Technical University of Crete
School of Electrical and Computer Engineering

A Distributed Complex Event Processing (CEP) System Based on the Esper Engine

---

Konstantinos Kyriakopoulos

Thesis Committee:
Professor Antonios Deligiannakis (Supervisor)
Professor Vasilis Samoladas
Professor Minos Garofalakis

September 24th, 2018

Abstract

The evolution of Big Data in the recent years has been posing continuous challenges to the Data Science field. More specifically, the multiple sources of information along with the continuous growth and fast transmission of data have raised the need for real-time data analysis. In order to fill this need complex event processing has emerged.

Complex event processing is a technique used for analyzing multiple streams of data in a pattern-based manner. The concept behind complex event processing is the establishment of relationships between events of information and the correlation of the events for specific patterns. Events may derive from multiple streams so CEP fulfills the need for distributed event detection.

In this thesis, a distributed complex event processing (CEP) system based on the Esper engine is implemented. Esper is an open-source software suitable for complex event processing and real time data analysis. The Esper engine lies on top of the Ferari project which serves as a framework for distributed streaming processing and utilizes the functionalities of various platforms such as the Apache Storm.The goal of this work is to integrate the Esper engine into the Ferari project and conduct experiments while taking advantage of Ferari's services and components. These experiments demonstrate Esper's capabilities and its efficiency as a CEP engine.

Περίληψη

Η επανάσταση των μεγάλων δεδομένων τα τελευταία χρόνια διαρκώς θέτει προκλήσεις στην επιστήμη των δεδομένων. Ειδικότερα, οι πολλαπλές πηγές πληροφοριών σε συνδυασμό με τη διαρκή ανάπτυξη και γρήγορη μετάδοση των δεδομένων, έχουν δημιουργήσει την ανάγκη για ανάλυση των δεδομένων σε πραγματικό χρόνο. Προκειμένου να καλυφθεί αυτή η ανάγκη, ήρθε στο προσκήνιο η πολύπλοκη επεξεργασία δεδομένων.

Η πολύπλοκη επεξεργασία δεδομένων είναι μία τεχνική που χρησιμοποιείται για την ανάλυση πολλαπλών ροών δεδομένων βασιζόμενη σε ανίχνευση μοτίβων. Η αντίληψη πίσω από την πολύπλοκη επεξεργασία δεδομένων, είναι η εφαρμογή συσχετισμών μεταξύ γεγονότων πληροφοριών και η σύνδεση των γεγονότων για συγκεριμένα μοτίβα. Τα γεγονότα πληροφοριών ενδεχομένως να προέρχονται από πολλαπλές πηγές πληροφορίας, οπότε η πολύπλοκη επεξεργασία δεδομένων εξυπηρετεί την κατανεμημένη ανίχνευση γεγονότων.

Στην παρούσα διπλωματική, έχει υλοποιηθεί ένα κατανεμημένο σύστημα πολύπλοκης επεξεργασίας δεδομένων βασιζόμενο στη μηχανή Esper. Η συγκεκριμένη μηχανή είναι ένα λογισμικό ανοικτού κώδικα, κατάλληλο για πολύπλοκη επεξεργασία δεδομένων και ανάλυση σε πραγματικό χρόνο. Η μηχανή Esper έχει εγκατασταθεί πάνω στην υποδομή του Ferari που λειτουργεί ως ένα σύστημα κατανεμημένης επεξεργασίας ροών δεδομένων και το οποίο με τη σειρά του αξιοιοποιεί τις δυνατότητες διαφόρων συστημάτων όπως το Apache Storm. Σκοπός αυτής της δουλειάς είναι η ενσωμάτωση της μηχανής Esper στο σύστημα Ferari και η διεξαγωγή πειραμάτων αξιοποιώντας τις υπηρεσίες και τις δυνατότητες των μηχανισμών του Ferari. Τα πειράματα αυτά αναδεικνύουν τις δυνατότητες της μηχανής Esper και την αποδοτικότητα της ως μια μηχανή πολύπλοκης επεξεργασίας δεδομένων.

Acknowledgments

First and foremost, I would like to thank my family, my parents and my brother for their contribution in the completion of my studies. Their unconditional support, both emotional and material, has been the cornerstone of my academic development. I dedicate this work to them because their love and faith in me all these years, helped me try harder in any circumstances.

Of course, I would like to express my gratitude towards Professor Antonios Deligiannakis who supervised this thesis. It was his personality and his teaching method that inspired me to get involved with the Data Science field. For the trust he showed in me by assigning me this topic, his advices and mentorship, I will be forever grateful.

Furthermore, I would like to thank Professor Minos Garofalakis and Professor Vasileios Samoladas for the priceless knowledge they offered me through their courses as well as for the time they spent reviewing this thesis.

Last but not least, i need to express my thanks towards Ioannis Flouris and Vasiliki Manikaki for the assistance and the advices they provided me. I am grateful for the time they spent, trying to help me overcome any obstacles that occured in this work.

"What you get by achieving your goals

is not as important as what you become

by achieving your goals."

Henry David Thoreau

Contents

List of Figures

Chapter 1

Introduction

The rapid development of various IoT applications in the past years has altered the concept behind data analysis. In the past years,before the explosion of Big Data, the main method for handling information had been the storing and retrieval of static data through the use of conventional databases. Today, various applications such as applications in finance and business, monitoring applications and applications for sensor networks require faster and more elegant techniques for data processing.

Complex event processing (CEP) is the term which refers to the conduction of real-time data analysis and is used to describe the technique of detecting patterns in streams of events. Important factors in applications such as the throughput, the latency and the complexity in the detection patterns played an important role in the development of this technique. For instance, these applications need to be able to cope with high throughput which implies processing huge amounts of messages per second. Also, they need to have low latency which indicates fast responses from the applications. Complex detection patterns refer to the complex ways that the information is combined. CEP engines are built to match these exact needs.

Esper is an engine suitable for complex event processing and streaming analytics. The engine guarantees memory efficiency, low latency, in-memory computing and high scalability. Also, Esper uses its own event processing language (EPL) which is based on the SQL syntax. These characteristics have classified Esper as one of the leading complex event processing engines.

1.1 Thesis Contribution

The objective of this work is to provide a solid, reliable and robust system architecture which supports complex event processing over distributed sites, supporting the well known push-pull paradigm, and based on the Esper engine. The basis of the system is the FERARI project which supports stream processing in distributed network topologies. During the system initialization, in order for the FERARI's components to communicate effectively with Esper, a wide range of configurations

take place in the CEP engine.
Several features of the engine such as Event Representations, Data Windows, EPL queries and others are presented during the experiments. The following chapters inlcude all the necessary, theoretical and practical, information about the functionality of Esper and the required preferences for its integration into the Ferari project.

1.2 Thesis Outline

The thesis is organized in the following chapters:

Chapter 2 describes the concept behind the functionality of the Esper engine and analyses many of its features.

Chapter 3 presents the architecture of the FERARI implementation,introduces the frameworks used in the project and provides an insight into its various components.

Chapter 4 contains the necessary configurations for Esper's integration into the Ferari project. It includes the system design and several experimental results.

Chapter 5 concludes the thesis.

Chapter 2

Esper

2.1 Overview

Esper is designed to operate in a continuous manner with the use of queries which act as filters for the incoming data. Rather than stocking the information and running queries against static data, the engine stores queries and processes the data through them. Event stream queries and event patterns are the two fundamental techniques which are provided by the engine.

The event stream queries technique supports aggregation, joining, various functions and windows which are useful for streaming analysis. This method uses the EPL language which is based on the SQL syntax. However, the languages differ in the use of views and tables as EPL is built to use views for data structuring. The event pattern method utilizes the detection of specific patterns and sequences in the processed data. Raw or combined events and the existence of patterns in the events are exploited by the technique.

2.2 Event Representations

The term event type is used by Esper to describe the type information available for an event representation. An event is an immutable record of a past occurrence of an action or state change. Event properties capture the state information for an event.

2.2.1 Event Underlying Objects

An event can be represented by any of the following underlying Java objects:

- **java.lang.Object**

  Any Java POJO (plain-old java object) with getter methods following JavaBean conventions. Legacy Java classes which are not following JavaBean conventions can also serve as events.

- **java.util.Map**

  Map events are implementations of the java.util.Map interface where each map entry is a propery value.

- **Object[] (array of object)**

  Object-array events are arrays of objects (type Object[]) where each array element is a property value.

- **Application classes**

  Plug-in event representation via the extension API.

- **org.w3c.dom.Node**

  XML document object model (DOM).

- **org.apache.axiom.om.OMDocument or OMElement**

  XML - Streaming API for XML (StAX) - Apache Axiom (provided by EsperIO package).

It is important to note that all event representations support nested, indexed and mapped properties. In fact,the nesting level is unlimited. The representations allow transposing the event itself and parts of all of its property graph into new events. The term transposing refers to selecting the event itself or event properties that are themselves nestable property graphs, and then querying the event's properties or nested property graphs in further statements.

All event representations provide event type metadata. Type metadata for nested properties is also included. Supertypes are allowed by the Java object,Object-array and Map representations.

2.2.2 Event Properties

The state information for an event are captured by Event Properties. These properties can be either simple or indexed or mapped or nested. The following table depicts the various types of event properties and their syntax. A short description of each type,its syntax and an example are inluded in the table. Queries against Map events,XML structures and JavaBean object graphs can be executed with this syntax.

Table 2.1: Event Properties

Type	Description	Syntax	Example
Simple	A property that has a single value that may be retrieved.	name	itemId
Indexed	An indexed property stores an ordered collection of objects (all of the same type) that can be individually accessed by an integervalued, non-negative index (or subscript).	name[index]	item[0]
Mapped	A mapped property stores a keyed collection of objects (all of the same type).	name('key')	item('pen')
Nested	A nested property is a property that lives within another property of an event.	name.nestedname	item.price

Source: Reprinted from the Esper Reference Documentation.

2.3 Processing Model

The *UpdateListener* method is responsible for delivering the processed data from the Esper engine to the listeners of a specific query. The results are encapsulated in *EventBean* instances. Specific getter methods are used for the retrieval of the results from the *EventBean* instance. These methods are the following:

- **get**(String propertyName):Object

  The *get* method can be used to retrieve result columns by name.

- **getUnderlying()**:Object

  The *getUnderlying* method allows update listeners to obtain the underlying event object.

- **getEventType()**:EventType

  The *getEventType* method provides metadata for the event.

