PaaS-CEP
A Query Language for Complex Event Processing and Databases

Ricardo Jiménez-Peris[1], ValerioVianello[2] and Marta Patiño-Martinez[2]
[1]LeanXcale, Madrid, Spain
[2]Universidad Politécnica de Madrid, Madrid, Spain

Keywords: CEP, DBMS, SQL.

Abstract: Nowadays many applications must process events at a very high rate. These events are processed on the fly, without being stored. Complex Event Processing technology (CEP) is used to implement such applications. Some of the CEP systems, like Apache Storm the most popular CEPs, lack a query language and operators to program queries as done in traditional relational databases. This paper presents PaaS-CEP, a CEP language that provides a SQL-like language to program queries for CEP and its integration with data stores (database or key-value store). Our current implementation is done on top of Apache Storm however, the CEP language can be used with any CEP. The paper describes the architecture of the PaaS-CEP, its query language and the algebraic operators. The paper also details the integration of the CEP with traditional data stores that allows the correlation of live streaming data with the stored data.

1 INTRODUCTION

Nowadays, enterprises face the problem of processing large amount of unstructured data almost in real time. Complex event processing is an emerging technology that has the potential to process huge amounts of data in real time. In the last decade several implementations of CEP came out on the market from both academia and industry (Borealis, 2007), (Flink, 2015), (Spark, 2016). Among those, Apache Storm (Storm, 2015) is considered state of the art in distributed complex event processing. Storm is a distributed, reliable and fault-tolerant computation system currently released as an open source project by the Apache foundation. Storm is a distributed stream processing engine that can process on-the-fly data coming from different data sources to produce new streams of data as output. However, Storm does not provide a language for describing queries and operators on the streams. Everything is done programmatically. This approach although flexible, it is error prone and time consuming. More over, the integration of the CEP with data stores (relational databases or key-value data stores) is not fully addressed by CEP systems. This paper presents the complex event processing language available in the CoherentPaaS suite (CoherentPaaS, 2013) used in other settings like the LeanBigData project (LeanBigData, 2014). Paas-CEP language provides:

- A query language for the creation of CEP queries.
- A set of algebraic operators.
- Integration with external data stores.
- Pluggable CEP language for any CEP.

The current implementation of PaaS-CEP is based on Storm although, other CEPs can be plugged-in PaaS-CEP.

The rest of the paper presents the architecture of PaaS-CEP (Section 2), then the algebraic operators are presented in Section 3. Section 4 is devoted to the interaction with data stores. Section 5 presents the query language and query compiler. Section 6 concludes the paper.

2 ARCHITECTURE

PaaS-CEP is a parallel-distributed engine able to read and write raw data from/to external data stores and to materialize the results of continuous queries in such data stores. One of the main issues in the integration of the CEP with external data stores is the impedance mismatch between CEP queries that are continuous and SQL queries that are point-in-time. A CEP query is deployed and then, it is delivering results continuously till it is decommissioned. However, a

SQL query is a point-in-time query that processes existing data and delivers the result.

In order to solve this impedance mismatch PaaS-CEP integrates two new mechanisms. The first one enables CEP queries to correlate events in real-time with data stored in external data stores, for example to enrich the event with stored information, or to check whether there is related information in a data store or for storing some events. That is, the output of a CEP query can be used by the data stores queries. For this to happen, it materialization operators are needed. These operators can store the output of a CEP query in external data store.
Furthermore, in order to ease the use of the CEP, PaaS-CEP also offers a SQL-like language to formulate CEP query. Figure 1 shows the main components of PaaS-CEP using a block diagram.

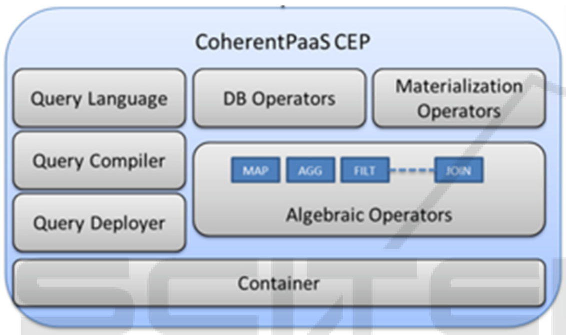


Figure 1: PaaS-CEP components.

