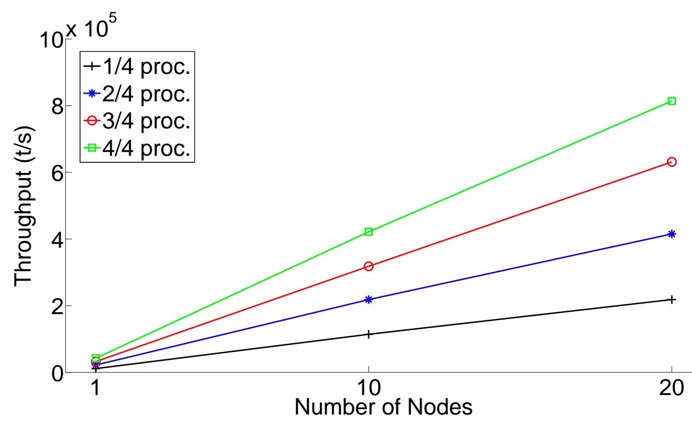


Figure 3.13: Join maximum throughput vs. number of *StreamCloud* instances per node.

Part IV

STREAMCLOUD DYNAMIC LOAD BALANCING AND ELASTICITY

Chapter 4

StreamCloud Dynamic Load Balancing and Elasticity

As introduced in Section 3.1, the number of instances assigned to run a parallel-distributed query might be inadequate depending on the current system input load. In order to avoid under-provisioning (i.e., the number of assigned instances cannot cope with the system load) or over-provisioning (i.e., assigned instances are not running at their full capacity), the system should be able to dynamically provision and decommission instances depending on the current system load.

In this chapter, we presents *StreamCloud* dynamic load balancing and elasticity protocols. It should be noticed that elastic capabilities should be combined with dynamic load balancing, to make sure that instances are provisioned (resp. decommissioned) only when the system as a whole cannot cope with the current incoming load (resp. when the system as a whole is running below its full capacitiy).

We first introduce *StreamCloud* architecture, presenting its different composing units and how they interact. Subsequently, we present the protocol used to transfer operators state across nodes and, finally, the conditions upon which dynamic load balancing or elastic reconfiguration actions are taken.

4.1 *StreamCloud* Architecture

This section presents the main components of *StreamCloud* architecture. We first briefly discuss which are tasks the system needs to address in order to attain elasticity and dynamic load balancing; subsequently, we present which components have been designed to address these tasks.

The first task that is needed by a system that provides dynamic load balancing and elastic capabilities is *monitoring of running instances*, in order to continuously check whether a reconfiguration action should be triggered. In case of a dynamic load balancing, provisioning or decommissioning action, decisions about how to reconfigure the system are taken based on the current state of each running instance. Hence, the system will need to define periodically *reports* that can be used to decide how to redistribute the load during a reconfiguration action. Finally, in order to provision or decommission instances, the system must also define a *pool* of available instance, from where instances are taken in case of provisioning and to where instances will be maintained once they are decommissioned. It should be notice that, due to the partitioning of queries into subqueries (discussed in Section 3.2), decisions about dynamic load balancing, provisioning and decommissioning reconfiguration actions cannot be taken at the whole “query level", but must be taken independently for each query subcluster.

Figure 4.1 illustrates a sample configuration with *StreamCloud* elastic management components.

In the example, we consider the query presented in section 3.2, computing the number of mobile phones that, on a per-hour basis, make N phone calls whose price is greater than P, for each $N \in [N_{min}, N_{max}]$. Following *StreamCloud* parallelization technique, the query has been partitioned into two subqueries, one for each stateful operator. More precisely, subquery 1 contains operators $M, F1, A1$ while subquery 2 contains operators $F2, A2$. In the example, subclusters 1 and 2 have been deployed over 2 and 3 *StreamCloud* instances, respectively.

StreamCloud's architecture includes the following components: StreamCloud *Manager* (SC-Mng), *Resource Manager (RM)* and *Local Managers (LMs)*. Each *StreamCloud* instance runs a LM to monitor the instance resource utilization (i.e., CPU consumption) and its incoming load. Each LM sends periodical reports to SC-Mng. Furthermore, each LM is able to reconfigure the local query when nodes are provisioned, decommissioned or a dynamic load balancing action is triggered. LMs reports are collected by SC-Mng that aggregates them on a per-subcluster basis. Depending on the collected data, SC-Mng may decide to reconfigure the system triggering a dynamic load balancing action, provisioning new instances or decommissioning part of the existing ones. Reconfiguration actions are taken and executed independently for each subcluster. Whenever instances must be provisioned or decommissioned, SC-Mng interacts with the RM. The latter maintains a pool of assigned and available *StreamCloud* instances. Each time an instance is assigned it moves its ID from the available instances pool to the assigned instances pool. Similarly, each time an instance is decommissioned, it moves its ID from the assigned instances pool to the available one. *StreamCloud* Resource Manager has been implemented as a generic interface so that the system is able to interact with any cloud data center module. The Resource Manager can interact with a public cloud based on the infrastructure as a service model (e.g., Amazon EC2 [Ama]). On the other hand, the resource manager can also interface with a private cloud infrastructure, like OpenNebula [Ope] or Eucalyptus [Euc]. In this

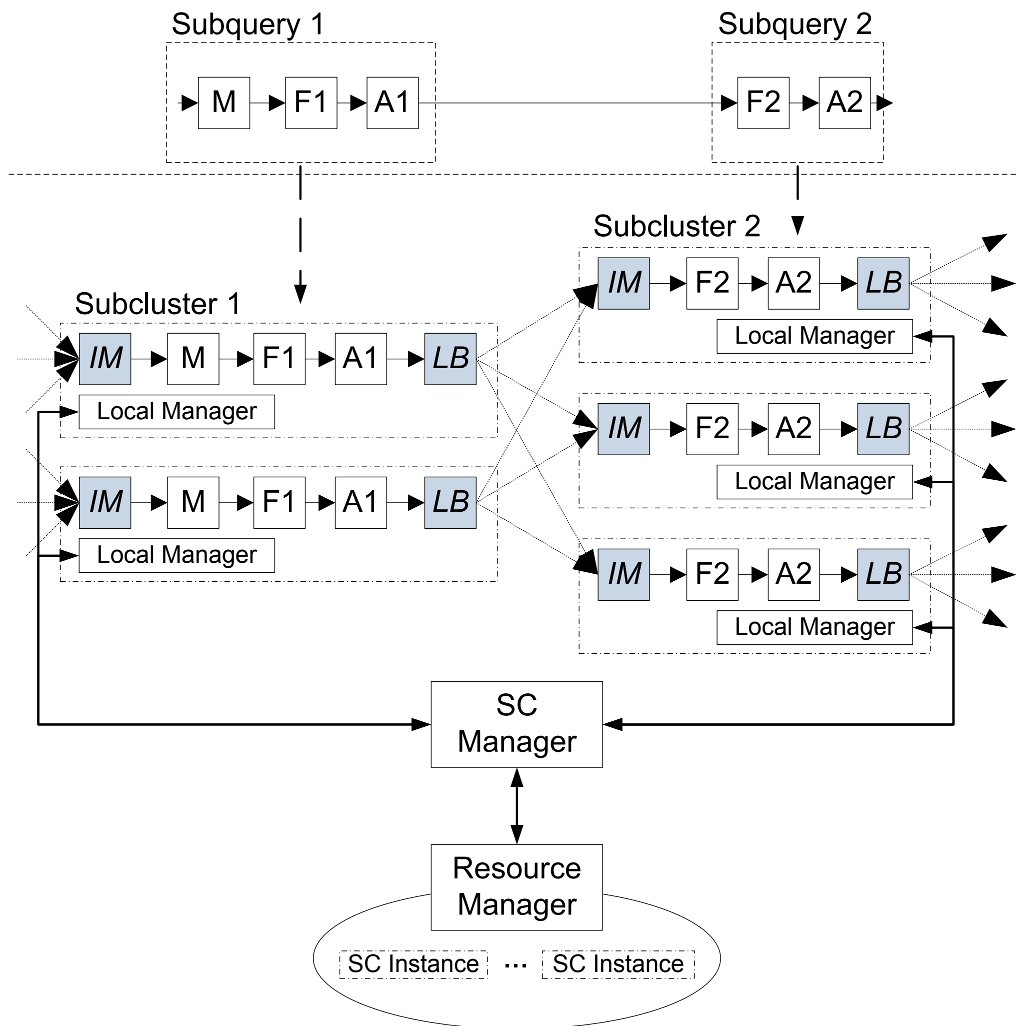


Figure 4.1: Elastic management architecture.

second case, *StreamCloud* provides an implementation for the Resource Manager. The implementation allows the user to maintain a pool of available instances that can optionally have *StreamCloud* software running on them without any query deployed. If no query is deployed, resources consumed by *StreamCloud* are negligible: the CPU consumption of an idle *StreamCloud* instance is in the order of 0.002% while its memory footprint is around 20MB. Hence, while in the available pool, a node can be used by other applications that will not be affected by the idle *StreamCloud* instance. *StreamCloud* software is kept active in available instances to reduce the provisioning time of new instances, accounting only for deployment time.

As motivated in 3.1, dynamic load balancing and elasticity are really essential for a parallel-distributed SPE in order to avoid under-provisioning and over-provisioning. A parallel-distributed SPE is under-provisioned whenever the available nodes cannot cope with the system input load. On

the other hand, it is over-provisioned if the available nodes are not fully utilized. It should be noticed that under-provisioning and over-provisioning depend on the current system input load. Due to variations in the system load, a given deployed parallel-distributed query may suffer under or over-provisioning. *StreamCloud* complements elastic resource management with dynamic load balancing to guarantee that new instances are only provisioned when a subcluster as a whole is not able to cope with the incoming load. Both dynamic load balancing and elasticity techniques boil down to the ability to reconfigure the system in an online and non-intrusive manner. The next section is devoted to this topic.

4.2 Elastic Reconfiguration Protocols

With respect to dynamic load balancing, provisioning or decommissioning, a reconfiguration action is the series of steps taken to move part of the computation of an instance to another instance. In order to provide a way to transfer only a portion of an instance computation, we need to define which is the minimal distribution unit. As present in Section 3.2.1, operators parallelization is performed partitioning streams into buckets. In *StreamCloud*, the minimal distribution unit that can be transferred between two distinct instances is a single bucket. Hence, a subcluster is reconfigured transferring the ownership of one or more buckets from an *old owner* instance to a *new owner* instance. For instance, one (or more) buckets owned by an overloaded instance may be transferred to a less loaded instance or to a new instance. When transferring a bucket, the idea is to define a point in time p so that tuples $t : t.ts < p$ are processed by the old owner while tuples $t : t.ts \geq p$ are processed by the new owner. This is straightforward for stateless operators: as they process incoming tuples individually and do not maintain any state, transferring a bucket only implies changing the destination instance to which tuples are routed. Tuples $t : t.ts < p$ will be routed to the old owner instance; starting from tuple $t : t.ts = p$, tuples will be routed to the new owner instance. However, state transfer is more challenging when reconfiguring stateful operators. The challenge arise from two aspects: (1) reconfiguring a stateful operator not only involves a modification about how tuples are routed but must also define a way for transferring the operators state and (2) due to the sliding window semantic, a single tuple may contribute to several windows.

Figure 4.2 presents the windows evolution of an aggregate operator having window $size = 3600$ and $advance = 600$. The first window includes tuples t having timestamps $0 \leq t.ts < 3600$. The second window includes tuples t having timestamps $600 \leq t.ts < 4200$, and so on. As shown in the figure, a tuple t with timestamp $t.ts = 2100$ contributes to 4 consecutive windows.

When reconfiguring a subcluster, *StreamCloud* triggers one (or more) reconfiguration actions. Each action transfers the ownership of a bucket from the old owner to the new owner within the same subcluster. Each reconfiguration action affects the old owner and the new owner instances and the LBs

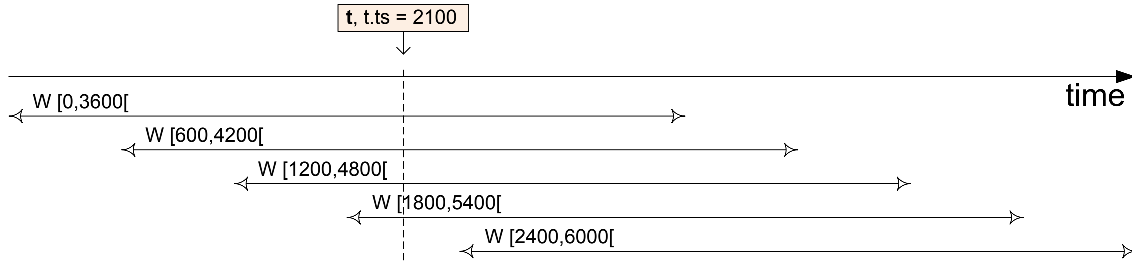


Figure 4.2: Example of tuple contributing to several windows

of the upstream subcluster. *StreamCloud* defines two different elastic reconfiguration protocols that trade completion time for communication between the instances being reconfigured. When presenting the protocols, we refer to a generic bucket b being transferred between the old owner instance A and the new owner instance B. Both protocols share the initial steps. We first present this common prefix and, subsequently, the two protocols individually. As introduced in Section 3.2.1, each subcluster employs a *Bucket Registry* BR. Table 4.1 presents the parameters used in the following protocols.

4.2.1 Reconfiguration Start

This section presents the common prefix protocol of the two reconfiguration protocols provided by *StreamCloud*. As presented in the previous section, both reconfiguration protocols define a point in time (referred to as $startTS$) so that tuples having timestamp earlier than $startTS$ are processed by the old owner instance A while tuples having timestamp greater than or equal to $startTS$ are processed by the new owner instance B. In order to define $startTS$, we need to communicate the start of a reconfiguration action to all the involved operators and make sure that all of them agree on the same value of $startTS$. This requires some communication between the involved operators. We present below the detailed description of the initial phase of the reconfiguration protocols.

The ownership transferring action is triggered by the SC-Mng in case a new instance is provi-

BR	Bucket registry
$BR[b].dest$	*StreamCloud* instances to which b tuples are forwarded
$BR[b].owner$	*StreamCloud* instance owning bucket b
$BR[b].state$	State of bucket b. If b is being transferred, $state = transferring$
$BR[b].startTS$	Timestamp from which ownership transferring starts
$BR[b].switchTS$	Timestamp from which new owner instance starts processing b tuples (in case of ownership transferring)
$BR[b].endTS$	Timestamp from which old owner instance stop processing b tuples (in case of ownership transferring)
OP_{id}	id of an operator

Table 4.1: Parameters used by elasticity protocols

sioned, an instance is decommissioned or the load between two instances has to be balanced. The reconfiguration of a subcluster starts sending a `reconfigCommand` to all the LBs of its upstream peer. Command `reconfigCommand` specifies which bucket b will be transferred from the old owner instance A to the new owner instance B. The goal of this first protocol is to obtain a common reconfiguration start timestamp ($startTS$) shared by both instances A and B. Each LB proposes the latest forwarded tuple timestamp, the highest timestamp becomes the logical start of the reconfiguration.

Algorithm 3 shows the pseudocode common to both reconfiguration protocols. The main actions performed by each LB consist in updating their bucket registry entry for bucket b and in proposing a timestamp to both A and B as a candidate for $startTS$. Upon reception of the `reconfigCommand`, each LB updates the destination to which tuples belonging to bucket b are forwarded, setting as destinations both A and B (Alg. 3 line 1). Subsequently, it updates its bucket registry entry for bucket b setting parameter $endTS$ to ∞. Parameter $endTS$ specifies the end timestamp of the reconfiguration action. It is initially set to ∞ as the exact value will provided by instance A (or instance B) once it computes the $startTS$ timestamp (Alg. 3 line 2). Subsequently, the bucket registry specifying the $owner$ instance of bucket b is the to B (Alg. 3 line 3). Finally, the state of bucket b is set to $reconfiguring$ and a control tuple is sent to both reconfiguring instances. The control tuple carries the information related to the timestamp ts proposed as $startTS$ by each LB (set as the timestamp of the last tuple forwarded), the bucket being reconfigured and the new owner instance B (Alg. 3 lines 4-6).

Upon reception of all the `controlTuple` messages, both A and B set $startTS$ as the highest timestamp proposed by LBs (Alg. 3, lines 7-10). $startTS$ will be the same for the old owner instance and the new owner instance as the both process all the control tuples produced by their upstream LBs. Despite both A and B compute $startTS$ in the same way, we present them separately in the algorithm to stress that each of them play a different role (old or new owner) and find it out comparing its operator id OP_{id} with the $newOwner$ carried by the control tuples. Once $startTS$ has been set, both A and B update their bucket registry entry $owner$ to the new owner instance B. The pseudo code for instances A and B is shown in 3 lines 7-10.

Figure 4.3 shows a sample execution with the information exchanged between the instances involved in the reconfiguration and their upstream LBs. In the example, we suppose two upstream LBs exist, namely LB1 and LB2. For the sake of simplicity, the figure only considers tuples and control messages related to bucket b. However, we stress that all the involved instances (i.e., LBS, A and B) might simultaneously process tuples belonging to other buckets.

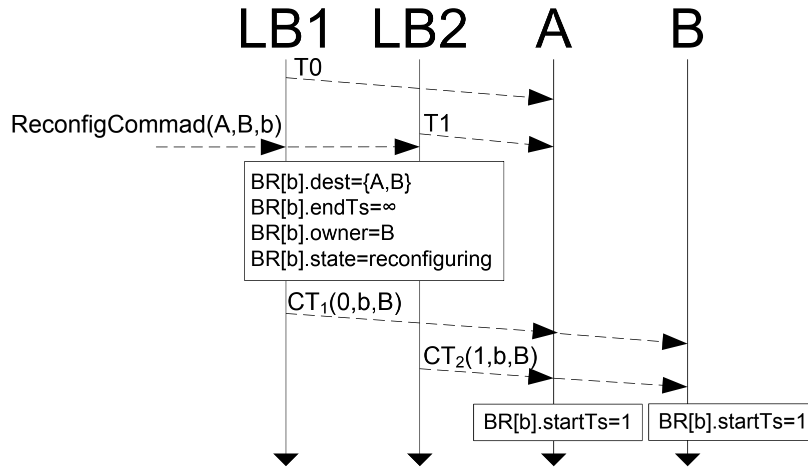


Figure 4.3: Example execution of reconfiguration protocols shared prefix

Algorithm 3 Reconfiguration Start Protocol.

LB

Upon: receiving of `reconfigCommand`(A,B,b)

1: $BR[b].dest = \{A, B\}$
2: $BR[b].endTS = \infty$
3: $BR[b].owner = B$
4: $BR[b].state = reconfiguring$
5: ts = timestamp of last sent tuple
6: send `controlTuple`(ts,b,B) to $BR[b].dest$

Old owner A

Upon: receiving all `controlTuple`(ts_i,b,$newOwner$) $\wedge$ $OP_{id} \neq newOwner$

7: $BR[b].startTS = max_i\{ts_i\}$
8: $BR[b].owner = newOwner$

New owner B

Upon: receiving all `controlTuple`(ts_i,b,$newOwner$) $\wedge$ $OP_{id} = newOwner$

9: $BR[b].startTS = max_i\{ts_i\}$
10: $BR[b].owner = newOwner$

Initially, LBs forward tuples belonging to bucket b to their destination instance A. Tuple $T0$ (timestamp 0) is forwarded by LB1 while tuple $T1$ (timestamp 1) is forwarded by LB2. Upon reception of the `reconfigCommand`, both LBs update their $BR[b]$ registry entries and send control tuples CT_1 and CT_2 to both A and B. Control tuple CT_1 carries timestamp 0 as the latter is the timestamp of the last tuple forwarded by LB1. Similarly, control tuple CT_2 carries timestamp 1.

Upon reception of both control tuples, instances A and B set $BR[b].startTS$ to 1, as it is the highest timestamp forwarded by any control tuple.

In the following sections we provide the detailed description of the two reconfiguration protocols provided by *StreamCloud*.

4.2.2 Window Recreation Protocol

The Window Recreation Protocol has been designed to avoid any communication between the instances being reconfigured. The idea is to let A process the tuples belonging to its current window while feeding B in parallel with the same tuples. When A processes the last tuple belonging to its current windows, the new incoming tuples are only sent to B, resuming regular processing. The period during which tuples are sent both to A and B is proportional to the window size. This means that this protocol is of interest for short-length window (i.e., in the range of seconds or tens of seconds).

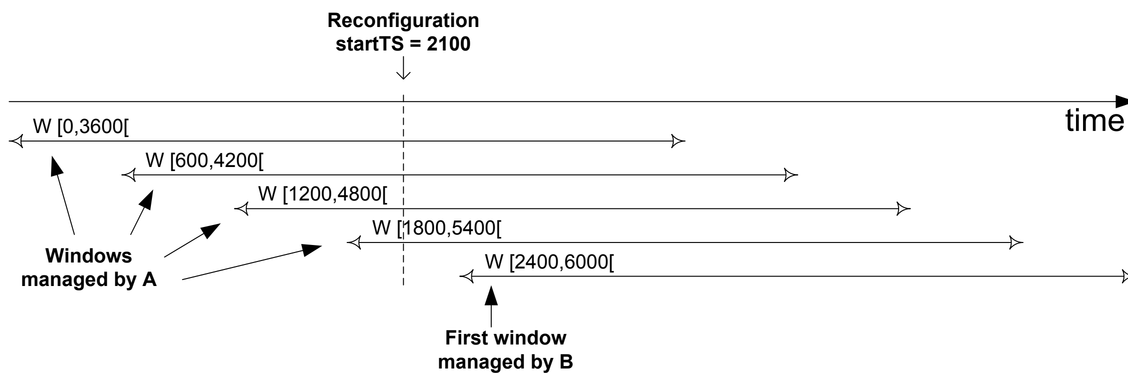


Figure 4.4: Example of which windows are managed by old owner or new owner instances during Window Recreation protocol

An example of the windows that are managed by the old owner instance A and the one managed by the new owner instance B is presented in Figure 4.4. In the example, we consider the same windows evolution of the aggregate operator presented in Figure 4.2. Timestamp $startTS$ is the to 2100.

Once the reconfiguration action has started as presented in the previous section, upstream LBs are sending to both old owner instance A and new owner instance B. In order to complete the reconfiguration, the old owner instance A that is currently maintaining b window must inform upstream LBs about the timestamp representing the end of the reconfiguration (referred to as $endTS$). Input tuples having timestamps greater than or equal to $endTS$ will be forwarded by LBs only to the new owner instance B. The old owner instance A must also inform the new owner instance B about the first window the latter will have to process. The Pseudocode for the Window Recreation protocol is shown in Algorithm 4. A is in charge of processing all windows with an initial timestamp earlier

than $BR[b].startTS$, while B is in charge of processing the following ones. Given $BR[b].startTS$, window $size$ and $advance$, A computes $BR[b].endTS$ using function `computeEndTS` [1]. (Alg. 4, line 8). $BR[b].endTS$ represents the highest tuples timestamp A will process (i.e., tuples having timestamp $t.ts \geq BR[b].endTS$ will be discarded, Alg. 4, lines 10-14).

Algorithm 4 Window Recreation Protocol.

LB

Upon: receiving t for bucket b
1: **if** $BR[b].status = reconfiguring \wedge t.ts > BR[b].endTS$ **then**
2: $BR[b].dest = BIM[b].owner$
3: $BR[b].status := normal$
4: **end if**
5: Send t to $BR[b].dest$
Upon: receiving `EndOfReconfiguration`(b, ts)
6: $BR[b].endTS := ts$

Old owner A

Upon: $BR[b].startTs = max_i\{ts_i\}$
7: $BR[b].endTS :=$ `ComputeEndTS`$(BR[b].startTS)$
8: Send `EndOfReconfiguration`$(b$, $BR[b].endTS)$ to upstream LBs
Upon: receiving tuple t for bucket $b \wedge OP_{id} \neq BR[b].owner$
9: **if** $t.ts < BR[b].endTS$ **then**
10: process t
11: **else**
12: discard t
13: **end if**

New owner B

Upon: $BR[b].startTS = max_i\{ts_i\}$
14: $BR[b].switchTS :=$ `computeSwitchTS`$(BR[b].startTS)$
Upon: receiving tuple t for bucket $b \wedge OP_{id} = BR[b].owner$
15: **if** $t.ts < BR[b].switchTS$ **then**
16: discard t
17: **else**
18: Start regular processing of bucket b
19: **end if**

After computing $BR[b].endTS$, A sends an `endOfReconfiguration` message to upstream LBS (Alg. 4, line 8). The latter update their $BR[b].endTS$ entry (Alg. 4, line 7). As soon as an incoming tuple t timestamp is equal to or greater than $BR[b].endTS$, entries $BR[b].dest$ and $BR[b].status$ are updated and, starting from t, tuples are sent only to B (Alg. 4, lines 1-6). After computing $BR[b].startTS$, new owner instance B computes $BR[b].switchTS$ using function

[1] All windows of the buckets being reconfigured share the same $startTS$. This is because *StreamCloud* enforces that all windows are aligned to the same timestamp as in [AAB+05a].

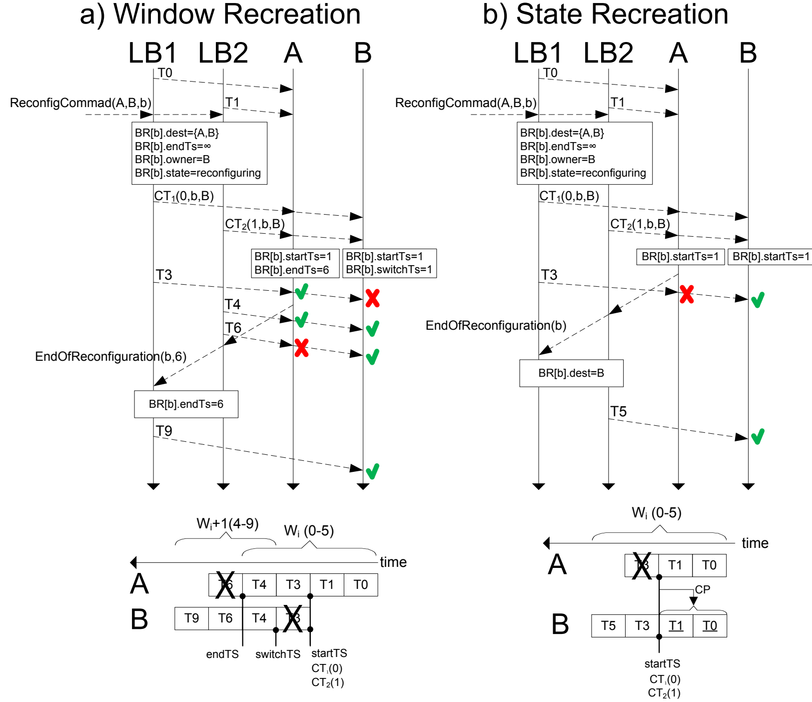


Figure 4.5: Sample reconfigurations.

`computeSwitchTS` (Alg. 4, line 15). Similarly to `computeEndTS`, `computeSwitchTS` depends on $BR[b].startTS$, window $size$ and $advance$. $BR[b].switchTS$ represents the earliest tuple timestamp B will process (i.e., tuples having timestamp $t.ts < BR[b].switchTS$ will be discarded, Alg. 4, lines 16-20).

Figure 4.5.a shows a sample execution where windows are time-based and have size and advance set to 6 and 2, respectively. The example is the continuation of the example 4.3. The bottom part of Fig. 4.5.a shows the windows managed by A and B, respectively.

Initially, both A and B compute the reconfiguration start time (1 in the example). Subsequently A computes $BT[b].endTS = 6$ ($BT[b].endTS$ depends on $BT[b].startTS$, window $size$ and $advance$). Similarly, B computes $BR[b].switchTS = 4$. A becomes responsible for all windows up to W_i since its starting timestamp (0) is lower than $BR[b].startTS$ (1). B becomes responsible for all windows starting from W_{i+1} since its starting timestamp (4) is greater than $BR[b].startTS$ (1). After computing $BT[b].endTS = 6$, A sends the `endOfReconfiguration` message to upstream LBs. Tuples $T3$ to $T4$ are sent from LBs to both instances because their timestamp is earlier than $BR[b].endTS$. Tuple $T6$ should be sent only to B (its timestamp being 6) but it is sent to both instances because it is processed by LB2 before receiving the `endOfReconfiguration` message. Tuples $T3$ is discarded by B ($T3.ts < BR[b].switchTS$). Tuples $T6$ is discarded by A ($T6.ts = BR[b].endTS$). Starting from tuple $T9$, LBs only forward tuples to B.

4.2.3 State Recreation Protocol

Algorithm 5 State Recreation Protocol.

LB

Upon: receiving t for bucket b
1: Send t to $BR[b].dest$
Upon: receiving `EnfOfReconfiguration`(b)
2: $BR[b].dest = BR[b].owner$

Old owner A

Upon: $BR[b].startTS = max_i\{ts_i\}$
3: Send `EndOfReconfiguration`(b) to upstream LBs
4: cp:= `Checkpoint`(b)
5: Send cp to $BT[b].owner$
Upon: receiving t for bucket $b \wedge OP_{id} \neq BR[b].owner$
6: Discard t

New owner B

Upon: receiving t for bucket $b \wedge OP_{id} = BR[b].owner$
7: Buffer t
Upon: receiving checkpoint cp
8: `install`(cp)
9: process all buffered tuples $t : t \in b \wedge t.ts \geq BR[b].startTS$
10: start regular processing of bucket b

The protocol presented in the previous section has been designed to avoid any communication between instances A and B. This leads to a completion time that is proportional to the window size. Hence, the protocol is not suitable when operating with stateful operators defining large windows (e.g., 1 hour). Complementary to this protocol, the State Recreation protocol has been designed to transfer the ownership of a bucket minimizing the completion time independently of the window size.

Once the reconfiguration has been triggered and the common prefix of both reconfiguration protocols has been completed, the old owner instance A performs two main tasks: (1) it alerts the upstream LBs that they can start sending tuples only to the new owner instance B and (2) it transfer its state to the new owner instance B. The state recreation protocol must take into account that, due to the fact that the serialized state might be received by the new owner instance B after tuples have already been processed, some buffering mechanism must be defined. The pseudocode for State Recreation protocol is shown in Algorithm 5. Once $BT[b].startTS$ has been set, A sends the `EndOfReconfiguration` to upstream LBs (Alg. 5, line 3). In this case, the message only carries the information about bucket b, but no $endTS$ is sent. This is because, as soon as the `EndOfReconfiguration` is received by any LB, $BR[b].dest$ is immediately updated to $BR[b].owner$ and new incoming tuples are only sent to B (Alg. 5, lines 1-2). After sending the `EndOfReconfiguration`, A also serializes the state associated to bucket b and sends it to B

(Alg. 5, lines 4-5). All tuples with timestamp later than or equal to $BT[b].startTS$ are discarded by A (Alg. 5, line 6). B buffers all tuples waiting for the state of bucket b. Once the state has been received and installed, B processes all buffered tuples having timestamp equal to or greater than $BR[b].startTS$ and ends the reconfiguration (Alg. 5, lines 8-10).

Figure 4.5.b shows a sample execution of the State Recreation protocol. The execution resembles the one in the example of the Window Recreation protocol up to the time when $BT[b].startTS$ is computed. Tuple $T3$ is forwarded to both A and B because LB1 processes it before receiving the `EndOfReconfiguration` message. The tuple is discarded by A. B processes $T3$ because it has already received the state associated to bucket b (denoted as CP). Tuple $T5$ is only sent to B since it is processed by LB2 after the `EndOfReconfiguration` message has been received.

4.3 Elasticity Protocol

As introduced in Section 3.1, a static system is not an appropriate solution for parallel-distributing SPEs as variations in the input load might lead to either under-provisioning (i.e., allocated SPE instances cannot cope with the system load) or over-provisioning (i.e., allocated SPE instances are running below their full capacity). To address this problem, we allow the definition of elasticity rules driving the scale-up and scale-down of the system. *StreamCloud* specifies different thresholds that trigger provisioning, decommissioning or dynamic load balancing actions. Given a subcluster, a provisioning action is triggered if its average CPU utilization exceeds the Upper-Utilization-Threshold (UUT). On the other hand, a decommissioning action is triggered if its average CPU utilization is below the Lower-Utilization-Threshold (LUT). Whenever instances are allocated (or deallocated), the number of *StreamCloud* instances composing the subcluster after the reconfiguration action is computed to achieve a new average CPU utilization lower than or equal to to the Target-Utilization-Threshold (TUT). In order to get as close as possible to TUT in case of a provisioning action, *StreamCloud* features a *load-aware* provisioning strategy. When provisioning instances, a naïve strategy would be to allocate one instance at a time (*individual provisioning*). Such solution might lead to cascade provisioning (i.e., new instances are continuously allocated) if the additional computing power of the provisioned instance does not decrease the average CPU utilization below UUT.. To overcome this problem, *StreamCloud* load-aware provisioning takes into account the current subcluster size and load to decide how many new instances to provide in order to reach for TUT.

A dynamic load balancing action is triggered whenever the standard deviation of the CPU utilization is above the Upper-Imbalance-Threshold (UIT). A Minimum-Improvement-Threshold (MIT) specifies the minimal performance improvement to start a new configuration. That is, the new configuration is applied only if the imbalance reduction is above the MIT.

In general, *StreamCloud* continuously monitors each subcluster and tries to keep its the average

CPU utilization within upper and lower utilization thresholds and its standard deviation below the upper imbalance threshold.

The protocol for elastic management is illustrated in Algorithm 6. In order to enforce the elasticity rules, the SC-Mng periodically collects monitoring information from all instances on each subcluster via the LMs. The information includes the average CPU usage (U_i) and number of tuples processed per second per bucket (T_b). The SC-Mng computes the average CPU usage per subcluster, U_{av} (Alg. 6, line 1). If U_{av} is outside the allowed range, the number of instances required to cope with the current load is computed (Alg. 6, lines 2-4). If the subcluster is under-provisioned, new instances are allocated (Alg. 6, lines 5-6). If the subcluster is over-provisioned, the load of unneeded instances is transferred to the rest of the instances by the *Offload* function and the unneeded instances are decommissioned (Alg. 6, lines 7-9). After computing U_{av}, SC-Mng computes the CPU standard deviation U_{sd}. Dynamic Load Balancing is triggered if $U_{sd} > UIT$ (Alg. 6, lines 14-16).

Algorithm 6 Elastic Management Protocol.

```
 ElasticManager
Upon: new monitoring period has elapsed
 1: U_av = (Σ_{i=1}^{n} U_i) / n
 2: if U_av ∉ [LUT, UUT] then
 3:     old:= n
 4:     n:=computeNewConfiguration(TUT, old, U_av)
 5:     if n > old then
 6:         provision(n-old)
 7:     end if
 8:     if n < old then
 9:         freeNodes:= offload(old-n)
10:         decommission(freeNodes)
11:     end if
12: end if
13: U_sd = sqrt(Σ_{i=1}^{n} (U_i − U_av)^2)
14: if U_sd > UIT then
15:     balanceLoad(U_sd)
16: end if
```

Whenever a provisioning, decommissioning or dynamic load balancing action is triggered, *StreamCloud* employs a greedy algorithm[2] to decide which buckets will be transferred during the reconfiguration action. Initially, instances are sorted by U_i and, for each instance, buckets within each instance are sorted by T_b. At each iteration the algorithm identifies the most and least loaded instances; the bucket with the highest T_b owned by the most loaded instance is transferred to the least

[2]Optimal load balancing is equivalent to the bin packing problem that is known to be NP-hard. In fact, each instance can be seen as a bin with given capacity and the set of tuples belonging to a bucket b is equivalent to an object "to be packed". Its "volume" is given by the sum of all T_b at each instance of the subcluster.

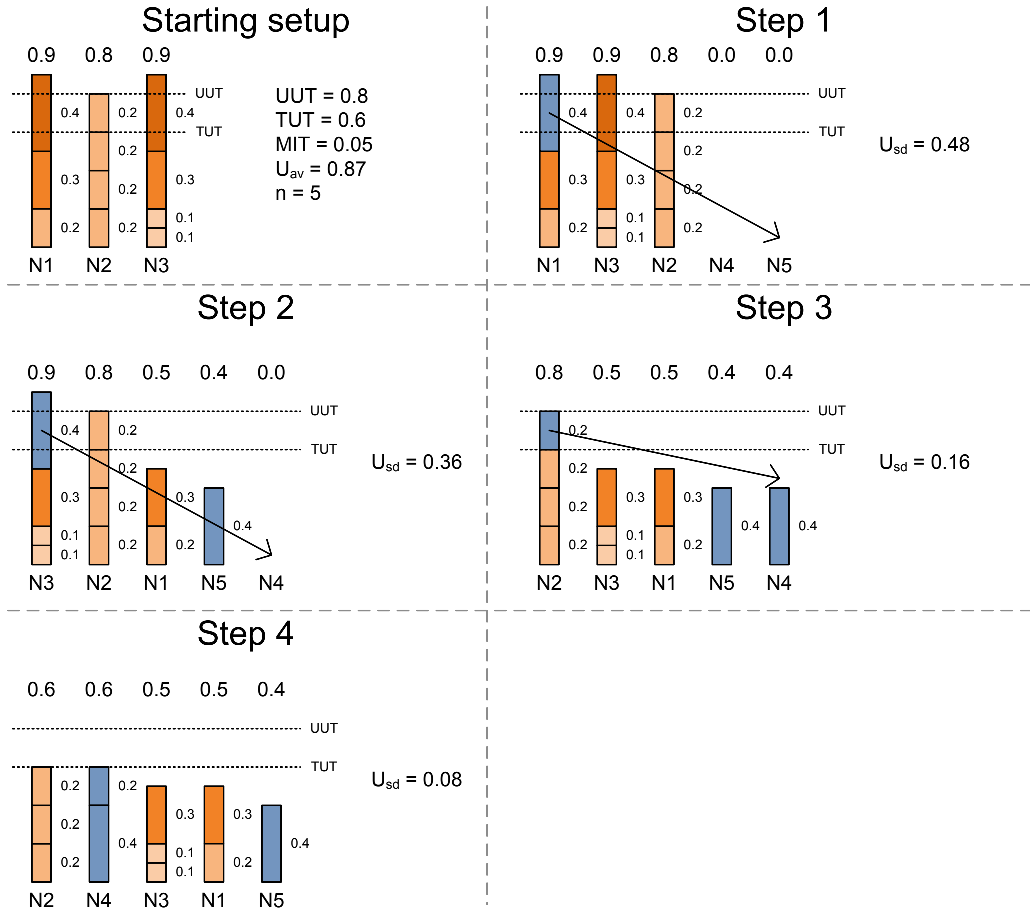


Figure 4.6: Sample execution of the buckets assignment algorithm

loaded one. The CPU usage standard deviation (U_{sd}) is updated and the loop stops when the relative improvement achieved (i.e., difference of standard deviation between two consecutive iterations) is lower than MIT.

Figure 4.6 shows a sample execution of the bucket assignment algorithm. In the example, a subcluster is running over 3 instances, namely $N1$,$N2$ and $N3$ and new nodes are going to be provisioned as the average CPU utilization $U_{av} = 0.87$ exceeds the Upper-Utilization-Threshold $UUT = 0.8$. The required number of nodes to process the incoming load with an average CPU utilization lower than or equal to the Target-Utilization-Threshold $TUT = 0.6$ is 5. Hence, two new *StreamCloud* instances $N4$ and $N5$ will be provisioned. At this point, the dynamic load balancing algorithm is used to determine which buckets will be transferred. At step 1, the most loaded instance $N1$ is offloaded of its heaviest bucket (responsible for 40% of its load) that is transferred to $N5$. The actual CPU standard deviation U_{sd} is equal to 0.48. At step 2, after the bucket has been reassigned to $N5$, the updated U_{sd} is equal to 0.36. The most loaded instance $N3$ is offloaded of its heaviest bucket (responsible for 40% of its load) that is transferred to $N4$.

At step 3, after the bucket has been reassigned to $N4$, the updated U_{sd} is equal to 0.16. The most loaded instance $N2$ is offloaded of its heaviest bucket (responsible for 20% of its load) that is transferred to $N4$. At step 4, after the bucket has been reassigned to $N4$, the updated U_{sd} is equal to 0.08. At this point, the algorithm stops as a new iteration will not decrease U_{sd} anymore. Altogether, 3 buckets will be transferred during the provisioning action, one from each of the *StreamCloud* instances $N1$,$N2$ and $N3$.