2.3.1 The Select Clause

A basic statement which detects all events with a given name and processes them to the listeners via the *UpdateListener* method is the *Select* statement:

```
Select * from Transactions
```

This statement singles out every incoming Transaction event without the use of a window or a filter clause. Its functionality is depicted below:

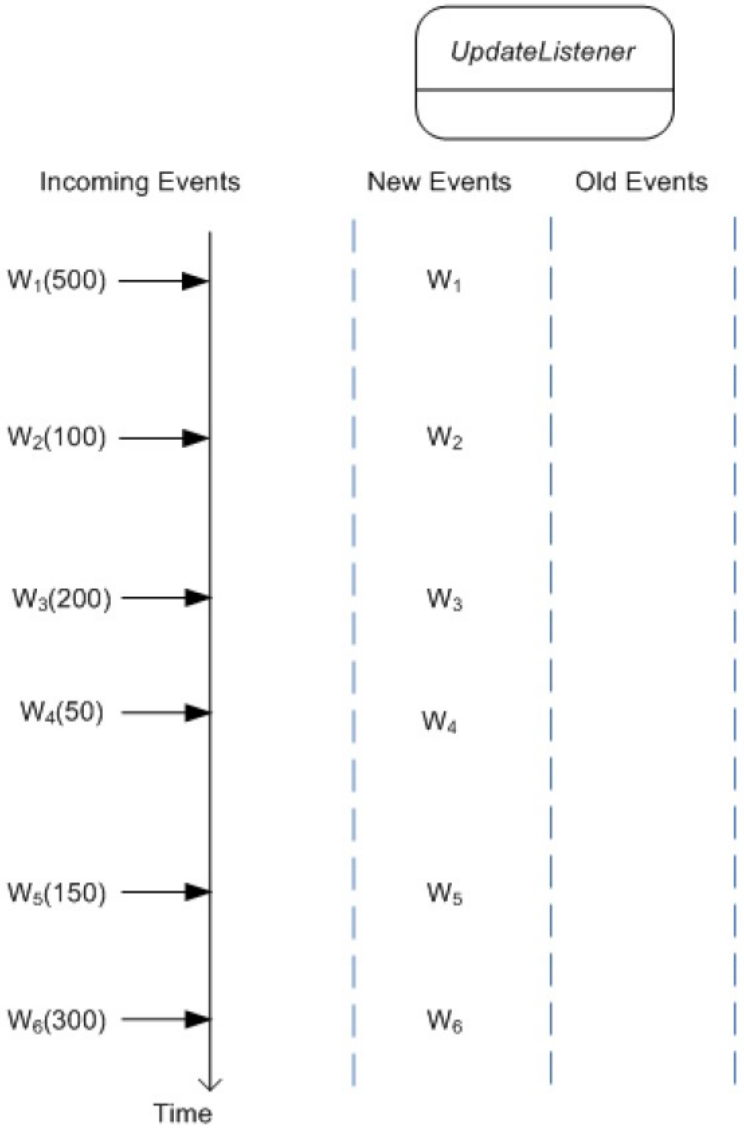

Figure 2.1: Functionality of the *Select* statement.
Source: Reprinted from the Esper Reference Documentation.

2.3.2 Windows

The *Windows* concept is used to identify old and new events. Events which enter a window are classified as new events whereas events leaving the window are labelled as old events. Depending on the manner that events enter and leave the windows,there are certain window categories:

Length Windows

Length windows are designed to maintain the last N events for a stream. The following statement is used to determine a length window onto the Transactions event stream:

```
select * from Transactions.win:length(5)
```

In the specific statement,the length of the window is defined as five. This implies that five events will be maintained in the window. As long as the number of five events is reached,the FIFO(First In,First Out) method is applied to the window. The event that entered first will exit the window and labelled as old event. On the contrary,the latest event will enter the window and categorised as new event.

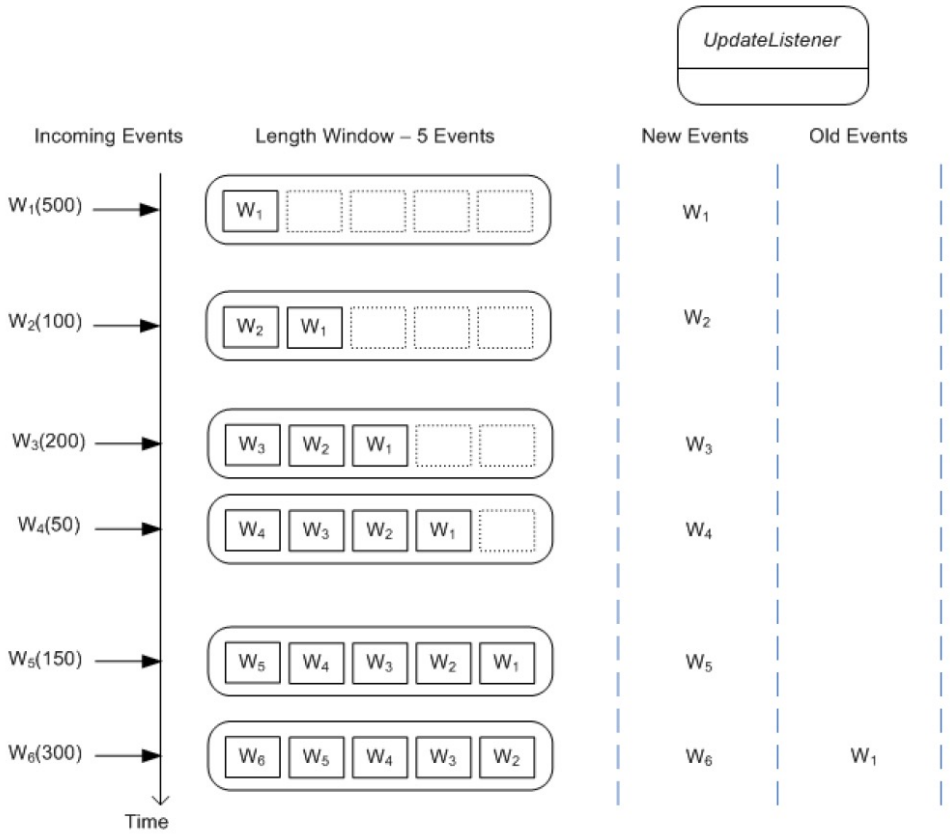

Figure 2.2: Legth Window
Source: Reprinted from the Esper Reference Documentation.

Time Windows

Time windows are built to maintain events for specific time intervals. The following statement is used to determine a time window onto the Transactions event stream:

```
select * from Transactions.win:time(4 sec)
```

According to the above statement,the time window will maintain all the Transactions events which have been detected in the last four seconds. As time goes by,events with a time span that exceeds the four second limit will be pushed out of the window. A time window while processing events is illustrated in the following diagram:

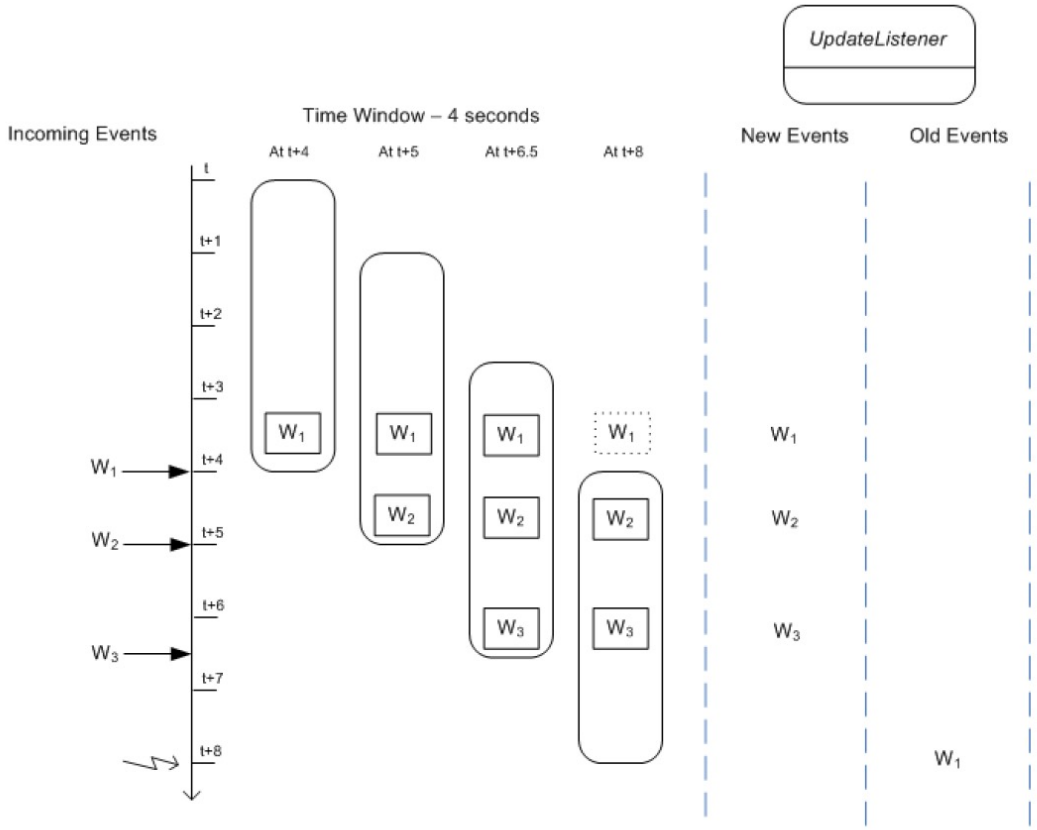

Figure 2.3: Time Window
Source: Reprinted from the Esper Reference Documentation.

Time Batch Windows

Time Batch windows combine the functionality of time and length windows. The window collects all the selected events for a specific time interval and releases all these events once the time period expires. A typical time batch window is presented in the following statement:

```
select * from Transactions.win:time_batch(4 sec)
```

All the events which are detected during the first four seconds will be placed inside the window. After the time interval of four seconds, the collected events will be instantly released and the window will repeat the previous procedure for the specific time period. The functionality of the previously declared time batch window is presented in the following figure:

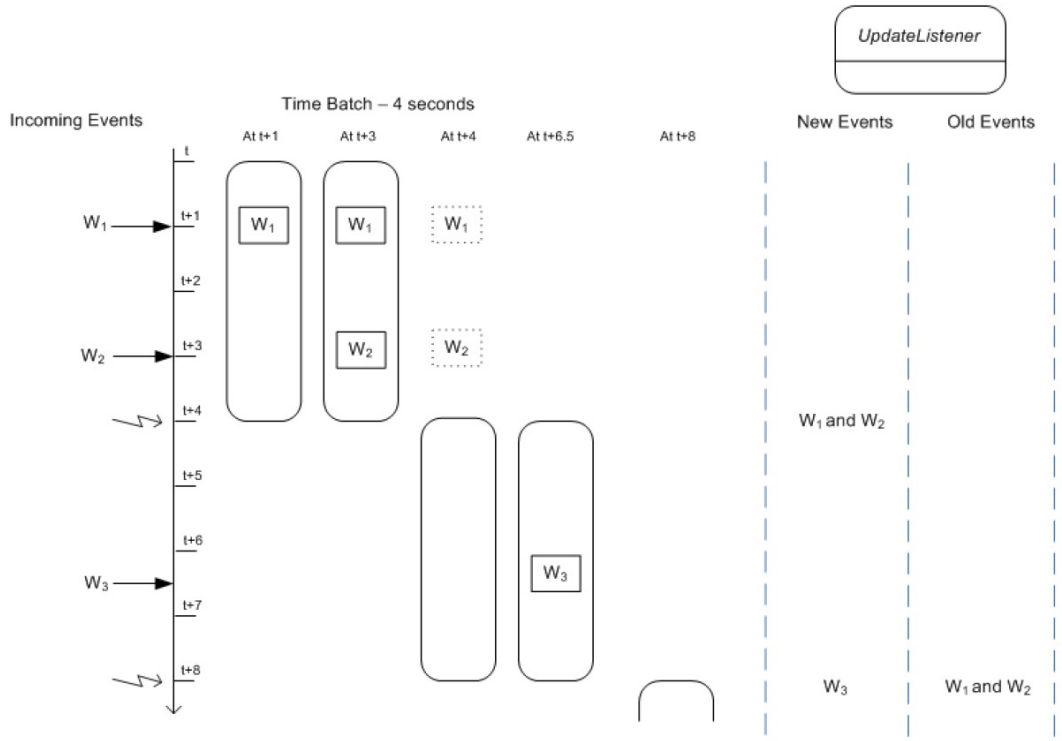

Figure 2.4: Time Batch Window
Source: Reprinted from the Esper Reference Documentation.