The query language, together with the query compiler and deployer, allows programmers to define a query in a SQL-like language, translate it into the corresponding CEP query and deploy the resulting query in the container. The container can be any CEP, in this case we are using Storm although other CEPs can be used. Operators are the building blocks of the CEP continuous queries and they are classified in three main categories:

- CEP algebraic operators. They provide the basis for supporting CEP queries.
- CEP database operators. They enable to correlate events with information stored in external data stores.
- CEP materialization operators. They allow storing the output of CEP queries in external data stores.

3 ALGEBRAIC OPERATORS

CEP continuous queries are modelled as an acyclic graph where the nodes are streaming operators and arrows are streams of events. Streaming operators are computational boxes that process events received over the incoming stream and produce output events on the outgoing streams. Algebraic operators can be either stateless or stateful, depending on whether they operate on the current event (tuple) or on a set of events (window).

Stateless operators process incoming events one by one. The output of these operators, if any, only depends on the last received event. Stateless operators provide basic processing functionalities such as filtering and projection transformations. The stateless operators in PaaS-CEP are: *map, filter, multi-filter* and *union*. Their definition is presented in Table 1.

Table 1: Stateless operators.

Map	it is a generalized projection operator defined as: $Map(S) = \{ A'_1 = f_1(t), A'_2 = f_2(t), . . . , A'_n = f_n(t) , O\}$ It requires one input stream and one output stream. The schema of these two streams may be different. The *map* operator transforms each tuple *t* on the input stream *S* by applying a boolean and/or arithmetic expression (f_i). The resulting tuple has attributes $A'_1, . . . , A'_n$ where, $A'_i = f_i(t)$, and is sent through the output stream *O*.
Filter	it is a selection operator defined as: $Filter(S) = \{(P(t), O)\}$ The *filter* operator requires one input stream and one output stream with the same schema. It verifies the match of tuples *t* on the input stream *S* with the user defined predicate *P*. When *P(t)* is satisfied the tuple *t* is emitted on the output stream *O*.
MultiFilter	it is a selection and semantic routing operator defined as: $MultiFilter(S) = \{(P_1(t), O_1), (P_2(t), O_2), . . . , (P_n(t), O_n)\}$ The *multiFilter* operator requires one input stream and at least one output stream, all with the same schema. The multiFilter emits a tuple *t* on all the output streams O_i for which the user defined predicate, $P_i(t)$ is satisfied.
Union	it is a merger operator defined as: $Union(S_1, S_2, . . ., S_n)\{O\}$ The *union* operator requires at least one input stream and only one output stream, all with the same schema. It is used to merge different input streams S_i with the same schema into one output stream *O*.

Stateful operators perform operations over a set of incoming events called *sliding window*. A sliding window is a volatile memory data structure. PaaS-CEP defines three types of sliding windows:

- Tuple-based window: it stores up to *n* tuples.
- Time-based window: it stores the tuples received in the last *t* seconds.
- Batch-based window: it stores all the tuples received between a start and stop conditions.

Tuple and time based windows must be configured with the *size* and *advance* parameters. The *size* parameter defines the capacity of the window (number of events/time in seconds) and the *advance* parameter defines which events must be removed from the window when the window is full. Two statefull operators are defined in PaaS-CEP : *aggregate* and *join*. Table 2 presents the definition of these operators.

Table 2: Stateful operators.