The provisioning strategy is encapsulated in the *ComputeNewConfiguration* function (Alg. 6, line 4). The interaction with the pool of free instances (e.g., a cloud resource manager) is encapsulated in functions *Provision* and *Decommission*. The dynamic load balancing algorithm is abstracted in the *BalanceLoad* function (Alg. 6, line 12).

4.4 *StreamCloud* Dynamic Load Balancing and Elasticity Evaluation

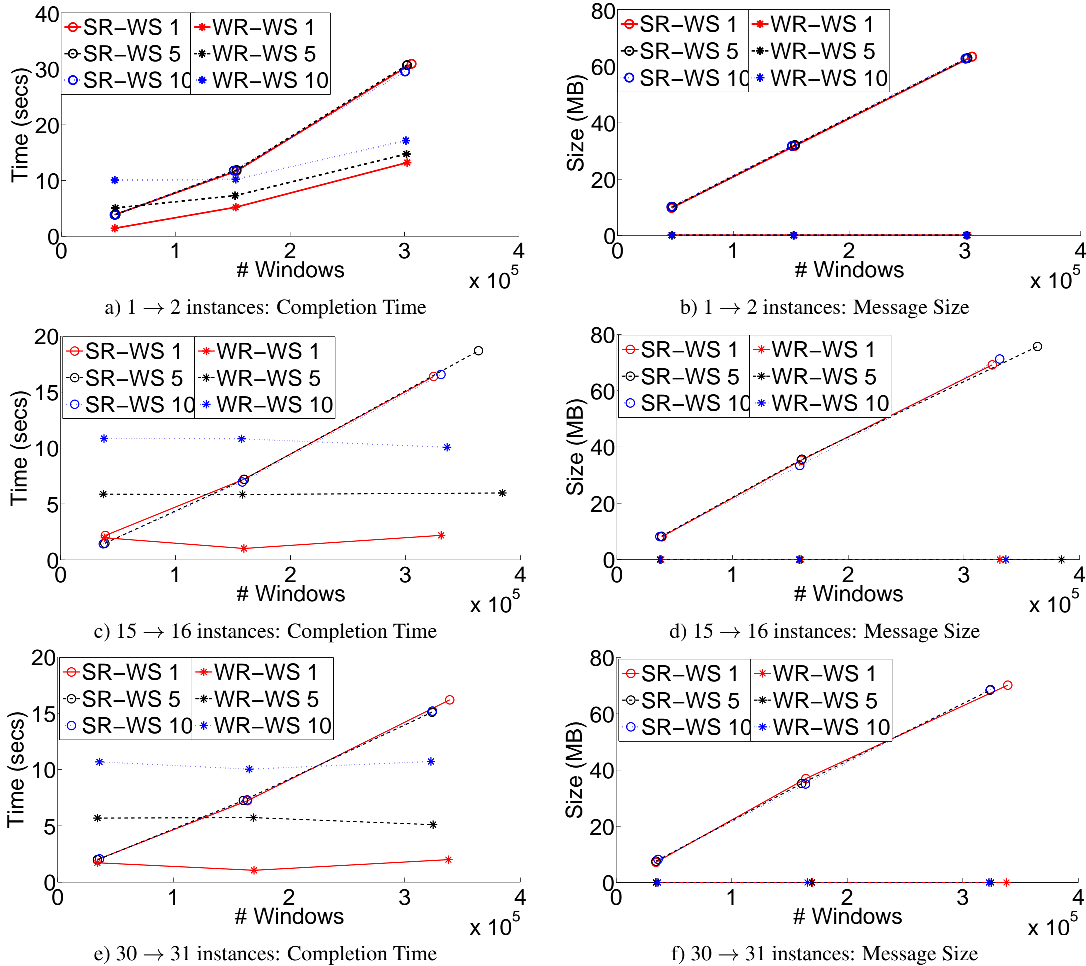
This section presents the experiments performed to evaluate *StreamCloud* dynamic load balancing and elastic features. The evaluation has been performed using the setup presented in 3.3.1. A first set of experiments shows the trade-off between the two elastic reconfiguration protocols of Section 4.2. A second set of experiments evaluates the adaptability of the dynamic load balancing protocol during changes of the workload and a third set of experiments evaluates provisioning and decommissioning strategies. For all the experiments, we used the high mobility fraud detection query presented in Fig. 2.4, used to spot mobile phone that, given two consecutive phone calls, cover a suspicious space distance with respect to their temporal distance. In particular, we focus on the query stateful subquery, composed by one aggregate operator and two stateless operators. In the last two sets of experiments, we have used the State Recreation protocol as it has shown to perform better.

4.4.1 Elastic Reconfiguration Protocols

This set of experiments aims at evaluating the trade-off between Window Recreation protocol and State Recreation protocol (Section 4.2). In this experiment, we run the aggregate operators with window size of 1, 5 and 10 seconds (WS labels in Fig. 4.7), respectively. UUT threshold is set to 80%. Hence, SC-Mng will provision a new node whenever the average CPU utilization of the stateful subcluster is equal to or greater than 80%.

For each reconfiguration protocol, Fig. 4.7 shows the completion times and the amount of data transferred between instances for an increasing number of windows being transferred. Completion time is measured from the sending of the reconfiguration command to the end of the reconfiguration at the new owner instance. Figures 4.7.a, 4.7.c, and 4.7.e show the time required to complete a

reconfiguration from 1 to 2, from 15 to 16 and from 30 to 31 instances, respectively.



a) 1 → 2 instances: Completion Time

b) 1 → 2 instances: Message Size

c) 15 → 16 instances: Completion Time

d) 15 → 16 instances: Message Size

e) 30 → 31 instances: Completion Time

f) 30 → 31 instances: Message Size

Figure 4.7: Evaluation of the elastic reconfiguration protocols.

State Recreation (SR) protocol exhibits a completion time that grows linearly with the number of windows being transferred. This is because all the windows of the buckets being transferred must be serialized and sent to the new owner. On the other hand, the Window Recreation (WR) protocol takes a time proportional to the window size, regardless of the number of windows to be transferred. The completion time shown in Fig. 4.7.a increases with a steeper slope with respect to Fig. 4.7.c and Fig. 4.7.e. This is because there is only one instance transferring a large number of buckets; for configurations with a higher number of instances, this effect disappears and the completion time only depends on the window size. Figures 4.7.b, 4.7.d, and 4.7.f show the amount of data transferred to the new owner in each configuration. With the SR protocol, data received by the new instance grows linearly with the number of transferred windows; using the WR protocol no data is exchanged between instances being reconfigured.

Comparing the results of this set of experiments, we conclude that SR provides better performance

as long as the completion time (dependent on the windows being transferred) does not exceed the time to fill up a window.

4.4.1.1 Dynamic Load balancing

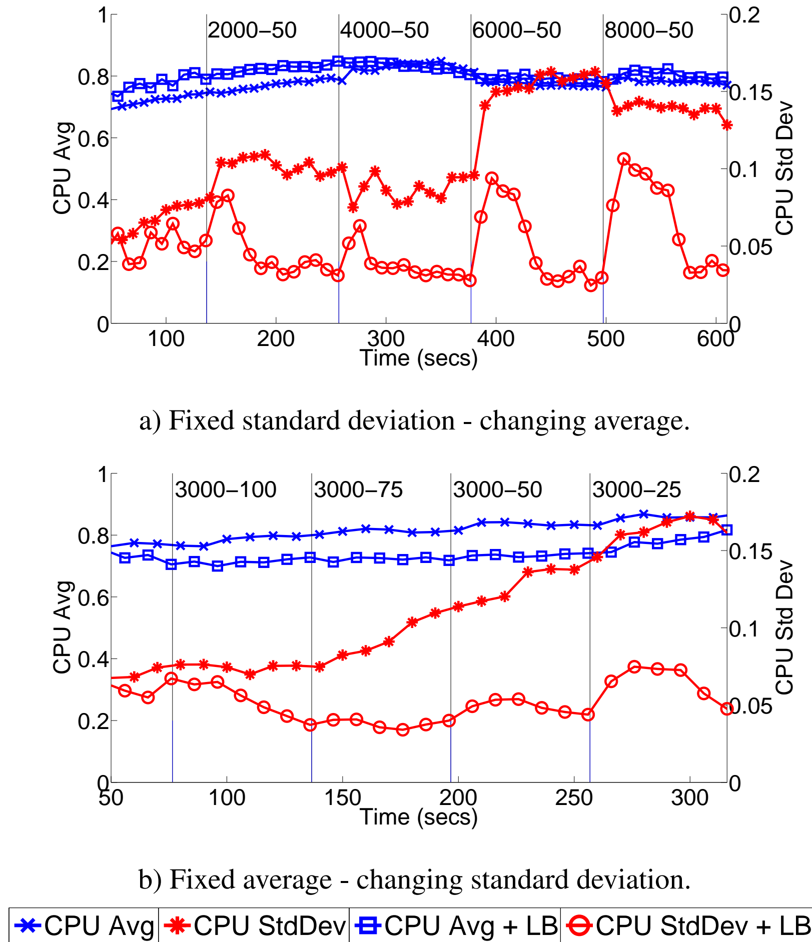
The goal of this set of experiments is to evaluate the effectiveness of dynamic load balancing when the data distribution of a subcluster changes over time. We monitor the evolution of the stateful subquery deployed in a subcluster of 15 instances that process a constant input load of $150,000$ t/s.

Input tuples have $10,000$ different phone numbers, i.e., the $Group-by$ parameter of the Aggregate operator has $10,000$ different keys. Input tuples are generated according to a normal distribution that varies over time, either its average μ or its standard deviation σ. The goal is to show that dynamic load balancing allows keeping a balanced CPU utilization rate of the allocated instances, despite the variability of the input load.

Figure 4.8.a compares system performance with and without dynamic load balancing for an input load that varies its average μ value. The experiment is divided in periods. During the first period, input data follows a uniform distribution. In all other periods, we use a normal distribution with μ that changes periodically from $2,000$ to $8,000$ by steps of $2,000$. In Fig. 4.8.a, periods are separated with vertical lines and, for each of them, μ and σ are specified. Dashed lines show the CPU average utilization rate (primary Y-axis) and its standard deviation (secondary Y-axis) when dynamic load balancing is not enabled. It should be noticed that the standard deviation grows each time μ changes. Solid lines show the performance when dynamic load balancing is enabled. The standard deviation exhibits a peak after the beginning of each new period, which is reduced after *StreamCloud* triggers a dynamic load balancing action. Dynamic load balancing keeps the standard deviation constant, despite the changes in the input distribution. Due to the fixed input load rate, both configurations show a constant average CPU utilization. Figure 4.8.b provides results of the experiment where μ is kept constant and σ changes periodically from 100 to 25 in steps of 25. That is, as time passes, input load concentrates in fewer different keys (i.e., in fewer *StreamCloud* instances). Without dynamic load balancing, the imbalance among instances increases as σ decreases. With dynamic load balancing, the load is redistributed and the standard deviation is constantly kept below the upper imbalance threshold.

4.4.1.2 Self-Provisioning

In this set of experiments, we evaluate the effectiveness of provisioning and decommissioning instances on-the-fly. We set $LUT = 0.5$, $UUT = 0.9$ and $TUT = 0.6$. The load is increased (resp. decreased) linearly to observe the effectiveness of provisioning (resp. decommissioning) actions. Figure 4.9.a shows the behavior of the individual provisioning strategy, i.e., when provisioning only a single instance at a time. We initially deploy the subcluster to a single instance and we study



a) Fixed standard deviation - changing average.

b) Fixed average - changing standard deviation.

Figure 4.8: Elastic Management - Dynamic Load Balancing.

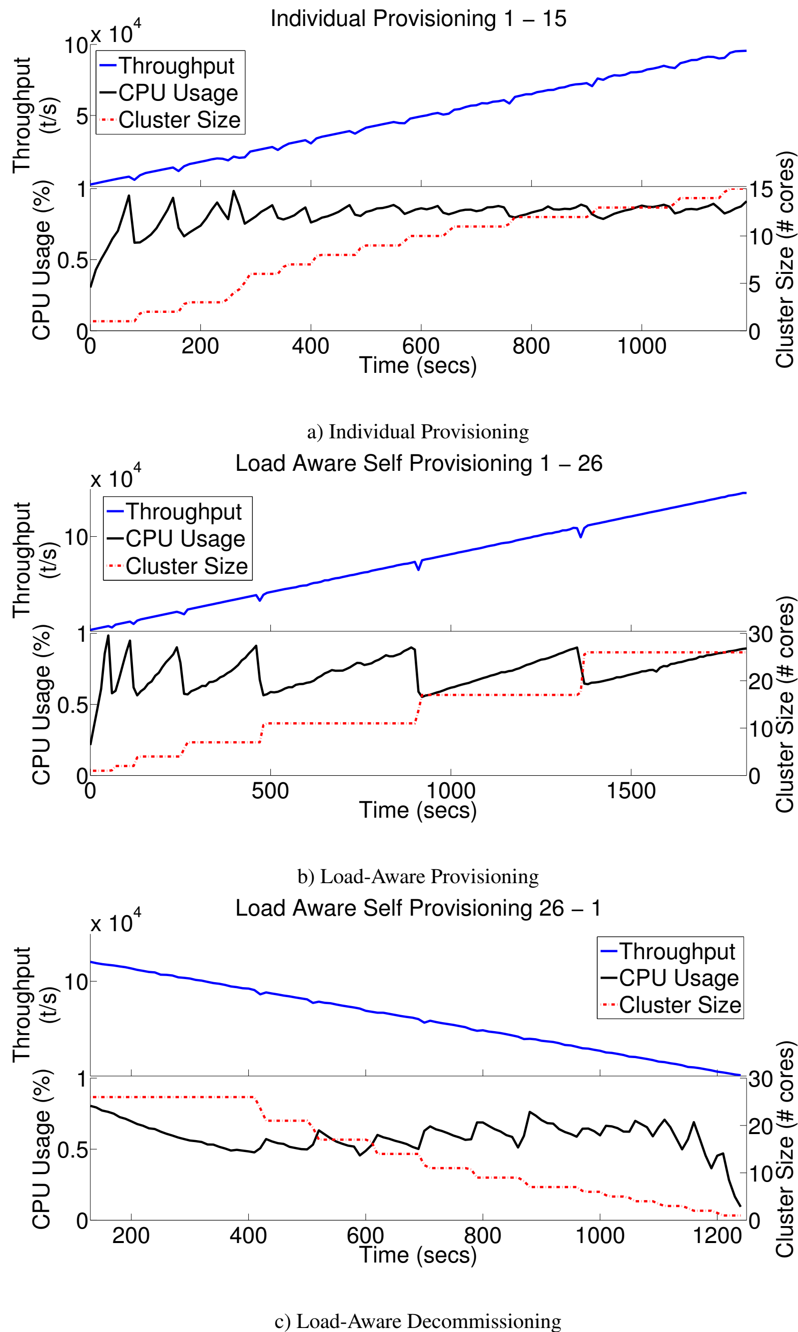
how it grows up to 15 instances. The throughput increases linearly with the input load, despite negligible variations at each provisioning step. However, the target utilization is achieved only when moving from 1 to 2 instances. Starting from size 3, each provisioning step does not provide enough additional computing power to decrease the average CPU utilization to the target one (TUT). For larger configurations (e.g., 15 nodes), provisioning of one instance results in a negligible increase of the overall computing power, leading to an average CPU utilization close to the upper threshold. As soon as the average CPU utilization stays above the upper threshold, the system suffers from cascade provisioning. Figure 4.9.b, shows the effectiveness of the *StreamCloud* load-aware provisioning strategy. In this experiment, we initially set only one instance for the subcluster and let SC-Mng provision multiple nodes at a time. As the number of provisioned nodes is computed on the current subcluster size and load, each provisioning step causes the new average CPU utilization to get close to the target utilization threshold. Moreover, load-aware provisioning affords less frequent reconfiguration steps than individual provisioning. Hence, the system can reach higher throughput with fewer reconfigu-

Table 4.2: Static vs. Elastic configurations overhead.

Configuration	CPU usage (%)	Throughput (t/s)
Static (11 instances)	0.81	64,506
Elastic (11 instances)	0.80	64,645
Static (17 instances)	0.85	100,869
Elastic (17 instances)	0.84	102,695

ration steps. In other words, load-aware provisioning is less intrusive than individual provisioning. Once 27 instances are reached, we start decreasing the load and show the system behavior in Fig. 4.9.c. Decommissioning works as effectively as provisioning. The decommissioning intrusiveness is even lower than the provisioning one due to the fact that instances that are going to be deallocated have a low CPU utilization.

In order to show that the parallelization technique performance is not affected by the elastic resource management, we ran the stateful subquery for two static configurations of 11 and 17 *StreamCloud* instances and we compared throughput and CPU consumption with the results of Fig. 4.9.b. For each of the two cases, we compare the throughput that is achieved with a average CPU utilization greater than 80%. Results are provided in Table 4.2. It should be noticed that for both cluster sizes, the same throughput is reached with similar CPU usage, independently of how subcluster instances have been allocated (i.e., statically at deploy time or dynamically on-the-fly).



a) Individual Provisioning

b) Load-Aware Provisioning

c) Load-Aware Decommissioning

Figure 4.9: Elastic Management - Provisioning Strategies.

Part V

STREAMCLOUD FAULT TOLERANCE

Chapter 5

StreamCloud Fault Tolerance

In this chapter, we discuss *StreamCloud* fault tolerance protocol. In distributed environments, the need for fault tolerance comes from the observation that the higher the number of nodes, the higher the probability of experiencing faults. For this reason, fault tolerance has been a research topic that has attracted significant attention in the context of distributed SPEs. Existing fault tolerance protocols have not addresses aspects such parallel execution of operators, dynamic load balancing and elasticity, which are the innovative aspects of *StreamCloud*. This lack of appropriate solutions motivated our study to define a novel fault tolerance technique that fits with *StreamCloud*. We first discuss the main fault tolerance approaches that have been introduced for distributed SPEs and introduce the challenges being addressed by our solution. Subsequently, we discuss the protocols details and we provide a complete evaluation based on the Linear Road benchmark.

5.1 Existing Fault Tolerance solutions

Fault tolerance in the field of distributed DBs has been studied intensively and well-known techniques are commonly used. As SPE have evolved from DB solutions, these techniques have inspired the initial proposals, improved in order to address the requirements of SPEs.

In general, a fault tolerance protocol must address two main aspects: (1) detection of failures and (2) protocols to recreate the failed instance lost state to mask the failure. While detection and replacement are reactive actions, recreation of the lost state requires a proactive mechanism that continuously maintains information that might be lost. As presented in [HBR$^+$05], three main base

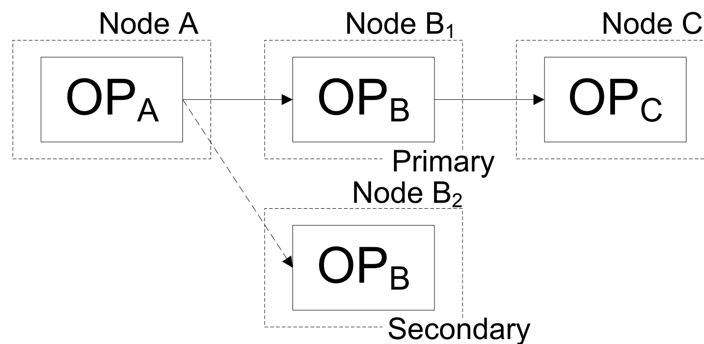


Figure 5.1: Active Standby Fault Tolerance

techniques can be adopted in order to provide fault tolerance to the operators of a query: *active standby*, *passive standby* and *upstream backup*. The three basic approaches differ primarily in how to maintain the information that might be lost in case of failure (i.e., the state of the queries operators).

The *Active standby* (or active replicas) protocol provides fault tolerance for an operator by means of a secondary copy of the operator that is fed with the same tuples forwarded to its primary peer. That is, an exact copy of the operator we want to make fault tolerant is available in order to replace the primary node in case of failure. In Figure 5.1, node B_1 (running operator OP_B) has been replicated and tuples generated by node A (running operator OP_A) are being forwarded to both the primary node and the secondary node B_2. Node B_2 is not connected to the downstream node C (running operator OP_C). It will be connected to it only in case of failure of node B_1. This fault tolerance technique incurs in a high overhead. The main overhead is given by the resources used to maintain replicas, as they are not exploited for active processing of the data. Furthermore, additional runtime overhead is imposed by the fact that tuples must be forwarded to multiple nodes (in the example, node A forwards tuples to both B_1 and B_2). Finally, additional overhead might be imposed if replicas must process tuples in the same order as the primary node (e.g., if we replicate an operator outputting the first tuple of each group of 5 consecutive tuples, both the primary and the replica operator will have to process tuples in the same order to produce the same results). On the other hand, the recovery time of this technique is very low, as it just implies the switching between the primary output stream to the secondary node one.

With the *Passive standby* fault tolerance technique, the state of the operators belonging to the node we want to protect are periodically copied to a secondary node. Copies can be continuously installed at replica nodes or they can be stored in a dedicated server and installed in a replacement instance in case of failure. The periodic copy of an operator state is known as *checkpointing*. Figure 5.2 shows an example of the *passive standby* approach where a replica node B_2 is being maintained copying to it the state of operator OP_B. Basic checkpointing techniques can copy the entire state maintained by the primary node to the secondary while more efficient technique (*delta-checkpointing*)

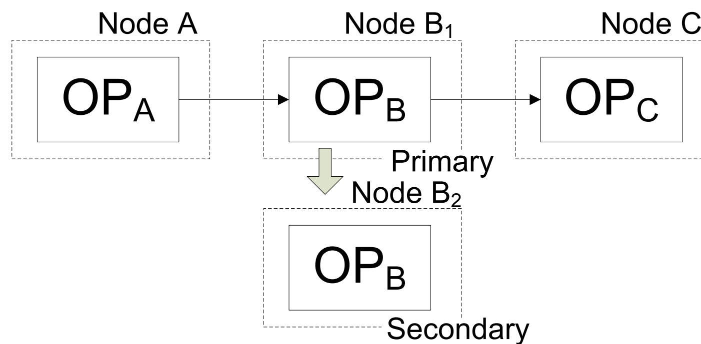


Figure 5.2: Passive Standby Fault Tolerance

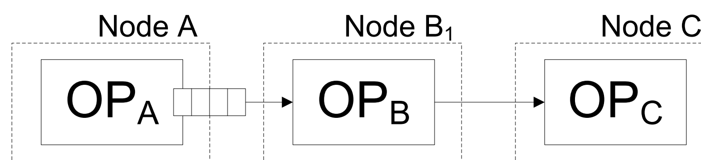


Figure 5.3: Upstream Backup Fault Tolerance

might update the state of the secondary nodes via “deltas" of the previous state. With respect to the *active standby*, this fault tolerance technique reduces the runtime overhead due to the fact that the information periodically exchanged between the primary and secondary nodes condensates multiple tuples, resulting in a lower overhead. On the other hand, it increases the recovery time: all the tuples forwarded to the primary operator during the time interleaving the last checkpoint and the failure are not maintained at the replica operator; hence, they need to be replayed at the secondary node.

Finally, *Upstream Backup* defines a different approach where no replicas nodes are used. The idea is to maintain the tuples forwarded to the primary operator in order to replay them to the replacement operator in case of failure. As the name suggests, tuples need to be maintained by the operator upstream peer (tuples will be lost if maintained by the same instance we want to protect from faults). In Figure 5.3, the node running the operator OP_A is using a buffer to maintain the tuples being sent to the node running operator OP_B. In case of failure, these tuples will be forwarded again to the replacement node. With this approach, the runtime overhead is further decreased, as the only requirement is the space needed to maintain the upstream tuples. On the other hand, the recovery time increases as the state of the failed primary operator has to be recreated processing again each tuple individually (the completion time will depend on the period of time covered by the primary node state). We present a detailed description of how existing work has improved all this basic techniques in the related work (Chapter 8).

As discussed in [HBR$^+$05], not only different approaches exist with respect to how to tolerate faults, but also with respect to which guarantees to provide. The weakest guarantee is provided by the *gap* fault tolerance protocol. With *gap* recovery, the state and the tuples (if any) lost due to a failure are

simply ignored. A replacement instance is provided with an initial empty state. This technique will result in the loss of part of the query results or in results whose values differ from the ones of a ideal fail-free execution. A stronger guarantee is provided by the *Rollback recovery*. In this case, all the tuples are processed even in case of failures but the resulting query outputs might differ from the ones of an ideal fail-free execution. As an example, using the *active standby* approach without making sure primary and secondary nodes process tuples in the same order might lead to different outputs (although all tuples are still processed). Finally, the strongest (and more challenging) guarantee is provided by the *Precise recovery*. In this case, any failure is completely masked to the final user and the results produced by the query are the same produced by an ideal fail-free execution.

In order to discuss which solution has been adopted to provide fault tolerance in *StreamCloud*, we first outline the requirements of the solution and then discuss which approach has been chosen (and improved) and why. The fault tolerance protocol we need must comply with the following requirements:

1. **Low runtime overhead and fast recovery.** The most important requisite of any fault tolerance protocol for an SPE is to provide low runtime overhead and fast recovery. Runtime overhead must be kept as lower as possible as the protocol is useful as long as its impact on the normal processing (e.g., its impact on the result latency) does not violate the application requirements. On the other hand, recovery time should be as short as possible in order to reduce the quality loss and the user satisfaction upon failures. We stress that, among the two requirements, low runtime overhead is usually more important than recovery time as failures are considered to happen occasionally. That is, the runtime overhead is a price that is being paid continuously while recovery time is a price paid sporadically.

2. **Precise Recovery.** We look for a fault tolerance protocol that guarantees precise recovery as we think most of the near real time applications that motivate *StreamCloud* do not allow for data loss. As an example, data loss is not acceptable in fraud detection applications.

3. **No replicas.** Although replicas can lead to fast recovery times, its overhead in terms of computation and resources is too high. In an environment as the cloud, where the number of nodes is usually optimized to reduce costs, duplicate (or triplicate) the system size to maintain replicas is not a feasible solution. As an example, a system that requires two replicas for each primary node will result in an overhead of 200%.

4. **Not rely uniquely on main memory.** As discussed in Chapter 2, streams are defined as potentially unbounded sequences of tuples. Thus, solutions that limit the scope of a stream are mandatory when relying on main memory (e.g., the windowing mechanism). When discussing a fault tolerance protocol that does not rely on replicas, we cannot make the assumption that all

the states maintained by all the instances running a set of queries can be aggregated using the main memory of a small number of dedicated servers. For this reason, we look for a solution that leverages the use of persistent storage.

5. **Decouple state maintenance from topology.** One of the novel features introduce by *StreamCloud* are elasticity and dynamic load balancing. As discussed in Chapter 4, both techniques are based on state transfer protocols used to move part of the state of an operator (the state of a set of *buckets*) to another one. Due to the fact that the state of an operator is not statically assigned to it but it can rather be partitioned and transferred at runtime, the fault tolerance protocol will need to be designed decoupling state maintenance from any particular topology. That is, the protocol will not maintain a copy of the state of a particular physical node, it will rather maintain a copy of the whole state of the parallel-operator independently of the number of machines being used to process it at any point in time.

6. **Tolerate failures happening during reconfigurations of the system.** The last requirement of *StreamCloud* fault tolerance protocol is that it must tolerate failures happening while the system is being reconfigured (i.e., failures happening while the state of an operator is being transferred or while nodes are being provisioned or decommissioned). This constitutes a new challenge that has not been studied previously.

Considering the possible basic fault tolerance techniques, we discard the *active standby* technique as, being based on replicas, is against requirement 2. For the same reason, we discard any variant of the *passive standby* approach if checkpoints of operators states are being installed at replica nodes. As discussed in the related work, solutions where checkpoints are maintained by server nodes rather than replicas have been studied. Such solutions could in principle be candidate solutions, as the number of servers used to maintain checkpoints is usually small with respect to the one used by replicas. As discussed previously, the *passive standby* approach might require partial re-forwarding of past tuples depending on the interleaving time between the last checkpointing of the primary node state and the actual failure. This means that *passive standby* protocol must be used together with a protocol that resembles the *Upstream Backup* (the protocol that defines which tuples the upstream peers should buffer would be slightly different in this case). This consideration motivates us to focus simply on the *Upstream Backup* technique. Several improvements must be defined for the basic approach: (1) nodes cannot rely uniquely on main memory to maintain tuples that could be re-forwarded (requirement 4) and (2) the protocol must be extended to comply with requirements 5 and 6.

In the following sections, we proceed describing how the basic *upstream backup* approach has been enriched in *StreamCloud*. We first provide an intuition about the proposed fault tolerance protocol and proceed then with a detailed description of the protocol and the related evaluation.

5.2 Intuition about Fault Tolerance protocol

In this section, we provide an intuition about how fault tolerance is provided to data streaming operators using an improved version of the *upstream backup* technique. Without focusing on how to implement such technique, let us first consider that we have a means for maintaining all the tuples that are being exchanged between two nodes A and B. In case of failure of node B, we can recreate the lost state replaying the past tuples to a replacement instance B'. Nevertheless, the following considerations hold: (1) in order to recreate the same state, tuples should be reprocessed in the same order they were processed by the failed instance and (2) in order to reduce the recovery time, the tuples that should be forwarded again should be only the ones that contributed to the state of node B just before its failure. If we now move to a real system where we have limited resources, more requirements arise. Suppose we want to implement a naïve solution where all the tuples forwarded by the upstream node are being persisted to disk (complying therefore with requirement 4 that states the protocol should not rely uniquely of main memory) and where, upon failure of an instance, all the previous tuples are replayed. First, the need for reducing the amount of tuples being replayed to B' is not only motivated by the need of reducing the recovery time, but also by the **impossibility of maintaining all the past history of a stream** (streams are potentially unbounded sequences of tuples). This implies we need to **maintain some information previous to the current point in time from where tuples should be replayed in case of failures**. Once we imposed the limitation about the amount of tuples that we can maintain, we could define some smart technique to maintain them in order to reduce the runtime overhead; as an example, we could maintain batches of tuples. In this case, **it might be only possible to replay tuples from specific past points in time**. If tuples are replayed from a point in time earlier than the required one, **any duplicate tuple must be discarded in order for the protocol to be precise**. The last aspect we must consider is related to the peer nodes upon which the node we want to protect from failures relies. The node **relies on its upstream peers** in order to maintain past tuples (it cannot rely on itself otherwise it will lose them upon failure). Consider now a scenario where the tuples being forwarded between the node we want to protect from faults and its downstream peer are being extremely delayed due to a congestion in the channel (e.g., the tuples produced minutes ago have not yet been received by the downstream node). Upon the failure of a node, if we just recreate the latest state maintained by the node, part of the tuples being forwarded that were not received by the downstream node might get lost. For this reason, the node **relies on its downstream peers** in order to make sure that, each time we update the point in time from where to replay tuples, all the previous output tuples generated by the node have been received. The last aspect we must take into account is related to the information maintained in a generic stateful operator state. While data is being processed, new windows are created and existing windows are slid. At the same time, windows might become obsolete. As an example, the information related to

a phone number that is no longer active is obsolete if kept waiting for new incoming calls. For this reason, **obsolete state must be garbage collected to reduce the state that must be recovered upon a failure**.

We introduce the following sample execution of an aggregate operator to give an idea about how the operator can be recovered in case of failure. In the example, the operator is used to compute the average price of the calls being made by mobile phones. The input CDR tuples have the schema presented in Section 2.1, composed by fields $Caller$ and $Callee$ (representing the phone number making and receiving the call, respectively), fields $Time$, $Duration$ and $Price$ and coordinates $Caller_X, Caller_Y$ and $Callee_X, Callee_Y$. The operator is defined as follows:

$$Agg\{time, Time, 180, 120, AvgPrice \leftarrow avg(Price)\}(I, O)$$

The operator defines a time-based window with size and advance of 180 and 120 seconds, respectively. That is, every 2 minutes, the operator outputs the average price of calls being made by all the mobile phones during the last 3 minutes. The output tuples schema is composed by fields $\langle Time, AvgPrice\rangle$.

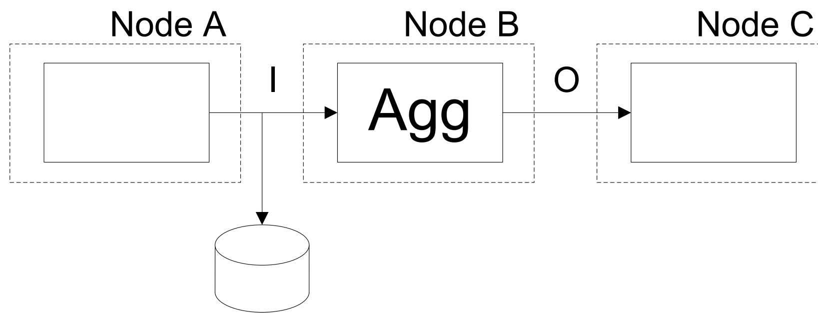


Figure 5.4: Example of fault tolerance for aggregate operator

As presented in Figure 5.4, assume the operator is deployed at node B, its input tuples are forwarded by an operator deployed at node A while its output tuples are being sent to the operator running at node C. Assume also that tuples exchanged between nodes A and B are being persisted and, at any point time, stream I can be replayed starting from a given tuple.

Table 5.1 presents the sequence of tuples consumed and produced by the operator. Input tuples are denoted as t_i while output tuples are denoted as T_i. For the ease of the explanation, we only consider fields $Time$ and $Price$ for the operator input tuples.

The first window managed by Agg contains tuples $t_i : t_i.Time \in [0, 180[$; in the example, tuples t_1, t_2, t_3. The second window managed by A contains tuples $t_i : t_i.Time \in [120, 300[$; in the example, tuples t_3, t_4, t_5, and so on. Tuple T_1 is produced upon reception of tuple t_4 as the latter causes window $[0, 180[$ to slide ($t_4.Time$ falls outside the window). Tuple T_1 carries the average call price computed from tuples t_1, t_2, t_3. After sliding the window, tuples t_1, t_2 are discarded. Similarly,

		t_1	t_2	t_3	t_4	t_5	t_6
In	$t_i.Time$ (secs)	0	60	120	180	240	300
	$t_i.Price$	7.8	8.2	8	7.5	7.3	8.1
Out		-	-	-	T_1	-	T_2
	$T_i.Time$ (secs)	0	60	120	180	240	300
	$T_i.AvgPrice$	-	-	-	8	-	7.6

Table 5.1: Sample input and output tuples for operator A_2

tuple T_2 is produced upon reception of tuple t_6. Consider how A state changes before and after processing tuple t_4. Before processing t_4, three tuples are currently maintained by A (i.e., tuples t_1, t_2, t_3). After processing t_4, the new state includes tuples t_3, t_4. Imagine that the instance running operator Agg fails just after outputting tuple T_1. If tuple T_1 has been received by node C, the node replacing B can be fed starting from tuple t_3. This way, the first tuple the replacement instance will produce is T_2. Doing this, from the point of view of the operator deployed at node C the failure has been completed masked. If tuple T_1 has not been received by node C, the failure will be masked if B replacement node is fed starting from tuple t_0. Doing this, the replacement operator will first output T_1 and later on T_2. Notice that, even if T_1 has been received by node C, B replacement node can still be fed starting from tuple t_0, as far as the duplicate tuple T_1 is not being forwarded to the operator of node C.

We denote as the *earliest timestamp* of an operator the smallest timestamp from where tuples should be replayed in case of failure of operator OP. With respect to the previous example, if T_1 has been received by node C after node B failure, then $Agg.et = 180$ (i.e., tuples should be replayed starting from timestamp 180 in order to provide precise recovery). On the other hand, if T_1 has not been received by node C after node B failure, then $Agg.et = 0$ (i.e., tuples should be replayed starting from timestamp 0 in order to provide precise recovery). It can be noticed that the *earliest timestamp* of an operator depends on both the tuples maintained at its state and the tuples received by its downstream peer. More precisely, the *earliest timestamp* represents the earliest tuple timestamp maintained by an operator if all the tuples that have been previously produced and that depend on earlier tuples have been received by the downstream instance.

5.3 Components involved in the Fault Tolerance protocol

In this section, we present an overview of the *StreamCloud* components used to provide fault tolerance to query operators. Figure 5.5 resembles the sample aggregate operator presented in Figure 5.4, presenting how the operator (and its upstream and downstream peers) are parallelized in *StreamCloud*.

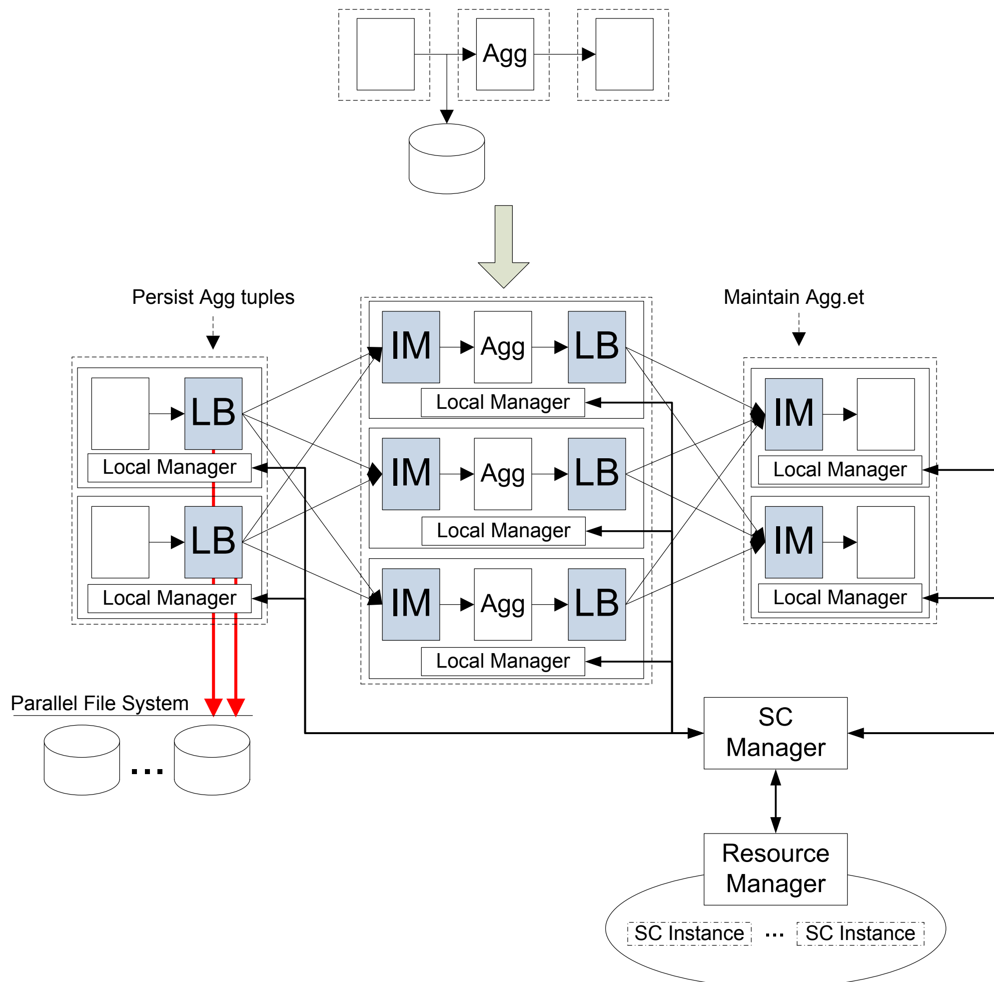


Figure 5.5: StreamCloud fault tolerance architecture

Following the operator-set-cloud parallelization strategy, a subquery is defined for the aggregate operator while a separate subquery is defined for its preceding operator (independently of the operator type, stateless or stateful). For the ease of the explanation, assume the operator following the aggregate is stateful, so that a dedicated subquery is defined for it. The instances of the subcluster containing the aggregate defines an input merger and a load balancer. Tuples are routed to the subcluster by the upstream LBs while they are collected by the downstream IMs.