2.4 Event Processing Language

Esper uses the Event Processing Language (EPL) which follows the rules of the SQL syntax. It provides SQL-like clauses such as the FROM ,ORDER BY, HAVING, SELECT and WHERE clauses. Combining and handling stream of events is supported by the EPL language as it allows aggregation, filtering and joins.Also, clauses like OUTPUT and PATTERN which are supported by EPL, promote a more elegant approach to event processing.

A key element of the language is the INSERT INTO clause which is used to dispatch events to other streams. Views,like tables in the SQL language, are used to process all the available data and have various representations. They can symbolize a window over a stream of events or depict a group of events or manage certain property values. Further use of the same views is allowed between different EPL statements for better performance.

2.4.1 The EPL Syntax

The structure of an EPL statement is the following:

```
[annotations]
[expression_declarations]
[context context_name]
[into table table_name]
[insert into insert_into_def]
 select select_list
 from stream_def [as name] [, stream_def [as name]] [,...]
[where search_conditions]
[group by grouping_expression_list]
[having grouping_search_conditions]
[output output_specification]
[order by order_by_expression_list]
[limit number_of_rows]
```

The expressions in brackets are optional which means that only the *select* and *from* clauses are necessary for the declaration of an EPL statement. Annotations add information to a query while the Expression Declarations is useful for declaring expressions in the statement. Context declaration is used to bind an EPL statement to a specific context name. The limit and output clauses are useful to determine the number of the rows and the rate at which events are output.

2.4.2 The On Select Clause

The *on select* clause performs a one-time, non-continuous query on a named window or table every time a triggering event arrives or a triggering pattern matches. The query can consider all rows, or only rows that match certain criteria, or rows that correlate with an arriving event or a pattern of arriving events. The syntax for the *on select* clause is as follows:

```
on event_type[(filter_criteria)] [as stream_name]
[insert into insert_into_def]
select select_list
from window_or_table_name [as stream_name]
[where criteria_expression]
[group by grouping_expression_list]
[having grouping_search_conditions]
[order by order_by_expression_list]
```

2.4.3 The Where Clause

The where clause is an optional clause in EPL statements. Event streams can be joined and correlated by utilising the where clause. Also,any expression can be placed in the clause. Typically you would use comparison operators $=, <, >$ $, >=, <=, != , <>$,*is null, is not null* and logical combinations via *and* and *or* for joining, correlating or comparing events. Sample queries with the *where* clause are the following:

```
select iD from Items,orders where items.iD = orders.iD
```

and:

```
select * from Values where measurement > 100
```

2.4.4 The Group By Clause

The group by clause is optional in all EPL statements. The functionality of the *group by* clause is to divide the output of a statement into groups. One or more event property names, or the result of complex expressions can be used in the clause. In the case of aggregate functions, the *group by* retrieves the calculations in each subgroup. Aggregate functions can be avoided in the clause, although they may result into complicated results.The syntax of the clause is presented below:

```
group by aggregate_free_expression [, aggregate_free_expression] [,
    ...]
```

and a typical example of the group by clause:

```
select symbol, sum(price)
from StockTickEvent.win:time(30 sec)
group by symbol
```

2.4.5 The Create Variable Clause

The create variable statement is used to declare a new variable by determining the name and the type of the variable. It ts important to note that variables can be declared during runtime. A variable can be of type from the following:**string, char, character, bool, boolean, byte, short, int, integer, long, double, float, object**.An example of creating a variable is the following statement:

```
create variable integer price = 5
```

Price is the name and integer is the type of the variable according to the above statement. Also, although that this is optional, the variable is initialized to number five.

The constant option allows the declaration of a constant type string as it is shown in the following statement:

```
create constant variable string eventName = 'Transfers'
```

A constant variable with the name eventName, of type string and with value **Transfers** is declared.

2.4.6 The Output Clause

The rate at which events are output and the suppression of output events are handled by the *output* clause. While the EPL language provides several different ways to control output rate, the clause is optional. Below, the syntax for the output clause that specifies a rate in time interval or number of events:

```
output [after suppression_def]
[[all|irs |last|snapshot] every output_rate [seconds|events]]
[and when terminated]
```

For example, the following statement outputs, every 30 seconds, the average temperature from sensor 1 in the 60 minute time window:

```
select avg(temperature) from SensorEvent.win:time(60 min)
where sensorID = "sensor_1"
output snapshot every 30 seconds
```

2.4.7 The Limit Clause

The limit clause is typically used together with the order by and output clause to limit the query results to those that fall within a specified range. It can be used either to receive a range of result rows or to receive the first given number of result rows. The syntax is the following:

```
limit row_count [offset offset_count]
```

where the optional offset count parameter specifies the number of rows that should be skipped at the beginning of the result set. An example:

```
select orderID, count(*) from OrderEvent
group by orderID
output snapshot every 1 minute
order by count(*) desc
limit 10
```

which outputs the top 10 counts per property 'orderID' every 1 minute.

2.4.8 The Having Clause

The *having* clause is used in combination with the *order by* clause. It is used in order to pass or reject events which are defined by the *group* by clause. The way the *where* clause manipulates the *select* clause is identical to the way that the *having* clause handles the *order by* clause. The only difference is the fact that *having* supports aggregate functions while *where* doesn't.

This statement is an example of a *having* clause utilizing an aggregate function. It posts the total price per sensor for the last 10 seconds of flight events for only those flights in which the total value exceeds 300. Also,all sensors where the total price is equal or less then 300 are eliminated.

```
select flightID, sum(value)
from FlightsEvent.win:time(10 sec)
group by flightID
having sum(value) > 300
```

Furthermore, the having clause can combine the conditions with *and, or or not*. The previous statement's having clause is enhanced with an additional condition of an average flight duration bigger than 6 hours:

```
select flightID, sum(value),avg(duration)
from FlightsEvent.win:time(10 sec)
group by flightID
having sum(value) > 300 and duration > 6
```

2.4.9 Subqueries

Esper includes the *subquery* functionality. A subquery is a select statement nested into another select statement. Subqueries are suppported by the Esper engine in the *select, having , where, having* clauses and in stream and pattern filter expressions.

Subqueries are a useful tool to avoid complicated *joins* in the case of complex data processing. Also, compared to complex joins, subqueries promote simplicity and robustness. Esper supports both correlated and simple subqueries. In a simple subquery, the inner query is not correlated to the outer query. An example of a simple subquery is presented below:

```
select ID, (select measurement from CollectedValues.std:lastevent())
   as SensorValues from SensorEvent
```

2.5 Tables

Tables are used by Esper to provide extra functionality for data handling. They do not replace the functionality of Windows in the processing model. Instead,tables are useful for manipulating the data deriving from events and storing them in the proper form in order to be later used in Queries, Subqueries and Fire-And-Forget queries. It is important that there is no syntax to drop or remove a table. Only when the application destroys the statement that creates the table and also destroys all statements referring to the table, is the table removed. The structure supports various SQL functionality such as the *Select*,*Order By*,*Group By* clauses, column naming, aggregation functions and others.

2.5.1 The Create Table Clause

The *create table* statement creates a table. A new table starts up empty and it must be explicitly aggregated-into using into table, or populated by an on-merge statement, or populated by insert into. The syntax for creating a table provides the table name, lists column names and types and designates primary key columns as follows:

```
create table table_name [as] (column_name column_type [primary key]
[,column_name column_type [primary key] [,...]])
```

The *primary key* keyword can be added after each column type in order for the column to be declared as a primary key of the table. Apart from the standard column types like *int*, *long*, *float* and *string*, tables in Esper also support all aggregation functions. For instance, *[simpleCounter count(*)]* is a valid column declaration.

2.5.2 Inserting Into Tables

The insert into clause inserts rows into a table while it is mandatory that the column names and types of the processed event match the declared column names and types of the declared table. An example of populating a table with data from an event:

```
insert into NewTable select column_1, column_2 from NewEvent
```

2.5.3 Select From Tables

Tables support the standard *select from* clause. However, it is important to note that views and the combination of data from a *join* is not supported. A usual select statementcan be:

```
select * from NewTable
```

while the statements:

```
select * from NewTable.win:time(30 sec)
```

and:

```
select * from IntrusionCountTable unidirectional, MyEvent
```

are invalid.

2.6 Functions

Esper includes the use of functions. More specifically, the CEP engine, supports Single-Row, Aggregation and User-Defined functions.

2.6.1 Aggregation Functions

Aggregation functions take into consideration value points or sets of events. In order for the result-set to be grouped by one or more columns, aggregation functions are combined with the *group by* clause. The syntax and overview of some aggregation functions are presented below:

- The ***count(*)*** function.

  This function returns a value of long type which corresponds to the number of events.

- The ***count([all— distinct] expression)*** function.

  This function returns a value of long type which represents the number of the (distinct) non-null values in the expression.

- The ***sum([all— distinct] expression)*** function.

  This function returns the sum of the,distinct,values in the expression.

- The ***avedev([all— distinct] expression)*** function.

  This function returns a value of double type,representing the mean deviation of the (distinct) values in the expression.

- The ***avg([all— distinct] expression)*** function.

  This function returns a value of double type,representing the average of the (distinct) values in the expression.

- The ***max([all— distinct] expression)*** function.

  This function calculates the highest (distinct) value in the expression, returning a value of the same type as the expression itself returns.

- The ***min([all— distinct] expression)*** function.

  This function calculates the lowest (distinct) value in the expression, returning a value of the same type as the expression itself returns.

- The ***median([all— distinct] expression)*** function.

  This function calculates the median (distinct) value in the expression, returning a value of double type. Note that double Not-a-Number (NaN) values are ignored during the computation.

- The ***stddev([all— distinct] expression)*** function.

  This function calculates the standard deviation of the (distinct) values in the expression, returning a value of double type.

2.6.2 User-Defined Functions

User-defined functions are a single-row function that can be called upon either within an expression or as an expression itself. The function reference is determined at statement creation time by the Esper engine and must be a public static (Java) method. Note that Java class names have to be fully qualified (e.g. java.lang.Math). However, Esper provides a mechanism for user-controlled imports of classes and packages. The CEP engine auto-imports the following Java library packages:

- **java.lang.***
- **java.math.***
- **java.text.***
- **java.util.***

The user can import additional packages or libraries by specifying the configuration files or through the API and adding them to the configuration. User-defined functions may also be chained which implies that if a user-defined function returns an object then the object can itself be the target of the next function call and so on.

2.6.3 Single Row Functions

Single-row functions return a single value. They are stateless functions and not expected to aggregate rows like the aggregation functions. These functions can appear in any expressions and any number of parameters may be passed into them. In order to develop and use a custom single-row function with Esper the following steps must be followed:

- Implement a class providing one or more public static methods accepting the number and type of parameters as required.
- Register the single-row function class and method name with the engine by supplying a function name, via the engine configuration file or the configuration API.