Aggregate	It computes aggregate functions (e.g., sum, average, min, count, ...) on a window of events. It is defined as: *Aggregate(S) = { A'*$_1$ *=* f_1*(t ,W), . . . , A'*$_n$ *=* f_n*(t ,W) , s, adv, t, Group-by(A*$_1$*, . . . , A*$_m$*), O}* The aggregate operator accepts only one input stream and defines one output stream. It supports both time based sliding windows and tuple based sliding windows. Parameters *s*, *adv* and *t* define the size, the advance and the type of the sliding window. The Group-by parameter indicates how to cluster the input events; that is, the operator keeps a separate window for each of cluster defined by the attributes *(A*$_1$*, . . . , A*$_m$*)*. Any time a new event *t* arrives on the input stream and the sliding window of the corresponding cluster is full, the set of aggregate functions $\{f_i\}_{i1\leq i\leq n}$ are computed over the events in that sliding window *W*. The resulting tuple with attributes *A'*$_1$*, . . . , A'*$_n$ where, *A'*$_i$ *=* f_i*(t ,W)*, is inserted in the output stream *O*. Finally, after producing the output tuple, all the windows are slid according with the advance *adv* parameter.
Join	It joins events coming from two input streams. It is defined as: *Join(*S_l*,*S_r*) = {A'*$_1$ *=* f_1*(t ,*W_l*,*W_r*),.. , A'*$_n$ *=* f_n*(t ,* W_l*,*W_r*) , P,* w_l*,* w_r*, Group-by(A*$_1$*, .., A*$_m$*), O}* The join operator accepts two input streams and define one output stream. S_l identifies the left input stream and S_r identifies the right input stream. *P* is a user defined predicate over pairs of events t_l and t_r belonging to input streams S_l and S_r, respectively; w_l and w_r define the size and the advance of the left and right sliding windows while de group-by defines the clustering as in the aggregate operator. In order to be deterministic the join operator only supports time based sliding windows. In the following we consider the simplified situation where the group-by parameter is empty and there is only one sliding window per stream. For each event t_l received on the input stream S_l (respectively t_r from stream S_r) the concatenation of events $t_l \mid t_i$ is emitted on the output stream *O,* if these conditions are satisfied: (1) t_i is a tuple currently stored in W_r (respectively in W_l) (2) P is satisfied for the pair t_l and t_i (respectively t_r and t_i) The attributes *A'*$_1$*, . . . , A'*$_n$ of tuples that are indeed inserted in the output stream *O* are a subset of the concatenation of events $t_l \mid t_i$ where, *A'*$_i$ *=* f_i*(t ,*W_l*,*W_r*)*. After that all the output tuples triggered by the tuple t_l (respectively t_r) received on the input are produced, the sliding window W_r (respectively in W_l) is slid according with the advance parameter.

Figure 2 shows an example of the *Map* operator. In the example the *Map* is used to transform the input tuples, with the schema [idcaller, idreceiver, duration, timestamp] representing a simplified Call Description Record (CDR) by adding a new field *cost* evaluated with the expression cost=duration*10 + 10

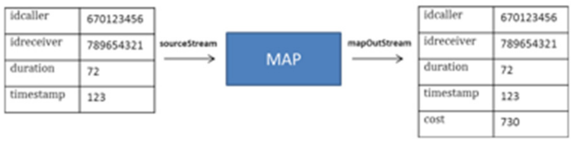


Figure 2: Example of Map operator.

4 CEP INTEGRATION WITH DATA STORES

CEP systems, since are in-memory processing systems, do not have the notion of transactions. Additionally, tuples are handled fully independently which makes difficult to define the notion of a transaction. However, CEP queries need many times to access transactional data stores for correlating the incoming data with the stored data. Relational databases provide to ways to interact with them: the so called auto-commit mode, where each SAL sentence executed on the database is a transaction, or bracketing mode, where a set of sentences is executed as a transaction. In this paper we only address the auto-commit mode and define operators for accessing a data store in that mode.

4.1 Database Operators

Data store operators provide the CEP with the capability of reading and writing tuples from/to an external data store. PaaS-CEP accesses the data stores using operators that issue queries written in SQL. Operators that access the data stores must be able to retrieve data at high rates in order to correlate stored data with the large amount of events produced by the CEP. At the same time the data stores must be able to store the results produced by the CEP at very high rates. The available data store operators are described in Table 3.

Table 3: Data store operators.

ReadSQL	The *ReadSQL* operator requires one input stream, S, and one output stream. The schema of these two streams may be different. The operator is configured with a parameterized query to be run against a data store. The parameterized query must be a *SELECT* statement. For each tuple, *t,* received on the input stream, *S,* the operator replaces the parameters in the query with the values read from the corresponding fields in the input tuple t and then, it executes the query. The operator produces as many tuples on the output stream as tuples has the result set of the query executed on the data store. Each output tuple is created either using fields of the incoming tuple, *t*, or fields of the result set row or a combination of them.
UpdateSQL	The *UpdateSQL* operator is in charge of storing results of the CEP query in a data store. It requires one input and one output stream. The schema of these two streams may be different. This operator is also configured with a parameterized query that must be an update statement, that is, it modifies, inserts or deletes data in the data store. For each tuple, *t*, received on the input stream S, the *UpdateSQL* operator replaces the parameters in the query with the values from the corresponding fields in the input tuple and then, it executes the query updating the data store. The *UpdateSQL* operator creates one output tuple for each input tuple. The output tuple can be either a copy of the input tuple or the number of modified rows in the data store or a concatenation of the two.