As discussed previously, the instances rely on their upstream peers to maintain past tuples. In *StreamCloud*, we employ the LBs to persist to disk tuples while forwarding them in parallel. As shown in the figure, tuples are persisted to a parallel file system; we discuss in the following section why we rely on such system to maintain streams past tuples. While LBs are being employed to persist tuples, the IMs of the downstream peers are used to maintain the earliest timestamp of the operator for which we provide fault tolerance. As presented in Section 4.1, each *StreamCloud* instance runs

a Local Manager that is used to share information with the *StreamCloud Manager* and to modify the running query. With respect to fault tolerance, Local Managers are used to forward the information related to the tuples being persisted and the earliest timestamp of operators to the *StreamCloud Manager*.

5.4 Fault Tolerance protocol

In this section, we provide a detailed description of *StreamCloud* fault tolerance protocol. We first focus on single instance failures and, subsequently, extend the protocol to multi-instance failures. As discussed in Section 3.2.1, the minimum data distribution unit used to route tuples across two subclusters is the bucket. In the following, we discuss protocols using buckets b as the minimum of tuples for which we provide fault tolerance (i.e., we discuss how to provide fault tolerance for individual buckets or groups of them).

As discussed in the previous section, the earliest timestamp that specifies which tuples should be replayed in case of failure depends on the operator running at the instance we are protecting against failures. For each type of operator (i.e., stateful or stateless) and for any query that can be deployed at a *StreamCloud* instance, we must define how the overall earliest timestamp is computed. Stateless operators do not maintain any tuple; hence, we define the earliest timestamp of the operator as the timestamp of the tuples being forwarded. This means that, upon failure, we plan to resume tuples processing starting from the last tuple sent by the operator (if the latter actually reached the downstream instance, we will need to discard the duplicate tuple). With respect to stateful operators, the earliest timestamp is set to the timestamp of the earliest tuple maintained by the operator. With respect to the possible operators deployed at a given *StreamCloud* instance, we discussed in Section 3.2 that a subquery contains no more than one stateful operator. The earliest timestamp must be computed by the operator maintaining the earliest tuple. For this reason, if the subquery contains a stateful operator, the latter is used to compute the earliest timestamp, otherwise, the earliest timestamp of computed by the last subquery stateless operator.

In the following, we denote as I_F the instance for which fault tolerance is provided and as I_R the instance replacing the former upon failure. I_F upstream subcluster is referred to as U while downstream subcluster is referred to as D. Given a tuple T outputted by I_F, $b.et$, referred to as *bucket B earliest timestamp*, represents the earliest tuple timestamp of bucket b once T is produced. Each output tuple T schema is enriched with field et, set to the value of bucket earliest timestamp ($T.et = b.et$). Hence, if I_F fails after T has been received by D, we can recreate I_F lost state replaying its input tuples starting from timestamp $T.et$.

StreamCloud detects that an instance I_F has failed if it stops answering to a series of consecutive heart-beat messages. Once failure has been detected, a replacement instance I_R is allocated by the

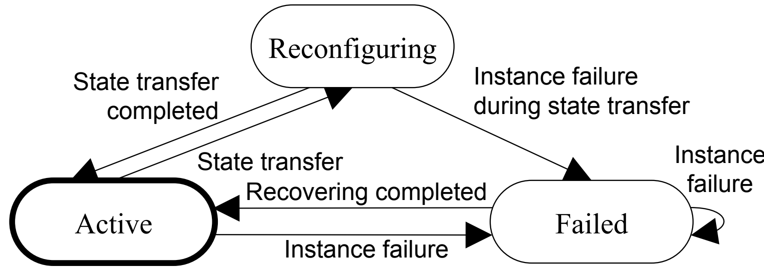


Figure 5.6: Bucket state machine.

Resource Manager and the query previously deployed at I_F is re-deployed at I_R. In order to know which tuples should be replayed in case of failure, $b.et$ is continuously updated by the stateful operator of I_F (or the first stateless one if no stateful operator is defined) and communicated to D using output tuples.

StreamCloud fault tolerance protocol ensures that, upon failure of the instance owning bucket b, b tuples are replayed starting from a timestamp $ts \leq b_{et}$ and that duplicate tuples are discarded, therefore providing precise recovery. In order to recover b in case of a failure, the protocol must (a) persist past tuples in order to replay them in case of failure, and (b) maintain the latest value of b_{et}. In our protocol, task (a) is performed by U LBs (denoted as $U.LBs$) while task (b) is performed by D IMS (denoted as $D.IMs$).

Figure 5.6 presents the possible states that define each bucket b. During regular processing, b state is set to *Active*. If b ownership is being transferred (e.g., a provisioning, decommissioning or dynamic load balancing action is triggered), its state moves to *Reconfiguring*. Once the reconfiguration has been completed, its state is set back to *Active*. Notice that reconfiguration actions are taken only for *Active* buckets.

Failures might happen for a bucket in *Active* or *Reconfiguring* state. If failure happen while b is *Active*, its state moves to *Failed*. Bucket b remains in this state until recovery ends (i.e., it remains in *Failed* state also if a second failure takes place before the first one has been completed solved). b state is moved to $Failed$ also if the failure occurs while b is in *Reconfiguring* state.

In the following sections, we present which are the tasks performed by I_F and by its upstream downstream peers U and D depending on b state. With respect to the $Reconfiguring$ state, we refer the reader to Section 4.2.3, presenting the *State Recreation* protocol, the state transfer protocol for which fault tolerance is provided in *StreamCloud*.

Table 5.2 summarizes the principal variables used in the following algorithms.

Buf	Buffer used by LBs to maintain forwarded tuples
BR	Bucket Registry
$BR[b].state$	State of bucket b
$BR[b].et$	Bucket b earliest timestamp
$BR[b].last_ts$	Bucket b latest tuple timestamp
$BR[b].dest$	Instance to which bucket b tuples are forwarded
$BR[b].owner$	Bucket b owner
$BR[b].buf$	Buffer associated to bucket b
I_F	Failing instance
Q	Query deployed at I_F
I_R	Replacement instance
U	I_F upstream subcluster
$U.LBs$	U load balancers
$U.LB_{prefix}$	Name prefix shared by LBs at U
D	I_F downstream subcluster
$D.IMs$	D input mergers

Table 5.2: Variables used in algorithms

5.4.1 Active state

While b state is *Active*, $U.LBs$ are responsible for forwarding and persisting tuples being sent to I_F, the stateful operator (or the last stateless one) deployed at I_F is responsible for computing the instance earliest timestamp and use it to enriching output tuples (setting field $T.et$) while $D.IMs$ are responsible for maintaining b_{et}.

With respect to the task performed by $U.LBs$, the protocol must define an efficient way to persist and forward tuples (i.e., it cannot be blocking, tuples must be forwarded and persisted in parallel). Furthermore, persisting individual tuples might result in a high overhead; hence, LBs should first buffer them and, subsequently, persist at once multiple tuples. In the proposed solution, $U.LBs$ use a buffer Buf to maintain tuples being forwarded. Each incoming tuple is added to the buffer that is persisted periodically. Similarly to time-based windows, buffers define attribute $size$ to specify the extension of the time period they cover. Nevertheless, the periods of time covered by Buf do not overlap, they rather partition the input stream of each LB into chunks of $size$ time units. All the LBs at U share the same $Buf.size$ and have aligned buffers (i.e., at any moment, all the buffers cover the same period of time). Given an input tuple t, and being $t.ts$ its timestamp (expressed in seconds, or other time units, from a given date), the buffer to which t belongs to will have boundaries $[\left\lfloor \frac{t.ts}{Buf.size} \right\rfloor, \left\lfloor \frac{t.ts}{Buf.size} \right\rfloor + Buf.size[$. We refer to the left time boundary of buffer Buf as $Buf.start_ts$. Each time the buffer is full (i.e., $t.ts - Buf.start_ts \geq Buf.size$), it is asynchronously written to disk. LBs rely on a parallel-replicated file system to persist Buf. The reason is twofold: (a) file system replication prevents information loss due to disk failures while and (b) being

distributed, tuples persisted by an LB can be accessed also by the other *StreamCloud* instance running the query (as presented in the incoming Section 5.4.2). Algorithm 7 presents LBs protocol. Each LB maintains a Bucket Registry (BR). For each bucket b, $BR[b].state$ defines b state while $BR[b].dest$ defines the instance to which b tuples are forwarded. For each incoming tuple belonging to bucket b, function `buffer` is invoked to add the incoming tuple to Buf and, if the buffer is full, to persist it. Subsequently, if b state is $Active$, t is forwarded to its destination instance. Given an incoming tuple t, function `buffer` checks whether Buf should be persisted and, eventually, stores tuple t. Buf is serialized if $t.ts - Buf.start_ts > Buf.size$. The file to which the buffer is persisted is identified by the LB name persisting it and $Buf.start_ts$. Considering how LBs persist their incoming streams, each incoming tuple is either maintained in the LB memory (if it belongs to the current buffer) or written to disk. Nevertheless, this consideration might not hold if, upon failure of instance I_F, $U.LBs$ continue persisting the tuples they are now buffering. To avoid this, function `buffer` stops serializing Buf if one (or more) of the downstream instances fails. That is, tuples are still being forwarded to the active instances and buffered at Buf, but not persisted to disk. It can be noticed that, in Algorithm 7 - line 6, the condition checked by the `buffer` function in order to persist Buf makes sure no bucket b is in $Failed$ state (i.e., no downstream instance has failed). The overhead introduced by the persistence of tuples is negligible. It is only caused by the time spent to create a copy of each incoming tuple and, whenever Buf is full, by the time it takes to issue the asynchronous write request.

Algorithm 7 Active State - LBs protocol

LB

Upon: Arrival of tuple $t \in b$
1: `buffer`(t)
2: $BR[b].dest = BR[b].owner$
3: **if** $BR[b].state = Active$ **then**
4: `forward`(t,$BR[b].dest$)
5: **end if**

`buffer`(t)
6: **if** $\nexists b : BR[b].state = Failed \wedge t.ts - Buf.start_ts \geq Buf.size$ **then**
7: $fileName$ = `concatenate`(LB.name,$Buf.start_ts$)
8: `async`(Buf,$fileName$)
9: Buf.`clear`()
10: **end if**
11: Buf.add(t)

$D.IMs$ are responsible for maintaining bucket b earliest timestamp and for discarding duplicate tuples. Algorithm 8 presents IMs protocol. For each incoming tuple $t \in b$, b earliest timestamp is updated to $t.et$. Similarly to LBs, each IM maintains a Bucket Registry BR for each bucket b. $BR[b].et$

defines b earliest timestamp, $BR[b].last_ts$ the timestamp of the latest tuple received and $BR[b].buf$ buffer is used to temporarily store incoming tuples. If we consider how to detect duplicates, the assumption about timestamp ordered input streams (Section 2.1) and the timestamp sorting guarantees provided by the IMs (Section 3.2.1.2), permit to spot a duplicate when its timestamp is earlier than the previous tuple or, if equal, if the tuple is a copy of another tuple sharing the same timestamp. $D.IMs$ use $BR[b].buf$ to temporarily store all the tuples sharing the same timestamp (the buffer is empty each time a new incoming tuple has a new timestamp) in order to check for duplicated tuples. Function isDuplicate in Algorithm 8 presents the pseudo-code used to check for duplicated tuples.

We have discussed which are the protocols for $U.LBs$ and $D.IMs$ with respect to b *Active* state. If we analyze the interaction between these components from a subcluster point of view, we have that U serializes all the tuples forwarded to I_F, partitioning them on a per time period, per LB basis. At the same time, D maintains b_{et} on a per IM basis. If, due to a failure of I_F, the lost state must be recreated starting from timestamp ts, the tuples will be read from the files persisted by $U.LBs$. More precisely, the files to read will be the ones having name $[lb][start_ts]$, where lb is the name of any of $U.LBs$ and

$$start_ts = max_i T_i : \left\lfloor \frac{T_i}{Buf.size} \right\rfloor \leq ts$$

The tuples being forwarded by I_F and carrying information about $b.et$ are routed to the different $D.IMs$ instances. At any point in time, given a bucket b, b_{et} is computed as the $min(b_{et_i}), \forall IM_i \in D.IMs$. That is, the earliest timestamp is the smallest one maintained by any $D.IMs$.

Bucket earliest timestamps maintenance and cleaning of stale information. Two important aspects related to the actions taken by the operators involved in the maintenance of bucket b while in state *Active* must be considered. If tuples are continuously persisted, they will eventually saturate the capacity of the parallel file system. Moreover, if an instance of D fails, its information associated to b_{et} will be lost. To address both problems, the *StreamCloud Manager* periodically connects to D instances and retrieves the earliest timestamps of bucket b. With this information, files that only contains tuples having timestamps earlier than b_{et} can be safely discarded. Furthermore, upon failure of I_F, the earliest timestamp indicating which tuples should be replayed is known by *StreamCloud* even if one or more D instances are not reachable (e.g., due to a multiple-instance failure). It should be noticed that, even if b_{et} is not updated to its latest value, the recovery will still be precise, as discussed in Section 5.4. Algorithm 9 presents the *StreamCloud Manager* protocol.

5.4.2 Failed state

In this section, we present the main steps performed by *StreamCloud* to replace instance I_F in case of failure. We first discuss the overall sequence of steps and proceed then with a detailed

description of each one. The failure of an instance I_F is discovered by the *StreamCloud Manager* when the former stops answering a given number of consecutive heart-beat messages. At the same time than the *StreamCloud Manager*, I_F upstream subcluster U discovers the instance has failed as soon as the TCP connections between them fails. A replacement instance I_R is taken from the pool of available instances maintained by the Resource Manager and the query previously deployed at I_F is re-deployed at I_R. While deploying the query, the lost state is recreated reprocessing past tuples persisted to the parallel file system. Once the query has been deployed, its state has been recovered and operators have been connected to their upstream and downstream peers, upstream LBs are instructed by the *StreamCloud Manager* to forward buffered tuples and resume regular processing.

Algorithm 8 Active State - IM.

```
IM
Upon: Arrival of tuple t from stream i
 1: buffer[i].textttenqueue(t)
 2: if ∀i buffer[i].textttnonEmpty() then
 3:     t0 = earliestTuple(buffer)
 4:     b = getBucket(t0)
 5:     if ¬isDuplicate(t0,b) then
 6:         forward(t)
 7:         BR[b].et = t.et
 8:     end if
 9: end if

    isDuplicate(t,b)
10: result=false
11: if t.ts < BR[b].last_ts then
12:     result=true
13: else if t.ts = BR[b].last_ts then
14:     if BR[b].buf.contains(t) then
15:         result=true
16:     else
17:         BR[b].buf.add(t)
18:     end if
19: else
20:     BR[b].last_ts = t.ts
21:     BR[b].buf.clear()
22:     BR[b].buf.add(t)
23: end if
24: return result
```

As soon as I_F failure is detected by $U.LBs$, the state of each bucket b owned by I_F is changed to *Failed*. As presented in Section 5.4.1, this implies that tuples that were previously persisted to the parallel file system by $U.LBs$ are now only maintained using main memory (as long as the failure has

been recovered). LBs pseudo-code for the action performed upon failure is presented in algorithm 10, lines 1-3.

Algorithm 9 Active State - *StreamCloud Manager*

StreamCloud Manager

Upon: Monitoring period expired for subcluster C

1: **for all** b **do**
2: $BR[b].et$=`getBucketET`(b)
3: **end for**
4: $T = \left\lfloor \frac{min(BR[i].et)}{Buf.size} \right\rfloor$
5: **for all** file $[LB][ts]$: LB.`hasPrefix`(LB_{prefix}) $\wedge ts < T$ **do**
6: remove file
7: **end for**

`getBucketET`(b)

8: b_{et}=∞
9: **for all** $im \in D.IMs$ **do**
10: b_{et} =`min`(b_{et},$im.BR[b].et$)
11: **end for**
12: **return** b_{et}

Algorithm 10 Failed State - LB

LB

Upon: Downstream instance I failure

1: **for all** b : $BR[b].dest = I$ **do**
2: $BR[b].state = Failed$
3: **end for**

Upon: Downstream instance I recovered

4: **for all** b : $BR[b].dest = I$ **do**
5: $BR[b].state = Active$
6: $T = Buf$.`get`(b,ts)
7: **for all** $t \in T$ **do**
8: `forward`(t,$BR[b].dest$)
9: **end for**
10: resume regular processing
11: **end for**

Algorithm 11 presents the steps followed by the *StreamCloud Manager* after discovering instance I_F has failed. We refer to the earliest timestamp from which tuples will be replayed as et. Timestamp et will be computed as the earlier among the earliest timestamps of any bucket b previously owned by I_F (Algorithm 11, lines 1-5). Once et has been computed, the *Resource Manager* is instructed to deallocate instance I_F and to allocate a replacement instance I_R. Once I_R has been allocated, the

query previously deployed at I_F is re-deployed at I_R (Algorithm 11, lines 6-8). In order to recreate the lost state, the IM deployed at I_R is instructed to replay persisted tuples belonging to I_R buckets, starting from et. Once the lost state has been recovered and I_R has been connected to its upstream and downstream peers, $U.LBs$ are instructed to forward any buffered tuple and resume regular processing (Algorithm 11, lines 10-12). As shown in Algorithm 10, lines 4-10, each LB changes the state of I_R buckets back to active, gets buffered tuples and, for each of them, forwards it if it belongs to I_R buckets. Once tuples have been replayed, $U.LBs$ resume regular processing.

Algorithm 11 Failed State - *StreamCloud Manager*

StreamCloud Manager

Upon: Instance I failure

1: $et = \infty$
2: **for all** $b \in I_F$ **do**
3: et =`min`(et,`getBucketET`(b))
4: R_B.`add`(b)
5: **end for**
6: `release`(I)
7: I_R = `allocate`()
8: `deploy`(Q,I_R)
9: I_R.IM.`replay`(ts,R_B,$U.LB_{prefix}$)
10: **for all** $b \in I, lb \in U.LBs$ **do**
11: lb.`recovered`(b)
12: **end for**

Algorithm 12 presents the pseudocode for the IM deployed at I_R. In order to replay R_B tuples, input merger at I_R first looks for all the Buf units persisted by upstream LBs and containing tuples t having timestamp $t.ts \geq et$. Each of these files contains tuples persisted in timestamp order. As multiple timestamp ordered files are read, the IM will have to merge-sort them in order to create a unique timestamp order stream of tuples. As we discussed, tuples forwarded to I_F are persisted by $U.LBs$ on a per-load balancer, per-time basis. This implies that, when reading them in order to recreated I_F lost state, tuples belonging to buckets that were not owned by I_F must be discarded.

Algorithm 12 Failed State - IM

IM

Upon: `replay`(ts,R_B,LB_{prefix})

1: $fileNames$ =`getFileNames`(ts,LB_{prefix})
2: F =`read`($fileNames$)
3: `mergedSort`(F)
4: **for all** $t \in F : t \in R_B$ **do**
5: `forward`(t)
6: **end for**

5.4.3 Failed while reconfiguring state

This section presents how *StreamCloud* fault tolerance protocol deals with failures happening during reconfiguration actions (i.e., while load is being balanced or a provisioning or decommissioning action has been triggered). In the following, we analyze separately how *StreamCloud* fault tolerance protocol covers the failure of each instance involved in a reconfiguration action. We refer to a reconfiguration action where a bucket is being transferred from instance A to B.

If, during a reconfiguration action, one of the upstream LBs fails, the failure can happen before or after all the control tuples that trigger the bucket ownership transfer have been received by A. In the former case, the reconfiguration action is postponed after recovering the failed LBs (the instances involved in the reconfiguration will not transfer any state due to the fact that they did not receive all the control tuples). In the latter case, no extra action must be taken. When recreating the failed LB at the replacement instance, it will be instructed to send incoming tuples to B (instance A will have sent the bucket state to B already, as they both received all the control tuples).

In case of failure of instance A, we can identify two possible cases: the failure happens before or after the bucket state has been sent (entirely) to B. In the former case, B will not receive the state of the bucket being transferred. To solve this problem, the *StreamCloud Manager* will instruct B to recreate the bucket building its state starting from the persisted tuples (*StreamCloud* fault tolerance protocol allows for the recovery of individual buckets). In the latter case, when the state has been already sent by instance A, the reconfiguration is not affected by instance A failure. State transfer is not affected if the failing instance is B. All buckets owned by B must be recovered, both if they were already owned or if they were being transferred when the instance failed.

5.4.4 Recovering state involved in previous reconfigurations

In this section, we analyze how past reconfiguration actions of the failed instance upstream subcluster triggered in the time interleaving the earliest timestamp and the failure affect recovery. That is, being t_{last} the last tuple processed by a failed instance I_F and being et the earliest timestamp from where to replay tuple, we analyze how reconfiguration actions involving I_F upstream subcluster taken during the period $[et, t_{last}]$ affect its recovery.

If I_F upstream subcluster has changed its size during period $[et, t_{last}]$ (e.g., the number of instances has decreased), then part of the tuples to replay has been persisted by an LB that does not longer exists. Nevertheless, this does not affect the protocol. As presented in Algorithm 12, the IM deployed at replacement instance I_R detects which files must be read depending on timestamp ts and LBs prefix name LB_{prefix}. Due to the fact that all the LBs of I_F upstream subcluster share the same prefix name (we discuss in Section 6.3 the operators naming convention), the files previously persisted by an LB that no longer exists are still considered in order to re-build I_F lost state.

5.4.5 Multiple instance failures

This section presents how *StreamCloud* fault tolerance protocol deals with multiple instance failures. As we discussed, fault tolerance is provided for a given instance relying on its upstream and downstream peers. Hence, if the two instances involved in the failure are not consecutive (i.e., one subcluster is the upstream of the other), their recovery can be executed in parallel as actions triggered by the *StreamCloud Manager* will not interfere. Similarly, the recovery of two failed instances is executed in parallel if the two instances belong to the same subcluster.

Opposite to these cases, if two failing instances I_{F_1}, I_{F_2} belong to consecutive subclusters (I_{F_1} preceding I_{F_2}), their recovery must be conducted in a specific order. The deployment of I_{F_1} operators and the connection to their upstream peers is executed in parallel with the deployment of I_{F_2} operators and the connection to their downstream peers. Nevertheless, connections between I_{F_1} and I_{F_2} can be established only after the operators of each instance have been recovered. This synchronization overhead does not significantly affects the recovery time. This is due to the fact that, among all the actions taken to recover a failed instance, connection to upstream and downstream peers takes a time negligible with respect to buckets recovery actions.

5.5 Garbage collection

As discussed in Section 5.2, operators obsolete state must be garbage collected in order for the fault tolerance protocol to be effective. The reason is that, if no garbage collection is provided, whenever the state of bucket b has to be recreated due to its owning instance failure its state will always be re-built starting from the oldest obsolete window.

Consider a stateful operator maintaining a window W, assume t_l is the latest tuple added to W. Given parameter $timeout$ and incoming tuple t_{in}, we say W is *obsolete* if $t_{in}.ts - t_l.ts > timeout \wedge t_{in} \notin W$, i.e., we say window W is obsolete if no tuple has been added to it in the last $timeout$ units.

In many scenarios, the query specified by the user implicitly considers timeouts for any stateful operator. As an example, the user might be interested in the average speed value of four consecutive position reports belonging to a vehicle, but might not be interested in such result if position reports span a time period of several days.

An interesting aspect is related to how obsolete windows are purged by the SPE. Given an obsolete window W we might want to produce its aggregated result even is not all the necessary tuples have been processed (e.g., a tuple based window having size 4 but containing only 2 tuples). On the other end, the user might be not interested in any result produced from obsolete windows.

As we said, streams are ordered by tuple timestamps. If an obsolete window is discovered due

to garbage collection, we must ensure that, in case its corresponding tuple is produced, the stream timestamp ordering guarantee is not violated. In the following sections we describe how garbage collection is implemented in *StreamCloud* depending on the window type.

5.5.1 Time-based windows

Two solutions are implemented in *StreamCloud* to manage time-based obsolete windows. If the user decides that obsolete windows should produce a result, then, whenever an incoming tuple causes it corresponding window to slide, all the open windows maintained by the operator are slid immediately. This solution does not affect results correctness. If an incoming tuple causes its windows to slide, the following incoming tuple belonging to a different window will also cause its window to slide. The adopted solution simply anticipates windows sliding in order to prevent obsolete windows to remain in memory.

If the user is not interested in results produced from obsolete windows, then it specifies $timeout$ value as a multiple of the window $advance$ parameter. A window covers intervals $[advance * i, advance * i + size[, \forall i = 1 \ldots n$. If $timeout$ is set, any time a window is slid to $[advance * i, advance * i + size[$, all windows up to $[advance * (i - timeout), advance * (i - timeout) + size[$ will be considered obsolete and therefore removed. For instance, defining a window with size and advance of 10 and 2 seconds, respectively, and $timeout = 2$, a window covering period $[0, 10[$ will be stale when receiving a tuple $t | t.ts \geq 4$, which will slide one of the operators windows to $[4, 14[$. Parameter $timeout$ is used to specify how frequently obsolete windows should be removed (setting $timeout = 0$, all windows are slid together). To reduce the impact of the garbage collection on the regular processing, open windows are maintained using a list sorted by their start timestamp. Doing this, obsolete windows to be removed will always be at the tail of the sorted list.

5.5.2 Tuple-based windows

The solution adopted in *StreamCloud* to process obsolete tuple-based windows consist in discarding them without producing any result. A window is considered obsolete (and is therefore erased) if no tuple has been added to it in the last $timeout$ time units. This solution is adopted because, contrary of time-based tuples, tuple-based window results cannot be anticipated. Outputting tuples computed over obsolete windows violates the stream timestamp ordering guarantee. Nevertheless, we stress that this decision does not lead to incomplete results. Parameter $timeout$ is set by the user and must be therefore chosen accordingly to the results expected by him.

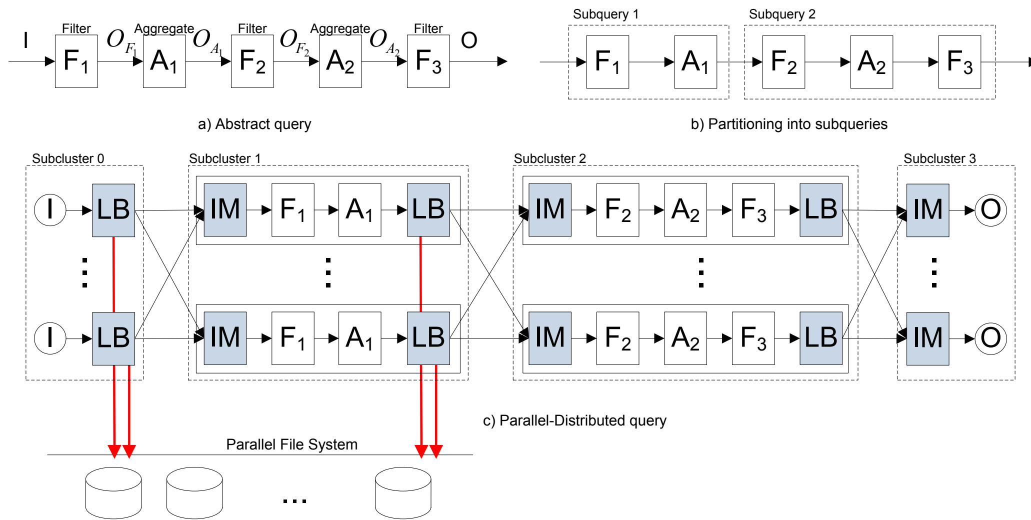


Figure 5.7: Linear Road.

5.6 Evaluation

In this section, we presented the evaluation of *StreamCloud* fault tolerance protocol. We evaluate the protocol studying its runtime overhead, its recovery time, the scalability of the replicated-distributed file system and the effectiveness of the garbage collection protocol for varying setups.

5.6.1 Evaluation Setup

The evaluation was performed in a shared-nothing cluster of 100 nodes (blades) with 320 cores. The details about the machines composing the cluster are presented in Section 3.3.1. All the experiments have been conducted using a query extracted from the Linear Road benchmark. Linear Road [ACG+04], [JAA+06] is the first and most used SPE benchmark. It has been designed by the developers of Aurora [ACC+03] and Stream [STRa]. Linear Road simulates a toll system for expressways of a metropolitan area. Tolls are computed considering aspects such traffic congestions and accidents. In order to evaluate the performance of an SPE, the system running Linear Road must be capable of processing the various position reports of a given number of highways generating tolls and accident alerts with a maximum delay of 5 seconds. The performance attained by an SPE is expressed in number of highways. The overall Linear Road query is composed by several modules used to check for the presence of accidents, to maintain statistics about each segment of the highway in order to compute the corresponding toll, a module to notify tolls and so on. In our evaluation, we focus on one of the Linear Road modules, the accident detection module, shown in Figure 5.7.a.

This portion of the Linear Road query is used to detect accidents. An accident happens if at least two stopped vehicles are found in the same position at the same time. Following Linear Road

Field Name	Field Type
Time	*integer*
Vehicle_ID	*integer*
Speed	*integer*
Position	*integer*
Type	*integer*

Table 5.3: Linear Road tuple schema

specifications, a vehicle is considered as stopped if four consecutive position reports of the same vehicle are related to the same position and all have speed equal to zero. In the following, we provide the details about the query input tuples schema and the operators definition.

The schema of the input tuples is presented in table 5.3. Field $Time$ specifies the time at which the position report is generated (expressed in seconds). $Vehicle_ID$ is the unique identifier of each vehicle. $Speed$ field represents the speed at which the vehicle is moving when the report is created. $Position$ identifies the position of the vehicle. Finally, $Type$ is used to distinguish between position reports tuples ($Type = 0$), toll request ($Type = 1$) and other reports types. In Linear Road specification, the position of a vehicle is given by several fields ($Highway$,$Segment$ and $Direction$ among others). Furthermore, the input schema contains additional fields. In the description, we are presenting only the fields relevant to our query, defining a single $Position$ field in order to simplify the description (without any loss).

The query consists of 5 operators. Filter F_1 is used to pass only position report tuples ($Type = 0$). The operator does not modify the tuples schema and is defined as:

$$F\{Type = 0\}(I, O_{F_1})$$

The aggregate operator A_1 is used to compute, for each distinct vehicle (i.e., $Group - by$ is set to field $Vehicle_ID$) the average speed and initial and final position of each group of 4 consecutive position reports (i.e., window $size$ and $advance$ are set to 4 and 1 tuples, respectively). The operator is defined as:

$$A\{tuples, 4, 1, Avg_Speed \leftarrow avg(Speed), First_Pos \leftarrow first_val(Position),$$
$$Last_Pos \leftarrow Last_val(Position), Group - by = Vehicle_ID\}(O_{F_1}, O_{A_1})$$

The output schema consists of fields $\langle Vehicle_ID, Avg_Speed, First_Pos, Last_Pos\rangle$.

Next to operator A_1, filter F_2 is used to pass only tuples referring to stopped vehicles. The operator does not modify the input tuples schema and is defined as:

$$F_2\{Avg_Speed = 0 \wedge First_Pos = Last_Pos\}(O_{A_1}, O_{F_2})$$

Aggregate A_2 is used to group together each pair of aggregated position reports referring to the same position in order to look for an accident (i.e., two distinct cars stopped at the same position). The operator is defined as:

$$A\{tuples, 2, 1, Vehicle_A \leftarrow first_val(Vehicle_ID),$$
$$Vehicle_B \leftarrow last_val(Vehicle_ID), Group - by = First_Pos\}(O_{F_2}, O_{A_2})$$

The output tuples schema consists of fields $\langle First_Pos, Vehicle_A, Vehicle_B\rangle$

Finally, filter F_3 is used to forward accidents alerts (i.e., tuples referring to a pair of distinct vehicles stopped at the same position). The operator does not modify the input tuples schema and is defined as:

$$F_3\{Vehicle_A \neq Vehicle_B\}(O_{A_2}, O)$$

Figure 5.7.b presents how the original query is partitioned into subqueries. A subquery is defined for each stateful operator. In this case, we adopted a slight variation of the *StreamCloud* subquery partitioning approach where the stateless prefix of the query (operator F_1) has been merged with the subquery containing A_1 and where the stateless operator following A_1 has been assigned to the following subquery. That is, subquery 1 is composed by operators F_1 and A_1 while subquery 2 is composed by operators F_2,A_2 and F_3.

Figure 5.7.c shows the global parallel-distributed query. The number of nodes assigned to each subcluster changes depending on the experiment. In the following sections, we define the subcluster size for each experiment. In our evaluation, we provide fault tolerance for subclusters 1 and 2 (subcluster 0, containing the input sources, and subcluster 1, containing the output receivers, are external applications for which we do not provide fault tolerance). As shown in the figure, tuples being forwarded to subcluster 1 by subcluster 0 LBs (and to subcluster 2 by subcluster 1 LBs) are being persisted to disk.

Linear Road benchmark provides a data simulator that is used to create the position reports for any given number of highways. Data created by the simulator covers a period of 3 hours. In our experiments, we generated position reports for 20 distinct highways. Figure 5.8 presents how the input data load evolves during the 3 hours period.

5.6.2 Runtime overhead

With this experiment, we evaluate the overhead introduced by *StreamCloud* fault tolerance protocol. We are interested in measuring how tuples processing latency is affected if LBs are forwarding and persisting tuples in parallel. For this reason, latency is measured during a fail-free period. As

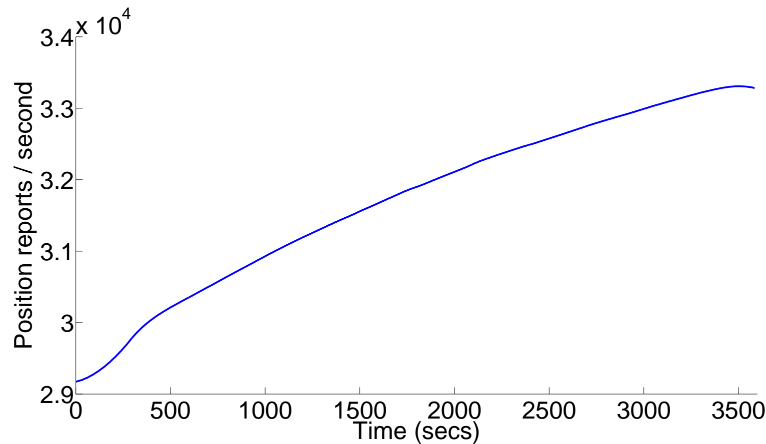


Figure 5.8: Input rate evolution of data used in the experiments

discussed in the previous section, LBs at subcluster 0 are persisting the tuples forwarded to subcluster 1 while LBs at subcluster 1 are persisting the tuples forwarded to subcluster 2. We measure the processing latency at subclusters 0 and 1 with and without fault tolerance. Rather than measuring the latency at the query end, measurements are taken at the end of each subcluster in order to precisely quantify the runtime overhead, excluding the latency introduced by following subclusters. In this experiment, each subcluster has been deployed over 10 *StreamCloud* instances. Figure 5.9 presents the latency measured at subcluster 0 with (solid blue line) and without fault tolerance (dotted black line). When fault tolerance is not active, the average latency is around 1.5 milliseconds. When fault tolerance protocol is active, the latency grows up to 3 milliseconds, approximately.

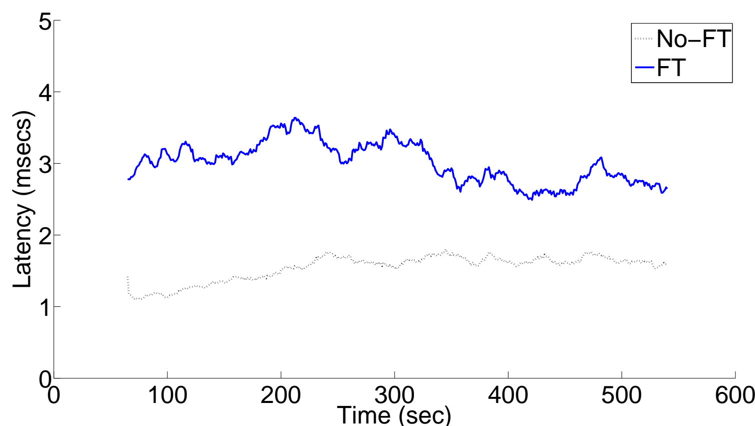


Figure 5.9: Latency measured at subcluster 0

Figure 5.10 shows the latency measured at subcluster 1 (as before, the solid blue line represents the latency when fault tolerance is activate while the dotted black line when it is not active). When fault tolerance protocol is not active, the average latency is around 55.5 milliseconds. When fault

tolerance protocol is active, the latency grows up to 57 milliseconds, approximately. It should be noticed that, in these experiments, whenever fault tolerance is activated, we activate it for both subclusters. That is, when measuring the extra latency introduced by LBs at subcluster 1 (providing fault tolerance for subcluster 2), the measurement includes the extra latency caused by LBs at subcluster 0 (providing fault tolerance for subcluster 1).

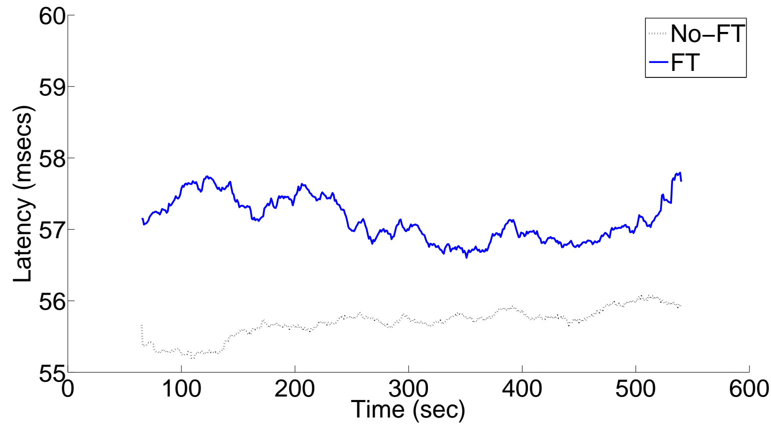


Figure 5.10: Latency measured at subcluster 1

As shown in the figures, we experience an increase of the processing latency at subcluster 0 but the extra overhead measured at subcluster 1 is negligible. This is due to the different number of operators deployed at the different subclusters. Subcluster 0 instances are running a single stateless operator, the LB used to distribute each data source tuples. In this case, the extra operations performed by the LBs have a noticeable impact on the latency, increasing it of approximately 2 milliseconds. Subcluster 1 instances run two operators (stateless filter F_1 and stateful aggregate A_1) plus an input merger and a load balancer. In this case, the extra operations performed by the LBs have a negligible impact with respect to the computations performed by the 4 operators. This is noticed as the increase in the latency (approximately of 2 milliseconds) is mainly caused by subcluster 0 LBs.

5.6.3 Recovery Time

This experiment has been conducted to measure the recovery time of single-instance failures. The recovery time is mainly defined by the *deploy* time (i.e., the time it takes to deploy the replacement instance) and the *state recovery* time (i.e., the time it takes to read persisted tuples are recreated the lost instance state). The *deploy* time is measured as the interleaving time between the detection of a failing instance I_F and the replacement of its query at I_R while the *state recovery* time is measured as the interleaving time between the request to recreate the state reading persisted tuples and the instant when I_R state has been fully recovered. In the evaluation, we take into account single instance failures of subcluster 1. In all the experiments, *StreamCloud* fault tolerance protocol performs well, showing

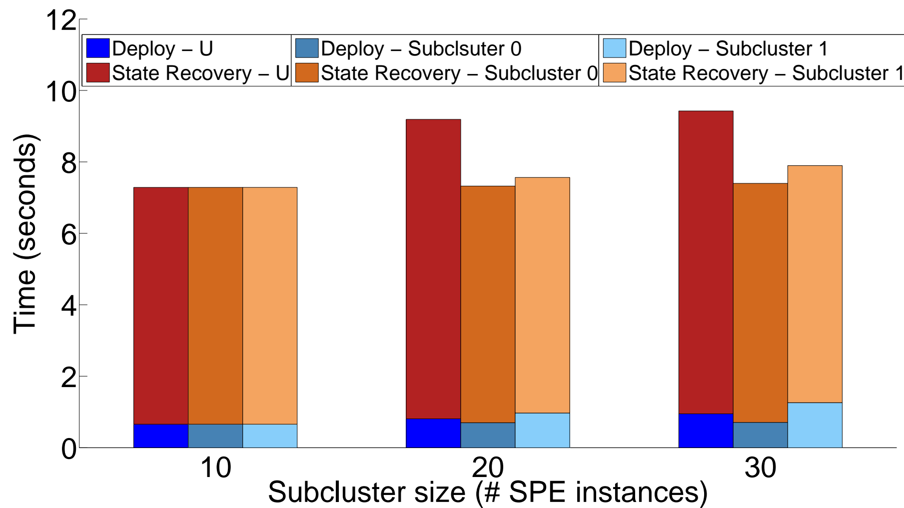


Figure 5.11: Deploy and State Recovery times for changing subcluster sizes

a small recovery time of approximately 7 seconds. To study in detail the different phases of the recovery, both *deploy* and *state recovery* times have been studied with respect to changing subcluster sizes and changing number of buckets. We take as base scenario the setup where each subcluster is deployed at 10 *StreamCloud* instances and, subsequently, we vary the size of each subcluster to 20 and 30 nodes. Moreover, we consider 3 different scenarios where the traffic sent from subcluster 0 to subcluster 1 is partitioned into 300, 600 and 900 buckets, respectively. We first discuss the expected results and proceed subsequently presenting the evaluation measurements. When deploying a query, its deploy time depends on the number of upstream and downstream instances to which it has to connect to. Hence, we expect the *deploy* time to increase when the upstream or the downstream subclusters of subcluster 1 change their size, while we should see a negligible variation with respect to varying sizes of subcluster 1. Considering now the *state recovery* time, we expect it to increase together with the upstream subcluster size. The rationale is that, the bigger the number of nodes persisting tuples on their own files, the higher the price we pay to open all the persisted files in parallel in order to recover the state.