A single-row function can't override a built-in function. Instead, the single row function must have a different name then any of the built-in functions. Single-row function classes have no further requirement than provide a public static method. The following sample single-row function calculates a percentage value based on two number values.

```
public class SampleClass {

   public static double calculatePercentage(double amount, double total) {

      return amount / total * 100;

   }
}
```

It is necessary, for the utilization of the function, that the function name, the class name of the class and the method name of the new single-row function is added to the engine configuration. This is feasible via the configuration API or the XML configuration file. A configuration example through XML is the following.

```
<esper-configuration
<plugin-singlerow-function name="percent"
function-class="mycompany.MyUtilityClass"
    function-method="computePercent" /
>
</esper-configuration>
```

It is important to mention that the function name and method name are not necessarily the same. A statement exploiting the the new single-row function is below:

```
select percent( fulfilled, total ) from NewEvent
```

2.7 Patterns

Patterns apply in case of an event or multiple events occuring which match the pattern's definition then. Pattern atoms and pattern operators consist the pattern expressions. The key factor in constructing blocks of patterns are the pattern atoms. They can either be observers for time-based events or filter expressions or plug-in custom observers that observe external events which lie outside of the engine. Pattern operators handle expression cycle of life and combine atoms logically or temporally. There are 4 types of pattern operators:

- Temporal operators that operate on event order: $->$ (followed-by)
- Logical operators: *and, or, not*
- Operators that control pattern sub-expression repetition: every, every-distinct, [num] and until
- Guards are where-conditions that control the lifecycle of subexpressions. The *while* expression and *timer:within* are examples. Custom plug-in guards are also accepted.

2.7.1 Pattern Syntax

An EPL statement allows a pattern to appear anywhere in the query including subqueries and joins. Clauses like the *group by, having, where, insert into and output* clauses are completely compatible with patterns. Also, a data window view can be declared onto a pattern. A data window declared onto a pattern solely aims to store pattern matches. An example demonstrating a statement with pattern matching:

```
select a.customerId, avg(a.value + b.value)
from pattern [every a=Order ->
b=Product(customerId = a.customerId)
where timer:within(1 min)].win:time(2 hour)
where a.name in ('Repair', b.name)
group by a.customerId
having avg(a.value + b.value) > 100
```

The above statement presents the idea of selecting a total price per customer over pairs of events (ServiceOrder followed by a ProductOrder event for the same customer id within 1 minute), occurring in the last 2 hours, in which the sum of price is greater than 100, and using a where clause to filter on name.

2.7.2 Pulling Data from Patterns

It is important to mention that data can also be retrieved from pattern statements through the safeIterator() and iterator() methods on EPStatement. This is available only if the pattern had fired at least once and the @IterableUnbound annotation is declared for the statement.If this condition is served, the last event for which the iterator fired,will be returned. The hasNext() method can then be used to determine if the pattern had fired. The methods mentioned:

```
if (myPattern.iterator().hasNext()) {
ServiceMeasurement event = (ServiceMeasurement)
view.iterator().next().get("price");
... // some more code here to process the event
}
else {
... // no matching events at this time
}
```

2.7.3 Pattern Expressions Priority

The following table demonstrates the precedence between operators and the correlation between events and pattern operators:

Expression	Equivalent	Reason
every A or B	(every A) or B	The `every` operator has higher precedence then the `or` operator.
every A -> B or C	(every A) -> (B or C)	The `or` operator has higher precedence then the `followed-by` operator.
A -> B or B -> A	A -> (B or B) -> A	The `or` operator has higher precedence then the `followed-by` operator, specify as (A -> B) or (B -> A) instead.
A and B or C	(A and B) or C	The `and` operator has higher precedence then the `or` operator.
A -> B until C -> D	A -> (B until C) -> D	The `until` operator has higher precedence then the `followed-by` operator.
[5] A or B	([5] A) or B	The `[num]` repeat operator has higher precedence then the `or` operator.
every A where timer:within(10)	every (A where timer:within(10))	The `where` postfix has higher precedence then the `every` operator.

Figure 2.5: Equivalent Pattern Expressions.
Source: Reprinted from the Esper Reference Documentation.

Chapter 3

Ferari

The FERARI (Flexible Event pRocessing for big dAta aRchItectures) project provides a large scale, distributed, streaming framework suitable for Big Data processing. Complex event processing is supported by the architecture and the components of the system. The Apache's Storm computation system is the basis of the Ferari project.

3.1 Storm

Storm is a distributed, fault-tolerant, efficient and reliable computation platform suitable for handling and analyzing streams of data. It was built by Nathan Marz and it is written in Java and Clojure. There are some certain characteristics which make Storm quite special:

- It is rather **simple** and **easy to use**. The basic concept behind Storm's functionality and its components is rather simple to be perceived by the common user.
- It is **fault-tolerant**. This means that it can handle failures by rescheduling tasks.
- It supports **all programming languages**. Although it is easier to work with a JVM-based language, Storm is compatible with many other languages.
- It is **reliable**. Storm posseses the mechanisms which guarantee that even if a message is lost, it will be retransmitted, until it reaches its destination.
- It is **easy to scale**. If the user wants to expand the capabilities of Storm, all that is needed is the machines to the cluster. Storm will handle the rest of the process.
- It has **low latency**. Storm has mili-second latency.

3.1.1 Storm Architecture

In a Storm cluster,two kind of nodes exist:

- The **Master** node. A daemon named Nimbus is run by the Master node and is accountable for the assignment of task to each worker node, failure monitoring and code distribution around the cluster.
- The **Worker** node. A daemon named Supervisor is run by the Worker node and is accountable for executing a certain topology part.

Due to the fact that all cluster states are kept either on local storage or in Zookeper,possible failure of the daemons doesn't afflict the system's performance.

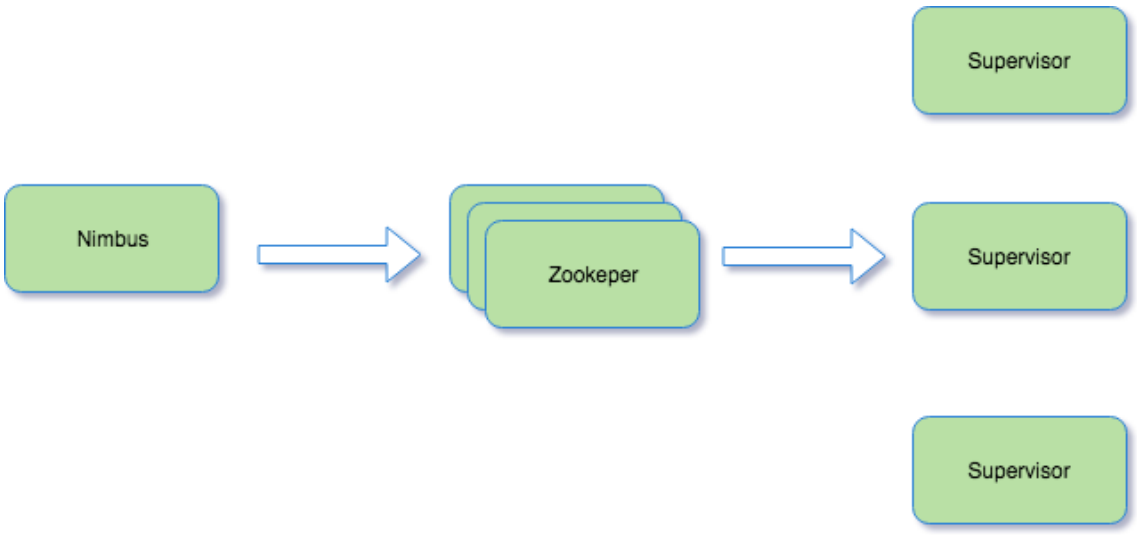

Figure 3.1: Interaction of Nodes in Storm cluster

3.1.2 Storm Concepts

Streams,bolts and spouts are Storm's main concepts.

Streams

Streams is a term to describe the series of tuples which are channeled between the Storm's components, bolts and spouts. Tuples refer to data structures consisting of multiple parts of information.In order for bolts to connect in an efficient and productive manner with the streams, stream grouping is defined:

- **Shuffle Grouping**:It is the most popular stream grouping method as it delivers, randomly, an equal number of tuples to all subscribing bolts.
- **Custom Grouping**:Allows the user to choose which bolt(s) will collect each tuple.
- **Global Grouping**:A specific task of the bolt, usually the one with the smallest ID, is the target of the stream.
- **Direct Grouping**:In direct grouping, the source of the stream selects the exact receiver (bolt) of the stream.
- **Fields Grouping**:Allows the user to determine which specific fields of the tuple in the stream will be delivered to the bolt.
- **All Grouping**:This method is used to transmit the tuples of the stream to all tasks.
- **None Grouping**:Currently, has identical functionality to the fields grouping method.

Bolt

Storm's Bolt is a unit which receives tuples and processes them. Once the tuple is processed, it will be forwarded to other components of the Storm framework. Processing includes several data manipulation techniques such as filtering, applying functions, aggregations and others. Bolts are regarded as the factories of Storm.

Spout

The role of the Spout in the Storm ecosystem is the transmission of data to the bolts. Spouts are supplied with data from external sources such as files, message brokers, databases and others. They are able to deliver data to bolts through multiple streams.

A storm topology,consisting of streams,bolts and spouts,is depicted in the following figure.

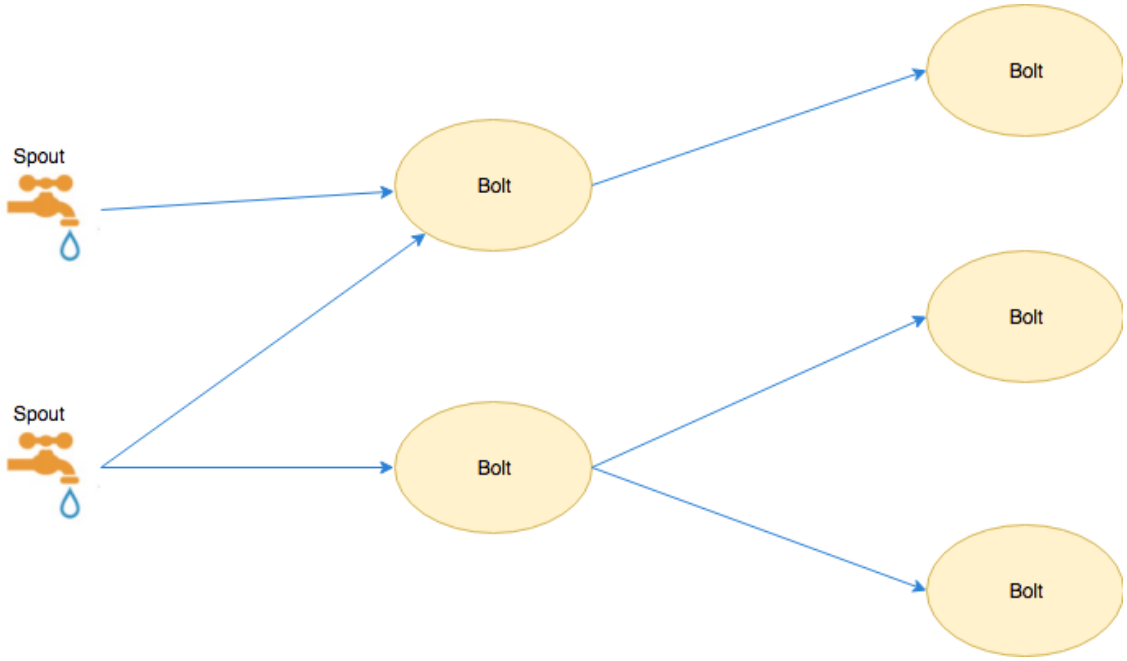

Figure 3.2: Storm Topology

3.2 The architecture of Ferari

The Ferari intra-site architecture consists of several components which collaborate with each other in order to provide a flexible and efficient streaming analyzer platform. Communication between sites is cost-effective and performed through Redis.

3.2.1 Communicator Bolt

Communications between the sites of the network is handled by the communicator bolt. The bolt is responsible for interacting with the other bolts of the Storm topology and other sites. The interaction with the the other sites of the network is achieved through the Redis message broker.

The communicator bolt's main tasks are the transmission of pull requests and the handling of occuring violations.Once,the bolt receives a violation message from the GateKeeper, it will inform the TimeMachine bolt to set its state accordingly.