Figure 5.11 presents the results of the evaluated *deploy* and *state recovery* times with respect to changing subcluster sizes. The first group of three bars (subclusters of size 10) shows the same *deploy* and *state recovery* times for the three configurations (in the three cases the setup is equal to the base setup where each subcluster is deployed at 10 instances). For this configuration, the *deploy* time is of approximately 0.66 seconds while the *state recovery* time is of 6.63 seconds. The second group of bars shows the *deploy* and *state recovery* times when one of the three subcluster is deployed at 20 *StreamCloud* instances. If subcluster 0 is deployed at 20 instances (first bar), both *deploy* and *state recovery* time increase with respect to the base case. The *deploy* time grows to 0.81 seconds

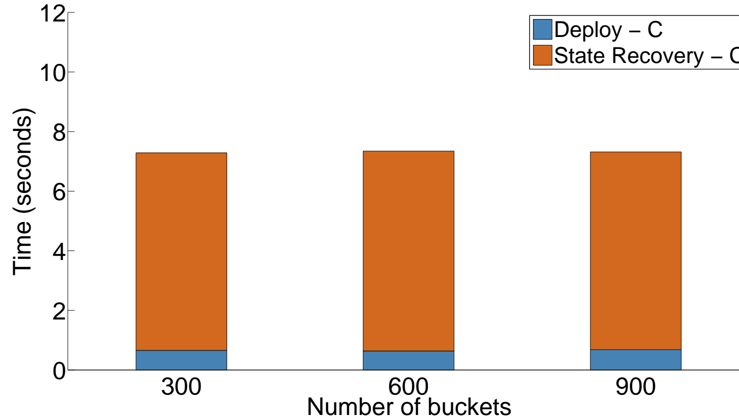


Figure 5.12: Deploy and State Recovery times for changing number of buckets

while the *state recovery* times grows to 8.38 seconds. The growth of the *deploy* time is due to the increased number of upstream LBs to which the query being deployed at subcluster 1 must connect. The growth of the *recovery state* time is due to the increase in the number of persisted files that must be read in order to recreate I_F lost state (as discussed in 5.4.1), tuples are persisted on a per-time period, per-LB basis). As expected, if subcluster 1 (second bar) is deployed at 20 *StreamCloud* instances, neither the *deploy* time nor the *state recovery* time change from the base case. Finally, if subcluster 2 is deployed at 20 instances (third bar) we can observe that only the *deploy* time increases to 0.97 seconds. The *state recovery* dos not change due to the unvarying number of persisted files that must be read to recreate subcluster 1 state with respect to the base case. The considerations done for the three subclusters when deployed over 20 *StreamCloud* instances hold also when the subclusters are deployed at 30 *StreamCloud* instances. In this case, when deploying subcluster 0 at 30 instances both *deploy* and *state recovery* time grow to 0.95 and 8.48 seconds, respectively. When deploying subcluster 1 at 30 instances *deploy* and *state recovery* time do not vary significantly (0.71 and 6.69 seconds, respectively). Finally, when deploying subcluster 2 at 30 instance only the *deploy* time grows (1.26 seconds) while the *state recovery* time is comparable to the one of the base case (6.64).

Figure 5.12 presents the *deploy* and *state recovery* times when increasing the number of buckets used to route the traffic flowing from subcluster 0 to subcluster 1. As it can be seen, both times are independent from the number of buckets being used by subcluster 0.

5.6.4 Garbage Collection

This experiment measures the effectiveness of *StreamCloud* garbage collection protocol. With respect to the query taken into account in this evaluation, we study how the size of the memory managed by aggregate A_1 changes with respect to 4 different configurations: $No-GC$, $GC-1200$, $GC-600$ and $GC-30$. $No-GC$ refers to a configuration where no garbage collection is defined.

$GC - 1200$ refers to a configuration where garbage collection timeout is set to 1200 seconds (i.e., if a window has not received any tuple in the last 1200 seconds, then it is removed). Similarly, $GC - 600$ and $GC - 30$ refer to configurations where garbage collection timeout is set to 600 and 30 seconds, respectively. Due to Linear Road semantics, 30 is the minimum timeout that can be set without incurring in any result loss. Position reports referring to the same vehicle have time distance of 30 seconds, therefore, if no position report for a given vehicle has been received in the last 30, its corresponding window can be discarded without affecting results.

Figure 5.13 presents the experiment result. Four lines are depicted, one for each configuration. It can be noticed that, with respect to the $No - GC$ configuration (solid line), the number of open windows increases linearly, reaching the highest value of roughly 1.5 million windows during the 3 hours of data. With respect to configurations $GC - 1200$ (dashed line), $GC - 600$ (dotted line) and $GC - 30$ (dash-dot line), the number of open windows increases linearly in the first phase, continuing after with a milder slope. Considering, as an example, configuration $GC - 1200$, it can be noticed that the number of open windows starts growing slower around second 1200. This is as expected, any obsolete window maintained will be discarded after 1200 seconds of inactivity. The rationale is that, after the warm up phase, the simulated traffic contains cars that are always entering the highway while cars leaving it; each time a car leaves the highway, its state becomes obsolete. Looking at time 2000, the number of open windows maintained using configurations $No - GC$, $GC - 1200$, $GC - 600$ and $GC - 30$ is, respectively, 1.26, 1.15, 1.05 and 0.95 millions. This implies that, if a failure happens at time 2000, having no garbage collection implies the recreation of 32% more windows with respect to a scenario where garbage collection timeout is set to 30 seconds. Notice that these extra windows that must be recovered are actually obsolete (i.e., it contain earlier tuples), which leads to a bigger serialized state to be read and replayed.

5.6.5 Storage System Scalability Evaluation

In this section, we provide an evaluation of the storage system scalability. We use term *server* to refer to one instance of the parallel file system managing the read and write operations to a single, dedicated physical disk while we use term *client* to refer to one LB instance persisting information on the parallel file system. We analyze which is the throughput of different setups with an increasing number of servers and clients (resp. 1, 4 and 8). Our evaluation shows that, even when the number of servers is lower than the number of available physical disks, the storage system does not usually constitutes a bottleneck as its throughput is higher than the throughput achieved by the *StreamCloud* instances running the query.

Figure 5.14 presents the storage system scalability evaluation. The *X* axes represents the number of servers of the parallel file system, the left *y* axes shows the throughput expressed in MB/s while the right *Y* shows the throughput expressed in Linear Road tuples / second (in our setup, one tuple is

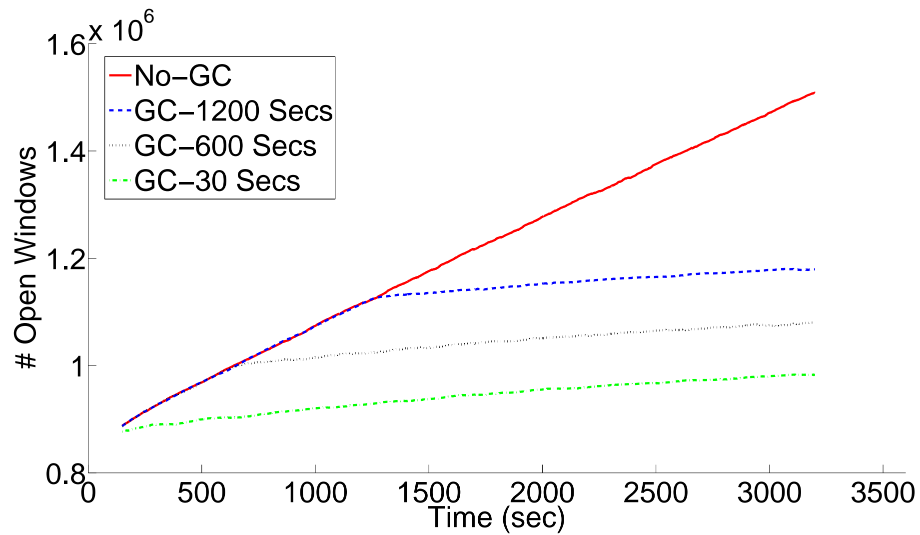


Figure 5.13: Garbage Collection evaluation

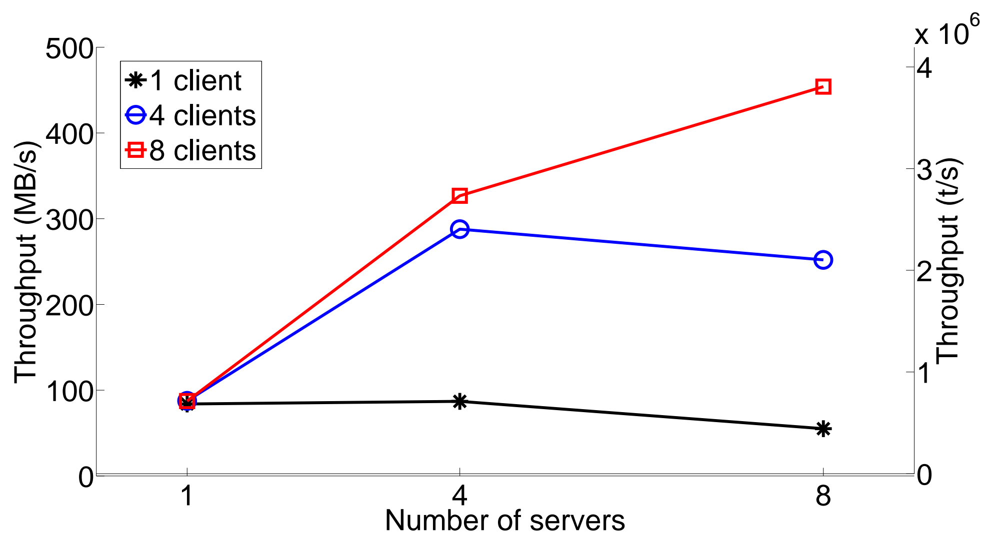


Figure 5.14: Storage System Scalability Evaluation

defined by 125 bytes). In all the experiments, clients issue write requests of 1 MB blocks. We consider 3 different setups of 1, 4 and 8 servers and 1, 4 and 8 clients. When defining a setup of a single server, the throughput achieved by a single client (line marked with black stars) is approximately 80 MB/s (or 670000 Linear Road tuples per second). This throughput does not change when having 4 or 8 writers accessing the parallel file system in parallel (lines marked with empty circles and squares, respectively). When defining 4 servers, the throughput achieved by 1 client does not change, while 4 and 8 clients achieve a throughput of approximately 300 MB/s (or 2.5 millions tuples per second). Finally, when defining a setup with 8 servers, the throughput slightly increases when using 4 clients while it grows up to almost 460 MB/s (or 3.9 millions tuples per second) when using 8 clients.

Part VI

VISUAL INTEGRATED DEVELOPMENT ENVIRONMENT

Chapter 6

Visual Integrated Development Environment

6.1 Introduction

One of the challenges in designing a parallel-distributed SPE, as discussed in Section 3.1 while presenting the evolution of SPEs, is to provide *syntactic transparency*. A syntactically transparent parallel-distribute SPE will provide functionalities to define parallel-distributed queries in the same way as centralized queries, simply providing additional information about the nodes at which queries can be deployed. *StreamCloud* Visual Integrated Development Environment (IDE) has been designed to specifically address this challenge. This IDE eases the user interaction with *StreamCloud* SPE. That is, it eases the programming of queries and automates the parallelization process. Furthermore, it eases the monitoring of running queries and provides utilities to inject data to a query reading it from text of binary files (containing the tuples to be sent). Four different tools have been developed in the context of *StreamCloud*, more precisely:

1. *Visual Query Composer:* A Graphical User Interface (GUI) application that eases the composition of queries providing a drag-and-drop interface where operators of a query can be easily added and interconnected. Most of the existing commercial SPEs define a GUI to simplify the programming of queries not only to ease the interaction with the user, but also because this activity is usually tedious and error-prone. For instance, a task that can be simplified and automatized is the assignment of query operators and streams name by means of a naming convention. The Borealis project, the SPE upon which *StreamCloud* is built, included a first

prototype of GUI for the programming of queries. Nevertheless, it was an early prototype that has been re-designed and implemented in order to include aspects covered by *StreamCloud*such parallelization and elasticity.

2. *Query Compiler:* an application that transforms an *abstract* query (i.e., a query that contains no information about how to distribute operators to the system nodes and which nodes to use) into its parallel-distributed counterpart. The Query Compiler is currently integrated with the Visual Query Composer.

3. *Real Time Performance Monitoring Tool:* a web-based application that integrates with *Stream-Cloud* and shows run-time statistics such input rate, output rate or CPU consumption of query operators. Statistics are aggregated on a per-parallel operator basis. That is, the average CPU consumption of a parallel-operator is computed as the average CPU consumption of the nodes where the operator is running.

4. *Distributed Load Injector*: Once an *abstract* query has been defined and its parallel-distributed version has been compiled, the user still needs a way to inject tuples to it. For this reason, we provide a Distributed Load Injector. With respect to the distributed Load Injector tool, data can be forwarded at the rate defined by the tuples timestamps (i.e., the interleaving time between each tuple forwarding is equal to their time distance) or injected at a rate specified by the user, that can be adjusted manually at runtime.

Borealis, the SPE upon which *StreamCloud* has been built, provided some basic tools to ease the composition of data streaming applications. We present them in Appendix A.4.

In the following examples, we refer to the high-availability fraud detection query presented in Section 2.2, used to spot phone numbers that, between two consecutive phone calls, cover a suspicious space distance with respect to their temporal distance.

6.2 Visual Query Composer

The Visual Query Composer (VQC) GUI has been designed to ease the composition and deployment of queries. The steps performed by the user in order to compose a query consists in the definition of the query operators, the definition of how they interconnect and the definition of the system input and output.

The IDE provides a drag and drop interface that allows for an intuitive addition or removal of data streaming operators in order to create the operators that compose the query. Once an operator has been added to the query, the user specifies its semantic. As discussed in Appendix A, the Borealis project (and so *StreamCloud*) define queries by means of XML files. More precisely, a *query file* is

used to specify which operators compose the query and their attributes while a separate *deploy file* is used to specify the nodes at which each operator will run. Listing 1 presents the definition of the fraud detection query aggregate operator by means of the Borealis (and *StreamCloud*) XML syntax.

```
<box name="a" type="aggregate" >
        <in   stream = "O_U" />
        <out  stream = "O_U" />
        <parameter name = "aggregate−function.0"     value = "firstval(Time)" />
        <parameter name = "aggregate−function−output−name.0"  value = "Time" />
        <parameter name = "aggregate−function.1"     value = "firstval(Time)" />
        <parameter name = "aggregate−function−output−name.1"  value = "T1" />
        <parameter name = "aggregate−function.2"     value = "firstval(X)" />
        <parameter name = "aggregate−function−output−name.2"  value = "X1" />
        <parameter name = "aggregate−function.3"     value = "firstval(Y)" />
        <parameter name = "aggregate−function−output−name.3"  value = "Y1" />
        <parameter name = "aggregate−function.4"     value = "lastval(Time)" />
        <parameter name = "aggregate−function−output−name.4"  value = "T2" />
        <parameter name = "aggregate−function.5"     value = "lastval(X)" />
        <parameter name = "aggregate−function−output−name.5"  value = "X2" />
        <parameter name = "aggregate−function.6"     value = "lastval(Y)" />
        <parameter name = "aggregate−function−output−name.6"  value = "Y2" />
        <parameter name = "window−size"  value = "2"   />
        <parameter name = "window−size−by"  value = "TUPLES"   />
        <parameter name = "advance"  value = "1"   />
        <parameter name = "group−by"  value = "Phone" />
</box>
```

Listing 1: Aggregate operator definition

In *StreamCloud*, an operator is defined by a *box* XML element containing one (or more) *in* element(s), one (or more) *out* element(s) and several *parameter* elements. *Parameter* elements are defined as optional and used to define any property of an operator. As an example, we respect to the aggregate operator, the *parameter* element is used to specify the window semantics and the functions applied over the input data. With respect to the aggregate operator of the fraud detection query, the *box* XML element contains one *in* and one *out* element. With respect to the window semantics, size and advance parameters are referred to as *window-size* and *advance* while parameter *window-size-by* is set to *TUPLES* to specify that the window is tuple-based. The group-by field is specified using the parameter *group-by*. For each function that the user defines over the window data, a pair of parameters, *aggregate-function* and *aggregate-function-output-name*, is used to specify the function and the output field name containing its result.

Once the semantic of each operator has been defined, operators are interconnected using the GUI interface linking them visually. The last step performed by the user to complete the abstract query is the definition of its input and output streams. Input (and output) streams are defined and connected with the same drag and drop interface used to define query operators. When defining a system input (or output) stream, the user defines a *schema* XML element containing a *field* element for each attribute defined by the tuples schema. Listing 2 presents how the query input schema is defined using *StreamCloud* XML syntax.

```
<schema name="input_schema">
        <field name="Caller" type="string" size="9"/>
        <field name="Callee" type="string" size="9"/>
        <field name="Time" type="int"/>
        <field name="Duration" type="int"/>
        <field name="Price" type="double"/>
        <field name="Caller_X" type="double"/>
        <field name="Caller_Y" type="double"/>
        <field name="Callee_X" type="double"/>
        <field name="Callee_Y" type="double"/>
</schema>
```

Listing 2: Input Schema operator definition

We provide a complete example of how the query is defined by means of XML elements in Appendix A.1.

Figure 6.1 presents a snapshot of the Visual Query Composer application. The snapshot captures how the user defines the semantic of the query operators and how operators are interconnected. The VQC allows the user to add input and output streams, basic data streaming operators (Filter, Map, Union, Aggregate and Join) and table operators (Insert, Update, Delete, Select). Table 6.1 presents the different operators (and their icons) provided by the VQC. The top panel shown in Figure 6.1 lets the user connect operators while the low panel on the right permits the user to specify each their semantic. In the Figure, the user is defining the semantic of the aggregate operator (the XML is the one presented in Listing 1).

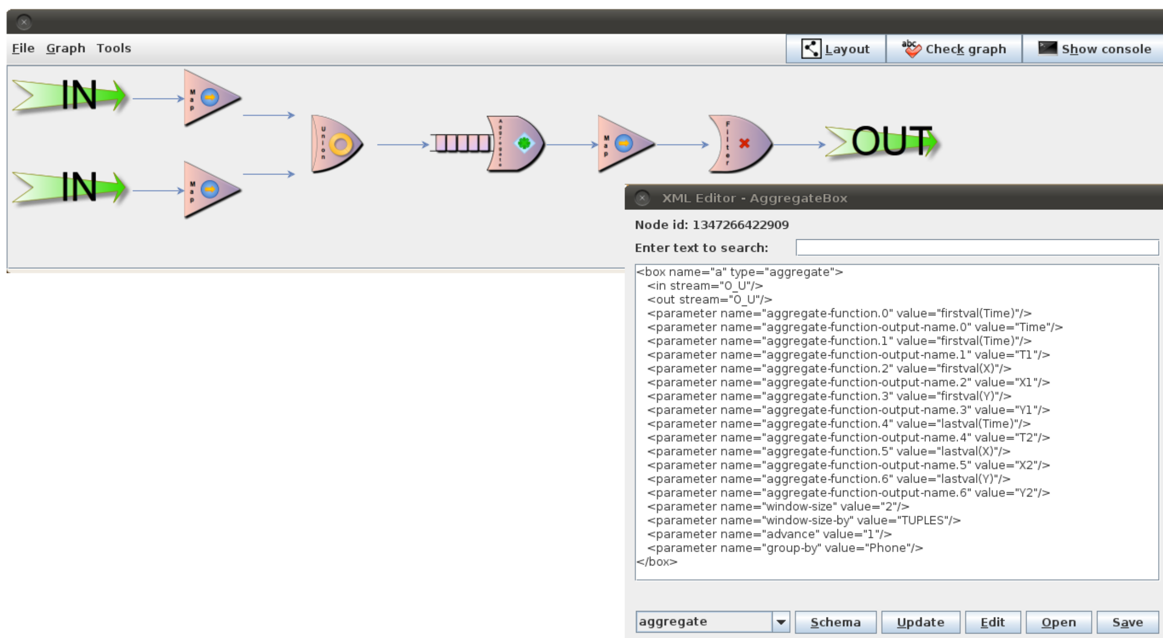


Figure 6.1: Abstract query definition

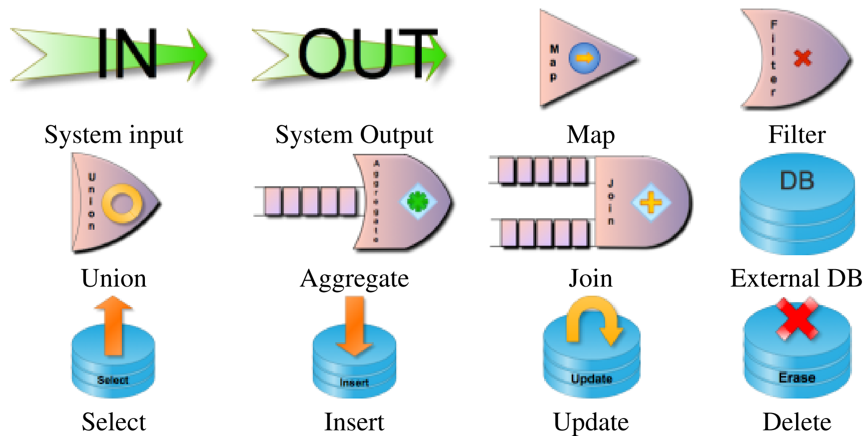


Table 6.1: VQC Operators legend

Together with the drag-and-drop interface that eases query composition, the VQC defines a set of syntactic and semantic correctness rules to fix possible user mistakes. In order to guarantee syntactic correctness, the application checks XML elements against the XSD schema definition, making sure that parameters and attribute names are nor misspelled or empty. Furthermore, syntactic correctness is guaranteed making sure that all the mandatory elements of each operator are defined. As presented before, attributes of an operator (like the windowing semantic of an aggregate operator) are defined by means of the *parameter* element. Each operator can define any number of *parameter* element, depending on the number of attributes it requires. For this reason, the VQC checks, for each operator, if all the mandatory attributes have been defined and if the number of input and output streams is correct (e.g., it checks whether the Union operator defines exactly one output stream). To guarantee semantic correctness, the VQC checks that the operations performed by each operator are based on tuple fields previously defined. That is, fields that are either defined by the previous operators or as system input streams. Similarly, the VQC checks whether the schema of the system output stream is consistent with the one of the operators generating the system output itself. Another condition that is checked with respect to streams is the correctness of names. That is, the VCQ checks that the stream connecting two operators has been defined with the same name in both XML definitions. The VQC also defines conditions to be checked with respect to the overall query. As an example, one condition that must be satisfied is that no loops are defined in the query (as discussed in Section 1.3, a query is defined as a directed acyclic graph - DAG).

6.3 Query Compiler and Deployer

In order to have a complete IDE, we have integrated two additional components: the *Query Compiler* and the *Deployer*. The Query Compiler transforms an *abstract* query (e.g., the query that

has been composed as described in the previous section) into its parallel-distributed counterpart. The Deployer generates all the input files and the script that can used by the user to execute the resulting application.

In order to compile an *abstract* query, the Query Compiler follows three steps: (1) partitioning into subqueries, (2) template creation and (3) parallelization. We introduce each step separately. Figure 6.2 presents an overview of these steps with respect to the fraud detection query (shown in Figure 6.2.a).

Partitioning into subqueries. The first step performed by the complier is the partitioning of the query into subqueries. Query partitioning is applied by the Query Compiler following the operator-set-cloud partitioning strategy presented in Section 3.2, where a subquery is defined for each stateful operator (and the stateless operators following it) and an additional subquery is defined for the query stateless prefix of operators. If the user decides to apply a different criteria to partition the query, it can specify manually how operators are partitioned into subqueries (selecting a group of operators and specifying they belong to the same subquery). Figure 6.2.b shows how the query is partitioned following the operator-set-cloud strategy. In the example, 3 subqueries are defined if partitioning is applied by the Query Compiler. One subquery contains the initial map operator M_1 and the following union operator U. A separate subquery contains the map operator M_2. Finally, a third subquery includes the aggregate operator A and its following stateless operators (map operator M_3 and filter operator F). Once subqueries have been defined the Query Compiler, the user is asked to specify which nodes will be used to run the parallel query in the initial deployment and if elasticity should be active while running the query (if elasticity should be provided, the user is asked to define a set of available instances that can be provisioned if necessary).

With respect to Figure 6.2.b, we suppose each subquery i will be deployed to a set of n_i nodes. Figure 6.3 presents a snapshot of the VQC application where the user is specifying which nodes are used for the initial deploy and which nodes can be provisioned. Node are expressed by means of an $IP{:}Port$ address. With respect to the subquery containing the aggregate operator, the user has specified that it will be deployed over 6 instances (the *Assigned* list contains 6 entries). Having defined 5 additional available instances(the *Available* list contains 5 entries), the user specifies that new instances can be provisioned, up to a maximum of 5 .

Template creation. During this second step, the Query Compiler converts each subquery into an XML template that is later on used to create the parallel-distributed query. For each subquery, its corresponding template represents the XML query that will be deployed at the subquery instances. The XML template includes the definition of the operators belonging to the subquery plus an input merger on each incoming edge (i.e., an input stream generated by an operator belonging to a different

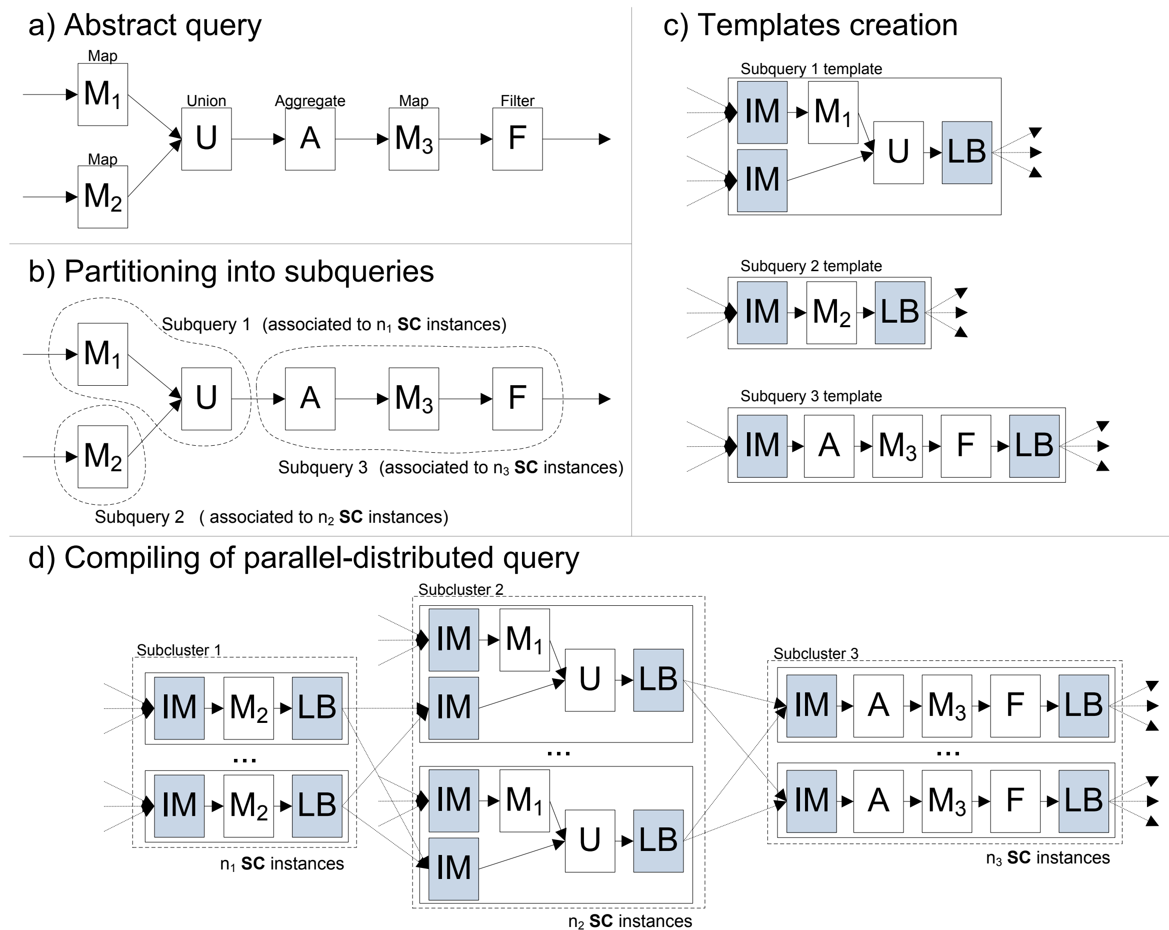


Figure 6.2: Compiling an abstract query into its parallel-distributed counterpart

subquery or by a system input) and a load balancer on each outgoing edge (i.e., an output stream feeding an operator belonging to a different subquery or a system output). All the operators name and streams names are enriched with a suffix that will be later used to enumerate them (a sample operator OP distributed over 3 *StreamCloud* instances will have names OP_1, OP_2 and OP_3). Figure 6.2.c shows how each subquery has been converted into its corresponding template.

Parallelization. During the last step, subqueries XML templates built during the previous stage are used to create the parallel-distributed query and create the files needed by *StreamCloud* to execute the query. As shown in Figure 6.2.d, each subcluster is created duplicating its corresponding subquery template (the number of duplicates being equal to the number of *StreamCloud* instances to which the subcluster will be deployed). Once each subquery has been duplicated the required number of times, operators and stream suffix names are updated and the resulting XML object is written to disk. The Query Compiler creates the following files starting from the abstract query:

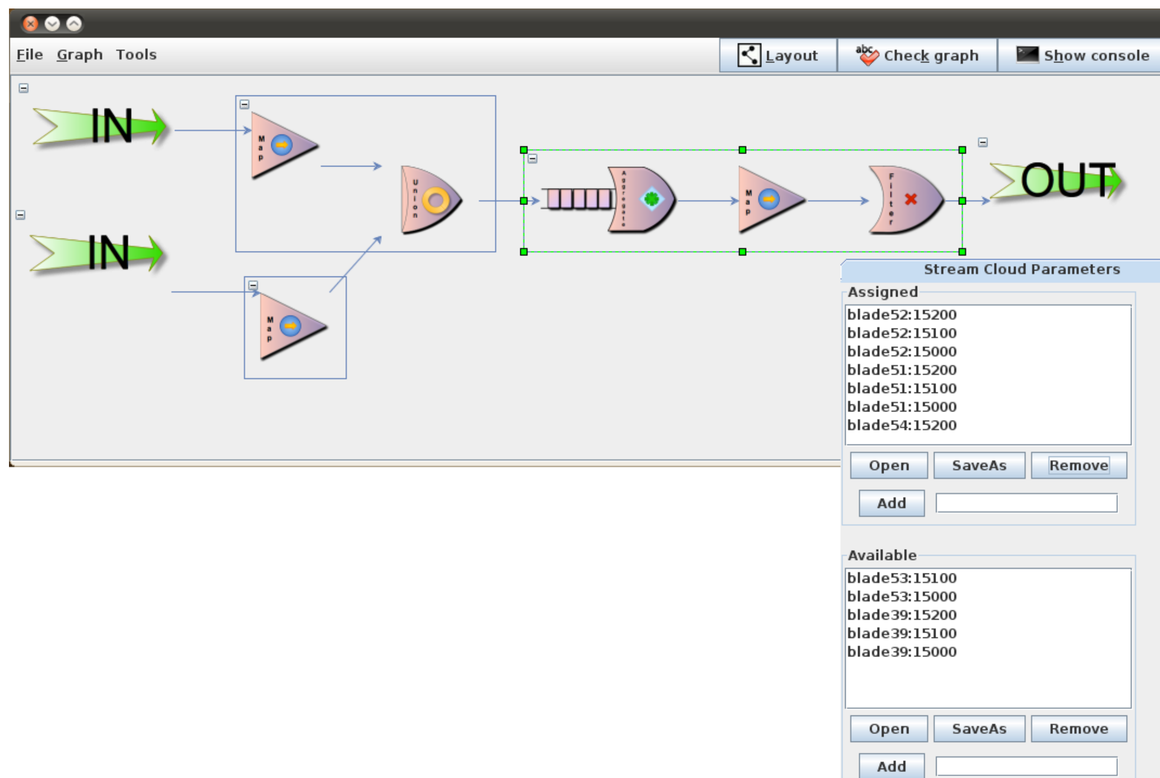


Figure 6.3: Subquery partitioning

query.xml The XML file defining the parallel-distributed query. The file contains the definition of all the operators belonging to the query; we provide an example of such file in Appendix A.

deploy.xml This XML file defines the *StreamCloud* instances where each subcluster is deployed. Deployment information is not maintained in the same file that defines the query so that different deployments can be defined referring to the same parallel-distributed query.

SC.xml / ResourceManager.xml these XML files contains parameters used by the *StreamCloud Manager* for provisioning, decommissioning, dynamic load balancing or fault tolerance actions. An excerpt of a sample SC.xml file is presented in Listing 3. The file contains information about the query and deploy files, the resource manager file, information about the recovery (in the example the heartbeat period is set to 1 millisecond) and information about the Load Injectors (in order to start or stop them). The second file created by the Query Compiler (ResourceManager.xml), contains information about the *StreamCloud* instances assigend to the operators and the ones that can be provisioned. An excerpt of a sample ResourceManager.xml file is presented in Listing 4. It can be noticed that different elements are used to define assigned and available instances. For each machine, the file specifies how many *StreamCloud* instances must be activated (and their ports). Furthermore, the file specifies additional

parameters, like the desired number of *StreamCloud* instance that should be kept in the pool of available instances.

```
<SC>
    <Query>path to query file</Query>
    <Deploy>path to deploy file</Deploy>
    <ResourceManager>path to resource manager file</ResourceManager>
    <FaultTolerance>
        <RecoveryType>precise</RecoveryType>
        <HeartBeatPeriod>1000</HeartBeatPeriod>
    </FaultTolerance>
    <LBs>
        <Instances>
            <LB>blade52:5000</LB>
            <LB>blade42:5008</LB>
            <LB>blade42:5009</LB>
        </Instances>
    </LBs>
</SC>
```

Listing 3: StreamCloud Parameters example

```
<ResourceManagerParameters>
   <AssignedNodes>
      <SCStarter Id="blade52" Port="8000" IP="blade52">
         <SCInstance Port="15200" IP="blade52"/>
         <SCInstance Port="15100" IP="blade52"/>
         <SCInstance Port="15000" IP="blade52"/>
      </SCStarter>
   </AssignedNodes>
   <AvailableNodes>
      <SCStarter Id="blade45" Port="8000" IP="blade45">
         <SCInstance Port="15200" IP="blade45"/>
         <SCInstance Port="15100" IP="blade45"/>
         <SCInstance Port="15000" IP="blade45"/>
      </SCStarter>
   </AvailableNodes>
   <DesideredPoolSize>3</DesideredPoolSize>
</ResourceManagerParameters>
```

Listing 4: Resource Manager Parameters example

Skeleton$_i$.xml these XML files (one for each template subquery i) contain XML templates that are used when provisioning a new instance or replacing a failed one. These XML files are referred to as Skeleton as they define the actual query but lack part of the information, as the latter is added by *StreamCloud* at deployment time. An example of lacking information is the address to which the input streams of a subquery should connect if the latter is deployed at a *StreamCloud* instance replacing a failed one. This information, computed at runtime by the *StreamCloud Manager*, depends on the particular failed instance, and is added to the subquery skeleton XML just before deploying it.

launch.sh a script that can be used by the user to start (or stop) the query and connect Load Injectors

to the running *StreamCloud* instances.

6.4 Real Time Performance Monitoring Tool

Another component of the IDE that aims at providing support during query execution is the Real Time Performance Monitoring Tool. This tool has been designed to let the user monitor the queries that have been deployed at the SPE instances maintained by *StreamCloud*. For each deployed query, the application allows the user to monitor the performance and the resource consumption of its composing operators. These statistics are computed as average aggregate statistics on a per-parallel operator basis (e.g., the CPU statistic of a parallel-operator is computed as the average CPU consumption of all the instances running the operator). Statistics are retrieved periodically from *StreamCloud* instances and, once aggregated, used to update the information displayed to the user (the frequency with which statistics are retrieved can be defined by the user as a parameter of the application) Given a parallel-operator, the following statistics are provided:

- **Input Stream Rate**: tuples/second consumed by the operator.
- **Output Stream Rate**: tuples/second produced by the operator.
- **Cost**: fraction of the operator processing time over the overall processing time of the operators deployed at the same instance. This statistics measure the percentage of resources consumed by an operator with respect to the resources consumed by all the operators running at the same *StreamCloud* instance.
- **Queue Length**: number of tuples currently queued at the operator input buffer. This statistic is useful as it behavior permits to know when a *StreamCloud* instance is getting close to saturation. When the computation being run does not saturate the available resource, this statistic is usually close to 0 (i.e., all the tuples are immediately processed and no queued). On the other hand, this statistics starts growing fast when the resources need for the computation of the query exceed the available ones.
- **CPU**: average CPU consumption of the SPE instances running the operator.
- **Size**: number of SPE instances running the parallel operator.

As introduced in Section 4.1, *StreamCloud* architecture defines Local Managers (LMs) component for each *StreamCloud* instance. This component is used by the *StreamCloud Manager* to retrieve information from the instance or to modify the running query. Information between each *StreamCloud* instance and the *StreamCloud Manager* is exchanged by means of periodic reports. These

reports include several statistics, such CPU consumption, load of each bucket, statistics about the running operators and so on. Statistics about the operators are maintained by the *Statistics sensor* unit, a sub-unit of the LM component. Figure 6.4 shows a sample parallel-distributed version of the high mobility fraud detection query. In the example, subqueries 1 and 2 have been deployed at subclusters 1 and 2, both composed by 2 *StreamCloud* instances; while subquery 2 has been deployed at subcluster 3, composed by 3 instances. As shown in the figure, each Local Manager includes the *Statistics Sensor* unit (depicted as a box contained in the LM box).

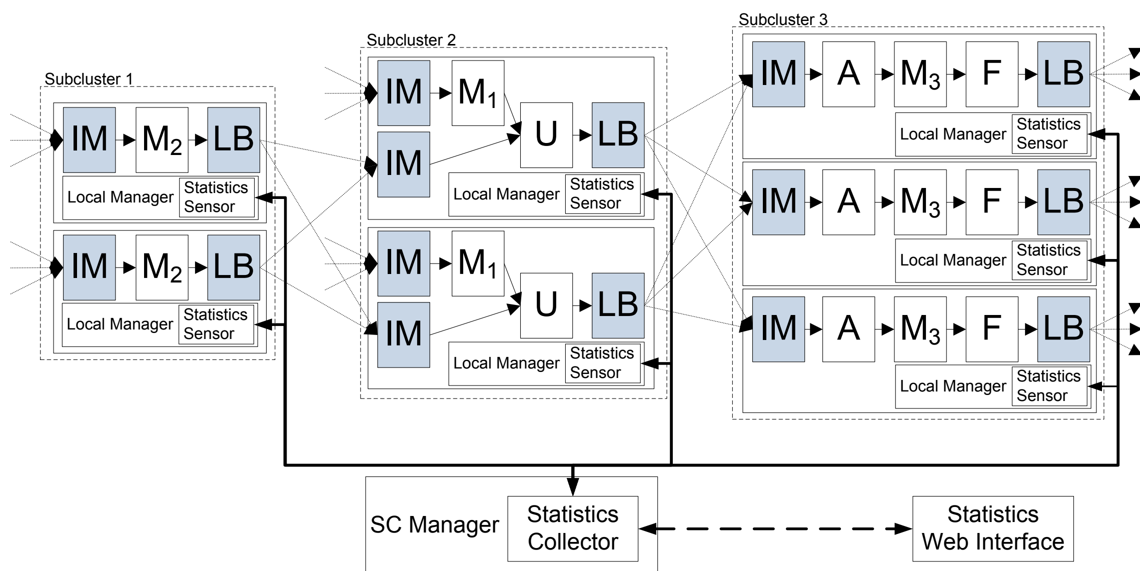


Figure 6.4: SC Statistics Monitor architecture

Periodic reports exchanged by the Statistic Sensor unit of each *StreamCloud* instance are collected by the *StreamCloud Manager*. More precisely, they are collected by the *Statistics Collector* module, a sub-module of the *StreamCloud Manager*, which aggregate statistics on a per-parallel operator basis. Whenever instances are provisioned, the *StreamCloud Manager* updates the Statistics Collector specifying which new statistics reports should be aggregated. Similarly, whenever instances are decommissioned, the *StreamCloud Manager* updates the Statistics Collector specifying which *StreamCloud* instances reports does not have to be aggregated anymore. While parallel operators statistics are being aggregated, they are also sent to the *Statistics Web Application*. The latter provides the user with an interface presenting all the queries maintained by *StreamCloud*. For each parallel-operator, the user can choose which of the presented statistics should be monitored. The application provides the user the possibility of visualizing multiple statistics at the same time.

Figure 6.5 presents a snapshot of the monitoring tool for the aggregate operator of the sample query presented in Section 6.2. Looking at the *Input Stream Rate* statistic, it can be noticed that the load injected to the operator is increasing linearly. During the time period between 10:32:00 and

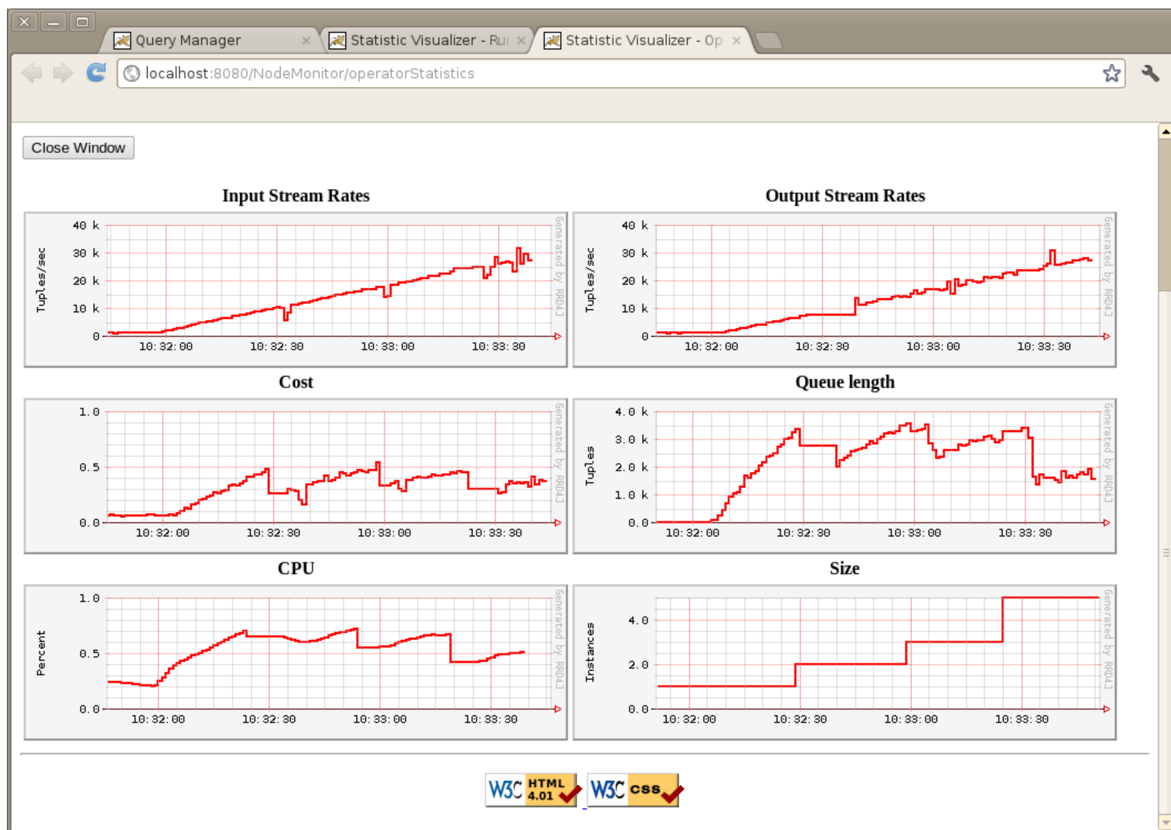


Figure 6.5: snapshot of Statistics Visualizer presenting the statistics of the Aggregate operator

10:33:30 the load moves from approximately 1000 t/s up to 25000 t/s. *Output Stream Rate* grows at the same rate as the *Input Stream Rate*. Looking at the *Size* statistic, it can be noticed that, initially, the aggregate operator has been deployed over a single SPE instance. New *StreamCloud* instances have been dynamically provisioned in order to cope with the incoming load. The operator has moved from 1 to 2 instances at time 10:32:30, from 2 to 3 instances at time 10:33:00 and, finally, from 3 to 5 instances at time 10:33:30. *Cost*, *Queue Length* and *CPU* statistics show a behavior that is similar to the one of the input and output stream rates. The measured values increase due to the growing load injected to the operator. In correspondence with each reconfiguration action, the measured statistics show a sudden drop. This is due to the fact that, as the number of instances increases, each operator processing cost decreases (processing is shared among more machines).

6.5 Distributed Load Injector

The last components of the IDE is the Distributed Load Injector.The distributed load injector is used to read tuples from text (or binary) files and forward them to *StreamCloud* instances. In order to used this component, the user defines a function that converts a text line into a tuple. This allows

the user to maintain tuples in any desired format, as long as it provide a way to translate them to binary objects. As introduced in Section 2.1, a system input stream is composed by tuples having non-decreasing timestamps. If data sources do not have synchronized clocks, the user can use the distributed load injector to timestamp tuples before being forwarded to *StreamCloud* instances. Doing this, the timestamp value of each tuple will be set to the current system timestamp (we suppose the machines used to inject the load belong to the same set of machines used to run *StreamCloud* instances). The Load Injector gives the user the possibility of defining a batch factor to group together tuples sent to *StreamCloud*. This is particularly useful when working with high volume input streams as batching of input tuples decreases the per-tuple serialization / deserialization overhead, leading to higher throughput. When using the Distributed Load Injector to send the input data of a query, multiple instances of the load injector run in parallel. The Distributed Load Injector defines commands to start / stop the injection of tuples and to adjust the injection rate. A centralized controller is connected with all the load injector instances so that commands such *start injection*, *stop injection* or *change injection rate* can be issued at the same time to all of them. Being distributed, the Load Injector allows for the injection of big loads, in the order of hundreds of thousand tuples per second. This high sending rate is achieved partitioning the input file containing the tuples and assigning a portion to each load injector instance.

Part VII

STREAMCLOUD - USE CASES

Chapter 7

StreamCloud - Use Cases

7.1 Introduction

In this chapter, we present application scenarios that have been studied while designing and developing *StreamCloud* SPE as potential target applications where *StreamCloud* has been deployed. Among this scenarios, we focus on fraud detection applications in the field of cellular telephony, fraud detection applications in the field of credit card transactions and in *Security Information and Event Management* (SIEM) systems.For each scenario, we motivate why data streaming, and in particular *StreamCloud*, is a good candidate solution and present some sample queries motivated by the use cases, describing their goal and how they can be implemented in *StreamCloud*. For each implementation of the sample queries, we provide a high level description of the operators that can be used to query the input data and proceed with a detailed description of each query operator.

7.2 Fraud Detection in cellular telephony

Fraud detection applications in the context of cellular telephony are one of the scenarios that motivated research in data streaming due to their need for high processing capacity with low latency constraints. Nowadays, several millions of mobile phone devices are used worldwide to communicate via voice calls, text messages or to exchange information accessing the Internet. As discussed in Chapter 1, the Spanish commission for the Telecommunications market [cmt] counts the number of mobile phones in Spain to exceed the 56 million units. In such scenarios, spotting fraudulent users is a hard task. The reason is that, as each mobile phone can be a potential fraudulent user or a

potential victim, all the traffic must be constantly monitored in order to look for suspicious activity. The complexity also raises from the need of checking for fraudulent activity in a real-time fashion. That is, it is not only important to spot fraudulent activity, but to spot it as soon as possible, as the larger the time it takes to spot a fraudulent activity, the higher the company loss and, even worse, the higher the probability of loosing a client.

StreamCloud fits with the discussed scenario for several reasons. By its nature, the infrastructure upon which mobile cellular telephony applications are built is distributed and the processing of the information generated by mobile phone antennas is highly parallelizable (that is, the overall amount of traffic is huge while the per-mobile phone traffic is small). The presence of distributed sources fits perfectly with *StreamCloud* as the latter has been designed exactly to overcome the limitation of centralized and distributed SPEs performing an end-to-end parallel analysis of the input data. Another reason why *StreamCloud* is a good candidate for managing fraud detection applications in cellular telephony is its high processing capacity and its scalability, that allows for processing of hundred of thousand messages per second. As presented in Section 1.3.1, when running parallel-distributed applications, input data arriving at fluctuating rates might cause under-provisioning or over-provisioning. When setting up a number of machines big enough to processes data at its highest rate (over-provisioning), the main shortcoming is that, most of the time, nodes are not used, incurring in unnecessary costs. The opposite solution, consisting in providing the number of machines needed to process the average system load (under-provisioning) leads to high processing latency during peak loads, resulting in a less effective (or even useless) analysis of the input data. Due to *StreamCloud* dynamic load balancing and elasticity protocols, the resource utilization (in terms of computation nodes) would be constantly adjusted to meet the processing requirements using the appropriate number of machines (i.e. reducing the overall application cost). Finally, *StreamCloud* is a good candidate for cellular telephony fraud detection applications as, thanks to its IDE (see Section 6 for further details), it reduces the final user task to the definition of the *abstract queries* to run and the nodes that can be used, automating all other steps such query parallelization, deployment and runtime management.

We continue this section presenting some sample application used for fraud detection in cellular telephony.

7.2.1 Use Cases

In this section, we present three sample queries used in cellular telephony fraud detection applications. For each query, we present the type of fraud it detects and introduce a possible implementation in terms of data streaming operators. All the sample queries refer to the same schema for the Call Description Record (CDR) input tuples. As presented in the Section 2.1, this information is usually generated by antennas to which mobile phones connect to. The schema is composed by the following 9 fields:

Field Name	Field Type
Caller	*text*
Callee	*text*
Time	*integer*
Duration	*integer*
Price	*double*
Caller_X	*double*
Caller_Y	*double*
Callee_X	*double*
Callee_Y	*double*

Table 7.1: CDR Schema

Fields $Caller$ and $Callee$ refer to the mobile phones making and receiving the call, respectively. Fields $Time$ represents the call start time while $Duration$ refers to its duration, expressed in seconds. Field $Price$ refers to the call price in €, while $Caller_X$, $Caller_Y$ and $Callee_X$, $Callee_Y$ represent the geographic coordinates of the caller and callee number, respectively.

The three sample queries we present in the following are the *Consumption Control* query, used to spot mobile phones whose number of calls exceeds a given threshold, the *Overlapping Calls* query, used to spot mobile phone that appear to maintain more than one call simultaneously and the *Blacklist* query, used to spot phone calls made by mobile phones known to be fraudsters.

Consumption Control Query This query, presented in Figure 7.1, is used to spot mobile phones whose number of calls exceed a given threshold. The goal is to isolate in real time mobile phone numbers that make a suspicious amount of calls for further investigating the presence of possible frauds. In the example, we want to spot mobile phones making 20 (or more) phone calls in a time period of 5 minutes. The query is composed by three operators. The first map operator M is used to transform the input tuples schema removing the fields that are not used by the following operators. The idea is to reduce the computational cost of the query removing unnecessary copies of information that are not used to compute the query result. This rule will be applied also in the following examples. Next to the map operator, the aggregate operator A is used to compute the number of phone calls made by each phone number over a period of time. Finally, the filter operator F is used to forward only mobile phone numbers whose number of calls exceed the given threshold. In the following, we proceed with a detailed description of each operator of the query.

Figure 7.1: Consumption Control Query

The fields of the input tuples needed to compute the query result are the $Caller$ phone number and the $Time$ when the call is started. The map operator M is used to discard all the remaining fields. The operator is defined as:

$$M\{Caller \leftarrow Caller, Time \leftarrow Time\}(I, O_M)$$

The output tuples schema is composed by fields $\langle Caller, Time\rangle$.

Once the unnecessary fields have been removed, the aggregate operator A is used to compute the number of phone calls made by each $Caller$ number over a period of 5 minutes, emitting results every minute. Field $Caller$ is set as the $group-by$ attribute and defines a time-based window with size and advance of 300 and 60 seconds, respectively. The operator is defined as:

$$A\{time, Time, 300, 60, Calls \leftarrow count(), Group-by = Caller\}(O_M, O_A)$$

The output tuples schema is composed by fields $\langle Caller, Time, Calls\rangle$.

Finally, the filter operator F is used to forward only tuples whose number of calls exceeds the given threshold (20 in the example). The operator does not modify its input tuples schema and is defined as:

$$F\{Calls \geq 20\}(O_A, O_F)$$

Overlapping Calls Query This query is used to spot mobile phones that appear to maintain more than one call simultaneously (i.e., mobile phones that could have been cloned). It should be noticed that, for each CDR we must consider both the $Caller$ and $Callee$ as possible cloned numbers. Hence, similarly to the *High Mobility* query presented in Section 2.2, we duplicate each incoming tuple into a pair of tuples, one referring to the $Caller$ number and one referring to the $Callee$ one. To duplicate input tuples, the query defines two initial map operators M_1 and M_2 and the union operator U. Map operators M_1 and M_2 are not only used to extract either the $Caller$ or the $Callee$ phone number from the input tuples but also used to remove the input schema fields that are not used by the following operators. The tuples forwarded by the union operator are processed by the aggregate operator A, used to extract, for each pair of consecutive tuples referring to the same phone number, the end time of the first call and the start time of the second one. Given the assumption the CDR are produced in timestamp order, we can spot two overlapping calls if, given two consecutive phone calls, the start time of the second is less than or equal to the end time of the the first one. In order to spot overlapping calls, the filter operator F compares the fields extracted by the aggregate operator to forward only mobile phone numbers appearing to maintain multiple calls simultaneously. In the following, we proceed with a detailed description of each operator of the query.

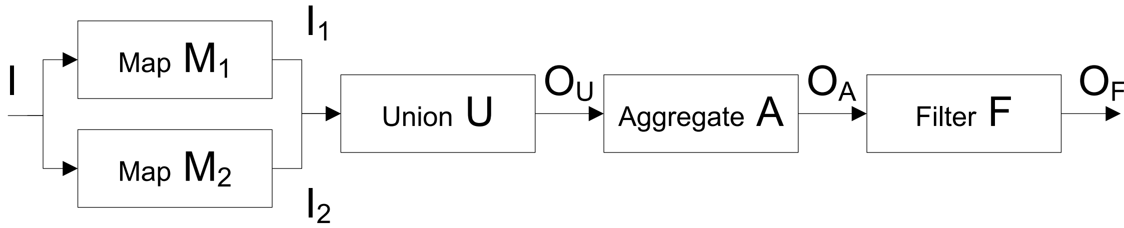


Figure 7.2: Overlapping Calls Query

The map operator M_1 is used to modify the input tuples schema keeping the $Caller$ field (renamed to $Phone$), the $Time$ field (renamed to $Start_Time$) and computing the End_Time field as the sum of the $Time$ and $Duration$. The operator is defines as:

$$M_1\{Phone \leftarrow Caller, Start_Time \leftarrow Time, End_Time \leftarrow Time + Duration,\}(I, I_1)$$

The output tuples schema is composed by fields $\langle Phone, Start_Time, End_Time\rangle$.

Similarly to operator M_1, the map operator M_2 produces tuples composed by fields $Phone$, $Start_Time$ and End_Time. In this case, field $Phone$ is set to be equal to the input tuple $Callee$ number. The operator is defined as:

$$M_2\{Phone \leftarrow Callee, Start_Time \leftarrow Time, End_Time \leftarrow Time + Duration,\}(I, I_2)$$

The union U is used to merge tuples produced by the two previous map operators. It should be noticed that this is possible as the map operators define the same output schema (that is why $Caller$ and $Callee$ fields are being renamed to $Phone$). The operator does not modify its input tuples schema is defined as:

$$U\{\}(I_1, I_2, O_U)$$

Tuples forwarded by the union operator are processed by the aggregate operator A. As the aggregate operator must extract fields for each consecutive pair of tuples, its window is set to be tuple-based and $size$ and $advance$ attributes are set to 2 and 1, respectively. The $group - by$ attribute is set to $Phone$ to match consecutive pairs of calls referring to the same $Phone$ number. Two functions are defined, $first_val(End_Time)$ is used to extract the end timestamp of the first call (function $first_val$ refers to the earliest tuple) while function $last_val(Start_Time)$ is used to extract the start timestamp of the second call (function $last_val$ refers to the latest tuple). The output tuples schema is composed by fields $Phone$, $First_Call_End$ and $Second_Call_Start$. The operator is defined as:

$$A\{tuples, 2, 1, First_Call_End \leftarrow first_val(End_Time),$$
$$Second_Call_Start \leftarrow last_val(Start_Time), Group - by = Phone\}(O_U, O_A)$$

The filter operator F is used to forward $Phone$ numbers appearing to have overlapping phone calls. The condition checked by the operator is $Second_Call_Start \leq First_Call_End$. The operator does not modify its input tuples schema is defined as:

$$F\{Second_Call_Start \leq First_Call_End\}(O_A, O_F)$$

Blacklist Query This query is used to spot mobile phone numbers that are known to be fraudsters. Differently from the two previous queries, the information needed to compute the query results is not carried entirely by the input tuples themselves. More precisely, the information about which mobile phone numbers are known to be fraudulent is provided by means of an external DB. For this reason, the query will mix basic data streaming operators with table operators to retrieve such information (see Section 2.3 for a description of table operators). Similarly to the previous query, the fraudulent mobile phone appearing in each CDR can be either the $Caller$ or the $Callee$ number. Hence, we need to define two initial map operators and a union operator to transform each incoming tuple in a pair of tuples, one carrying the $Caller$ number and one carrying the $Calle$ number. Tuples produced by the union operator are processed by the select operator. As presented in Section 2.3, this operator is used to retrieve tuples from a DB and defines up to three outputs. The first output is used to forward the records contained in the given relation that, for each incoming tuple, match the given SQL WHERE statement. The output tuples of this first output stream share the same schema of the table. The second (optional) output is used to forward input tuples when no record matches the given SQL condition; the tuples schema is the same as the input one. Finally, the third (optional) output is used to match, for each incoming tuple, the number of matching records in the table, the output tuples schema is composed by the fields defined in the SQL `WHERE` clause plus a $Count$ field. In the example, the third output is used to retrieve the number of matching tuples. The final filter operator is used to forward only tuples whose counter is 1, i.e., mobile phones that appear in the blacklist. In the following, we proceed with a detailed description of each operator of the query.

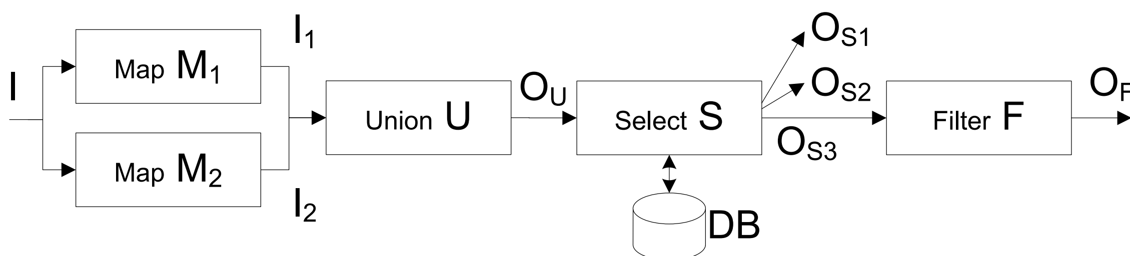


Figure 7.3: Blacklist Query

Map operators M_1 and M_2 are used to modify the input tuples schema keeping the $Caller$ and the $Callee$ field, respectively. They are defined as:

$$M_1\{Phone \leftarrow Caller\}(I, I_1)$$

$$M_2\{Phone \leftarrow Callee\}(I, I_1)$$

Tuples produced by map operators M_1 and M_2, both defining the same schema composed by field $Phone$ are merged by the union operator U. The operator does not modify its input tuples schema and is defined as:

$$U\{\}(I_1, I_2, O_U)$$

Tuples produced by the union operator are processed by the select operator S. In the example, we define a table referred to as DB, whose record schema is defined by the field $Phone$. Each unique record of the list refers to a known fraudulent mobile phone. The operator is used to query the table and extract the number of matching records for each incoming tuple. The third output stream is used to retrieve the number of matching records in the table. The schema of the tuples forwarded to this stream is composed by the field appearing in the SQL `WHERE` clause plus a $Count$ field containing the actual number of matching records. In our example, the $Count$ field will have value 0 if the mobile phone is not in the blacklist or 1 otherwise. The operator is defined as:

$$S\{DB, \text{SELECT * FROM DB WHERE DB.Phone = input.Phone}\}(I, O_{S1}, O_{S2}, O_{S3})$$

The output tuples schema is composed by fields $\langle Phone, Count\rangle$.

The last operator defined by the query is the filter operator F. This operator is used to forward only tuples whose $Counter$ field is 1. The operator does not modify its input tuples schema and is defined as:

$$F\{Counter = 1\}(O_{S3}, O_F)$$

An interesting consideration about this query is related to how it can be parallelized and run by *StreamCloud* providing dynamic load balancing and elasticity. As discussed in 2.3, the parallelization of table operators is not in the scope of the presented work; nevertheless, this query can be parallelized and run by *StreamCloud* with a small effort. This is because the table operator used in the query is only used to retrieve information. That is, as the information is not updated by the query, it is enough to provide access to the information contained in the DB to all the instances at which the select operator could be deployed. It should be noticed that, in order to scale, the access to the information contained in the table should not rely on a centralized DB. One possible solution is to simply provide

a copy of the blacklist DB to each instance running the select operator. Dynamic load balancing can be easily provided by this query as the select operator can be considered as a stateless operator; that is, dynamic load balancing for the select operator can be achieved just changing the routing policy of its input tuples, as far as all the operator instance have access to the same information contained in the blacklist DB (see Section 4.2 for further details on dynamic load balancing for stateless operators).

7.3 Fraud Detection in credit card transactions

In this section, we discuss a different fraud detection use case for credit card transactions. The need for real-time solutions that provide high processing capacities with low latency guarantees to prevent fraud clearly emerges from the huge number of cards used nowadays. As reported by The Nilson Report [Nil], the projection for the year 2012 estimates the number of debit cards holders in the U.S. is approximately of 191 million people, for a total of 530 million debit cards whose estimated purchase volume is 2089 billion dollars. The credit card transaction scenario share similarities with cellular telephony fraud detection applications: in both cases, applications are required to process huge amounts of data with low processing latency and, in both cases, processing of such traffic is highly parallelizable, as the per-credit card (or per-mobile phone) traffic is usually small. As for cellular telephony applications, the credit card transactions rates fluctuate depending on the particular period of time and in correspondence with mass events. This makes *StreamCloud* a good candidate for data streaming processing thanks to its dynamic load balancing and elastic capabilities.

One of the differences between applications in the domain of fraud detection application in cellular telephony and credit card transactions is that, in the second case, queries can be defined not only to detect fraudulent activity immediately after its completion, but to prevent it. The reason is that, when issuing transactions involving credit cards, a series of authentication steps interleave the transaction request from the transaction completion. Hence, if a fraudulent transaction can be spotted before authorizing the transaction itself, the money loss is completely prevented. Nevertheless, this possibility implies a more strict latency bound for tuples processing as authorization of credit card transactions usually defines time thresholds lower than the second.

In the following section, we present two sample applications used to detect frauds related to credit cards transactions.

7.3.1 Use Cases

In this section, we present two sample fraud detection applications related to credit card transactions. Both examples refers to the same input tuples schema, presented in Table 7.2. The schema is composed by four different fields; field $Card$ refers to the card number making the transaction,

field $Time$ represents the transaction start time, field $Price$ refers to the amount of money being transferred while field $Seller_ID$ refers to the entity to which the transaction is being made.

Field Name	Field Type
Card	*text*
Time	*integer*
Price	*double*
Seller_ID	*text*

Table 7.2: Credit Card Fraud Detection Tuple Schema

The two sample queries are presented: the *Improper-Fake Transaction* query, used to spot possibly fraudulent sellers that are issuing multiple transactions for the same credit card in a short period, and the *Restrict Usage* query, used to spot credit card holders whose expenses exceed the credit card usage threshold.

Improper-Fake Query This sample query is used to spot sellers that simulate proper transactions with possibly stolen credit cards. The condition to raise an alarm is that, in a short time period (e.g., 1 hour) at least two distinct credit cards are used multiple times (at least twice in the examples) by the same seller. As an example, this alarm should be raised if the owner of a shop copies credit cards numbers of its clients and simulates several purchases of its items in a short time period.

The query is composed by four operators. The first map operator is used to remove the input tuples fields that are not used by the following query operators. In the example, the fields needed by the query operators are $Card$, $Time$ and $Seller_ID$. Tuples produced by the map operator are then processed by the aggregate operator, who computes the number of transactions on a per-card, per-seller basis. In the example, the time window is set to one hour. The following join operator is used to match tuples sharing the same $Seller_ID$ field but referring to distinct $Card$ numbers. That is, for each pair of input tuples referring to transactions issued by the same seller but for different credit cards, an output tuple is produced. Similarly to the aggregate operator, the time window of the join operator has been set to one hour. Finally, the filter operator is used to output tuples where both credit cards have been involved in more than two transactions each. In the following, we proceed with a detailed description of each operator of the query.

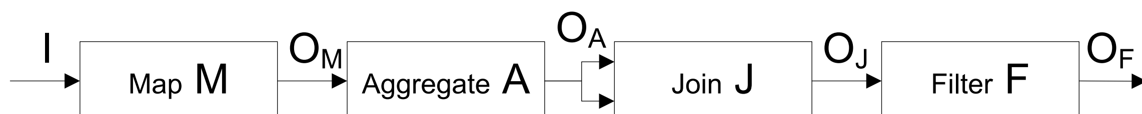


Figure 7.4: Improper Fake Transaction

The map operator M is used to maintain only the input tuples fields used by the following operator. In the example, fields $Card$, $Time$ and $Seller_ID$. The operator is defined as:

$$M\{Card \leftarrow Card, Time \leftarrow Time, Seller_ID \leftarrow, Seller_ID\}(I, O_M)$$

The output tuples schema is composed by fields $\langle Card, Time, Seller_ID \rangle$.

Tuples produced by the map operator are consumed by the aggregate operator A. This operator computes the number of transactions on a per-card, per-seller basis (i.e., the $group - by$ parameter is set to fields $Card$ and $Seller_ID$). The time-based window has size and advance of one hour and ten minutes, respectively. The operator is defined as:

$$A\{time, Time, 3600, 600, Transactions \leftarrow count(),$$
$$Group - by = Card, Seller_ID\}(O_M, O_A)$$

The output tuples schema is composed by fields $\langle Card, Seller_ID, Time, Transactions \rangle$.

The join operator J is used to match tuples produced by the aggregate operator referring to the same seller but to different credit cards. The join operator, that defines two input streams, is feed twice with the aggregate output stream, so that each output tuple can be compared with the other ones. As discussed in Section 2.1.1.0.5, the schema of the tuples produced by the join operator is the concatenation of the left stream and right stream tuples schema. Fields names of the left stream are modified adding the prefix $Left_$ while field names of the right stream are modified adding the prefix $Right_$. These prefix names are added in order to avoid conflicts between fields of the left and right input stream that share the same name. The operator is defined as:

$$J\{left.Seller_ID = right.Seller_ID \wedge left.Card \neq right.Card, time, Time, 3600\}(O_A, O_A, O_J)$$

The output tuples schema is composed by fields $\langle Left_Card$, $Left_Seller_ID$, $Left_Time$, $Left_Transactions$, $Right_Card$, $Right_Seller_ID$, $Right_Time$, $Right_Transactions \rangle$.

Finally, tuples produced by the join operator are consumed by the filter operator F. The condition that must be satisfied in order to generate the alarm is that both fields $Left_Transactions$ and $Right_Transactions$ values are greater than or equal to 2. The operator does not modify its input tuples schema and is defined as:

$$F\{Left_Transactions \geq 2 \wedge Right_Transactions \geq 2\}(O_J, O_F)$$

Restrict Usage Query This last sample query we present, referred to as *Restrict Usage* query, is used to spot credit card whose expenses over a period of time (e.g., 30 days) exceed the credit card limit. This application can have multiple goals, on one side, it could be used to alert users when exceeding their expense limits; on the other side, it might by used to spot stolen cards that are being

used to spend as much as possible in a short time period. Similarly to the *Blacklist* query presented in the previous section, this query needs external information related to the expense limit of each credit card and defines a table operator to retrieve such information. In the example, we suppose the DB containing such information defines a table whose records are composed by fields $Card$ and $Limit$. It should be noticed that, differently from the *Blacklist* DB with respect to mobile phone numbers, each credit card appearing in the query input stream has one (and only one) record in the DB.

The query is composed by 5 operators. The first map operator is used to remove the input tuples schema fields that are not needed by the remaining query operators. In the example, only fields $Card$, $Time$ and $Price$ are kept. The following aggregate operator is used to compute the total expense over the period of time to which the expense limit refers to. Tuples produced by the aggregate operator, containing the total amount of money spent by each credit card must be compared with the respective credit card limit. To do this, output tuples are consumed by the select operator to retrieve the expense limit information from the given DB and then joined by a join operator. The last filter operator is used to forward only tuples referring to credit cards whose expenses exceed their limit. In the following, we provide a detailed description of each query operator.

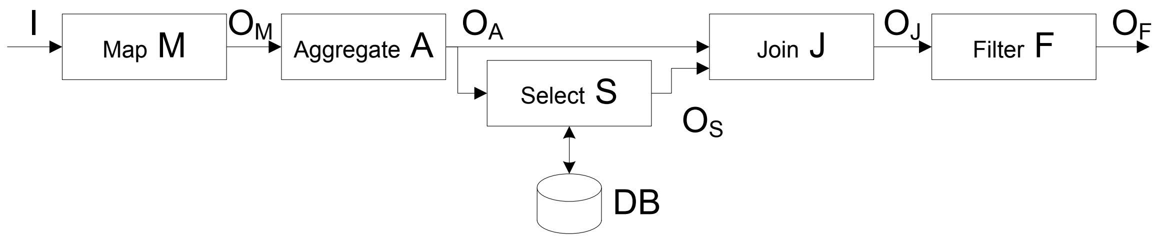


Figure 7.5: Restrict Usage Query

The map operator M modifies the input tuples schema removing the field $Seller_ID$, as the latter is not needed to compute the query result. The operator is defined as:

$$M\{Card \leftarrow Card, Time \leftarrow Time, Price \leftarrow Price\}(I, O_M)$$

The output tuples schema is composed by fields $\langle Card, Time, Price \rangle$

Tuple produced by the map operator are consumed by the aggregate operator A. This operator computes the overall expense of each credit card over a period of time. Hence, its windows are time-based and the $group - by$ parameter is set to field $Card$. In the example, we define a time window of 30 days and an advance of 1 day (expressed in seconds in the operator definition); that is, every day, the overall expense of the previous 30 days while be compared with the credit card expense limit. The operator is defined as:

$$A\{time, Time, 3600 \times 24 \times 30, 3600 \times 24, Expenses \leftarrow sum(Price),$$
$$Group - by = Card\}(O_M, O_A)$$

The output tuples schema is composed by fields $\langle Card, Time, Expenses \rangle$

Tuples produced by the aggregate operator are forwarded to the select operator S to retrieve the expense limit associated to each credit card. In the example, the select operator defines a single output stream, whose tuples share the same schema of the tuples stored in the table (i.e., fields $Card$ and $Limit$). The operator is defined as:

$$S\{DB, \text{SELECT * FROM DB WHERE DB.Card = input.Card}\}(O_A, O_S)$$

The join operator J is used to match tuples produced by the aggregate operator and the select operator. The predicate used to match the tuples defines a single equality between left and right field $Card$. when defining this operator, we must make sure that each tuple produced by the aggregate operator is checked against the one produced by the select operator. These two tuples will reach the join operator at different time: tuples produced by the aggregate will reach the join operator before the ones produced by the select operator. To assure no comparison and no matching is lost, we set the window of the join operator to be time-based and big enough to let both input tuples reach the operator; in the example, we set the window size to 10 seconds. The schema of the tuples produced by the join operator is the concatenation of the schema of the left and right stream tuples. The operator is defined as:

$$J\{left.Card = right.Card, time, Time, 10\}(O_A, O_S, O_J)$$

The output tuples schema is composed by fields $\langle Left_Card$, $Left_Time$, $Left_Expenses$, $Right_Card$, $Right_Limit\rangle$

Finally, tuples produced by the join operator are processed by the filter operator F and forwarded only if the overall expense of each credit card exceeds its limit. The operator does not modify its input tuples and is defined as:

$$F\{Left_Expenses < Right_Limit\}(O_J, O_F)$$

Similarly to the Blacklist query presented in the previous section, although this query defines a table operator and that parallelization, dynamic load balancing and elasticity for table operators is not in the scope of the presented work, such features can be provided for the query with little effort. As the information contained in the external DB is not modified my the select operator but simply read and, as the select operator can be seen as a stateless operator, parallelization, dynamic load balancing and elasticity can be provided as far as each instance at which the operator could possibly be deployed has access to the information contained in the DB. As stated before, the access should not rely on a centralized DB in order to scale.

7.4 Security Information and Event Management Systems

In the section, we introduce Security Information and Event Management (SIEM) systems. These solutions are designed to process security alerts generated by multiple monitoring devices (or applications) and look for specific patterns or series of events, generating alarms if necessary. As an example, a SIEM system can be used to check for intrusion detection analyzing the log files of a server in order to generate an alarm if multiple attempts to access the machine with a wrong password are followed by a successful login in a short time period. The heart of SIEM systems is the Complex Event Processing (CEP) engine, equivalent to a data streaming engine, that must provide the capability to process, aggregate and correlate real-time input data with high capacity and low processing latencies. In SIEMs, conditions that must be checked are expressed in terms of *directives*, each expressed as a tree of *rules*. Each *rule* is expressed as a predicate over the input data. With respect to the previous example, the directive could be expressed as a tree with two rules: the first rule (root node) looking for 100 consecutive attempts to login withing a time period of 10 minutes while the second rule (leaf node) looking for a successful login within 1 minute from the previous rule.

Nowadays, SIEM solutions rely on a centralized CEP system to process the information by the infrastructure being monitored. Our study focus on how to use *StreamCloud* as the base for a parallel-distributed SIEM system. With respect to this goal, the challenge relies in how to make such switch in the SIEM underlying CEP transparent to the final user; i.e., how to automatically translate traditional CEP directives into data streaming queries. As discussed in Chapter 6, it is important to ease as much as possible the interaction between a user and the system in charge of processing the input data. Hence, with respect to SIEM systems, the challenge not only relies in how to define a query whose results are equivalent to its CEP directive counterpart, but also in how to do it in terms of template queries that can be used to automatize the translation process.

In the following section, we first present how SIEM directives are defined, we discuss how they can be converted into data streaming queries and finally, present a sample SIEM directive and its data streaming query counterpart.

7.4.1 SIEM directives

SIEM directives are defined as a collection of non-empty trees of *rules*, each rule defined as a predicate over one or multiple input events. The series of events specified in a *directive* is analyzed from the root node to the leaves ones. Each time the predicate of the rule is satisfied (we say the rule *fires*), the directive starts checking the predicate of the rule child nodes. The definition of directive by means of tree structures allows for the definition of OR conditions inside a directive by means of sibling nodes. For each directive, an output message is forwarded to the final user each time a rule fires. As multiple messages are generated for each directive, each message defines a *Reliability* factor

that is used to evaluate the relevance of the message itself.

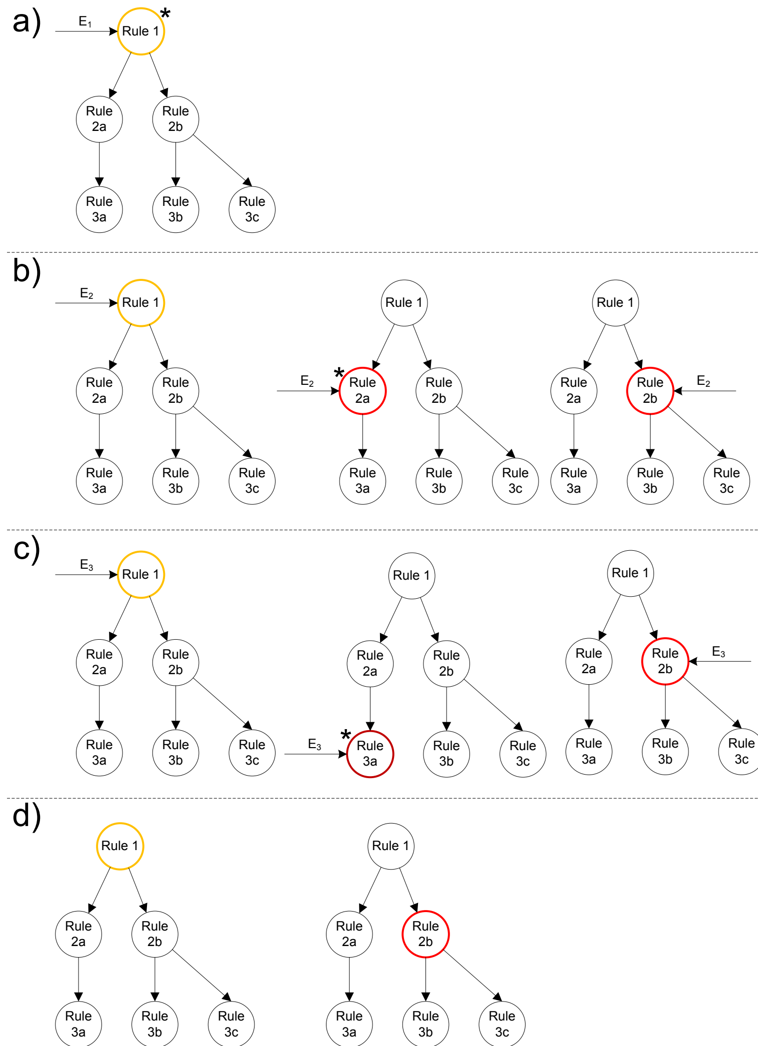


Figure 7.6: Rules firing and directives cloning example

It should be noticed that, at any point in time, multiple instances of the same directive can be active. As an example, consider the sample directive presented in the previous section, that defines a directive composed by two rules, the first looking for 100 consecutive failed login attempts within a time period of 10 minutes and the second rule looking for a successful login within 1 minute of the previous failed login attempts. Suppose this directive is used to monitor a network containing multiple server hosts; it is easy to see that each server should be monitored on its own. Furthermore, the same server can be monitored by multiple instances of the same directive. As an example, suppose 100 unsuccessful login attempts have been made and the directive is now waiting for the following successful login; it easy to see that the directive should still check for unsuccessful login events if the user password has not yet been found. In order to define how directives are instantiated, we

can imagine that each directive can invoke a `clone` and `remove` method. The `clone` function is invoked each time the firing rule is the root node or if it has multiple child nodes. For each firing node that invokes the `clone` functions, has many directive clones as child nodes are instantiated. The `remove` function is invoked each time a leaf node fires (as no further conditions must be checked for the given directive).