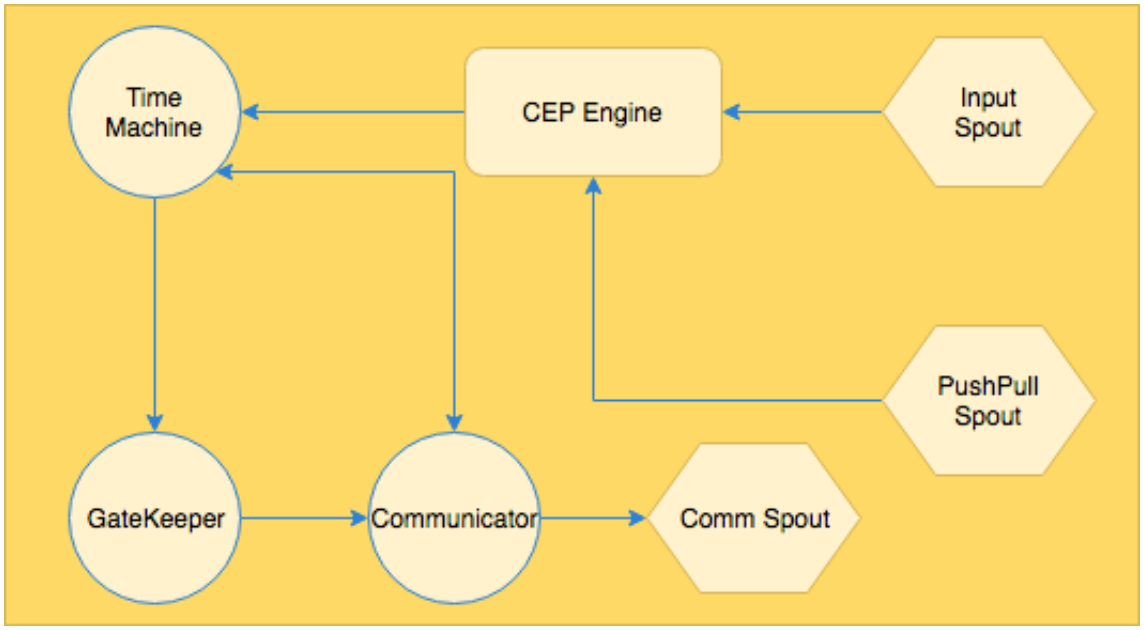

Figure 3.3: Site Architecture

3.2.2 Communicator Spout

The Communicator spout can be compared to the antenna of the site. The bolt is responsible for receiving all incoming messages of the site. Also,it handles messages which are processed directly from the communicator of the node. By the time,the communicator bolt receives a message,it will forward it to the appropriate components of the site.

3.2.3 Input Spout

The Input Spout is responsible for providing the framework with data. It can be compared to the site's gateway as all incoming data from external sources is processed by this spout.Manipulating external data and emitting the tuples to the CEP engine are its main workload.

3.2.4 Time Machine

Time Machine works like a buffer mechanism for the events processed by the CEP engine. Depending on the push and pull requests,the events are accordingly manipulated in order to be sent to the coordinator sites. The Time Machine bolt provides statistics of the events and operates in a state manner. Ideally,it operates in Play mode in which it assigns the requested events for forwarding to the coordinator site. If a violation occurs, the Time Machine bolt is set to Pause mode until the violation is fixed and then it returns to Play mode.

3.2.5 GateKeeper bolt

The GateKeeper bolt is responsible for observing and handling violations. It is assigned with detecting anomalies in the monitoring function which oversees the threshold of the value computed on event data. In case of a threshold violation, it informs the coordinator to perform specific actions.

3.2.6 PushAndPull spout

The PushAndPull spout is the component in charge of receiving pushed events from other sites and forwarding these events to the CEP engine for processing.

3.2.7 CEP-Engine Bolt

The CEP-Engine bolt is the bolt which contains the functionality of the CEP engine. The Ferari project has been designed to use the Proton On Storm CEP engine. In that implementation, the CEP-Engine consisted of three bolts. However, for the purposes of this work, the Esper engine will be used to conduct complex event processing. In this case, the CEP-Engine consists of a single bolt. The proper system configurations in order to adapt the Esper engine into the Ferari computation platform, will be presented in the following chapter.

3.2.8 Optimizer

The optimizer component is an autonomous Storm topology. Its tasks are the provision of network plans and queries to sites. More specifically, it assigns a specific role to each site and provides the necessary plans in order for them to set up properly. Also,it supplies the CEP engine with the queries based on which data analysis will be conducted. The topology is composed of the optimizer spout and the optimizer bolt. The spout receives the network parameters and the queries from external sources and forwards them to the optimizer bolt. Network plans,according to the parameters, are generated from the optimizer bolt and along with the queries are transmitted to the sites and the the CEP engine respectively.

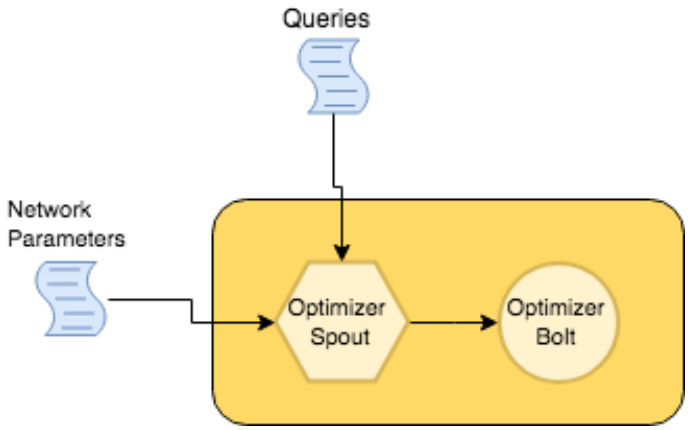

Figure 3.4: Optimizer Topology

Chapter 4

System Configuration

The integration of the Esper engine into the Ferari computation system requires a series of configurations. An instance of the engine has to be placed inside a Storm bolt, stream names and event types have to be declared and certain methods have to be applied in order for the bolt to support the push and pull mechanism. Support classes such as the MessageBean class are used to handle the input data.

4.1 The *MessageBean* Class

The incoming data are loaded from external files and are forwarded to the Esper engine by the Input Spout in the form of tuples. In our experiments, the used data can be classified as "tidy" as each tuple contains comma separated variables and their values. In order for the system,to be able to handle all kinds of data, the MessageBean class has been built. The class works like a parser which scans the tuple and extracts each variable and its value by using a regular expression pattern. Getter and setter methods are supported by the class. Due to the fact that the event processing language (EPL) language allows the import of Java functions, the MessageBean class is used to create objects of the tuples which contain the variables and their values.

```
MessageBean push_attributes_bean = MessageBean.parse(tuple
    .getValueByField(STORMMetadataFacade.ATTRIBUTES_FIELD).toString());

MessageBean attributes_bean = MessageBean.parse(tuple
    .getValueByField(STORMMetadataFacade.ATTRIBUTES_FIELD).toString());
```

The functionality of the MessageBean class is presented in the above sample code and will be further demonstrated in the following sections.

4.2 The *prepare* function

The prepare function is used, during the initialization and before the bolt starts processing tuples. In the Esper bolt, the *prepare* function is the following:

```
public void prepare(@SuppressWarnings("rawtypes") Map
    conf,TopologyContext context,OutputCollector collector){

        setup();
        StormOutputWriter.Instance().setPath(path);
        StormOutputWriter.Instance().setOut(NodeID);
        this.collector = collector;
        Configuration configuration = new Configuration();

        setupEventTypes(context, configuration);

        this.esperSink =
            EPServiceProviderManager.getProvider(this.toString(),
            configuration);
        this.esperSink.initialize();
        this.runtime = esperSink.getEPRuntime();
        this.admin = esperSink.getEPAdministrator();

        for (String stmt : statements) {
            EPStatement statement = admin.createEPL(stmt);

            statement.addListener(this);
        }
    }
```

As it can be seen,the setupEventTypes function is called inside the *prepare* function. The OptimizerBolt has sent the statements to the EsperBolt and these are stored in the ***statements*** list. All the declared statements are created inside the engine and a Listener subscribes to them. Several extra configurations take place during the configuration.

4.3 The *setup* function

The initiliaze function is used to initialize the necessary data structures and variables which will be used in the EsperBolt:

```
public void setup() {
    allstatistics = new HashMap<String, List<Statistics>>();
    statistics = new HashMap<String, Statistics>();
    hasPullEvens = new HashMap<String, Boolean>();
    eventsToPull = new HashMap<String, List<String>>();
    AllEvents = new HashMap<String, IEventType>();
    pushModeEvents = new HashMap<String, List<String>>();
    AlleventsToPull = new ArrayList<String>();
    DashboardEvents = new ArrayList<String>();
    pushEvents_ = new ArrayList<String>();
    event_map = new HashMap<String, Object>();
    all_event_map = new HashMap<String, Object>();
    firstEvent = true;
  }
```

4.4 Streams

In order for the Esper Bolt to communicate with other bolts, certain stream names have to be declared. The following code demonstrates this process:

```
public void declareOutputFields(OutputFieldsDeclarer declarer)
    {
      declarer.declareStream("Event", new
          Fields("type","Name","timestamp","attributes","value"));
      declarer.declareStream("Statistics", new Fields("type",
          "statistics"));
      declarer.declareStream("PushModeEvent", new Fields("type",
          "eventName", "nodeIDs"));
      declarer.declareStream("Pull", new Fields("type", "pullEvents",
          "startTime", "endTime", "nodeID"));
    }
```

4.5 Event Types

The function *setupEventTypes* is used to declare specific Event Types which will be used by the Esper engine:

```
private void setupEventTypes(TopologyContext context, Configuration
    configuration){
   Set<GlobalStreamId> sourceIds = context.getThisSources().keySet();
   for (GlobalStreamId id : sourceIds) {

      String eventTypeName = getEventTypeName(id.get_componentId(),
          id.get_streamId());
      updatedFieldDef.put("data", MessageBean.class);
      configuration.addEventType("UpdatedFieldType", updatedFieldDef);
        Map<String, Object> accountUpdateDef = new HashMap<String,
            Object>();
      accountUpdateDef.put("Name", String.class);
      accountUpdateDef.put("fields", updatedFieldDef);
      configuration.addEventType(eventTypeName, accountUpdateDef);
   }
}
```

4.6 The *execute* function

The *execute* method is called once per tuple received and is responsible for handling and processing the incoming tuples. In our design,this specific function is of great importance as it validates the initialization of the EsperBolt, processes the tuples based on the fact that the tuple can either be a primitive event or a pushed one, calculates statistics for each event and emits them to the TimeMachineBolt. The following subsections will demonstrate the functionality of the *execute* method.

4.6.1 Setup Validation

During the initialization of the system,the EsperBolt will receive configurations from the OptimizerBolt. Continous checks are perfomed by the EsperBolt to determine if these configurations are set, in order to start processing tuples. The tuple which contains the information will be delivered from the Optimizer topology to the EsperBolt through the CommunicatorSpout which lies in the EsperBolt's topology. Variables are assigned with certain values of the tuple and are used to support the functionality of the EsperBolt.

```
if(!isPropertiesSet){
   if(tuple.getSourceStreamId().equals("CommSpout_to_esper_properties")){
      List<String> properties = (List<String>)tuple.getValue(0);
      pull = Boolean.valueOf(properties.get(8));
      statisticsClass = properties.get(9);
      epoch = Integer.valueOf(properties.get(10));
      eType = properties.get(11);
      jarLocation = properties.get(12);
      isPropertiesSet = true;
      return;
    }
}
```

Note that the epoch parameter is of great importance for experimental phase as it determines the number of events after which the Statistics structure will be delivered to the TimeMachine for further processing.For instance, epoch = 1 means that Statistics will be updated in the TimeMachine Bolt after each detected event. The *eType* variable is set to *events*, while the *StatisticsClass* is set to *statistics.CountStatistics*.