Figure 7.6.a presents a sample directive composed by 6 rules. The root rule 1 defines two sibling child rules $2a$ and $2b$. Rule $2a$ defines a single child rule $3a$ while rule $2b$ defines two sibling rules $3b$ and $3c$. In the example, the root node is the currently active rule (marked by the yellow line). We suppose an input event E_1 satisfies rule 1 condition (marked with $*$ in the figure). That is, rule 1 fires. After rule 1 has fired, two clones of the directive are instantiated for the given directive, one for each child of rule 1. As shown in Figure 7.6.b, at this point, 3 directives are active, the first processes input events checking for rule 1, while the two clones check for rules $2a$ and $2b$ respectively. In the example, we consider an input event E_2 that causes rule $2a$ to fire. After rule $2a$ has fired, the resulting configuration is shown in Figure 7.6.c. At this point, the first cloned directive has changed its active rule to $3a$. Consider now an input event E_3 that satisfies the condition of rule $3a$. As shown in Figure 7.6.d, the first cloned directive has been removed as rule $3a$ is a leaf node.

In the following, we provide a detailed description about how rules are defined. We refer to the OSSIM directives semantics [ali]. Each rule is defined by the following parameters [1]:

Name	Type
Plugin_Id	*int*
Plugin_Sid	*int*
From	*IP,subnet,ANY*
To	*IP,subnet,ANY*
Port_From	*0-65636,ANY*
Port_To	*0-65636,ANY*
Occurrence	*int*
Time_Out	*int*
Reliability	*int*

Table 7.3: OSSIM rule parameters

Parameters $Plugin_Id$ and $Plugin_Sid$ are used to categorize the sensors producing input events; parameter $Plugin_Id$ is used to specify the sensor type while parameter $Plugin_Sid$ is used to specify the message type. Both fields are expressed as integer number, each number uniquely idenfies a sensor type or a message type. For each event, the rule allows for defining which are the required source and destination addresses (in terms of IP and port number) defining parameters $From, Port_From$ and $To, Port_To$. Source and destination IP addresses can be specified as a specific IP address or as subnets. If any of the two addresses is not required by the rule, parameters

[1]OSSIM directive defines further parameters, for the ease of the explanation, we refer only to this set of parameters

$From$ or To value can be set to ANY. Similarly, port numbers can be set to a specific number or ANY, otherwise. The OSSIM language defines a referencing system so that parameters of a rule can refer to the events that satisfied the parent rule. With respect to the sample directive used to spot a series of failing authentications followed by a valid one that allows the user to access a specific server, it is easy to see that the failed authentications and the valid one should refer to same server. In OSSIM, this is achieved preceding the value of the attribute by a number specifying the parent position of the referenced rule. As en example, "1:From" refers to the same source IP address of the parent rule. Each rule permits to define how many events of the same type should be received before firing by means of parameter $Occurrence$. If $Occurrence > 1$, parameter $Time_Out$ specifies which is the maximum interleaving time between the first and last event for the rule in order to fire. Finally, parameter $Reliability$ is used to specify the reliability associated to each rule; it can be expressed as the increase with respect to the previous rule or as an absolute value.

Listing 5 presents a sample OSSIM directive included in the set of directives available with the standard installation package. The directive is used to spot a possible threat using Server Message block (SMB) protocol. SMB protocol is used to share files and resources among hosts of a network. The directive looks for a new file shared among nodes that seems to be harmful.

```
<directive>
        <rule Plugin_Sid ="537,2465,2466" Plugin_Id="1001"
                Port_To="ANY" Port_From="ANY" To="ANY" From="ANY" Occurrence="1" Reliability="2">
                <rules>
                        <rule
                                Plugin_Sid ="2009033,2009034,2009035"
                                Plugin_Id="1001" Port_To="ANY" Port_From="ANY" To="1:DST_IP"
                                From="1:SRC_IP" Occurrence="2" Reliability="+1" Time_Out="10"/>
                </rules>
        </rule>
</directive>
```

Listing 5: Sample OSSIM directive

The directive is composed by two rules. The root rule fires is a single event ($occurence = 1$) generated by sensor 1001 (SNORT sensor) and of type 537, 2465 or 2466 (these IDs refer to a new file being shared) is sent between any two hosts (all parameters $From$, To, $Port_From$, $Port_To$ are set to ANY). When firing, the rule defines $Reliability$ to be equal to 2. The child rule fires if two events ($occurence = 2$), received in a time frame not exceeding 10 seconds ($timeout = 10$), generated by sensor 1001 (SNORT sensor) and of type 2009033,2009024 or 2009035 (these messages refer to a potentially harmful file) is exchanged between the same source and destination IP addresses seen in the previous rule ($from = 1 : SRC_IP$, $to = 1 : DST_IP$) using any port number (parameters $Port_From$, $Port_To$ are both set to ANY). When firing, the $Reliability$ associated to the rule is increased to 3 ($Reliability = +1$).

7.4.2 Directives translation

In this section, we discuss how OSSIM directives can be automatically converted to data streaming queries. A first challenge relies in how to cope with OSSIM cloning and removal of directives. In order to be scalable, it is not feasible to have a system where copies of the same query are continuously added and removed. Rather, we should have a single continuous query that processes all the input tuples producing the desired results. An important consideration is that, in an OSSIM directive, each rule is able to receive input tuples, to share information with its child rules (e.g., when looking for a given source IP that appeared in a previous rule) and to output tuples to the final user. These functionality should be addressed by the query template we define to translate each rule. That is, it must define an input stream to read the system input tuples, an input and output stream to receive and forwarded tuples from the previous and to the following rules and an output stream to forward tuples to the final user. Figure 7.7.a presents the generic structure of an OSSIM rule while Figure 7.7.b presents the generic structure of its query counterpart. It can be noticed that the root rule will only receive tuples from the input stream while the leaf rules will just forward tuples to the final user.

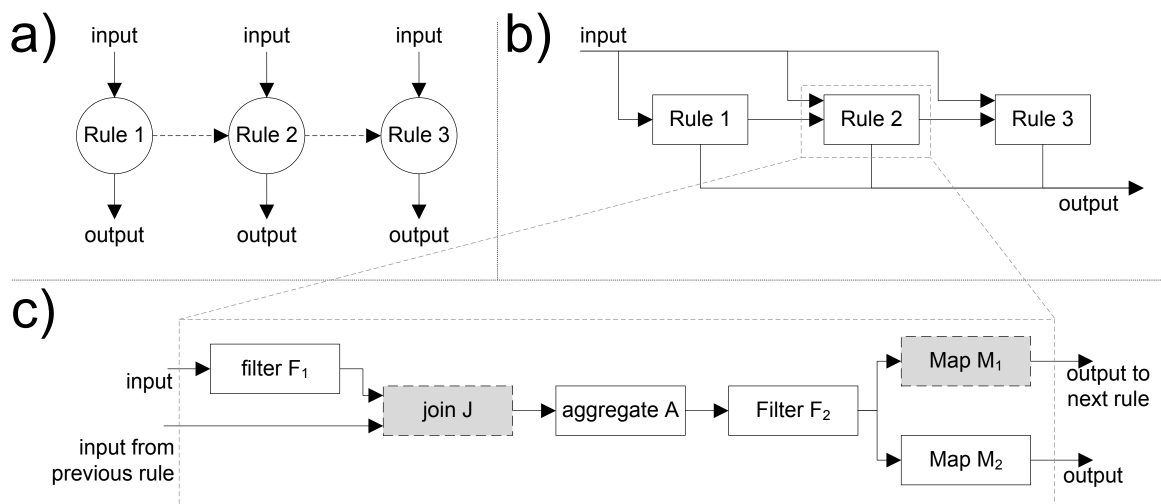


Figure 7.7: OSSIM directive translation guidelines

The last challenge to be addressed in order to convert OSSIM rules to data streaming queries, is how to implement each rule by means of data streaming operators. Two possible ways of achieving this goal are possible: on one hand, we could define a data streaming user-defined operator (see Section A.2 for operators extensibility) that exactly reproduces the semantic of an OSSIM rule; on the other hand, we can define a template query only composed by basic data streaming operators. We adopt this second strategy as it eases the parallelization of the resulting query. That is, as the parallelization of each basic data streaming operator is provided by the Parallel Query Compiler (Section 6.3), the converted query can be directly used by *StreamCloud*. Figure 7.7.c presents the

data streaming operators composing the template query for OSSIM rules. Consider a stream of input tuples whose schema resembles the parameters of the OSSIM rules. The first filter operator F_1 is used to forward only input tuples which $Plugin_Id$ and $Plugin_Sid$ are the ones specified by the OSSIM rule. The join operator J is used to forward both input streams (i.e., tuples coming from the input stream and the ones forwarded by the previous OSSIM rule) to the following operators. In principle, a union operator could have been used to merge two input streams (as far as they share the same schema), nevertheless, we adopt the join operator as it permits to forward tuples to the rule only if the previous rule has fired (i.e., only if there is a tuple on both input streams that can be used to match input tuples). In the template, the join operator is drawn with a gray shadow to indicate it is optional. Specifically, it will not be defined for the directive root rule. The aggregate operator A is used to wait for the number of events specified by the $Occurrence$ specifying a tuple based window with size and advance of $Occurrence$ and 1, respectively. The timestamp of the first and the last tuple can be extracted by the operator in order to check if their distance does not exceed the given $Time_Out$ parameter. Using the $group - by$ attribute, the aggregate operator can define separate windows for each possible pair of source and destination IP and port pairs. The filter operator F_2 is used to check if the time distance between the first and the last event is less than or equal to the $Time_Out$ parameter. Finally, two separate map operators M_1 and M_2 are used to forward output tuples to the next rule and the system output, respectively. We chose to use map operators so that the schema of tuples forwarded by the filter operator can be modified as needed in order to be forwarded to the next rule or the system output.

The tool used to convert OSSIM rules into data streaming queries has been integrated as a tool of the *StreamCloud* IDE presented in Chapter 6.

7.4.3 Directive translation example

In this section, we study how the sample OSSIM directive presented in Section 7.4.1 is converted to a data streaming continuous query.

The input tuples schema considered in the example is composed by the fields presented in Table 7.4.

Field Name	Field Type
Pid	*int*
Sid	*int*
IP_A	*text*
IP_B	*text*
P_A	*int*
P_B	*int*
$Time$	*int*

Table 7.4: OSSIM input tuples schema

Fields Pid and Sid represent the $Plugin_Id$ and $Plugin_Sid$ associated to the message generated by any monitoring device. Fields IP_A and IP_B represent the IP addresses of the source and destination hosts, while fields P_A and P_B represent their correspondent port numbers. Finally, field $Time$ carries the information related to when the report has been generated.

The output tuples schema considered in the example is composed by the fields presented in Table 7.5.

Field Name	Field Type
IP_A	*text*
IP_B	*text*
P_A	*int*
P_B	*int*
$Rule$	*int*
$Reliability$	*int*
$Time$	*int*

Table 7.5: OSSIM output tuples schema

Fields IP_A, IP_B, P_A and P_B represent the IP addresses and port numbers of the source and destination hosts for which the alarm has been raised. Field $Rule$ is used to specify the rule that generates the output tuple. In the example, the OSSIM directive is composed by two rules, hence, field $Rule$ will be set 1 when the output tuple is generated by the first rule while it will be set to 2 when generated by the second one. Finally, field $Reliability$ represents the reliability associated to the generated alarm.

Figure 7.8 presents the resulting continuous query obtained translating the sample OSSIM directive. In the following, we proceed with a detailed description of the operators of each rule.

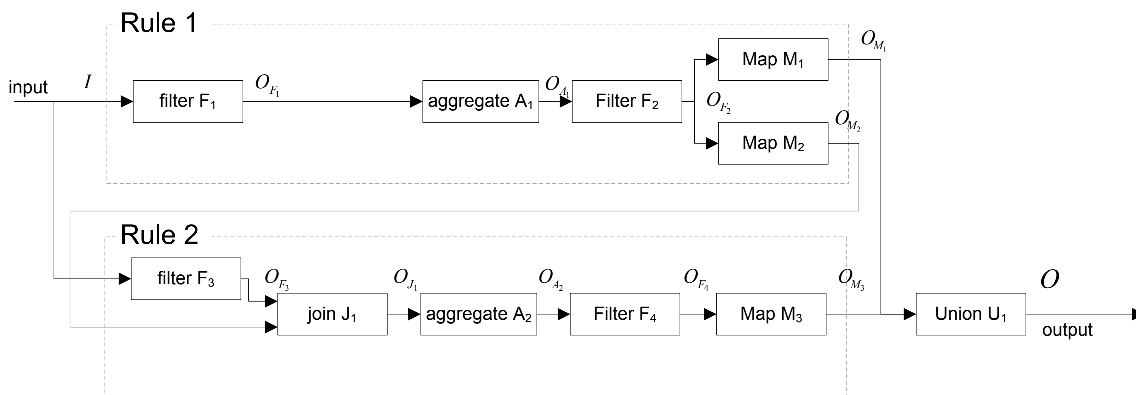


Figure 7.8: OSSIM Directive translation to continuous query

Rule 1 The first rule of the OSSIM directive is converted to the template presented in 7.4.2, except for the join operator, that is not necessary as the root rule only defines one input stream. The filter

operator F_1 is used to forward to the operators of the rule 1 only the tuple whose $Plugin_Id$ and $Plugin_Sid$ values are the ones defined in the OSSIM rule. The schema of the operator output tuples is the same as the input one. Operator F_1 is defined as:

$$F_1\{Pid = 1001 \text{ AND } (Sid = 537 \text{ OR } Sid = 2465 \text{ OR } Sid = 2466)\}(I, O_{F_1})$$

When only tuples referring to the desired events have been forwarded by operator F_1, the aggregate operator A_1 is used to wait for the right number of events (specified by the $Occurrences$ parameter). The first rule of the OSSIM directive specifies parameter $Occurrence$ as 1; hence, both window size and advance parameters are set to 1. In order to ensure the time distance between the first and the last event are within the specified $Time_Out$, function `first_val` and `last_val` are used to extract the timestamp of the earliest and latest tuple, respectively. Finally, in order to define separate windows for each possible pair of hosts (both in terms of IP and port numbers), $group - by$ parameter is set to IP_A, IP_B, P_A, P_B. The output tuples schema is composed by fields $\langle IP_A, IP_B, P_A, P_B, First_Time, Last_Time\rangle$. Operator A_1 is defined as:

$$A_1\{tuples, 1, 1, First_Time \leftarrow first_val(Time), Last_Time \leftarrow last_val(Time),$$
$$Group - by = IP_A, IP_B, P_A, P_B\}(O_{F_1}, O_{A_1})$$

Tuples produced by operator A_1 are filtered by the filter operator F_2. In this case, as the occurrence of the OSSIM rule is 1 (i.e., $First_Time$ and $Last_Time$ are equal), the filter condition is always $true$. It should be noticed that, for the specific directive rule taken into account, both operators A_1 and F_2 are not strictly mandatory, the same results can be achieved simply forwarding tuples generated by the filter operator F_1 to the final user and to the following rule. Nevertheless, for the ease of the example, we keep all the operators defined in the query template and do not discuss rule-specific optimizations. Operator F_2 is defined as:

$$F_2\{true\}(O_{A_1}, O_{F_2})$$

Tuples forwarded by the filter operator F_2 are sent to the final user and the following rule using map operators M_1 and M_2. Both operators modify their input tuples schema keeping only the fields related to the source and destination IP addresses and port numbers and field $Time$. Operator M_1, in charge of outputting tuples to the final user, modifies the tuples schema adding fields $Rule$ and $Reliability$. Field $Rule$ is set to 1, in order for the final user to identify the rule that generated the output. Field $Reliability$ value is set to 2, as defined by the OSSIM rule. The operator is defined as:

$$M_1\{IP_A \leftarrow IP_A, IP_B \leftarrow IP_B, P_A \leftarrow P_A, P_B \leftarrow P_B, Time \leftarrow Time,$$
$$Rule \leftarrow 1, Reliability \leftarrow 2\}(O_{F_2}, O_{M_1})$$

The output tuples schema is composed by fields $\langle IP_A, IP_B, P_A, P_B, Time, Rule, Reliability\rangle$

Map operator M_2 defines the same output schema of operator M_1 except for the $Rule$ field, that is not used by the operators of the following rule. The operator is defined as:

$$M_2\{IP_A \leftarrow IP_A, IP_B \leftarrow IP_B, P_A \leftarrow P_A, P_B \leftarrow P_B, Time \leftarrow Time,$$
$$Reliability \leftarrow 2\}(O_{F_2}, O_{M_2})$$

Rule 2 As shown in Figure 7.8, the second OSSIM rule is converted to the query template, except for one of the final map operators, as this leaf rule has no following rule to send output tuples to. In this case, the join operator is added to the query as the latter must process both tuples from the input stream and the previous rule.

Similarly to operator F_1, filter operator F_3 is used to foward only tuples whose $Plugin_Id$ and $Pluing_Sid$ values are specified in the OSSIM rule. The schema of the filter output tuples is the same as the input tuples one. The operator is defined as:

$$F_3\{Pid = 1001 \text{ AND } (Sid = 2009033 \text{ OR } Sid = 2009034 \text{ OR } Sid = 2009035)\}(I, O_{F_3})$$

Once tuples are forwarded by operator F_3, the join operator J_1 is used to match tuples from the input stream and the previous rule. The join operator is used to ensure that: (1) matched tuples share the same source and destination IP addresses, as specified by the OSSIM rule, and (2) to forward to the following operator only input tuples having timestamp greater than or equal to the timestamp of the tuple generated by the previous rule. That is, as far as the first rule has not outputted a tuple, input tuple are not being processed by the second rule. The schema of the output tuples is the union of the left and right input streams tuples schema, and is composed by fields $\langle Left_Pid$, $Left_Sid$, $Left_IP_A$, $Left_IP_B$, $Left_P_A$, $Left_P_B$,, $Left_Time$, $Right_IP_A$, $Right_IP_B$, $Right_P_A$, $Right_P_B$, $Right_Time$, $Right_Reliability\rangle$ Operator J_1 is defined as:

$$J_1\{left.IP_A = right.IP_A \text{ AND } left.IP_B = right.IP_B \text{ AND } left.Time \geq right.Time,$$
$$time, left.Time, 3600\}(O_{F_3}, O_{M_2}, O_{J_1})$$

Once events have been matched by operator J_1, the aggregate operator A_2 is used to wait for the number of events specified by the $Occurrence$ parameter of the OSSIM rule. The operator is similar to the aggregate operator A_1, except for the window size, that is set in this case to 2. The schema of the operator output tuples is $\langle Left_IP_A, Left_IP_B, Left_P_A, Left_P_B, First_Time, Last_Time\rangle$. The operator is defined as:

$$A_2\{tuples, 2, 1, First_Time \leftarrow first_val(Left_Time), Last_Time \leftarrow last_val(Left_Time),$$
$$Group - by = Left_IP_A, Left_IP_B, Left_P_A, Left_P_B\}(O_{J_1}, O_{A_2})$$

Tuples generated by operator A_2 are filtered by the filter operator F_4. The OSSIM rule specifies a $Time_Out$ parameter of 10 seconds as maximum interleaving time between the first and last event.

Hence, the predicate of the filter operator will check for the difference between fields $Last_Time$ and $First_Time$ to be less than or equal to 10. Operator F_4 is defined as:

$$F_4\{Last_Timestamp - First_Timestamp \leq 10\}(O_{A_2}, O_{F_4})$$

Tuples forwarded by the operator F_4 are modified by the map operator M_3 in order to be sent to the final user. The output tuples schema is the one defined by the map operator M_1, except for the values set to fields $Rule$ (set to 2) and $Reliability$ (set to 3). The operator is defined as:

$$M_3\{IP_A \leftarrow Right_IP_A, IP_B \leftarrow Right_IP_B, P_A \leftarrow Right_P_A, P_B \leftarrow Right_P_B,$$
$$Rule \leftarrow 1, Reliability \leftarrow 3\}(O_{F_4}, O_{M_3})$$

The last union operator is used to merge output tuples forwarded by the two previous rules to the final user. This is possible as both map operator M_1 and M_3 define the same output schema. The operator is defined as:

$$U_1\{\}(O_{M_1}, O_{M_3}, O)$$

Part VIII

RELATED WORK

Chapter 8

Related Work

8.1 Introduction

In this chapter, we present the existing relevant work that motivated our research in the field of data streaming and in the design and development of *StreamCloud*, the parallel-distributed SPE presented in this thesis. We first introduce the pioneer SPEs that have contributed to data streaming research and to the development of nowadays commercial products. We continue with a short introduction of some of the state of the art SPEs. Subsequently, we investigate existing work related to operators scheduling, load sampling and shedding techniques, parallelization of data streaming operators, load balancing, elasticity and fault tolerance techniques.

8.2 Pioneer SPEs

Several SPEs have been studied (approximately) since 2000. Among the existing ones, we think 5 SPEs are of particular interest because of the evolution they introduced with respect to the previous existing work, namely: *Borealis*, *STREAM*, *TelegraphCQ*, *NiagaraCQ* and *Cougar*. We present each of them separately, introducing their most innovative contributions.

8.2.1 The Borealis project

The Borealis project [AAB+05b] [ABC+05] defines one of the first distributed SPEs. It has been developed by Brandeis University, Brown University and MIT and is an evolution of two previously

existing SPEs developed by the same universities: Aurora [CCC+02] [ACC+03] [BBC+04] and Medusa [CBB+03] [SZS+03]. We first introduce Aurora and Medusa separately and, subsequently, we discuss how Borealis merges and improves them.

The Aurora project constitutes one of the firsts centralized SPEs (that is, queries are entirely deployed on a single instance in charge of processing all the input tuples and producing all the output results). The project focused on the weaknesses of the existing DB based solutions with respect to emerging data streaming applications. As presented in Chapter 1, SPEs were conceived to provide higher processing capacity with low processing latency to applications demanding for near real-time analysis of flows. Among others, interesting aspects that have been studied in the context of the Aurora project are related to how a data streaming based solution should cope with respect to imprecise and missing data, real-time requirements and graceful results degradation under the presence of peaks in the system input load. Aurora introduces a boxes and arrows model to define queries and provides a set of around 10 operators. A *Scheduler* is provided to decide which operator should run and how many tuples it should process at any point in time. The scheduler component cooperates with a *QoS Monitor* component and a *Load Shedder* component. The QoS Monitor tries to maximize continuously the quality of the outputs produced by each query. Such results depend on aspects such response time (i.e., the time it takes to create output tuples), tuple drops (i.e., how dropped tuples affect the output quality) and values produced (i.e., whether important values are being sent to the user application). Load shedding is applied any time the resource of the instance cannot cope with the incoming load. Rules to decide which information to discard are taken minimizing the output quality degradation. Operator scheduling policies and load shedding techniques are discusses in the following sections.

After developing Aurora, two distributed SPEs have been developed starting from it: (1) Aurora* allows to create distributed networks of Aurora instances within the same administrative domain. As nodes belongs to the same domain, there are no operational restrictions with respect to how queries operators can be distributed among them. On the other hand, Medusa has been designed to connect autonomous *participants*. Each participant represents a set of computing devices of an entity that can contribute to running queries in several ways: (1) providing data sources, (2) providing computational resources that can be used to deploy queries operators and (3) consuming queries results.

Aurora* introduces innovative aspects such as two load sharing mechanisms (namely, *box sliding* and *box splitting*) and a *high availability* protocol to face the increasing probability of failures inherent to distributed SPEs. These aspects will be presented in the following sections where we compare our parallelization, load balancing, elasticity and fault tolerance protocols with previous existing solutions.

The Borealis project represents the evolution of Aurora* and Medusa. It addresses new aspects such as dynamic revision of query results and dynamic query modification.

Dynamic revision of query results gives the user the possibility to receive corrections to results previously produced by the query. As an example, a correction can be produced in order to replace an output value computed only over part of the input streams (e.g., in order to avoid blocking the query processing). Such revision are based on particular *revision tuples* that, once processed by operators instructed to wait for them, cause the emission of updated output values. Dynamic query modification gives the user the possibility of changing, at execution time, the parameters that define how data should be processed and which data should be forwarded to the application (e.g., a filter condition that specifies which tuples are forwarded depending on a specific attribute).

The Borealis project is no longer an active research project, it has been shut down in 2008.

As discussed in Section 1.3, the Borealis SPE has been used as the underlying SPE of *StreamCloud*. We provide further details about Borealis SPE in Appendix A.

8.2.2 STREAM

Similarly to Borealis, STREAM (the STandford stREam datA Manager) [ABB+04] [ABW06] has been designed and developed to overcome the limitations of previous DB based solutions. As discussed in Chapter 1, one of the requirements of data streaming solutions is to provide a “bridge" from DB solutions to data streaming solutions providing an intuitive and easy way to define queries used to process the input data. With respect to this need, STREAM introduces a declarative language to specify queries: CQL (Continuous Query Language). CQL is based on SQL for its relational query language, while it windows specification language is derived from SQL-99. When defining a query, some of the constructs of the CQL query language allow for extracting windows of tuples from a stream and manage them as relations. These relations are transformed into other relations and, finally, transformed back to stream with relation-to-stream operators. That is, streams are first transformed to relations; data analysis is performed applying traditional SQL like query over these relations and final results are forwarded to the final user converting relations back to streams. An interesting feature of STREAM is that, contrary to Aurora, it allows for sharing of windows of data. That is, distinct operators fed by the same stream do not only share their input buffer, but they can also share the same window. Suppose two distinct aggregate operators (defining the same window semantic) are used to compute the max and min values of one attribute of the stream, the system will maintain a single window with indexed tuples while each aggregate operator will maintain the index of the tuple carrying the max and min value of the given stream attribute, respectively.

The scheduling protocol defined in STREAM (*Chain Scheduling*) takes primarily into account main memory consumptions as the criteria to decide which operator and how many tuples it should process. Query execution plans are built defining chains of consecutive operators that effectively reduce the runtime memory. The idea is to prioritize, if possible, operators that consume a lot of tuples while producing few. We discuss operator scheduling protocol in the following sections.

Research directions for STREAM are similar to the ones covered by Aurora, namely *scheduling*, *graceful degradation* of system outputs under peaks in the incoming load, *distributed stream processing* and *high availability*.

The STREAM project is no longer an active research project, it has been shut down in 2006.

STREAM people, together with the developers of Aurora, have introduced LinearRoad [JAA+06] [ACG+04] the first benchmark for SPEs. This benchmark is designed to evaluate the performance of an SPE in terms of the scale out at which the system if able to provide query answers within some given time constraints. The benchmark simulates a variable tolling system for an imaginary highway system. Position reports generated by the vehicles traveling in the highway are used to generate traffic and accident statistics later used to determine tolls charges for each highway segment. The performance of each SPE is measured in number of highways the system can process complying with the required QoS.

8.2.3 TelegraphCQ

TelegraphCQ [CCD+03] [Des04] has been designed and developed to address the same limitation of DB based solutions that motivated Borealis and STREAM (the inadequateness of existing DB based solution to cope with data streaming applications requirements). Nevertheless, TelegraphCQ presents a significantly different architecture from previous SPEs.

TelegraphCQ architecture is based on individual modules that communicate using the *Fjord* API. Modules are generic units that produce and consume tuples. Different types of modules exists, Ingress and Caching modules, that provide tuples to the system from external applications; Adaptive Routing modules, that, given some route criteria, distribute tuples to other modules and Query Processing modules, that transform incoming tuples to output tuples. Modules communicate using the Fjord API; the latter provides the glue between modules and allows for *push-based* and *pull-based* communication. TelegraphCQ does not define a specific scheduling unit that is also responsible for the quality of the system output. Adaptive Routing modules are in charge of measuring the system output quality and adapt their routing policies accordingly.

TelegraphCQ architecture might seem to share several properties with the boxes and arrows model employed by the Borealis SPE. Nevertheless, two important factors must be taken into account to distinguish between the two approaches. First, one of the goals in Borealis is to provide not generic boxes the user can specialize but rather provide a set of predefined operators that allow for rich analysis of data streams. Moreover, an important aspect of TelegraphCQ is that, differently from other SPEs, it has not been developed from scratch. On the contrary, it is based on PostgreSQL [pos]. TelegraphCQ exploits PostgreSQL facilities to store and manipulate data. This way, new code is only added to provide the data streaming functionalities PostgreSQL does not offer.

8.2.4 NiagaraCQ

NiagaraCQ [NDM+01] [CDTW00] is one of the earliest research projects that tries to improve existing DB-based technology to overcome its limitation with respect to data streaming applications. It is based on the Niagara project, a data management system developed at the Universtiy of Wisconsin and at the Oregan Graduate Institute. Initially, this research project does not propose a new data processing model to overcome DB-based solutions limitations. It rather tries to achieve a higher scalability for a distributed database system. The project focuses on queries over distributed XML data, defined using a query language like XML-QL [DFF+98].

NiagaraCQ proposes a novel approach to attain a scalable system with respect to the number of continuous queries the system is maintaining at the same time. The idea is to group queries so that overlapping computation is shared among them. The challenge in grouping continuous query relies in how to add and remove queries at runtime exploiting already defined groups of queries. NiagaraCQ defines a dynamic re-grouping protocol to address this challenge. As for the STREAM SPE, where different operators might share the same window structure, NiagaraCQ groups queries in order to minimize the processing overhead reducing redundant computations.

Moreover, as for the STREAM SPE, NiagaraCQ exploits existing querying language to ease the programming of continuous queries. The introduced command language is an extension of the ordinary XML-QL language that allows the user to add queries at runtime and specify the frequency with which tuples are emitted by the system.

8.2.5 Cougar

The Cornell GOUGAR research project [BGS01], similarly to NiagaraCQ and TelegraphCQ, proposes an SPE that still relies on a database engine. Nevertheless, the focus is on sensor databases systems used to maintain long running queries over the data provided by the sensors. Such systems usually maintain stored data (information about the available sensors) and sensor data (information gathered from the sensors).

The traditional centralized (warehousing) approach to process sensors data is a two-step process: data is initially collected from all the sensors into a centralized DB and, subsequently, queried in order to extract the desired information. Such system does not scale because (1) a huge amount of data has to be collected and (2) part of this data might by not included in any query (i.e., data has been collected pointlessly).

The solution COUGAR proposes is a distributed processing of sensor data. Sensors (modeled as Abstract Data Types) are queried only to extract the information required by queries (if any). Furthermore, depending on each query, results can be evaluated at the front-end server or at the sensor network.

COUGAR distributed processing introduces a key aspect of distributed SPEs: when running distributed queries, data moved across nodes should be kept to the minimum, transferring only the useful information. That is, the data being transferred between two nodes should contain only the information needed to run a given computation, removing unnecessary data or performing previous analysis as close as possible to the data sources. This concept turns out to be important not only to attain a higher throughput in distributed SPEs, it also relates to dynamic load balancing, as will be presented in the next section.

8.3 State Of the Art SPEs

In this section, we focus on existing SPEs presenting a brief summary of the most important ones. We take into account the following SPEs: Esper [espb], Storm [sto], StreamBase [strb], IBM InfoSphere (or System S) [inf], Yahoo S4 [yah] and Microsoft StreamInsight [msi].

8.3.1 Esper

The Esper SPE [espb] is a centralized SPE that allows for rich analysis over data streams. Similarly to the STREAM project, it adopts a declarative language that resembles SQL like statements to express the operations to be run over the input data. The Esper SPE is available as a Java based solution or as a .NET solution (under the name of NEsper). The good feature provided by the engine is the high throughput while its main limitation is that, currently, the engine has been designed as a centralized solution. That is, authors have focused on the richness of the provided query semantic rather than the scalability and parallelization of its operators. Esper provides an API interface to ease the interaction with other programs, adapters used for data input and output and allows the user for fine tuning of operators characteristics such window type and behavior.

8.3.2 Storm

The Storm project [sto] can be seen as the complementary project of Esper. With Storm, the focus is on the distribution, parallelization and fault tolerance guarantees while letting the specification of how to process tuples to the final user. That is, queries can be expressed similarly as for *StreamCloud*, using the boxes and arrows model; the system will take care of distributing such operators among the available instances but will let the task of defining (by means of classes and functions) how to process data to the final user. With Storm, queries are expressed by means of two kind of objects, Spouts and Bolts. Spout nodes are responsible for generating the system input streams while Bolts are in charge of processing those streams and generate output tuples results. This project can be seen as the data streaming alternative to the Map-Reduce paradigm. As for the Map-Reduce [DG08] paradigm, the

user task is to define the functions that are used to read and generate data (Spouts) and to process it in parallel (Bolts) while the system is in charge of managing the several multi-threaded instances. Storm relies on Zookeeper servers [zoo] to maintain the state of distributed setups. Zookeeper is a server used to provide coordination among distributed applications maintaining configuration information and providing distributed synchronization and group services.

8.3.3 StreamBase

StreamBase [strb] is the commercial evolution of the Aurora SPE. StreamBase comprises three different products: StreamBase CEP, the core Complex Event Processing engine, StreamBase Live-View, a real time analytics solution that is used together with the SPE to present the result of the running queries by means of a warehouse-like interface, and StreamBase Adapters, a set of more than 150 adapters that ease the task of connecting applications to external sources or data receiver applications (e.g., visualization tools). The product, including the three different applications, provides a comprehensive data streaming solution that, similarly to *StreamCloud*, has been designed to ease as much as possible the user task reducing it to the definition of the desired query while automatizing all the processes that go from parallelization, fault tolerance and so on to output results visualization. StreamBase allows the user to define queries by means of a graphical interface (as for *StreamCloud*) using the *EventFlow* semantics or using the declarative *StreamSQL* language.

8.3.4 IBM InfoSphere

The InfoSphere SPE [inf] represents another alternative parallel-distributed SPE. Two interesting aspects of this SPE are its capability for processing several input formats such as XML, text, voice, video and so on and its query language SPADE [GTY+]. SPADE is a declarative language that allows the user for the definition of queries, also including information related to how to distribute or parallelize the query operators. The language also allows for the specification of user-defined functions or operators. The interesting aspect of the language is that, once a query has been defined, it is compiled to specific code to run the given application. This allows for higher processing capacities (due to the low-level specialized code) and for query-specific optimizations.

8.3.5 Yahoo S4

The S4 [yah] SPE represents a comprehensive free data streaming solution that provides distributed processing and fault tolerance capabilities. It is a Java based solution that, similarly to STORM, relies on the user for the definition of classes used to process and produce streams tuples. Similarly to the Storm project, S4 relies on Zookeeper to maintain the state of a distributed setup. One of the features S4 shares with *StreamCloud* is that it allows for the parallel execution of data streaming

operators (referred to as *Symmetrical Deployment*. Nevertheless, differently from *StreamCloud*, S4 does not provide a dynamic load balancing protocol, leaving its definition to the user. S4 also provides a Fault Tolerance protocol that is based on state checkpointing (i.e., passive standby). As discussed in Section (5.1, passive standby fault tolerance technique need to be used together with a variant of the upstream backup technique in order to provide precise recovery (i.e., failures are completely masked to the final user). In S4, fault tolerance is provided simply relying on state checkpointing and can (as stated by the authors) lead to (although minimal) state loss.

8.3.6 Microsoft StreamInsight

Microsoft StreamInsight [msi] CEP is the first attempt by Microsoft to emerge in the market of SPEs. Since the release in 2008, this engine is provided as a product of the SQL Server engine. Queries are expressed using the declarative LINQ [lin] (Language-Integrated Query) data streaming language, a SQL-like language that defines statements to use in order to specify window constructs. The product defines a small set of adapters that can be used to retrieve data and forward produced results. In general, the product seems to be less mature than other available free or commercial alternatives, missing appropriate solutions for distributed or parallel processing, load balancing and fault tolerance.

8.4 StreamCloud related work

In this section we present related work related to the innovative features of StreamCloud including a discussion about load shedding and operator scheduling protocols, parallelization techniques, load balancing protocols, elasticity protocol and fault tolerance protocols.

8.4.1 Load Shedding and Operators Scheduling protocols

In this section, we discuss existing work related to load shedding and operators scheduling techniques. All these techniques have been studied in order to improve centralized and distributed SPE achieving higher throughputs despite of the single-node bottleneck problem (discussed in Section 3.1). Although the related work being presented in this section relates to a solution orthogonal to the one being proposed in this thesis (single-node bottlenecks are overcome parallelizing data streams processing), we consider prior research interesting to be discussed as it shares some of the considerations that motivated our work.

Load shedding refers to a solution to face spikes in the system input load that can exceed the computational resources of an SPE discarding part of the incoming information. As an example, the processing cost of a system can be decreased sampling the input stream forwarding only part of

the incoming tuples. With load shedding, the challenge relies in how to chose which information to discard in order to degrade as less as possible the query output results. One of the early works related to load shedding is discussed in [Tat02]; the work discusses how load shedding can be applied without violating a Quality of Service (QoS) agreement. The QoS is based on the percentage of tuples being forwarded to the final user, their delay and their specific data values. The authors proposed two alternative load shedding techniques: static load shedding is used to achieve the higher query throughput taking off-line decisions based on the query operators selectivities and the data streams properties while dynamic load shedding is used to further discard tuples depending on the current input load (i.e., in the presence of spikes). The authors define a *drop* operator to discard tuples based on their position in the stream while employ a *filter* operator to discard tuples based on their values. The presented work is designed for a centralized SPE (Aurora).

The same authors continued their research in load shedding with the work presented in [TcZ+03]. In this paper, the authors define two novel *drop* operators for both randomized dropping and value based dropping. The work focuses on the following load shedding aspects: (a) when to activate it, (b) where to insert drop operators and (c) how to discard tuples. As stated by the authors, there is a trade-off between the position where drop operators can be inserted for a given query and the resulting output approximation. The rationale is that, the earlier the drop operator is inserted, the lower the cost of dropping tuples. On the other hand, the earlier the drop operator is inserted, the higher the probability that, in case of operators with multiple output streams, the higher the degradation of the output results. It is interesting to notice that, although our solution addresses spikes in the incoming load by means of dynamic load balancing or provisioning actions, both our approach and the one presented in [TcZ+03] require continuous monitoring in order to know *when* and *where* to prevent operator overload.

The work presented in [BDM04] discusses an alternative technique to take decisions about what information to discard if load shedding is used to prevent resources saturation in case of spikes in the input load. The authors propose a probabilistic filtering approach where each tuple as a probability p of being forwarded. In this case, the resulting query output (approximated) is scaled depending on the percentage of discarded tuples.

The work presented in [CWY05] represents another alternative load shedding technique. In this case, the focus is on data mining algorithms based on data streaming. The major difference in this case is that, as data is being mined, no a-priori information about how to provide the higher quality of service can be provided. Hence, the authors study a load shedding technique where decisions about which information to maintain and which to discard are based on predictions of future data values.

In [TZ06a] and [TCZ07], the load shedding technique is, for the first time, studied in the context of the distributed SPE Borealis. As stated by the authors, the main difference between load shedding in the context of centralized SPEs and distributed SPEs is that decisions about what information to

discard affect the rate (and subsequently the quality) of the information processed by downstream nodes. Hence, decision about how to shed the incoming load should be taken by each node not only considering its own quality of service degradation but also the one of its child nodes. The authors proposed a distributed load shedding technique where each node informs periodically its upstream peers exchanging metadata (referred to as FIT, Feasible Input Table) containing a summary of what the nodes expect in terms of input streams rate. This information is used by the upstream node to take decisions about how to shed load.

The work presented in [TZ06b] defines a different type of load shedding that has not been considered before. Early load shedding techniques considered every tuple as possible candidate to be discarded, leading to query results where each output could possibly be approximated. With this work, the authors focus on a load shedding technique where only part of the results is produced by a query but where each result is precise rather than approximated. The idea is to discard windows of data so that stateful operators like the aggregate operator produce less but precise output tuples. This load shedding technique is challenging because, as discussed in Section 4.2, due to the sliding window semantic, a single tuple may contribute to several windows. That is, avoiding producing the result of a given window does not translate in simply discarding all the window tuples.

In [TLPY06] and [TP06], the authors focus on a load shedding technique designed to discard information so that the overall output streams delay is kept as low as possible. The innovation relies in the approach used to decide how to discard the information. The authors present a Feedback based solution where a dedicated controller operator is used to continuously take decisions about the input streams rates (i.e., the tuples being forwarded) based on the feedback measured in terms of output streams delay.

Finally, the work presented in [JMSS07] presentes a load shedding technique that, similarly to [TZ06b], is designed to guarantee that only a subset of the query results are generated with no approximation. In this case, the information discarded is not based on windows but rather on the semantics of the stateful operators. As an example, the load shedding technique could be defined so that particular values of the $group - by$ attribute of an aggregate operator are not considered for tuple processing.

Together with the study of load shedding techniques, research as focused on data streaming operators scheduling. The idea is that load shedding should be enabled only if the available resources cannot cope with the incoming load (similarly to *StreamCloud* provisioning of nodes, issued if the system as a whole cannot cope with the given load). This means the plan used to schedule the operators of the different queries being run by a centralized SPE can be optimized to reduce the results degradation.

As presented in [MWA$^+$02], the operator scheduling technique can be designed for different goals, like reducing the overall response time, maximizing the throughput, providing fairness among

queries execution or minimizing the overall memory consumption. In the presented work, the authors focus on this last goal and define a scheduling algorithm that prioritizes operators that consume the largest number of tuples per unit of time and the operators with low selectivity (i.e., operators producing less tuples than the ones consumed) as both operators basically reduce the amount of memory occupied by the query tuples. As stated by the authors, this strategy could under-utilize operators with low priority. For this reason, the authors proposed the scheduling strategy to consider chains of operators in order to provide a fair scheduling of all the query operators.

In [BBMD03], the authors present another operator scheduling strategy that tries to minimize the overall memory occupied by the queries tuples. The work focuses on single input stream queries. For each query, the authors define as operator path the chain of operators through which a tuple is forwarded from the query input stream to its output stream. For each operator path, a *progress chart* is built considering the selectivity of each operator in order to define the expected average size of each tuple depending on its position in the operators chain. As an example, suppose the query is defined by a chain of three operators having selectivities 1, 0.5 and 0.1, respectively. On average, for each batch of incoming tuples (e.g., 10000 tuples), the tuples produced by the first operator will be the same (i.e., 10000), the tuples produced by the second operator will reduce to half of them (i.e., 5000) while the ones produced by the third one will be one tenth of the previous ones (i.e., 500). On average, the size of each incoming tuple will reduce to the 50% after the second operator and to the 5% after the third operator. The operators scheduling policy is designed so that, at each operator selection step, the operator that results in a smaller memory consumption after all its input tuples are processed is chosen.

In [CCR+03], the authors propose a different operator scheduling policy that, similarly to the load shedding technique proposed in [Tat02], defines a QoS metric to take decisions about which queries, and which operators, to schedule at each moment. As discussed by the authors, operators scheduling should not rely on multi-threaded infrastructure as, being threads managed by the operating systems, fine-grained control over the scheduling policy is not feasible (furthermore, multi-threaded systems do not scale when defining a high number of parallel threads). In the proposed operator scheduling policy, two-level decision are taken about which query and which operators to execute. Similarly to [MWA+02], operators are grouped to *superboxes* in order to be scheduled as an atomic entity. This grouping is motivated by the need of reducing the tuples processing cost. The authors define both a static and a dynamic plan to adjust the scheduling policy depending on the current system load.

8.4.2 Parallelization techniques

To the best of our knowledge Aurora* [CCC+02] and Flux [JHCF02] are the only efforts for parallelizing data streaming in shared-nothing environments with respect to the pioneer SPEs. Nowadays, state of the art SPEs such S4 [yah] or Storm [sto] also provide semantic aware parallelization.

Aurora* provides a parallelization technique named box-splitting. The idea is to parallelize an operator preceding it by a filter operator and putting a union operator after it to collect the results produced by the parallel operator, as presented in Figure 8.1.

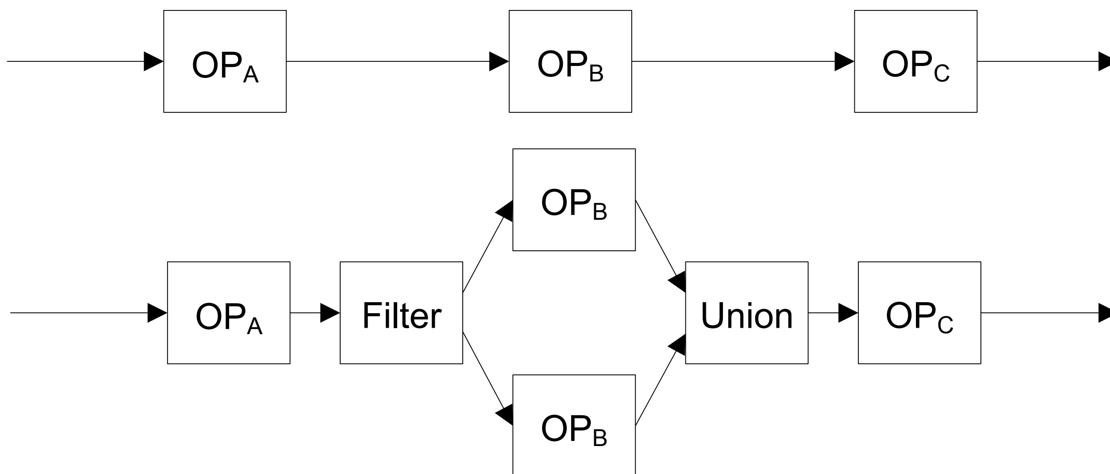


Figure 8.1: Box-Splitting example

This parallelization approach does not overcome the single-node bottleneck problem of any centralized or distributed SPEs as the parallel operator throughput is bounded by the capacity of its preceding filter. More precisely, this parallelization approach is useful only if the processing cost of the operator to be parallelized is greater than the filter or the union ones, and remains useful until the incoming stream (resp. the outgoing stream) of the parallel operator does not exceed the filter (resp. the union) capacity. Aurora* box-splitting has been introduced only for stateless operators. As presented in Chapter 3, the parallelization of stateful operators is more challenging than stateless ones as the former require semantic aware distribution of tuples in order to produce results that are equivalent to the ones of a centralized execution. The use of filter operators in order to route tuples in a semantic-aware fashion to an operator is possible, although the switch-like behavior of the filter operator (the condition of each output stream is checked sequentially in order to find the first matching output stream) will not provide a good scalability. Nevertheless, the use of Filter operators as routing operators of a parallel operator is not enough in the case of dynamic load balancing, as the operator is required to change dynamically its routing policy and manage special control tuples to orchestrate state transfer between instances of the same operator (as presented in Chapter 4).

Flux extends the exchange operator [Gra90] to a shared nothing environment for data streaming. The exchange operator has similar goals to our load balancer operator; nevertheless, as it has been designed for statistic configurations, it does not fit well with the variable nature of streams. Similarly to our load balancer, flux provides semantic aware tuples routing (referred to as content sensitive routing). Part of the innovative features of the Flux operator relate to dynamic load balancing, we

present a detailed comparison of these features in the following section. One of the main lacks of Flux[JHCF02] is that there is no evaluation on a real system for it. Evaluation has been conducted using a simulator and considering only a single operator query containing an aggregate operator. With Flux, aspects related to the parallelization of particular stateful operators are not considered. As an example, the parallelization of the Cartesian Product operator presented in Section 3.2.1.1.3 requires a particular routing policy that sends incoming tuples to multiple destination instances.

8.4.3 Load Balancing techniques

Several load balancing protocols have been proposed for data streaming applications. One of the earliest load balancing protocols has been proposed in Aurora*. The protocol, named *box sliding* or *horizontal load sharing*, is used to move operators among nodes at runtime. Suppose a chain composed by operators $OP_1, \ldots, OP_n$ is deployed at instance I, operator OP_1 is being fed by its upstream peer I_U and operator OP_n is feeding its downstream peer I_D. A box sliding action can be performed to move the first operators (e.g., OP_1 and OP_2) from I to I_U or the last operators (e.g., OP_{n-1} and OP_n) from I to I_D. Operators are usually moved across nodes depending on their selectivity. If the first operator of a node has very low selectivity (input stream rate $\gg$ output stream rate), the overall throughput of the query will increase if the operator is moved upstream (this will reduce the communication overhead between the instances). Similarly, if the last operator shows very high selectivity (input stream rate $\ll$ output stream rate) the query throughput will increase when moving the operator to its downstream peer node (as this will reduce the communication overhead between the instances). Nevertheless , this load balancing solution does not fit well with real world application as it only considers moving of operators at the extremes of the chain, while it does not cover cases where the overall throughput could increase if moving operators located in central positions of the chain. As example of the box sliding technique is presented in Figure 8.2.

In the example, a query composed by a chain of 5 operator $OP_A, \ldots, OP_E$ is deployed over 3 different nodes $Node_1, \ldots, Node_3$. Figure 8.2.a presents the initial deploy of the query: operator OP_A is deployed at $Node_1$, operator OP_E at $Node_3$ while remaining operators are deployed at $Node_2$. Figure 8.2.b presents a possible change in the original deployment where operator OP_B has been moved to $Node_1$. Such change in the deployment could happen if the selectivity of OP_A is greater than OP_B (i.e., the output stream rate of OP_A is greater than the one of OP_B). Figure 8.2.c presents a possible change in the original deployment where operators OP_C and OP_D have been moved to $Node_3$. Such change in the deployment could happen if the selectivity of operators OP_C and OP_D is lower than OP_B. It should be noticed that, in order to increase the overall query throughput, changes in the deploy cannot only be based on the selectivity of the operators but also on factors like their cost. That is, in the example of Figure 8.2.b, the overall throughput is increased as far as $Node_1$ has enough computational resources to maintain both operators at the same time.

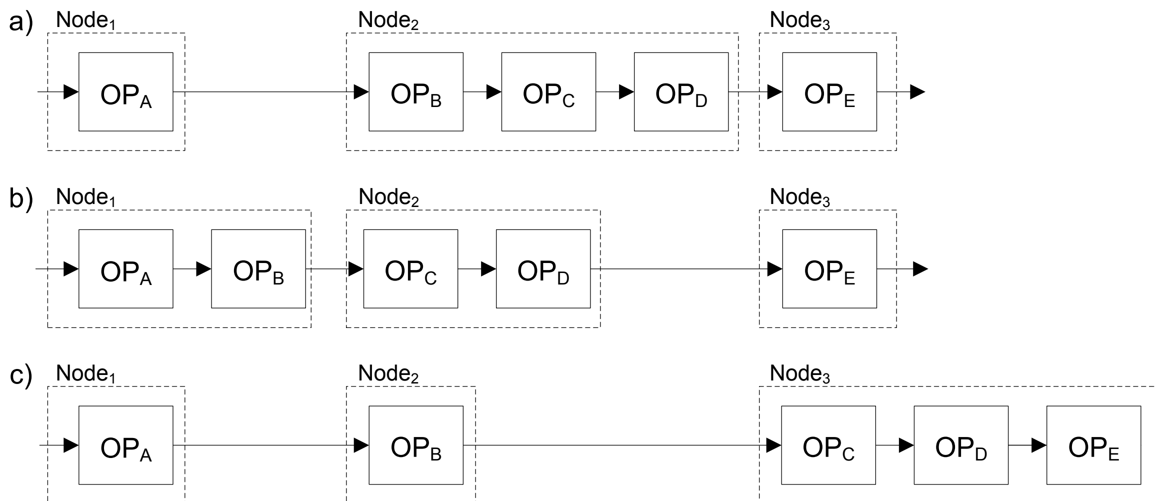


Figure 8.2: Box Sliding example

Load balancing is addressed in [XZH05] with a focus on the load correlation between operators. Load correlation between two operators is expressed as a value $\in [-1, 1]$. If correlation is -1 a burst in the load of one operator corresponds to a decrease in the load of the second. On the other hand, if correlation between the operators is 1, bursts happen at the same time for both of them. The proposed algorithm distributes existing operators among available instances trying to maximize the correlation across different instances. The rationale is that, upon a sudden burst in the input load, the additional processing cost caused by the spike will be balanced among nodes. The algorithm is composed by a first off line part used to determine the initial operators distribution plan plus an on-line component to adjust operators distributions at runtime in case of burst in the system input load.

In [CC09], the dynamic load balancing protocol is designed to redistributed the load of one operator among multiple instances whenever the operator is overloaded. The ideas is to deploy multiple instances of each operator composing a query among the instances used to run the query and, in case of saturation of on operator, to share its load among others idle instances. The protocol does not require any centralized component to monitor the state of each operator. Upon saturation, each operator sends a “backpressure” message to its upstream peer; the latter starts distributing its output tuples to multiple instances of the overloaded operator. The proposed solution presents several problems. First, multiple instance of each operator are allocated to the available nodes, implying running instances must share computational resources with idle operators (in *StreamCloud*, nodes are provisioned or decommissioned in order to make sure that extra computational resources are used only when necessary and are not kept idle). Moreover, the protocol considers only stateless operators, as seen in Section 4.2, changing how tuples are routed to a stateful operators is not trivial and requires a state transfer mechanism between the instances being reconfigured.

Flux defines distinct protocols for *short-term imbalances* and *long-term processing imbalances*.

To face short-term imbalances, the flux operator has been designed to “absorb" fluctuations in an operator output rate. Flux distributing operator is composed by two separate units: the *Ex-Prod* unit is in charge of taking output tuples produced by an operator and pass them to several *Ex-Cons* units, in charge of forwarding tuples to the downstream parallel operator. In order to face short-term fluctuations, the memory allocated by the Ex-Prod unit is not statically partitioned to the existing Ex-Cons units. On the contrary, it is dynamically adjusted to provide more available memory to the consumer units receiving more tuples.

Flux also defines a protocol to face the long-term imbalances. The protocol is similar to the one proposed in *StreamCloud*, nevertheless: Flux protocol is designed to be blocking, if state must be transferred between two instances, the system makes sure all its upstream flux operators have flushed their respective Ex-Cons units and the operator has processed all its input tuples. Furthermore, the proposed protocol defines generic `getPartitionState` and `installPartitionState` functions to move operators state across instances, while *StreamCloud* provides two working protocols for state transferring.

The work presented in [BBS04] discusses a load balancing algorithm for federated systems where autonomous participants might stipulate pairwise contracts pricing the costs of migrating computational units among them. That is, pairs of participants might fix prices to transfer part of the computation if, letting a peer processing part of the information results in a lower cost than processing it locally. This problem is quite close to the “classic" load balancing problem as often the processing being moved to external participants is caused by the saturation of the local resource, which incurs in high processing costs. Nevertheless, the problem being studied presents also differences from the original problem, as the balancing problem cannot be seen as the problem of achieving the best distribution for the system as whole due to the fact that each participant looks at minimizing its cost (i.e. maximizing its benefits). In the presented work, contracts are negotiated off-line for each pair of participants and can be either fixed (the unit processing cost is fixed) or defined by means of an interval (i.e., min and max price). In the second case, load balancing is more challenging as each participant can decide to further distribute the processing of another participants depending on the specific cost. From an implementation point of view, the algorithm as been designed as an on-line, distributed algorithm where each participant negotiates with the others transferring of processing units depending on the current system load. Processing units being transferred are intended as group of operators belonging to the continuous queries defined by each participant. One of the weak points in the proposed work is that, when transferring stateful operators, the state is not moved across instances but rather recreated at the destination instance, loosing the one previously maintained by the old participant. As discussed in Chapters 4 and in 5, state transfer is challenging and recreation of stateful operators states performed just resuming the processing at a different node (i.e., like the gap recovery protocol) might lead to incorrect or incomplete results.

The work discussed in [XHCZ06] presents a particular load balancing algorithm designed for systems that to not allow for the transferring of data streaming operators among nodes. Even if this system requirement is not a limitation in *StreamCloud*, it is interesting to study such problem as it could help in reducing the number of operators transfers among nodes hence reducing reconfiguration actions. Being the operators statically assigned to each node during all the execution time of a query, the decision about how to deploy operators must be taken off-line. As discussed by the authors, this off-line planning of the operators deployment might be enough to face short-term imbalances. This holds also for the work presented in this thesis, as we suppose the imbalance cause by a change in the system input stream characteristics has usually an impact which duration exceeds the time it takes to perform a reconfiguration action. As stated by the authors, the problem of finding the optimal planning given a set of operators and their expected input rates and selectivities is computationally hard. For this reason, the authors define a greedy algorithm to find sup-optimal deployment plans.

8.4.4 Elasticity techniques

Elasticity in the context of SPEs has not been studied at the same depth as other aspects such parallelization, dynamic load balancing and fault tolerance. In the context of traditional database systems, solutions like the ones proposed in [SAG06] and [CSA06] are designed to dynamically provision database servers in order to keep the average below an arbitrary service level agreement. These solutions are hard to compare with data streaming based solution as the two systems have different goals; more precisely, database system typically do not require near real-time processing.

The work presented in [GSP$^+$] contains some interesting analogies with *StreamCloud*. Although the focus is not on data streaming query processing as it is rather related to database solutions, the presented considerations hold with respect to data streaming queries. The motivations of the proposed work are the challenges that emerge from the possibility of obtaining resources on demand (e.g., from cloud infrastructures) in large distributed systems. As stated by the authors, new available resources might provide heterogeneous processing capacities, it is thus mandatory to continuously adjust the load distribution (or *horizontal imbalances* as referred by the authors) to meet the punctual best distribution adjusting the intra-operator parallelization policy. Furthermore, bottlenecks caused by consecutive operators (or *vertical imbalances* as referred by the authors) should be removed distributing the operators at distinct nodes adjusting the inter-operator parallelism. The research done in *StreamCloud* covers all these aspects. First, when defining how to partition a query, the parallel compiler defines subclusters that maximize the throughput (as presented in Chapter 3.2) trying to minimize the so-called *vertical imbalance*. Moreover, as presented in Chapter 4, dynamic load balancing and elasticity techniques have been designed to overcome *horizontal imbalances*. It should be noticed that, even if we do not study directly provisioning of heterogeneous resources, our solutions adapts to them as the fluctuations in the data streams rate and distribution lead to equivalent observed

behaviors (i.e., machine having different CPU consumption even in front of an even distribution in terms of tuples / second).

The work presented in [SAG+09] presents an elastic protocol where processing threads are provisioned or decommissioned depending on the node state. In the protocol, a single *work queue* is used to maintain the node incoming tuples while multiple processing threads run in parallel consuming the tuples stored in the queue. This solution has two main shortcomings: the first limitation is that multi-threaded systems do not scale well for big number of concurrent threads while the second one is that, independently of the scalability, processing capacity is limited to the resource of a single physical machine.

This elasticity protocol does not address several challenging aspects addressed by *StreamCloud*. Being the provisioned instance a different physical machine, several tasks must be performed before the machine can start processing tuples. Among others, tasks include connection establishment to and from the machine and deployment of the new operators. When provisioning only processing threads, provisioning completion time decreases drastically. This is because provisioning only consists in informing a new thread it can actually start processing tuples.

The second challenging aspect is related to how load is balanced among processing threads upon a provisioning or decommissioning action. Due to the shared resources among all the processing threads, the system is able to maintain a single queue for the incoming tuples, letting each processing thread consuming tuples at its maximum rate. This is not possible in *StreamCloud*, as it has been designed considering a shared-nothing architecture. That is, different instances of the same parallel operator are not sharing any resource.

Finally, the proposed solution only considers stateless operators. As it has been presented in 4.2, provisioning, decommissioning or dynamic load balancing actions are challenging when considering stateful operators.

In [LHKK12], the authors discuss elasticity (and fault tolerance, which will be discussed in the following section) in the context of a “streaming as a service" architecture. The authors propose a multi-tenant Cloud based architecture where users can register queries and input streams are receive results in a real-time fashion relying on the data streaming processing paradigm. The presented work cannot be directly compared with *StreamCloud* as it differs from the latter in the sense that elastic capability is not intended to increase the resources available even to a single operator but rather to increase the number of nodes where all the different queries (intended as an atomic processing unit) can be accommodated. That is, the solution is intended to scale with respect to the number of active queries and input streams. In the proposed solution, load balancing is applied re-assigning part of the queries run by a node to a different less-loaded node. Similarly to *StreamCloud*, if load cannot be balanced using the assigned nodes, additional nodes are added and part of the queries run by the previous nodes are re-assigned. The presented work introduces several limitation of the queries

that can be run by the system: queries can define multiple input streams but single output streams; furthermore, the authors make the assumption that the resource consumption of individual queries does not exceed the available resources of an individual node (i.e., a single query can be executed by a single node) reducing the scope of the application to small to medium size queries. The authors state the architecture is being implemented and do not provide preliminary results about system throughput or scalability.

Recently, the author of [Hei11] presented an high level overview of data streaming elasticity protocols and the planning for the development of elastic capabilities for the StreamMine system. The work has been presented at a doctoral symposium and introduces the aspects that are covered in the Ph.D. research. Most of the aspects related to elasticity are similar to the ones presented in this thesis; nevertheless, the high level description and the lack of details about the protocols does not permits a deep comparison between *StreamCloud* and the author system (StreamMine).

8.4.5 Fault Tolerance techniques

Several fault tolerance algorithms have been proposed in the context of data streaming. In this section, we present how each existing solution has inspired our work.

[HBR+05] presents a classification of fault tolerance guarantees and protocols. Three different guarantees are presented. *Precise recovery* completely hides any failure to the end user application (e.g., results produced in case of failure are identical to the ones produced during a fail-free execution). *Rollback recovery* avoids information loss (e.g., all the information is processed but results may differ from a fail-free execution). Finally, *Gap recovery* simply ignores state loss due to one (or more) instance failure, leading to partial (and possibly different) results. As discussed also in [HBR+03], three different protocols exists that provide fault tolerance capabilities to an SPE: *Active Standby*, *Passive Standby* and *Upstream backup*. *Active Standby* relies on replicas that, in parallel with a primary node, process the same tuples and whose state is continuously updated with respect to the one of the primary node. In case of failure, one of the primary replicas is taken as new primary node switching the input and output streams of the former to the latter. This protocol implies a high runtime overhead both in terms of resources (as replica nodes keep redundant state) both in term of processing costs (output tuples are being forwarded to multiple outputs). On the other hand, it provides fast recovery as the action taken to replace a failed node simply boils down to the redirection of some streams. *Passive Standby* defines a checkpointing mechanism used to persist continuously the state of the query operators. Checkpoints are usually maintained at backup server nodes and, in case of failures, installed to the replacement instance taking the place of the failed one. Opposite to the *Active Standby*, this approach reduces the runtime overhead in terms of resources (the number of backup server being used to maintain state checkpoints is usually much smaller than the number of replicas) but leads to a higher recovery time as the replacement instance needs first to install the lost state before

being able to process new incoming tuples (also in this case the recovery actions include redirection of the lost instance input and output streams). Finally, *Upstream backup* provides fault tolerance for a given instance relying on its previous and next peers (resp. upstream and downstream instances). That is, upstream instances define a protocol so that output tuples are maintained as long as the downstream instances confirms they can be deleted. In case of failure, upstream instances replay all tuples maintained in the output queues that have not being acked by the downstream instances to the replacement instance. This strategy further reduces the runtime overhead in terms of resources (only the active nodes are in charge of providing fault tolerance) but implies a recovery time comparable with the one provided by the *Passive Standby* approach.

In the following, we provide a description of the most relevant solution being presented to provide fault tolerance in the context of SPEs. All the solutions can be seen as modifications and improvements of the active standby or the passive standby methods. Rather than discussing why each solution is not optimal with respect the the parallel-distributed SPE presented in this thesis, we first proceed with a discussion about the limitations of these approaches and continue then with the presentation of the related work. We consider that active standby (i.e., replica) based solutions such [SHB04] or [BBMS08] are unsuitable for parallel-distributed SPEs. They are too expensive in terms of resource utilization (e.g., using a cluster to run data streaming applications having 3 replicas for each node leads to 25% utilization). Furthermore, whenever an SPE instance is added due to a provisioning action (or removed due to a decommission action) replicas should be setup accordingly, leading to high reconfiguration times. On the other hand, also passive standby solutions where operators state are being periodically checkpointed are not optimal. Even if not stated clearly in most of the presented works, passive standby protocols work only if applied together with upstream backup ones. Upon a failure of an instance, the information processed by it during the time interleaving the previous checkpoint and the failure has not been stored nowhere. The only way to avoid the loss of that information is keeping it at the upstream peers. This requirement of the passive standby protocol implies a duplication in the logic of the fault tolerance protocol, making its cost higher. This reason motivated our research in defining the fault tolerance protocol as an evolution of the upstream backup one. To the best of our knowledge, our fault tolerance protocol is the first protocol that takes into account elasticity and dynamic reconfigurations in general.

In [BBMS04] and [BBMS08], the authors describe an active replica based fault tolerance solution for the Borealis distributed SPE. Although the active replica protocol is not new, the authors improve it defining a user-configurable trade-off between availability and consistency. Suppose a node being fed by multiple upstream nodes suddenly stops receiving tuples from one of its upstream peers. The node could react in two different ways: on one hand, it could still produce results computed over the tuples forwarded by the remaining instances; on the other hand, the node could wait for the upstream node to recover as long as the time constraints (expressed as a QoS metric) of the tuples

being processed are not violated. The configurable trade-off is about the amount of time the node should wait hoping the failed node replacement completes before any imprecise tuple is produced. In the presented work, the authors define a *SUnion* operator that is used to make sure the primary node and replicas are processing incoming tuples in the same order, thus producing the same output tuples and maintaing the same state. This operator shares some of the design goals of the *Input Merger* operator defined by *StreamCloud*, discussed in Section 3.2.1. In the proposed works, the authors also discuss the possibility of correcting imprecise tuples by means of correction tuples generated after the healing of a failed instance. This mechanism can further reduce the impact of a failure but is effective as long as the imprecise tuples have not being already forwarded to the final user (in which case fault tolerance would depend on the external user).

In [BSS06] and [BS07], the authors present a *Passive Standby* approach (referred to as Efficient and Coordinated Checkpoint - ECOC) that is used in an SPE employed in health-care applications. The presented protocol does not improve significantly previously techniques relying on the *Passive Standby* technique. As stated by the authors, and as discussed in Chapter 5, the passive standby technique relies also on the upstream technique in order to recover the lost state referring to the information sent during the time interleaving the last checkpointing and the failure time. In the proposed work, the passive standby technique is justified by the need of reducing the communication overhead as much as possible due to the limited resources of the mobile devices used in health-care applications (like blood pressure monitor, heart beat monitors and so on).

In [HCZ07] and [HCCZ08], the authors discuss an active standby replica fault tolerance approach for wide-area network SPEs. The presented work introduces some improvements of the classical active standby protocol. The first improvement is related to how each node processes its incoming tuples. Rather than consuming the tuples forwarded by its primary upstream node, each node consumes the tuples of any upstream replica so to process the first available tuple. This approach implies a significant reduction of the recovery time as, in case of failure, downstream instances are already instructed to process tuples coming from other replicas. Nevertheless, this approach requires a more efficient way of maintaining replicas consistent. That is, an approach where each replica consumes its upstream tuples in the same order will slow down the processing to the slowest replica, making thus the proposed technique inefficient. The authors propose a punctuation based protocol where special tuples (punctuation tuples) are used to guarantee the stateful operator processing the incoming tuples that no more tuples will have a lower timestamp than the punctuation tuple one. With this information, stateful operators introduce a minimal delay in the computation of the output tuples that is negligible with respect to the one cause by a synchronization of the replicas input streams. The same protocol is presented in [MPH10a] and [MPH10b], where the authors discuss how the protocol is being used in the context of the *iFlow* SPE.

In [HXCZ07], the authors propose an improved algorithm for the passive standby fault toler-

ance protocol. The first idea is to define delta-checkpointing of the operators states. With delta-checkpoints, rather than persisting periodically the entire snapshot of an operator state, only the difference between the previous state and the current one is persisted. The second improvement relies in the partitioning of the checkpoints deltas and in distributing them at multiple peer nodes acting as servers. This improvement significantly reduces the recovery time as the lost state can be re-collected at the replacement instances by multiple sources simultaneously. For the same reason, the recovery time is also reduced in the presence of multiple instance failures as lost operators state can be recreated in parallel. The presented work studies how the partitions of the operators state should be distributed among the other peer nodes and, for each stateful operators, defines an ad-hoc delta checkpointing mechanism. As an example, the state of an aggregate operator is delta-checkpointed marking each window with a *dirty bit* that specifies if the window has been created after the previous checkpoint or, during the interleaving time, it has been modified.

[KBG08] also introduces a protocol based on delta-state checkpointing. While being updated processing incoming tuples, operators state is periodically checkpointed and persisted. In case of failure, a replacement instance is deployed and the lost state is re-created starting from the latest checkpoint. Checkpoints are written to a parallel, replicated file system. This solution introduces the idea of persisting state relying on disks instead of memory. We believe such solution is mandatory for a fault tolerance protocol to work with parallel-distributed SPEs. Nevertheless, the proposed solution might incur (even if with low probability) in delayed processing as the state might not be concurrently updated and checkpointed.

As stated in [GPYC08], all the existing fault tolerance techniques (either reactive or proactive) impose a runtime overhead. The idea of the proposed work is to define a predictive model that can be used to turn such fault tolerance mechanism on only in the presence of an incoming failure. The proposed protocol implies stream classifiers to continuously monitor the state of a node and mark it as *normal*, *alert* or *failure*. The idea is to activate the fault tolerance mechanism when the state of a node switches from normal to alert. In order to build the stream classifiers, the system defines a training period used by the protocol to prevent future failures. When the system detects a failure might happen in a short period of time, it first isolates the failing operator moving it to a dedicated node in order to reduce the impact of the failure and instantiating a replica operator to replace it in case of failure. When in *alert* state, the monitoring of a possibly failing operator is increased in order to collect information that can be useful to predict future failures for the same type of operator. The main limitation of the proposed work relies in the fact that the types of failures that can be predicted is usually small. Furthermore, the proposed technique is valid as long as the computational resource needed by the prediction protocol do not exceeds the ones needed by an alternative fault tolerance protocol.

In [BFF09b] and [BFF09a], the authors focus on a fault tolerance technique for non deterministic

operators. In data streaming, non determinism in the operator execution can be caused by functions that are sensitive to the input stream tuples arrival order or to functions that depends on the time. As stated by the authors, the information to be maintained in order to provide fault tolerance for such operators will also include ordering and extra information depending on the function being applied to the data. The authors also study how to maintain active replicas efficiently overcoming the limitations imposed by a strict synchronization of the replicas input tuples ordering and studying how to run replicas in a multi-threaded fashion.

In [GZY$^+$09], the authors propose an improvement for the passive stand-by fault tolerance protocol. With this protocol, referred to as *sweeping checkpointing* the authors define a checkpoint of an operator as the copy of its state and its output queues. The idea is to avoid scheduling operators checkpointing by means of a periodic task but rather let each operator checkpoint its state at the time when the task will have less impact on the overall computation. The idea is to checkpoint operators state just after their output queues have been trimmed of tuples no longer required by their downstream peers. Doing this, the checkpoint state will be smaller and the overall impact of the fault tolerance technique will be decreased.

In [ZGY$^+$10], the authors propose a hybrid fault tolerance protocol that mixes the active stand-by and the passive stand-by protocols. The idea is to periodically checkpoint the state of the query operators but, rather than storing them in a dedicated server, installing them in an idle copy of the operators. That is, a replica of the operators is being maintained by a separated node but, rather than receiving the same tuple being processed by the primary node, its state is continuously updated with incremental checkpoints. In case of temporary failure, the replica is activated and it starts processing the same tuples of its primary counterpart. If the failure results to be permanent, the query is instructed to start forwarding tuples only to the replica and start consuming its output tuples.

The authors of [SM11] present a fault tolerance protocol based on an asynchronous checkpointing mechanism (akin to fuzzy checkpointing) where, instead of checkpointing an operator's state and its output queues, output tuples are mixed with windows checkpoints (which are dedicated output tuples describing a window's intermediate state) and only output streams are persisted. In case of failure, the output queue of the failed operator is read to retrieve information about the latest window checkpoint and, therefore, compute from where to reply tuples of its input stream. As discussed by the authors, intermediate state checkpoints decrease the overall recovery time as they reduce the amount of tuples to be read (and replayed) in order to recreate the failed operator lost state. This approach is similar to *StreamCloud* in the sense that recovery requires the ability of computing from where to replay tuples in order to recreate an operator state. Nevertheless, *StreamCloud* improves on it as follows: (1) the earliest timestamp (i.e. the information about the point in time from where to replay tuples in case of failure) is not maintained mixing regular output tuples with checkpoint tuples but rather using output tuples header (finer granularity and lower overhead); (2) the earliest timestamp

in maintained on-line in *StreamCloud*, avoiding thus unnecessary read operations in the parallel file system to retrieve its value and finally, (3) [SM11] do not consider dynamic setups (elasticity) nor stateful operators garbage collection mechanisms. The authors of [SM11] leverage previous work [SM10] on how to efficiently persist a stream connecting two data streaming operators relying on a parallel file system. *StreamCloud* leverages and improves on this work by providing a better way to persist streams adopting a self-identifying naming convention for the persisted information; thus avoiding metadata maintenance as in [SM10] and reducing the runtime protocol impact (about 20ms in the proposed work to approximately 1ms in *StreamCloud*, as presented in 5.6.2).

Part IX

Conclusions

Chapter 9

Conclusions

In this thesis we have presented *StreamCloud*, a parallel-distributed Stream Processing Engine (SPE) with dynamic load balancing, elasticity and fault tolerance capabilities. We have discussed how research in data streaming emerged in the past decade to overcome the limitations of previous DB-based solutions that could not satisfy the required high processing capacity and low latency guarantees of nowadays real-time applications such as network monitoring, financial applications and fraud detection applications. We have presented the design goals and architecture of pioneer centralized and distributed SPEs and we discussed why all of them suffer from the single-node performance bottleneck. We have discussed the main challenges in designing a parallel-distributed SPE and each challenge has been addressed and constitutes a contribution of the presented work:

Parallelization of data streaming operators and queries. *StreamCloud* introduces a new parallelization technique for data streaming operators that guarantees semantic transparency. That is, tuples produced by a parallel operator are the same as the ones produced by its centralized counterpart. The parallelization technique has been designed, implemented and fully evaluated for all the main data streaming operators defined by an SPE. Furthermore, *StreamCloud* provides a parallelization and distribution protocol for continuous queries to achieve high processing throughputs. As for the data streaming operators, this protocol has been designed, implemented and fully evaluated. All the protocols are discussed in Chapter 3.

Elasticity and Dynamic Load Balancing protocols. *StreamCloud* defines protocols to provision and decommission nodes in an on-line fashion so that the resources being used are always adjusted

to the current system load. This overcomes the limitations of workaround solutions such over-provisioning (where the allocated resources are chosen to cope with the maximum system load) that results in under-utilization; or under-provisioning (where the allocated resources are chosen to cope with the average system load) that, in case of temporary spikes or long-term variations in the system input rate, might lead to high processing latencies. The elasticity protocols have been designed, developed and evaluated together with a dynamic load balancing protocol. The dynamic load balancing protocol has been defined in order to make sure that nodes are provisioned (or decommissioned) only if the system as a whole cannot cope (or is under-utilized) with respect to the system incoming load. All the protocols are discussed in Chapter 4.

Fault Tolerance protocol. In this thesis we have introduced a novel fault tolerance protocol designed for parallel-distributed SPEs with elastic capabilities. This protocol is designed to incur in a small runtime overhead while providing fast recovery for single or multiple instance failures. The presented fault tolerance protocol faces challenges that have not been addressed before, like failures of nodes being provisioned or failures of nodes whose state is being transferred to other nodes (e.g., due to a dynamic load balancing action). The presented protocol has been designed, implemented and fully evaluated and is discussed in Chapter 5.

Comprehensive IDE. Together with *StreamCloud*, we developed a comprehensive IDE to ease user tasks such query composition and application execution. The IDE provided with *StreamCloud* reduces the interaction of a user with the system to the definition of the abstract query to be run (i.e., with no specific information about how to deploy it) and information about the available nodes. The IDE compiles then the given query into its parallel-distributed counterpart and provides a series of scripts that can be used to execute the corresponding application. Several adapters have been designed to ease the injection and reception of information from different sources or destination applications. Furthermore, the IDE defines a graphical network interface that can be used to monitor the system state, visualizing which queries are being executed and presenting, for each query operator, statistics such input stream rate, output stream rate or CPU consumption. The IDE is presented in Chapter 6.

All the protocols presented in the thesis have been fully implemented and evaluated. During the development of *StreamCloud*, several real world applications have been used as a testbed of the system. Such applications are discussed in Chapter 7, presenting why *StreamCloud* is a good candidate for these applications and presenting the definition and possible implementation of real use-cases.

Part X

APPENDICES

Appendix A

The Borealis Project - Overview

In this chapter, we provide a short introduction to some of the features of the Borealis SPE engine. This introduction is not intended to give a complete description of the Borealis SPE, we only focus on particular aspects that strictly relate with the design and implementation of *StreamCloud*. We first introduce the Borealis SPE query algebra, presenting how *continuous queries* are defined by means of XML files. Subsequently, we discuss how new user-defined operators can be added to the set of available operators used to compose *continuous queries*. We continue discussing some details of the Borealis SPE tuple processing paradigm and we conclude introducing the tools provided by the Borealis SPE to ease the developing of applications.

A.1 Query Algebra

In this section we introduce the Borealis SPE query algebra, presenting how *continuous queries* and information related to the nodes where they are deployed is provided my means of XML files. We refer the reader to the Borealis application programmer's guide [bora] and the Borealis developer's guide [borb] for a detailed description of the Borealis SPE.

As presented in Chapter 2, a *continuous query* is defined by its composing operators. Each operator is defined by its type, a set of attributes and at least one input stream and one output stream. For each input (resp. output) stream, a schema specifies which are the composing fields of the input (resp. output) tuples. In Borealis, *continuous queries* (i.e., schema, streams and operators) are defined by XML elements. In the following, we present how the *High Mobility Fraud Detection*

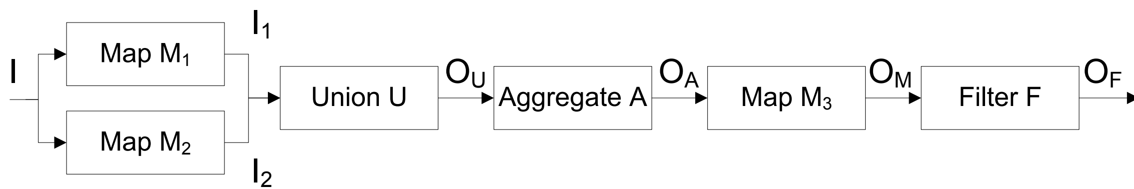


Figure A.1: High Mobility fraud detection query

continuous query introduced in Section 2.2, used in cellular telephony fraud detection applications to spot mobile phones that, between two consecutive phone calls, cover a suspicious space distance with respect to their temporal distance is defined using Borealis query algebra. The *continuous query*, shown in Figure A.1, consumes CDR tuples whose schema is composed by fields $Caller$, $Callee$, $Time$, $Duration$, $Price$, $Caller_X$, $Caller_Y$, $Callee_X$, $Callee_Y$. Field $Caller$ specifies the mobile phone number making the phone call while field $Callee$ specify the call receiver, fields $Time$, $Duration$ and $Price$ specify the time when the phone call starts, its duration and the overall price, fields $Caller_X$, $Caller_Y$ specify the $Caller$ geographic coordinates while $Callee_X$, $Callee_Y$ the $Callee$ ones.