4.6.2 Statistics

The *execute* function checks if the *pull* variable has been set to *true* though the configurations sent by the OptimizerBolt and whether the incoming tuple is labeled as an event or not.Once these conditions are met, the *updateStatistics* function is called with the *Name* of the event as an argument.

```
if(pull && tuple.getSourceStreamId().equals("Event")) {
   updateStatistics(tuple, tuple.getStringByField("Name"));
}
```

In the above *if* condition, the *updateStatistics* function is responsible for updating the statistics of the event and delivering these statistics to the TimeMachine. Presenting the *updateStatistics* function is skipped as it is rather a part of the FERARI implementation than a part of the Esper configuration.

4.6.3 Queries - Push And Pull Mechanism

The FERARI project supports the push and pull mechanism. This technique involves the dispatch of events from the peripheral nodes to the coordinator and the request of events to be pulled by the coordinator.In order for this functionality to be integrated into the EsperBolt, the execute method is responsible for handling the events which will either be in pull or push mode.

```
if(!isJsonSet) {

   if(tuple.getSourceStreamId().equals("metadatatoEsper")) {

      jsonString = (String)tuple.getValue(1);
      stmts = (String)tuple.getValue(2);
      addStatements(stmts);
      prepare();
      setPullStructures();
      isJsonSet = true;

      for(IEventType Event: AllEvents.values()){
         for(TypeAttribute atr :Event.getTypeAttributes()){

               String EventType = Event.getName();
               if(atr.getName().equals("PushToCoordinators")){

                  String nodeId_s = (String) atr.getDefaultValue();
                  pushEvents_.add(EventType);
                  List<Object> pullmsg = new ArrayList<Object>();

                  pullmsg.add("PushModeEvent");
                  pullmsg.add(EventType);
                  pullmsg.add(nodeId_s);

                  collector.emit("PushModeEvent", pullmsg);
                  System.out.println(NodeID + " set event: " + EventType +"
                      in pushMode for  sites: "+ nodeId_s);
                  StormOutputWriter.Instance().print(NodeID, NodeID + " set
                      event: " +  EventType +" in pushMode for sites: "+
                      nodeId_s + "<br />", "black");
                  break;
            }
         }
      }
   }
}
```

The Optimizer Bolt has built and delivered to the sites, specific plans which contain all the necessary information about the events, the push/pull mechanism and the queries which will be used by the EsperBolt. These plans are exploited by the *execute* function which monitors the *isJsonSet* boolean flag. The flag is set to *false* by default so once an tuple with the plans and the queries from the Optimizer Bolt arrive, the mechanism is established.

First, the JSON file which contains the plans is loaded into a variable. However, the code about the JSON manipulation is skipped for sake of reference. Then, the statements which stand for the queries,are assigned to a variable and this variable is passed into the function *addStatements*. The *addStatements* funtion separates the statements that are loaded in a single string and add them to a list named *statements*.

```
private void addStatements(String queries)
    {
        \\Manipulating the queries string depending on whether the
        statements inside the string are separated by commas, spaces,
        tab or other symbols.The code is skipped as it depends on the
        form of the tuple.The result will be a list containing the queries.
        for (String query : queries_list) {
            statements.add(query)
        }
    }
```

The *statements* is an ArrayList which is declared in the EsperBolt class. Afterwards, the **prepare** function is called because it is the method which will add a listener to each statement as presented in page 32. The step that follows is setting up the push and pull mechanism.

In order for the mechanism to be established, the *setPullStructures* function is called. As its name implies, this function builds the structure for the *pull* concept. Since it is mainly a FERARI's functionality, the demonstration of its code will be skipped. Afterwards, the *execute* function iterates the *AllEvents* list. Once an event has an attribute with the name *PushToCoordinators*, a tuple is built. This tuple contains the *name* of the event to be pushed, a *PushMode* flag and the name of the *coordinator* site where all these events from the peripheral nodes will be delivered. Every tuple is, then, emitted to the Communicator Bolt of the site which will, in turn, inform the other sites about the *Events* in *PushMode*.

4.6.4 Primitive (Raw) Events

Events which are delivered to the EsperBolt by the the InputSpout that lies in the same site topology, are called **Raw Events**. All the data from external sources are handled by the InputSpout which imports and forwards the data to the EsperBolt. So the term **Raw Events** corresponds to these events imported by external sources through the InputSpout anh have not been pre-processed. The *execute* function performs a check on the source of the incoming tuple and once the tuple is delivered from the InputSpout, it assigns the proper label to the tuple and imports it to the inner architecture of the Esper CEP engine. The following code demonstrates this procedure:

```
if(tuple.getSourceStreamId().equals("frs_to_esper_primitiveEvents")) {

   String eventType = getEventTypeName(tuple.getSourceComponent(),
       tuple.getSourceStreamId());

   Fields fields = tuple.getFields();
   int numFields = fields.size();
   Map<String, Object> attributes = null;

   attributes = (Map<String,Object>)tuple.
   getValueByField(STORMMetadataFacade.ATTRIBUTES_FIELD);

   try {
      MessageBean attributes_bean = MessageBean.
      parse(tuple.getValueByField(STORMMetadataFacade.
      ATTRIBUTES_FIELD).toString());
         event_map.put("event", attributes_bean);
   } catch (UnsupportedEncodingException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
     }

   all_events_map.put("Name", "Raw");
   all_events_map.put("fields", event_map);

   runtime.sendEvent(all_events_map, eventType);
   collector.ack(tuple);
}
```

4.6.5 Derived Events

Apart from the raw events, there may be incoming tuples from other topologies depending on the push/pull functionality. These events which are either pushed to or pulled by the EsperBolt are labelled as Derived Events. The *execute* method needs to handle these tuples accordingly so it performs a check on the source of the tuple. The FERARI's project architecture defines that all tuples delivered from other sites are received by the PushAndPullSpout and the processed to the EsperBolt of the same site. Once the source of the tuple is the PushAndPullSpout, the *execute* function processes the data of the tuple in the event mechanism of the Esper engine. The code responsible for handling **derived events**:

```
if(tuple.getSourceStreamId().equals("PushAndPullSpout_to_esper_push")) {

   String eventType = getEventTypeName(tuple.getSourceComponent(),
   tuple.getSourceStreamId());
   try {
      MessageBean push_attributes_bean = MessageBean.parse(tuple
         .getValueByField(STORMMetadataFacade.ATTRIBUTES_FIELD)
         .toString());
          event_map.put("event", push_attributes_bean);
   } catch (UnsupportedEncodingException e) {
       // TODO Auto-generated catch block
         e.printStackTrace();
     }

   all_events_map.put("Name", "PushPullEvent");
   all_events_map.put("fields", event_map);

   runtime.sendEvent(all_events_map, eventType);
   collector.ack(tuple);

}
```

It is important to note that, apart from the derived events which are delivered from other sites though the push/pull mechanism, there are other events which are produced in the EsperBolt by the raw events and are also labelled as **derived events**. However, these events are not forwarded to the Esper engine. Instead, they are handled internally by the CEP engine, so there is no need for the *execute* method to perform any actions.

4.7 The *update* method

This method is the one responsible for handling the events which have been detected by Esper. Every time a pattern is fired in Esper, an EventBean is generated. The properties of each EventBean contain the underlying events that provoked the pattern to fire.The *update* function:

```
public void update(EventBean[] newEvents, EventBean[] oldEvents)
{
        if (newEvents != null) {
            for (EventBean newEvent : newEvents) {
                  EventTypeDescriptor eventType =
                  getEventType(newEvent.getEventType().getName());

                  if (eventType == null) {
                    eventType = getEventType(null);
                  }

                  if (eventType != null) {
                    collector.emit(eventType.getStreamId(),
                    toTuple(newEvent, eventType.getFields(),
                    eventType.getName()));
                  }
             }
        }
}
```

The array of the EventBean is iterated and new events along with their properties and values are extracted. Then, every event is emitted to the TimeMachine Bolt after it has been processed with the toTuple function. The toTuple function is a simple method which determines the format of the tuple in order to be emitted.

Chapter 5

Experimental Assessment

In order to evaluate both the capabilities of the Esper engine and its integration into the Ferari project, we conducted various experiments. These experiments demonstrate specific functionalities of the Esper engine such as windows, aggregation and user-defined functions, *having* and *where* clauses and others. Also, they exhibit the collaboration of Esper with the Apache Storm stream processing computation framework. This collaboration is evidenced in the experiments through the use of the Ferari's mechanisms.

The basis for the experiments has been the Mobile Fraud example of the Ferari project. In the tests we run, the network of the sites is defined in the network parameters which are provided by the Optimizer. More specifically, our implementation consists of 4 peripheral sites and one central site which is categorized as the coordinator. The network graph is demonstrated below:

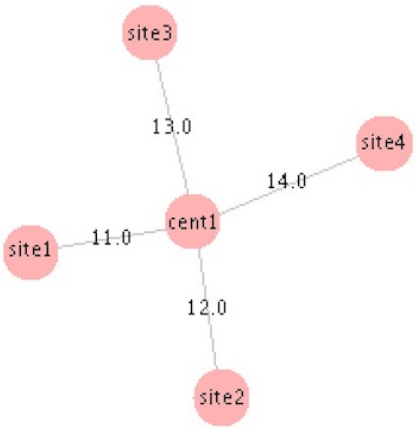

Figure 5.1: Network Graph

Each node detects the events which is assigned for and if it is a peripheral node, he is obliged to push the events which are in push mode to the coordinator. In turn, the coordinator, detects not only the events which are created in his topology but also the events which are delivered from other sites.

In the following section we demonstrate the event detection in the coordinator site. It allows us to monitor the process of receiving and detecting raw and derived events as well as confirm the accuracy of the results.

- **cent1**

cent1 received CallPOPDWH, generated at 1456938480000
cent1 detected CallPOPDWH at 1536531960445
cent1 detected halfFrequentLongCalls at 1536531960450
cent1 detected halfExpensiveCalls at 1536531960452
cent1 received CallPOPDWH, generated at 1456938484000
cent1 detected CallPOPDWH at 1536531960500
cent1 received CallPOPDWH, generated at 1456938485000
cent1 detected CallPOPDWH at 1536531960555
cent1 detected halfFrequentLongCalls at 1536531960555
cent1 detected halfExpensiveCalls at 1536531960556
cent1 detected halfFrequentEachLongCall at 1536531960557
cent1 detected LongCallAtNight at 1536531960557
cent1 received CallPOPDWH, generated at 1456938492000
cent1 detected CallPOPDWH at 1536531960612
cent1 detected halfFrequentLongCalls at 1536531960613
cent1 detected halfExpensiveCalls at 1536531960614
cent1 detected halfFrequentEachLongCall at 1536531960615
cent1 detected LongCallAtNight at 1536531960616
cent1 received CallPOPDWH, generated at 1456938494000
cent1 detected CallPOPDWH at 1536531960668
cent1 detected halfFrequentLongCalls at 1536531960669
cent1 detected halfExpensiveCalls at 1536531960670
cent1 detected halfFrequentEachLongCall at 1536531960671
cent1 detected LongCallAtNight at 1536531960672
cent1 detected FrequentLongCallsAtNight at 1536531960673
cent1 received CallPOPDWH, generated at 1456938495000
cent1 detected CallPOPDWH at 1536531960724
cent1 detected halfFrequentLongCalls at 1536531960725
cent1 detected halfExpensiveCalls at 1536531960726
cent1 detected LongCallAtNight at 1536531960727
cent1 received CallPOPDWH, generated at 1456938507000
cent1 detected CallPOPDWH at 1536531960780
cent1 detected halfFrequentLongCalls at 1536531960781
cent1 detected halfExpensiveCalls at 1536531960782

cent1 detected halfFrequentEachLongCall at 1536531960783
cent1 detected LongCallAtNight at 1536531960785
cent1 received CallPOPDWH, generated at 1456938510000
cent1 detected CallPOPDWH at 1536531961098
cent1 received CallPOPDWH, generated at 1456938523000
cent1 detected CallPOPDWH at 1536531961156
cent1 received CallPOPDWH, generated at 1456938534000
cent1 detected CallPOPDWH at 1536531961215
cent1 received CallPOPDWH, generated at 1456938545000
cent1 detected CallPOPDWH at 1536531961269
cent1 detected halfFrequentLongCalls at 1536531961271
cent1 detected halfExpensiveCalls at 1536531961272
cent1 detected halfFrequentEachLongCall at 1536531961272
cent1 detected LongCallAtNight at 1536531961274
cent1 received CallPOPDWH, generated at 1456938546000
cent1 detected CallPOPDWH at 1536531961325
cent1 detected halfFrequentLongCalls at 1536531961327
cent1 detected halfExpensiveCalls at 1536531961328
cent1 detected halfFrequentEachLongCall at 1536531961329
cent1 detected LongCallAtNight at 1536531961330
cent1 received CallPOPDWH, generated at 1456938547000
cent1 detected CallPOPDWH at 1536531961382
cent1 received CallPOPDWH, generated at 1456938555000
cent1 detected CallPOPDWH at 1536531961436
cent1 received halfFrequentLongCalls, generated at 1536531960432
cent1 received CallPOPDWH, generated at 1456938565000
cent1 received halfExpensiveCalls, generated at 1536531960433
cent1 received halfFrequentEachLongCall, generated at 1536531960434
cent1 detected halfFrequentLongCalls at 1536531961488
cent1 detected halfExpensiveCalls at 1536531961491
cent1 detected halfFrequentEachLongCall at 1536531961493
cent1 detected CallPOPDWH at 1536531961498
cent1 received LongCallAtNight, generated at 1536531960435
cent1 detected halfFrequentLongCalls at 1536531961499
cent1 received halfFrequentLongCalls, generated at 1536531960543
cent1 detected halfExpensiveCalls at 1536531961501
cent1 detected halfFrequentEachLongCall at 1536531961507
cent1 received halfExpensiveCalls, generated at 1536531960544
cent1 detected LongCallAtNight at 1536531961513
cent1 detected LongCallAtNight at 1536531961514
cent1 detected halfFrequentLongCalls at 1536531961515
cent1 detected halfExpensiveCalls at 1536531961516
cent1 received halfFrequentEachLongCall, generated at 1536531960545
cent1 detected halfFrequentEachLongCall at 1536531961518

cent1 received LongCallAtNight, generated at 1536531960546
cent1 detected LongCallAtNight at 1536531961523
cent1 received halfFrequentLongCalls, generated at 1536531960710
cent1 detected halfFrequentLongCalls at 1536531961525
cent1 received halfExpensiveCalls, generated at 1536531960711
cent1 detected halfExpensiveCalls at 1536531961529
cent1 received halfFrequentEachLongCall, generated at 1536531960712
cent1 detected halfFrequentEachLongCall at 1536531961532
cent1 received CallPOPDWH, generated at 1456938573000
cent1 received halfFrequentLongCalls, generated at 1536531960507
cent1 detected halfFrequentLongCalls at 1536531961545
cent1 detected CallPOPDWH at 1536531961546
cent1 received LongCallAtNight, generated at 1536531960713
cent1 detected LongCallAtNight at 1536531961549
cent1 received halfExpensiveCalls, generated at 1536531960508
cent1 detected halfExpensiveCalls at 1536531961552
cent1 received LongCallAtNight, generated at 1536531960509
cent1 detected LongCallAtNight at 1536531961556
cent1 received halfFrequentLongCalls, generated at 1536531960676
cent1 received halfExpensiveCalls, generated at 1536531960677
cent1 detected halfFrequentLongCalls at 1536531961558
cent1 received halfFrequentLongCalls, generated at 1536531960731
cent1 detected halfExpensiveCalls at 1536531961559
cent1 received halfExpensiveCalls, generated at 1536531960732
cent1 detected halfFrequentLongCalls at 1536531961560
cent1 detected halfExpensiveCalls at 1536531961561
cent1 received halfFrequentEachLongCall, generated at 1536531960733
cent1 received halfFrequentLongCalls, generated at 1536531961158
cent1 received LongCallAtNight, generated at 1536531960733
cent1 detected halfFrequentEachLongCall at 1536531961563
cent1 received halfExpensiveCalls, generated at 1536531961161
cent1 detected halfFrequentLongCalls at 1536531961564
cent1 detected LongCallAtNight at 1536531961565
cent1 detected halfExpensiveCalls at 1536531961566
cent1 received halfFrequentEachLongCall, generated at 1536531961162
cent1 detected halfFrequentEachLongCall at 1536531961567
cent1 received LongCallAtNight, generated at 1536531961164
cent1 detected LongCallAtNight at 1536531961571
cent1 received halfFrequentLongCalls, generated at 1536531961213
cent1 detected halfFrequentLongCalls at 1536531961574
cent1 received halfExpensiveCalls, generated at 1536531961214
cent1 detected halfExpensiveCalls at 1536531961578
cent1 received CallPOPDWH, generated at 1456938576000
cent1 detected CallPOPDWH at 1536531961601

cent1 received halfFrequentLongCalls, generated at 1536531960597
cent1 detected halfFrequentLongCalls at 1536531961626
cent1 received halfExpensiveCalls, generated at 1536531960598
cent1 received halfFrequentEachLongCall, generated at 1536531960599
cent1 detected halfExpensiveCalls at 1536531961629
cent1 detected halfFrequentEachLongCall at 1536531961630
cent1 received LongCallAtNight, generated at 1536531960600
cent1 detected LongCallAtNight at 1536531961633
cent1 received halfFrequentLongCalls, generated at 1536531960653
cent1 detected halfFrequentLongCalls at 1536531961635
cent1 received halfExpensiveCalls, generated at 1536531960654
cent1 detected halfExpensiveCalls at 1536531961638
cent1 received halfFrequentLongCalls, generated at 1536531960766
cent1 detected halfFrequentLongCalls at 1536531961642
cent1 received halfFrequentLongCalls, generated at 1536531960621
cent1 detected halfFrequentLongCalls at 1536531961644
cent1 received halfExpensiveCalls, generated at 1536531960622
cent1 detected halfExpensiveCalls at 1536531961646
cent1 received halfExpensiveCalls, generated at 1536531960767
cent1 detected halfExpensiveCalls at 1536531961648
cent1 received halfFrequentEachLongCall, generated at 1536531960622
cent1 detected halfFrequentEachLongCall at 1536531961650
cent1 received halfFrequentEachLongCall, generated at 1536531960768
cent1 detected halfFrequentEachLongCall at 1536531961654
cent1 received LongCallAtNight, generated at 1536531960624
cent1 received halfFrequentLongCalls, generated at 1536531961101
cent1 detected LongCallAtNight at 1536531961656
cent1 detected halfFrequentLongCalls at 1536531961657
cent1 received LongCallAtNight, generated at 1536531960769
cent1 detected LongCallAtNight at 1536531961658
cent1 received halfExpensiveCalls, generated at 1536531961101
cent1 detected halfExpensiveCalls at 1536531961660
cent1 received halfFrequentLongCalls, generated at 1536531961490
cent1 detected halfFrequentLongCalls at 1536531961661
cent1 received halfFrequentLongCalls, generated at 1536531961158
cent1 detected halfFrequentLongCalls at 1536531961664
cent1 received halfExpensiveCalls, generated at 1536531961492
cent1 detected halfExpensiveCalls at 1536531961665
cent1 received halfExpensiveCalls, generated at 1536531961162
cent1 detected halfExpensiveCalls at 1536531961667
cent1 received halfFrequentLongCalls, generated at 1536531961596
cent1 detected halfFrequentLongCalls at 1536531961670
cent1 received halfExpensiveCalls, generated at 1536531961597
cent1 detected halfExpensiveCalls at 1536531961671

cent1 received halfFrequentEachLongCall, generated at 1536531961597
cent1 detected halfFrequentEachLongCall at 1536531961672
cent1 received LongCallAtNight, generated at 1536531961598
cent1 detected LongCallAtNight at 1536531961674
cent1 received halfFrequentLongCalls, generated at 1536531961650
cent1 detected halfFrequentLongCalls at 1536531961675
cent1 received halfExpensiveCalls, generated at 1536531961651
cent1 detected halfExpensiveCalls at 1536531961677
cent1 received halfFrequentEachLongCall, generated at 1536531961651
cent1 detected halfFrequentEachLongCall at 1536531961679
cent1 received LongCallAtNight, generated at 1536531961652
cent1 detected LongCallAtNight at 1536531961681
cent1 received LongCallAtNight, generated at 1536531961876
cent1 detected LongCallAtNight at 1536531961891
****** cent1halfFrequentLongCalls generated at 1536531960787, received with delay: 0.113 sec
cent1 detected halfFrequentLongCalls at 1536531961937
****** cent1halfExpensiveCalls generated at 1536531960788, received with delay: 0.119 sec
cent1 detected halfExpensiveCalls at 1536531961938
****** cent1halfFrequentLongCalls generated at 1536531961218, received with delay: 0.127 sec
cent1 detected halfFrequentLongCalls at 1536531961941
****** cent1halfExpensiveCalls generated at 1536531961219, received with delay: 0.095 sec
cent1 detected halfExpensiveCalls at 1536531961942
****** cent1halfFrequentLongCalls generated at 1536531961326, received with delay: 0.129 sec cent1 detected halfFrequentLongCalls at 1536531961944
cent1 detected FrequentLongCalls at 1536531961944
****** cent1halfExpensiveCalls generated at 1536531961327, received with delay: 0.131 sec
cent1 detected halfExpensiveCalls at 1536531961946
****** cent1halfFrequentLongCalls generated at 1536531961494, received with delay: 0.137 sec
cent1 detected halfFrequentLongCalls at 1536531961949
****** cent1halfExpensiveCalls generated at 1536531961495, received with delay: 0.133 sec
cent1 detected halfExpensiveCalls at 1536531961951
****** cent1halfFrequentEachLongCall generated at 1536531961496, received with delay: 0.111 sec
cent1 detected halfFrequentEachLongCall at 1536531961953
****** cent1LongCallAtNight generated at 1536531961497, received with delay: 0.114 sec
cent1 detected LongCallAtNight at 1536531961955