Listing 6 presents the XML *schema* element that defines the input tuples schema. Each *field* element is defined by attributes *name* and *type* (plus the extra *size* attribute for string type) [1]. Once a schema has been defined, it can be associated to any input or output stream. In the listing, the schema is associated to the input stream named *input*.

```
<schema name="input_schema">
        <field name="caller" type="string" size="9"/>
        <field name="callee" type="string" size="9"/>
        <field name="time" type="int"/>
        <field name="duration" type="int"/>
        <field name="price" type="double"/>
        <field name="callerx" type="double"/>
        <field name="callery" type="double"/>
        <field name="calleex" type="double"/>
        <field name="calleey" type="double"/>
</schema>

<input stream="input" schema="input_schema" />
```

Listing 6: Input Schema and Input definition

For each incoming tuple, each of the first two map operators creates a tuple composed by fields $Phone$, $Duration$, $Time$, $Price$, X, Y. While fields values $Duration$, $Time$, $Price$, are just copied from the input tuple values, operator M_1 sets field $Phone$ value to be equal to the input field $Caller$ and fields X, Y to be equal to $Caller_X$, $Caller_Y$ while operator M_1 sets $Phone$ to be equal to the input field $Callee$ and fields X, Y to be equal to $Callee_X$, $Callee_Y$.

[1] In all the listing presenting XML code, streams and operators names are converted to lower-case letters and separation symbols such “_" are removed (i.e., operator name OP_3 is converted to op3).

```
<box name="m1" type="map" >
        <in stream="input"/>
        <out stream="i1"/>
        <parameter name="expression.0" value="caller"/>
        <parameter name="output−field−name.0" value="phone"/>
        <parameter name="expression.1" value="duration"/>
        <parameter name="output−field−name.1" value="duration"/>
        <parameter name="expression.2" value="time"/>
        <parameter name="output−field−name.2" value="time"/>
        <parameter name="expression.3" value="price"/>
        <parameter name="output−field−name.3" value="price"/>
        <parameter name="expression.4" value="x"/>
        <parameter name="output−field−name.4" value="callerx"/>
        <parameter name="expression.5" value="y"/>
        <parameter name="output−field−name.5" value="callery"/>
</box>

<box name="m2" type="map" >
        <in stream="input"/>
        <out stream="i2"/>
        <parameter name="expression.0" value="callee"/>
        <parameter name="output−field−name.0" value="phone"/>
        <parameter name="expression.1" value="duration"/>
        <parameter name="output−field−name.1" value="duration"/>
        <parameter name="expression.2" value="time"/>
        <parameter name="output−field−name.2" value="time"/>
        <parameter name="expression.3" value="price"/>
        <parameter name="output−field−name.3" value="price"/>
        <parameter name="expression.4" value="x"/>
        <parameter name="output−field−name.4" value="calleex"/>
        <parameter name="expression.5" value="y"/>
        <parameter name="output−field−name.5" value="calleey"/>
</box>
```

Listing 7: Map operators M_1 and M_2 definition

As presented in Section 2.2, these operators are defined as:

$$M_1\{Phone \leftarrow Caller, Duration \leftarrow Duration, Time \leftarrow Time,$$
$$Price \leftarrow Price, X \leftarrow Caller_X, Y \leftarrow Caller_Y\}(I, I_1)$$

$$M_2\{Phone \leftarrow Callee, Duration \leftarrow Duration, Time \leftarrow Time,$$
$$Price \leftarrow Price, X \leftarrow Callee_X, Y \leftarrow Callee_Y\}(I, I_2)$$

Their definition by means of XML elements is presented in Listing 7. Both operators are feed with stream *input*, as specified by the XML *in* element, Map operator M_1 produces output stream I_1 while Map operator M_2 produces output stream I_2 (*out* XML elements). The fields composing the output tuples schema are defined by pairs of *parameter* elements using attributes *expression.#* and *output-field-name.#* (where # represents the expression number, starting from 0). It can be noticed that the only difference in how the operators define the output schema is in attributes *expression.0*, *expression.4* and *expression.5*: operator M_1 defines $Phone$ field to be equal to $Caller$,X to be equal to $Caller_X$ and Y to be equal to $Caller_Y$ while operator M_2 defines $Phone$ to be equal to $Callee$,

X to be equal to $Callee_X$ and Y to be equal to $Callee_Y$.

The Union operator U merges tuples produced by both Map operators M_1, M_2 to a single stream. It is defined as:

$$U\{\}(I_1, I_2, O_U)$$

The XML representation of this operator is presented in listing 8. This operator defines one *in* element for each input stream and one *out* element. In this case, we specify that the operator consumes the tuples produced by Map operators M_1 and M_2 defining as *in* elements streams I_1 and I_2.

```
<box name="u" type="union" >
        <in stream = "i1"/>
        <in stream = "i2"/>
        <out stream = "ou"/>
</box>
```

Listing 8: Union operator U definition

Once tuples produced by Map operators M_1 and M_2 have been merged by the Union operator U we use the Aggregate operator A to extract X, Y and $Time$ fields for each pair of consecutive tuples referring to the same $Phone$ number. The Aggregate operators is defined as:

$$A\{tuples, 2, 1, Time \leftarrow first_val(Time), T_1 \leftarrow first_val(Time), X_1 \leftarrow first_val(X),$$
$$Y_1 \leftarrow first_val(Y), T_2 \leftarrow last_val(Time), X_2 \leftarrow last_val(X), Y_2 \leftarrow last_val(Y),$$
$$[Group - by = Phone]\}(O_U, O_A)$$

The XML representation of the Aggregate operator is presented in Listing 9.

The XML *parameter* is a generic XML element used to define the properties of an operator. With respect to the Aggregate operator, this element is used both to define the operator parameters such the window type, the window size and so on and to define the fields composing the output tuples. Similarly to the Map operator, pairs of elements with attributes *aggregate-function-output-name.#* and *aggregate-function.#* are used to specify the name and the function of each output schema field. The operator *group-by* field is specified using a *parameter* XML element with attribute *group-by*, window size and advance are defined by parameters *window-size* and *advance* while the window type (tuple-based in the example) is defined setting attribute *window-size-by* to *TUPLES*. As defined in Borealis, a time-based window is specified setting *window-size-by* attribute to be equal to *TIME*.

```
<box name="a" type="aggregate" >
        <in  stream = "ou" />
        <out stream = "oa" />
        <parameter name = "aggregate−function.0"    value = "firstval(time)" />
        <parameter name = "aggregate−function−output−name.0"  value = "time" />
        <parameter name = "aggregate−function.1"    value = "firstval(time)" />
        <parameter name = "aggregate−function−output−name.1"  value = "t1" />
        <parameter name = "aggregate−function.2"    value = "firstval(x)" />
        <parameter name = "aggregate−function−output−name.2"  value = "x1" />
        <parameter name = "aggregate−function.3"    value = "firstval(y)" />
        <parameter name = "aggregate−function−output−name.3"  value = "y1" />
        <parameter name = "aggregate−function.4"    value = "lastval(time)" />
        <parameter name = "aggregate−function−output−name.4"  value = "t2" />
        <parameter name = "aggregate−function.5"    value = "lastval(x)" />
        <parameter name = "aggregate−function−output−name.5"  value = "x2" />
        <parameter name = "aggregate−function.6"    value = "lastval(y)" />
        <parameter name = "aggregate−function−output−name.6"  value = "y2" />
        <parameter name = "window−size"  value = "2"   />
        <parameter name = "window−size−by"  value = "TUPLES"   />
        <parameter name = "advance"  value = "1"   />
        <parameter name = "group−by"  value = "phone" />
</box>
```

Listing 9: Aggregate operator A definition

The tuples produced by the Aggregate operator A are consumed by the Map operator M_3 in order to compute the speed at which the mobile phone moved between each pair of consecutive calls. The operator is defined as:

$$M_3\{Phone \leftarrow Phone, Time \leftarrow Time, Speed \leftarrow \frac{\sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}}{T_2 - T_1}\}(O_A, O_M)$$

Its representation by means of an XML element are presented in Listing 10. As for the previous two Map operators, we define the output tuples schema by means of elements attributes *expression.#* and *output-field-name.#*. The operator in the example defines three output fields, $Phone$, $Time$ and $Speed$. Field $Speed$ is computed as the division between the Euclidean distance and the temporal distance of consecutive phone calls.

```
<box name="m3" type="map" >
        <in stream="ao"/>
        <out stream="om"/>
        <parameter name="expression.0" value="phone"/>
        <parameter name="output−field−name.0" value="phone"/>
        <parameter name="expression.1" value="time"/>
        <parameter name="output−field−name.1" value="time"/>
        <parameter name="expression.2" value="sqrt(pow(x2−x1,2.0)+pow(y2−y1,2.0))/(t2−t1)"/>
        <parameter name="output−field−name.2" value="speed"/>
</box>
```

Listing 10: Map operator M_3 definition

The last operator defined in the *continuous query* is the Filter operator F. This operator is used to forward only the tuples referring to mobile phones whose speed exceeds a given threshold (110 in

the example). The operator is defined as:

$$F\{Speed \geq 110\}(M_A, O_F)$$

Its representation by means of a XML element is presented in listing 11. As presented in Section 2.1.1.0.2, the Filter operator may define multiple output streams, one for each filtering condition. Furthermore, it can define an extra output stream to forward all the tuples that do not satisfy any filtering condition. In the example, we are interested in forwarding only tuples for which the predicate $Speed \geq 110$ holds. We define this condition using element attribute *expression.0*.

```
<box name="f" type="filter" >
        <in stream="om" />
        <out stream="output" />
        <parameter name="expression.0" value="speed>110" />
</box>
```

Listing 11: Filter operator F definition

Once all the operators of the *continuous query* have been defined, we must specify which are the *continuous query* outputs and their respective schema. In the example, we define the output schema *output_schema* in the same way we define schema *input_schema*. We specify which are the outputs using the *output* element.

```
<schema name="output_schema">
        <field name="phone" type="string" size="9" />
        <field name="time" type="int"/>
        <field name="speed" type="double"/>
</schema>

<output stream = "output" schema = "output_schema"/>
```

Listing 12: Output Schema and Output definition

Once all the operators and the inputs and outputs have been defined, we can decide whether to deploy the *continuous query* on a single Borealis instance (centralized execution) or at multiple instance (distributed execution). To define how to distribute operators, we must provide additional information in the query XML file. With respect to the XML containing the definition of the operators, we must specify which groups of operators will be deployed at the same SPE instance. As an example, we can decide to group together the first three operators M_1, M_2 and U and to group together the remaining operators A, M_3 and F. We provide this information in the XML file setting *box* XML elements as child of the *query* element. Listing 13 presents a possible XML definition of the whole *continuous query*. For the ease of the understanding, we don't repeat the XML definition of each schema or operator presented before.

```
<borealis>

        <schema name="input_schema" ...

        <input stream="input" schema="input_schema" />

        <query name="group1">
                <box name="m1" type="map" ...
                <box name="m2" type="map" ...
                <box name="u" type="union"...
        </query>

        <query name="group2">
                <box name="a" type="aggregate"...
                <box name="m3" type="map" ...
                <box name="f" type="filter" ...
        </query>

        <output stream = "output" schema = "output_schema"/>

        <schema name="output_schema"...

</borealis>
```

Listing 13: Operators groups definition

The information related to the nodes to which each group will be deployed is provided in a separated file, referred to as the *Deploy* file. For each input stream, output stream and group of operators we must define the respective Borealis instance address by means of an IP:Port pair.

```
<deploy>

    <publish endpoint="blade39:15000" stream="input"/>

    <node endpoint="blade39:15000" query="group1"/>
    <node endpoint="blade55:15000" query="group2"/>

    <subscribe endpoint="blade55:25000" stream="output"/>

</deploy>
```

Listing 14: Deployment information

In the example, we specify that *input* will receive tuples at address blade39:15000. The operators belonging to *group1* will be deployed at the same address while the operators belonging to *group2* will be deployed at the Borealis instance running at blade55:15000. Finally, the output stream will output tuples at address blade55:25000.

A.2 Operators extensibility

In this section we provide a short overview about how user-defined operators can be added to the set of available operators to be used to define *continuous queries*. We present this extensibility of the

Borealis SPE as, due to it simplicity, it was one of the motivating choices in using Borealis as the base SPE for *StreamCloud*.

Two main functions must be implemented when defining a new data streaming operator. Function `setup_impl` is used to setup the operator using the parameters defined in the XML query file. Function `run_impl` is invoked by the Borealis SPE scheduler each time (at least) one tuple is available for one of the input streams of the operator. In the following, we focus on a sample operator used to duplicate each incoming tuple $duplicate - number$ times, where parameter $duplicate - number$ is provided in the XML query file. A possible definition of the operator by means of an XML element is presented in Listing 15.

```
<box name="d" type="duplicate" >
        <in stream="in"/>
        <out stream="out"/>
        <parameter name="duplicate−number" value="3"/>
</box>
```

Listing 15: Sample operator to duplicate input tuples

In the example, the operator defines one input stream *in* and one output stream *out*. Each tuple consumed from the input stream will be duplicated 3 times. The following code presents a possible implementation for the `setup_impl` function.

```
void setup_impl() throw (AuroraException) {

        if ( get_num_inputs() != 1 ) {
                Throw(aurora_typing_exception,
                        "Operator Duplicate requires exactly 1 input stream");
        }

        if ( get_num_outputs() != 1 ) {
                Throw(aurora_typing_exception,
                        "Operator Duplicate requires exactly 1 output stream");
        }

        _duplicate_number = param( "duplicate−number", PARAM_NON_EMPTY );

}
```

Listing 16: `setup_impl` implementation

In the example, we define exactly one input stream and one output stream for the operator and we throw an exception if the number of *in* or *out* elements provided in the XML definition is not correct. Subsequently, we store the value of the parameter `duplicate-number` into the variable `_duplicate_number`. We provide the `PARAM_NON_EMPTY` option to the `param` function to specify that the parameter is mandatory. Once the `setup_impl` function has been defined, we can proceed with the definition of the `run_impl` function. We provide a sample implementation of the function in Listing 17.

```
void run_impl(QBoxInvocation& inv) throw (AuroraException) {

    EnqIterator my_enq_iterator = enq(0);

    DeqIterator my_deq_iterator = deq(0);

    while (inv.continue_dequeue_on(my_deq_iterator, 0)) {

        for (uint i=0;i<_duplicate_number;i++) {
            memcpy(my_enq_iterator.tuple(),
                (char*) my_deq_iterator.tuple(),_output_tuple_size);
            ++my_enq_iterator;
        }

        ++my_deq_iterator;
        get_output(0).notify_enq();

    }

}
```

Listing 17: `run_impl` implementation

As shown Listing 17, Borealis provides two objects named `EnqIterator` and `DeqIterator`. Object `EnqIterator` is used to add tuples to the operator output stream queue. Each time a tuple is written to the output queue, the operator ++ is used to increment the position of the queue pointer where tuples are written. Similarly, object `DeqIterator` is used to read tuples from the operator input stream queue. Each time a tuple is read from the input queue, the operator ++ is used to increment the position of the queue pointer to the next available tuple position. Once objects `EnqIterator` and `DeqIterator` have been created, the function proceeds copying each input tuple from the input queue to the output queue `duplicate-number` times. Each time tuples are written to the output queue, the function `notify_enq` is invoked to alert the operators scheduler new tuples are available.

Once functions `setup_impl` and `run_impl` have been implemented, the last step consists in adding the operator name (`duplicate` in the example) to the list of available operators and in recompiling Borealis source code with the new added classes.

A.3 Borealis Tuples Processing Paradigm

As presented in Section 3.3.4, *StreamCloud* scales with respect to the available number of machines but also with respect to the number of cores available in each machine. To better understand why *StreamCloud* scales with the number of available cores, we provide in this section an overview of the Borealis tuple processing paradigm, presenting how tuples are processed and transferred across nodes and discussing how the original Borealis SPE paradigm has been improved in *StreamCloud*.

As introduce in the Borealis application programmer's guide [bora] and in the Borealis developer's guide [borb] and as analyzed in [AMB12], the Borealis SPE defines 4 main threads to process

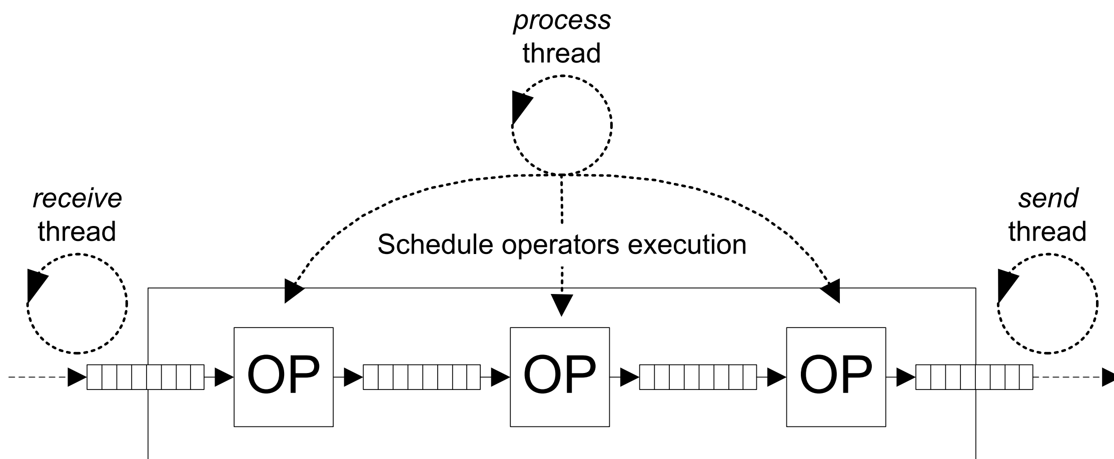


Figure A.2: *StreamCloud* threads interaction

incoming tuples. The first thread, referred to as *receive* in [AMB12], is used to take tuples from the network and to store them in the SPE instance input queues. Tuples are exchanged between instances as *events* that can contain single or multiple tuples (*batching*). Once tuples are available in the SPE instance input queues, a second thread, referred to as *process*, is in charge of scheduling the operators deployed at the SPE instance in order for them to process all the input tuples and produce the respective output tuples. Between each pair of operators, tuples are stored using inter-operator queues. Tuple produced by the last operators of the local query deployed at each SPE instance (i.e., operators forwarding tuples to operators that are deployed at different SPE instances or forwarding tuples to the end user application) are processed by the *prepare* thread, in charge of serializing them back to events. Finally, serialized events that must be forwarded to other SPE instances or to the end user application are sent back to the network by the *send* thread.

During the design and the development of *StreamCloud* starting from the Borealis SPE, the paradigm for tuple processing has been improved. One of the modification applied to the original Borealis SPE was the removal of the *prepare* thread, whose functionalities have been added to the *send* thread. After this modification, tuple processing is now managed by a thread that takes tuples from the network, a thread used to process tuples locally and a thread used to send output tuples back to the network. Figure A.2 presents *StreamCloud* tuples processing paradigm.

In order to understand why *StreamCloud* scales with respect to the number of available cores several considerations must be made about the three threads defined to process input tuples. The first consideration is related to the fact that, even if all the threads could run at the same time, there is a clear dependency between them. Thread *process* will run only after thread *receive* will have added at least one tuple to the SPE input queue. Similarly, thread *send* will run only after thread *process* will have scheduled and run the operator whose output tuples must be forwarded to a different node. The second consideration is related to the CPU cycles consumed by each thread. A first observation

is that the *process* thread usually consumes more CPU threads than the other two. The reason is that the CPU consumption is not only related to the reading and writing of tuples across operators queues, but also related to the number of operators running at the SPE instance, and the computation run by each of them. As presented in Section 3.2.1, when running a parallel-distributed *continuous query* each local subquery is enriched with an input merger and a load balancer, that is, at least 3 operators are managed by the *process* thread. A second observation is that, even if the *receive* and *send* threads CPU consumption should be comparable (deserialization and serialization of tuples), the CPU consumption of the *send* thread is usually lower than the one of the *receive* one. The reason is that the *selectivity* (i.e. the output stream rate divided the input stream rate) is usually lower than one. That is, the tuples produced by a query are usually less than the consumed ones. This is due to the nature of data streaming applications, where a huge amount of data is processed in order to extract only useful information or to generate alarms. The last consideration to justify why the threads processing tuples do not exceed the consumption of approximately a single CPU is related to how they interact when a *StreamCloud* instance is close to saturation (i.e., when the instance cannot cope with the incoming load). In order to avoid the memory saturation at the node where the SPE instance is running, inter-operator queues define a maximum number of tuples to be stored. When this maximum size is reached, flow control will avoid the processing of new tuples. This limitation on the queues sizes implies that, under bursty conditions, the *receive* thread will add tuples to the input queue at a faster rate than the *process* thread one. Upon saturation, the *receive* thread will start waiting for the *process* thread to schedule operators in order to free the input buffer, therefore reducing dramatically its CPU consumption.

A.4 Borealis Application Development Tools

In this section we provide an overview of the tools provided by Borealis to develop applications running *continuous queries*. As discussed in Chapter 6, it is important to provide tools to ease the interaction of the user with the SPE. Even if the tools originally provided by Borealis have been re-designed and re-implemented in order to provide a better IDE for *StreamCloud*, we provide a brief description of them as they help understanding which are the steps a user performs to convert a *continuous queries* into an application.

The main tools provided by Borealis are the *Marshal* and the *BigGiantHead*. The *Marshal* tool has been introduced to generate code to send and receive tuples starting from a query and Deploy XML files. The schema of the input and output streams (i.e., the fields composing the input and the output tuples) depend on the particular *continuous query* being run. The code generated by the *Marshal* tool provides `send` functions for each input stream defined by the *continuous query* and `receive` functions for each output stream. The code that must be provided by the user is related

to how to create input tuples (e.g., reading them from a file) and what to do with output tuples (e.g., store them in a file). The limitation of the code generated by the *Marshal* tool is due to the fact that both receiver and sender functions are designed to run at the same application instance. In the case of a distributed setup involving many nodes and running a *continuous query* with multiple input and output streams, the application might constitute a bottleneck as it cannot sustain the inputs and output rates. Together with the *Marshal* tool, the *BigGiantHead* tool has been provided with Borealis to manage the deployment of *continuous queries*. This tool accepts as input the pair of XML files containing the query and the deployment information and coordinates all the SPE instances sending instructions about which operators to deploy and which input and output streams to subscribe to. The tool can be used to simply deploy a given *continuous query* but can also be executed as a stand alone application that maintains a global catalog of the *continuous queries* being run by Borealis. The *BigGiantHead* tool has two main limitations: (1) the global catalog has not been designed to be updated with information related to dynamic load balancing, elasticity and fault tolerance actions and (2) instructions to deploy operators or subscribe streams are sent to the SPE instances as synchronized actions. When provisioning a new node, or when replacing a failed one, it is fundamental to reduce the deploy time as much as possible. For this reason, in *StreamCloud*, operators are deployed invoking asynchronous functions to reduce the duration of reconfiguration actions.

Together with the *Marshal* and the *BigGiantHead* tools, Borealis provided some bash scripts to automate the deployment and execution of applications running the *continuous queries* defined by the user.

Figure A.3.a presents the steps performed by the user in order to prepare and run the application generated starting from the *continuous query*:

1. The first step consists in the preparation of the XML query and deploy files.

2. When the files have been prepared, the *Marshal* tool is invoked to generate the code with the functions to send to and receive data from the application that will later run the query.

3. The auto generated code must be specialized defining how tuples being sent are created and what to do with output tuples produced by the *continuous query*. As an example, the user might want to read tuples from a file and also to persist received tuples in a file; in this case, he will implement the needed functionalities.

4. When the auto-generated code has been enriched with the user-defined functions, the application is compiled and linked into an executable.

5. The last step consists in preparing the script used to execute the application. The information provided by the user is basically related to which SPE instance should be activated in order to deploy the query.

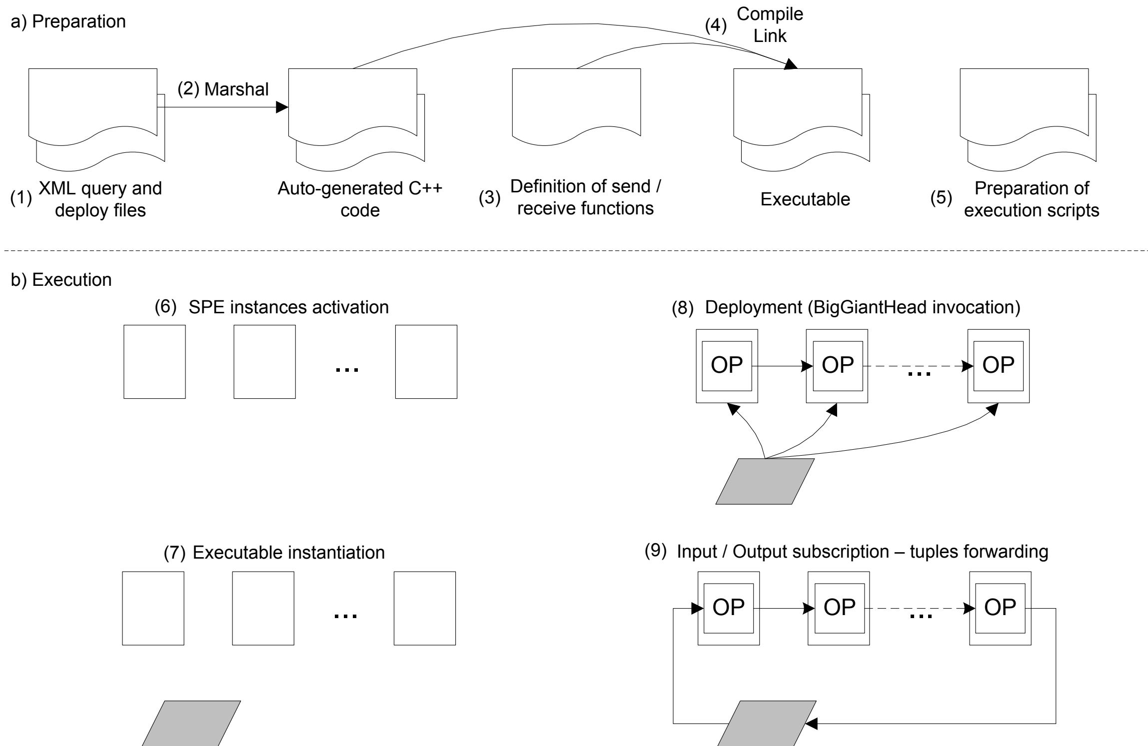


Figure A.3: Steps performed by the user to create an application starting from a *continuous query*

Once the application has been prepared, the script is executed. The steps followed by the script are presented in Figure A.3.b:

6. The first step consists in the instantiation of all the SPE instances specified by the user at step (5). At this point, each SPE instance has no operator deployed at it and is waiting for instruction about what to deploy and to which streams to subscribe.

7. Subsequently, the script activates the executable created at step (4). From this point on, the deployment and the execution of the *continuous query* will be managed by the application.

8. The executable starts deploying the query invoking the *BigGiantHead* tool.

9. Finally, the application subscribes to the query output streams and starts injecting tuples to the *continuous query* input streams. As discussed previously, all the tuples being sent by the application and all the tuples being received are processed at the same time by the application.

The application runs until all the tuples have been sent (e.g., if the user is reading them from a file) or until it is killed by the user.

Bibliography

[AAB+05a] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[AAB+05b] 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 *In CIDR*, 2005.

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. Springer, 2004.

[ABC+05] Yanif Ahmad, Bradley Berg, Ugur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stan Zdonik. Distributed operation in the borealis stream processing engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, New York, NY, USA, 2005. ACM.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2), June 2006.

[ACC+03] Daniel J. Abadi, Don Carney, Ugur Cetintemel, 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), August 2003.

[ACG+04] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04. VLDB Endowment, 2004.

[ali] Alienvalut LLC. Alienvault documentation. `http://www.alienvault.com/documentation/`.

[Ama] Amazon EC2. `http://aws.amazon.com/ec2/`.

[AMB12] Shoaib Akram, Manolis Marazakis, and Angelos Bilas. Understanding and improving the cost of scaling distributed event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, New York, NY, USA, 2012. ACM.

[BBC+04] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stan Zdonik. Retrospective on aurora. *The VLDB Journal*, 13(4), December 2004.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, New York, NY, USA, 2002. ACM.

[BBMD03] 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, New York, NY, USA, 2003. ACM.

[BBMS04] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Mike Stonebraker. Availability-consistency trade-offs in a fault-tolerant stream processing system. Technical report, 2004.

[BBMS08] Magdalena Balazinska, Hari Balakrishnan, Samuel R Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1), March 2008. ACM ID: 1331907.

[BBS04] Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker. Contract-based load management in federated distributed systems. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[BDD+10] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. SECRET: a model for analysis of the execution semantics of stream processing systems. *Proc. VLDB Endow.*, 3(1-2), September 2010.

[BDM04] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 350 – 361, march-2 april 2004.

[BFF09a] A. Brito, C. Fetzer, and P. Felber. Multithreading-enabled active replication for event stream processing operators. In *28th IEEE International Symposium on Reliable Distributed Systems, 2009. SRDS '09*, September 2009.

[BFF09b] Andrey Brito, Christof Fetzer, and Pascal Felber. Minimizing latency in fault-tolerant distributed stream processing systems. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, Washington, DC, USA, 2009. IEEE Computer Society.

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management*, MDM '01, London, UK, UK, 2001. Springer-Verlag.

[bora] The Borealis Application Programmer's Guide. `http://www.cs.brown.edu/research/borealis/public/publications/borealis_application_guide.pdf`.

[borb] The Borealis Developer's Guide. `http://www.cs.brown.edu/research/borealis/public/publications/borealis_developer_guide.pdf`.

[Borc] The Borealis Project. `http://www.cs.brown.edu/research/borealis/public/`.

[BS07] Gert Brettlecker and Heiko Schuldt. The OSIRIS-SE (stream-enabled) infrastructure for reliable data stream management on mobile devices. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, New York, NY, USA, 2007. ACM.

[BSS06] Gert Brettlecker, Heiko Schuldt, and Hans-Jorg Schek. Efficient and coordinated checkpointing for reliable distributed data stream management. In Yannis Manolopoulos, Jaroslav Pokorny, and Timos Sellis, editors, *Advances in Databases and Information Systems*, volume 4152 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006.

[cai] The Cooperative Association for Internet Data Analysis. `http://www.caida.org/home/`.

[CBB+03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *In CIDR*, 2003.

[CC09] Rebecca L. Collins and Luca P. Carloni. Flexible filters: load balancing through back-pressure for stream programs. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, New York, NY, USA, 2009. ACM.

[CCC+02] Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02. VLDB Endowment, 2002.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, New York, NY, USA, 2003. ACM.

[CCR+03] Don Carney, Ugur Cetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03. VLDB Endowment, 2003.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for internet databases. *SIGMOD Rec.*, 29(2), May 2000.

[cmt] Spanish Commission for the Telecommunication Market - Comisión del Mercado de las Telecomunicaciones. `http://www.cmt.es/inicio`.

[CSA06] null Jin Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of backend databases in dynamic content web servers. In *Autonomic Computing, International Conference on*, volume 0, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[CWY05] Yun Chi, Haixun Wang, and Philip S. Yu. Loadstar: Load shedding in data stream mining. In *In Proc. Int. Conf. on Very Large Data Bases (VLDB*, 2005.

[Des04] Amol Deshpande. An initial study of overheads of eddies. *SIGMOD Rec.*, 33(1), March 2004.

[DFF+98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. *A Query Language for XML*. 1998.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), January 2008.

[DM07] Mayur Datar and Rajeev Motwani. The sliding-window computation model and results. In Charu C. Aggarwal and Ahmed K. Elmagarmid, editors, *Data Streams*, volume 31 of *The Kluwer International Series on Advances in Database Systems*. Springer US, 2007.

[Espa] Esper. `http://esper.codehaus.org/`.

[espb] Esper - Complex Event Processing. `http://esper.codehaus.org/`.

[Euc] Ubuntu Eucalyptus. `http://www.ubuntu.com/cloud`.

[GJPPM+12] 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*, 99(PrePrints), 2012.

[GJPPMV10] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Patrick Valduriez. Streamcloud: A large scale data streaming system. In *ICDCS 2010: International Conference on Distributed Computing Systems*, pages 126–137, June 2010.

[GPYC08] Xiaohui Gu, Spiros Papadimitriou, Philip S. Yu, and Shu-Ping Chang. Toward predictive failure management for distributed stream processing systems. In *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, ICDCS '08, Washington, DC, USA, 2008. IEEE Computer Society.

[Gra90] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. *SIGMOD Rec.*, 19(2), May 1990.

[GSP+] Anastasios Gounaris, Jim Smith, Norman W. Paton, Rizos Sakellariou, Alvaro A. A. Fernandes, and Paul Watson. Adaptive workload allocation in query processing in autonomous heterogeneous environments. *Distrib. Parallel Databases*.

[GTY+] Bu ?gra Gedik, Ibm Thomas, Philip S. Yu, Henrique Andrade, Ibm Thomas, Myungcheol Doo, and Kun-lung Wu. Spade: the system s declarative stream processing engine. In *in SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*.

[GZY+09] Yu Gu, Zhe Zhang, Fan Ye, Hao Yang, Minkyong Kim, Hui Lei, and Zhen Liu. An empirical study of high availability in stream processing systems. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[HBR+03] Jeong-hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. A comparison of stream-oriented high-availability algorithms. Technical report, Brown CS, 2003.

[HBR+05] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Uğur Çetintemel, Michael Stonebraker, and Stan Zdonik. High-Availability algorithms for distributed stream processing. In *Data Engineering, International Conference on*, volume 0, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[HCCZ08] Jeong-Hyon Hwang, Sanghoon Cha, Ugur Cetintemel, and Stan Zdonik. Borealis-r: a replication-transparent stream processing system for wide-area monitoring applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, New York, NY, USA, 2008. ACM.

[HCZ07] Jeong-Hyon Hwang, U. Cetintemel, and S. Zdonik. Fast and reliable stream processing over wide area networks. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*, April 2007.

[Hei11] Thomas Heinze. Elastic complex event processing. In *Proceedings of the 8th Middleware Doctoral Symposium*, MDS '11, New York, NY, USA, 2011. ACM.

[HXCZ07] Jeong-Hyon Hwang, Ying Xing, U. Cetintemel, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *IEEE 23rd International Conference on Data Engineering, 2007. ICDE 2007*, April 2007.

[inf] IBM Infosphere. `http://www-01.ibm.com/software/data/infosphere/`.

[JAA+06] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, New York, NY, USA, 2006. ACM.

[JHCF02] Mehul Shah Joseph, Joseph M. Hellerstein, Sirish Ch, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *In ICDE*, 2002.

[JMSS07] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. Query-aware sampling for data streams. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*, April 2007.

[KBG08] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *Proc. VLDB Endow.*, 1(1), August 2008. ACM ID: 1453920.

[LHKK12] 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*, EDBT-ICDT '12, New York, NY, USA, 2012. ACM.

[lin] Microsoft LINQ. `http://msdn.microsoft.com/en-us/library/bb397926`.

[Mil03] David L. Mills. A brief history of ntp time: memoirs of an internet timekeeper. *Computer Communication Review*, 33(2):9–21, 2003.

[MPH10a] Christopher McConnell, Fan Ping, and Jeong-Hyon Hwang. Detouring and replication for fast and reliable internet-scale stream processing. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, New York, NY, USA, 2010. ACM.

[MPH10b] Christopher McConnell, Fan Ping, and Jeong-Hyon Hwang. iFlow: an approach for fast and reliable internet-scale stream processing utilizing detouring and replication. *Proc. VLDB Endow.*, 3(1-2), September 2010.

[msi] Microsoft StreamInsight. `http://msdn.microsoft.com/en-us/library/ee362541.aspx`.

[MWA+02] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation ina data stream management system. http://ilpubs.stanford.edu:8090/549/, 2002.

[NDM+01] Jeffrey Naughton, David Dewitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, and Stratis Viglas. The niagara internet query system. *IEEE Data Engineering Bulletin*, 24, 2001.

[Nil] The Nilson Report. `http://www.nilsonreport.com/`.

[Ope] OpenNebula. `http://opennebula.org/`.

[OPR] Options Price Reporting Authority. `http://www.opradata.com/`.

[ÖV11] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[pos] PostgreSQL. `http://www.postgresql.org/`.

[PS06] Kostas Patroumpas and Timos Sellis. Window specification over data streams. In Torsten Grust, Hagen Höpfner, Arantza Illarramendi, Stefan Jablonski, Marco Mesiti, Sascha Müller, Paula-Lavinia Patranjan, Kai-Uwe Sattler, Myra Spiliopoulou, and Jef Wijsen, editors, *Current Trends in Database Technology EDBT 2006*, volume 4254 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006.

[SAG06] Gokul Soundararajan, Cristiana Amza, and Ashvin Goel. Database replication policies for dynamic content applications. In *In EuroSys06*. ACM, 2006.

[SAG+09] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, Washington, DC, USA, 2009. IEEE Computer Society.

[ScZ05] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4), December 2005.

[SHB04] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, New York, NY, USA, 2004. ACM.

[SM10] Z. Sebepou and K. Magoutis. Scalable storage support for data stream processing. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2010.

[SM11] Z. Sebepou and K. Magoutis. CEC: continuous eventual checkpointing for data stream processing operators. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, June 2011.

[sto] Storm Project. `http://storm-project.net/`.

[STRa] Standford Stream Data Manager. `http://infolab.stanford.edu/stream/`.

[strb] StreamBase. `http://www.streambase.com/`.

[Strc] StreamBased Systems, Inc. `http://www.streambase.com/`.

[SZS+03] Stan Zdonik Sbz, Stan Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur C. Etintemel, Magdalena Balazinska, and Hari Balakrishnan. The aurora and medusa projects. *IEEE Data Engineering Bulletin*, 26, 2003.

[Tat02] Nesime Tatbul. QoS-Driven load shedding on data streams. In *Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, EDBT '02, London, UK, UK, 2002. Springer-Verlag.

[TcZ+03] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 309–320. VLDB Endowment, 2003.

[TCZ07] Nesime Tatbul, Ugur Cetintemel, and Stan Zdonik. Staying FIT: efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07. VLDB Endowment, 2007.

[TLPY06] Yi-Cheng Tu, Song Liu, Sunil Prabhakar, and Bin Yao. Load shedding in stream databases: a control-based approach. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06. VLDB Endowment, 2006.

[TP06] Yi-Cheng Tu and Sunil Prabhakar. Control-based load shedding in data stream management systems. In *Proceedings of the 22nd International Conference on Data Engineering Workshops*, ICDEW '06, Washington, DC, USA, 2006. IEEE Computer Society.

[TZ06a] N. Tatbul and S. Zdonik. Dealing with overload in distributed stream processing systems. In *22nd International Conference on Data Engineering Workshops, 2006. Proceedings*, 2006.

[TZ06b] Nesime Tatbul and Stan Zdonik. Window-aware load shedding for aggregation queries over data streams. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06. VLDB Endowment, 2006.

[XHCZ06] Ying Xing, Jeong-Hyon Hwang, Ugur Cetintemel, and Stan Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06. VLDB Endowment, 2006.

[XZH05] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, Washington, DC, USA, 2005. IEEE Computer Society.

[yah] Yahoo S4. http://incubator.apache.org/s4/.

[ZGY+10] Zhe Zhang, Yu Gu, Fan Ye, Hao Yang, Minkyong Kim, Hui Lei, and Zhen Liu. A hybrid approach to high availability in stream processing systems. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, Washington, DC, USA, 2010. IEEE Computer Society.

[zoo] Apache Zookeeper. http://zookeeper.apache.org/.

Funding - Acknowledgments

This Ph.D. has been partially funded by the following research projects:

Stream: Scalable Autonomic Streaming Middleware for Real-time Processing of Massive Data Flows (FP7-216181)
Funding Programme: Seventh European Framework (FP7) (2008-2011)

MASSIF: Management of Security Information and Events in Services Infrastructures (FP7-257475)
Funding Programme: Seventh European Framework (FP7) (2010-2013)

IOLanes: Advancing the Scalability and Performance of I/O Subsystems in Multi-core Platforms (FP7-248615)
Funding Programme: Seventh European Framework (FP7) (2010-2013)

CLOUDS: Cloud Computing para Servicios Escalables, Confiables y Ubicuos (S2009TIC-1692)
Funding Programme: Comunidad Autónoma de Madrid (2010-2013)

CloudStorm: Scalable and Dependable Cloud Service Platforms (TIN2010-19077)
Funding Programme: Ministry of Science and Innovation (MICINN) (2010-2013)

Highly Scalable Platform for the Construction of Dependable and Ubiquitous Services (TIN2007-67353-C02)
Funding Programme: Ministry of Education and Science (MEC) (2010-2013)