****** cent1halfFrequentLongCalls generated at 1536531961818, received with delay: 0.100 sec
cent1 detected halfFrequentLongCalls at 1536531961958
****** cent1halfExpensiveCalls generated at 1536531961819, received with delay: 0.105 sec
cent1 detected halfExpensiveCalls at 1536531961959
****** cent1halfFrequentEachLongCall generated at 1536531961820, received with delay: 0.140 sec
cent1 detected halfFrequentEachLongCall at 1536531961961
****** cent1LongCallAtNight generated at 1536531961820, received with delay: 0.98 sec
cent1 detected LongCallAtNight at 1536531961963
****** cent1halfFrequentLongCalls generated at 1536531961874, received with delay: 0.118 sec
****** cent1halfExpensiveCalls generated at 1536531961875, received with delay: 0.123 sec
cent1 detected halfFrequentLongCalls at 1536531961966
cent1 detected halfExpensiveCalls at 1536531961967
****** cent1halfFrequentEachLongCall generated at 1536531961876, received with delay: 0.137 sec
cent1 detected halfFrequentEachLongCall at 1536531961969
cent1 received halfFrequentLongCalls, generated at 1536531961929
cent1 detected halfFrequentLongCalls at 1536531961970
cent1 received halfExpensiveCalls, generated at 1536531961930
cent1 detected halfExpensiveCalls at 1536531961971
cent1 received halfFrequentEachLongCall, generated at 1536531961930
cent1 detected halfFrequentEachLongCall at 1536531961973
cent1 received LongCallAtNight, generated at 1536531961931
cent1 detected LongCallAtNight at 1536531961975
cent1 detected FrequentLongCallsAtNight at 1536531961976
cent1 received halfFrequentLongCalls, generated at 1536531961983
cent1 detected halfFrequentLongCalls at 1536531961988
cent1 received halfExpensiveCalls, generated at 1536531961984
cent1 detected halfExpensiveCalls at 1536531961990
cent1 received halfFrequentEachLongCall, generated at 1536531961985
cent1 detected ExpensiveCalls at 1536531961991
cent1 detected halfFrequentEachLongCall at 1536531961992
cent1 received LongCallAtNight, generated at 1536531961985
cent1 detected LongCallAtNight at 1536531961993
cent1 detected FrequentLongCallsAtNight at 1536531961994
cent1 received halfFrequentLongCalls, generated at 1536531962039
cent1 detected halfFrequentLongCalls at 1536531962045
cent1 received halfExpensiveCalls, generated at 1536531962039
cent1 detected halfExpensiveCalls at 1536531962047

cent1 received halfFrequentEachLongCall, generated at 1536531962040
cent1 detected ExpensiveCalls at 1536531962048
cent1 received LongCallAtNight, generated at 1536531962041
cent1 detected halfFrequentEachLongCall at 1536531962050
cent1 detected LongCallAtNight at 1536531962051
cent1 detected FrequentLongCallsAtNight at 1536531962052
cent1 received halfFrequentLongCalls, generated at 1536531962095
cent1 detected halfFrequentLongCalls at 1536531962103
cent1 received halfExpensiveCalls, generated at 1536531962096
cent1 received halfFrequentEachLongCall, generated at 1536531962097
cent1 detected halfExpensiveCalls at 1536531962105
cent1 detected ExpensiveCalls at 1536531962106
cent1 detected halfFrequentEachLongCall at 1536531962107
cent1 received LongCallAtNight, generated at 1536531962098
cent1 detected LongCallAtNight at 1536531962112
cent1 detected FrequentLongCallsAtNight at 1536531962113
cent1 received halfFrequentLongCalls, generated at 1536531962205
cent1 received halfExpensiveCalls, generated at 1536531962206
cent1 detected halfFrequentLongCalls at 1536531962215
cent1 detected halfExpensiveCalls at 1536531962218
cent1 detected ExpensiveCalls at 1536531962220
cent1 received halfFrequentEachLongCall, generated at 1536531962207
cent1 detected halfFrequentEachLongCall at 1536531962224
cent1 received LongCallAtNight, generated at 1536531962208
cent1 detected LongCallAtNight at 1536531962225
cent1 detected FrequentLongCallsAtNight at 1536531962226
cent1 received halfFrequentLongCalls, generated at 1536531962317
cent1 detected halfFrequentLongCalls at 1536531962328
cent1 received halfExpensiveCalls, generated at 1536531962318
cent1 detected halfExpensiveCalls at 1536531962332
cent1 received halfFrequentEachLongCall, generated at 1536531962319
cent1 detected ExpensiveCalls at 1536531962335
cent1 received LongCallAtNight, generated at 1536531962320
cent1 detected halfFrequentEachLongCall at 1536531962337
cent1 detected LongCallAtNight at 1536531962338
cent1 detected FrequentLongCallsAtNight at 1536531962339
cent1 received halfFrequentLongCalls, generated at 1536531962372
cent1 received halfExpensiveCalls, generated at 1536531962373
cent1 detected halfFrequentLongCalls at 1536531962385
cent1 detected halfExpensiveCalls at 1536531962387
cent1 received halfFrequentEachLongCall, generated at 1536531962375
cent1 detected ExpensiveCalls at 1536531962388
cent1 detected halfFrequentEachLongCall at 1536531962390
cent1 received LongCallAtNight, generated at 1536531962376

cent1 detected LongCallAtNight at 1536531962392
cent1 detected FrequentLongCallsAtNight at 1536531962392
cent1 received halfFrequentLongCalls, generated at 1536531962427
cent1 detected halfFrequentLongCalls at 1536531962433
cent1 received halfExpensiveCalls, generated at 1536531962427
cent1 detected halfExpensiveCalls at 1536531962436
cent1 detected ExpensiveCalls at 1536531962438
cent1 received halfFrequentEachLongCall, generated at 1536531962429
cent1 received LongCallAtNight, generated at 1536531962430
cent1 detected halfFrequentEachLongCall at 1536531962444
cent1 detected FrequentEachLongCall at 1536531962445
cent1 detected LongCallAtNight at 1536531962446
cent1 detected FrequentLongCallsAtNight at 1536531962448
cent1 received halfFrequentLongCalls, generated at 1536531962540
cent1 detected halfFrequentLongCalls at 1536531962545
cent1 received halfExpensiveCalls, generated at 1536531962540
cent1 detected halfExpensiveCalls at 1536531962548
cent1 received halfFrequentLongCalls, generated at 1536531962647
cent1 detected halfFrequentLongCalls at 1536531962655
cent1 received halfExpensiveCalls, generated at 1536531962648
cent1 detected halfExpensiveCalls at 1536531962659
cent1 received halfFrequentLongCalls, generated at 1536531962758
cent1 detected halfFrequentLongCalls at 1536531962763
cent1 received halfExpensiveCalls, generated at 1536531962758
cent1 detected halfExpensiveCalls at 1536531962764
cent1 received LongCallAtNight, generated at 1536531962759
cent1 detected LongCallAtNight at 1536531962766

The above section allowed us to oversee the functionality of the Esper engine in the coordinator site. In fact, after reviewing the experimental data that we provided to the sites as input data, we concluded that that our system design accurately detected the events and successfully supported the push/pull mechanism. The relevant section from the other sites has been skipped for sake of bravity.

Also, it is important that the results were confirmed by the statistics which were generated in the TimeMachine Bolt of the coordinator topology. The statistics are shown below:

TimeMachine Statistics: cent1

halfFrequentLongCalls −− > count: 43
FrequentEachLongCall −− > count: 1
halfExpensiveCalls −− > count: 43
halfFrequentEachLongCall −− > count: 28
FrequentLongCallsAtNight −− > count: 9
ExpensiveCalls −− > count: 7
CallPOPDWH −− > count: 17
LongCallAtNight −− > count: 31
FrequentLongCalls −− > count: 1

Note that apart from the above experiment and its various forms, we tested the pull mechanism through the Synthetic Data experiment. Although that in this case the queries were rather simple, the experiment allowed us confirm the efficiency of the Esper engine to support the pull mechanism.

Moreover, we applied stress testing to determine the robustness of the Esper CEP engine. Since the initial tuple transmission of the InputSpout was 50 ms, we decided to increase the size of our input data to 10000 times the size of the initial input data, in the Mobile Fraud experiment, and gradually increase the tuple transmission rate. Starting from 50 ms, we gradually lowered the delay between tuple transmission down to 0.5 ms. The results showed no tuple loss and confirmed that Esper is capable of handling large throughput with high incoming rate.

Last but not least, we practiced the stress testing described previously, while increasing the parallelism of the EsperBolt. Starting from the default parallelism value of 1, we increased the parallelism each time by 1 until we reached parallelism 5 and run the previous stress test with the InputSpout. Also in this case, the results were accurate and Esper supported its reliability.

Chapter 6

Conclusion

In this work, the main idea behind complex event processing and the concept upon which Esper is built is demonstrated. Windows, clauses, tables, functions and the EPL language are some of the Esper's aspects that were presented in the thesis.

Furthermore, a reliable and efficient a complex event processing system using the Esper engine was designed and tested. In order for this system to be implemented, the resources of the Ferari project were exploited. Esper was integrated into the Storm architecture which supports the Ferari project after certain configurations in the bolt hosting the CEP engine,took place. Consequently, the components and the functionality of each Ferari's module was described and analyzed. All the necessary modifications in terms of Java code about the Ferari-Esper sustainability, are available in this work. Also, Esper proved its efficiency to support the push/pull mechanism since it possesses all the necessary tools to face any challenge. Finally, through the experiments which are described and presented in the thesis, the robustness, reliability and scalability of the Esper engine is evidenced.

6.1 Future Work

Due to Esper's extended capabilities, there is definitely space for expansion of the current work. More specifically, in this work we focused on Time Windows which means that there can be relevant work with Length and Time Batch Windows in terms of experimental evaluation. Moreover, our implementation can be expanded by the use of tables which in combination with Fire-And- Forget queries, will provide extra functionality. Of course, patterns offer great effectiveness in match recognition, so there is room for future development in queries. Finally, since Esper supports user-defined functions, there is room for personal intervention in the overall Esper functionality.

References

[1] Ioannis Flouris, Vasiliki Manikaki, Nikos Giatrakos, Antonios Deligiannakis, Minos N. Garofalakis, Michael Mock, Sebastian Bothe et al. "FERARI: A Prototype for Complex Event Processing over Streaming Multi-cloud Platforms." *In Proceedings of the ACM SIGMOD/PODS Conference (2093-2096) San Francisco, June 2016*

[2] Ioannis Flouris, Vasiliki Manikaki, Nikos Giatrakos, Antonios Deligiannakis, Minos N. Garofalakis, Michael Mock, Sebastian Bothe et al. "Complex event processing over streaming multi-cloud platforms: the FERARI approach" *In Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS 2016: 348-349) Irvine, CA*

[3] Sebastian Bothe, Vasiliki Manikaki, Antonios Deligiannakis and Michael Mock. "Towards Flexible Event Processing in Distributed Data Streams." *In Proceedings of the Event Processing, Forecasting and Decision-Making in the Big Data Era (EPForDM) Workshop held in conjunction with EDBT/ICDT(111-117) Brussels, Belgium, March 2015*

[4] Ioannis Flouris, Nikos Giatrakos, Minos N. Garofalakis and Antonios Deligiannakis. "Issues in Complex Event Processing Systems." *Journal of Systems and Software 127: 217-236 , 2017*

[5] Vasiliki Manikaki. "Architecture and Implementation of a Distributed Complex Event Processing System." *Master Thesis, Technical University of Crete, Chania, April 2017*

[6] Esper Reference Documentation.

[7] http://www.espertech.com.

[8] Getting Started With Storm. *Jonathan Leibiusky, Gabriel Eisbruch, Dario Simonassi.*

[9] Storm Real-Time Processing Cookbook. *Quinton Anderson.*

[10] Stream Data Processing: A Quality of Service Perspective: Modelling, Scheduling, Load Shedding and Complex Event Processing *Sharma Chakravarthy, Qingchun Jiang.*

[11] Event Processing in Action. *Opher Etzion, Peter Niblett.*