Figure 3 shows an example of the *ReadSQL* operator. The operator receives CDRs and fetches from an external data store the monthly plan (idplan) of the user making the phone call. The output tuple is composed by the fields idcaller, duration and timestamp of the input tuple plus the idplan field read from the data store.

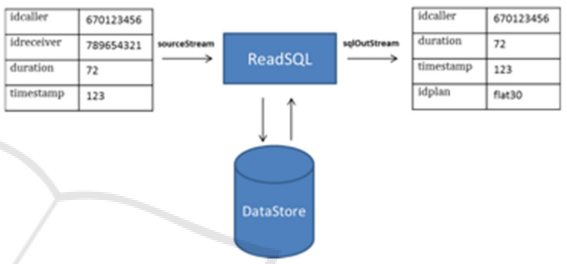


Figure 3: Example of ReadSQL operator.

5 QUERY LANGUAGE

Programmers can use the operators to generate continuous queries however, most programmers are familiar with SQL language. Our proposal is to provide CEPs with a SQL-like language to ease the programmer task. PaaS-CEP Query Language (CPL) is defined as a subset of the traditional SQL language where tables are replaced by continuous streams.

It allows to feed and filter events from one stream to another one, to aggregate information of a stream, to join or merge events from different streams. The query language is similar to the one of Esper (Esper, 2006) however, Esper is a centralized CEP while our approach is generic and can be plugged into any available CEP, and targets distributed CEP.

CPL queries always start with the declaration of streams and windows structures followed by the definition of one or more statements over these streams and windows. Each statement requires at least one input and one output stream. Input streams must be defined either before any statements or as output stream of previous statements. Streams can be defined at the beginning of a CPL query using the *CREATE STREAMSCHEMA* clause and they can have an arbitrary number of fields:

CREATE STREAMSCHEMA s1 WITH fieldname fieldType [, fieldname fieldType] [, ….]

FieldTypes can be chosen among several common types such as boolean, integer, double, char, varchar, etc. Each statement must include the three mandatory clauses *INSERT INTO, SELECT* and *FROM. INSERT INTO* is used to name the output stream of a statement:

INSERT INTO streamName

streamName is the name to give to the output stream and it can be any literal string. The *FROM* clause specifies the source streams for a statement and its syntax is:

FROM streamName [, streamName] [, …]

streamName is the name of the input stream of the statement. The number of parameters in a *FROM* clause depends on the number of input streams of the statement. If the statement is doing a projection, transformation or aggregation then, there is only one parameter. The *SELECT* clause is used to picking data from streams. Depending on the number of source streams defined in the statement, the *SELECT* clause is used to (i) select or (ii) rename fields, (iii) evaluate expressions over multiple fields and (iv) compute aggregation functions over single fields. As an example, the syntax to evaluate aggregation function over an input stream is:

SELECT aggFunc(fieldname) AS newFieldname [, aggFunc(fieldname) AS newFieldname] [, …]

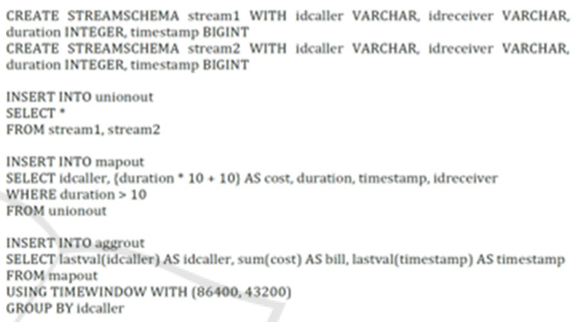
where *aggFunc* is an aggregation function (maximum, minimum, sum, average, etc) and *AS* is the clause used to name the result of the function in the output stream. Some other clauses available in CPL language are: *WHERE, HAVING, GROUP-BY* and *ON SQL.* In particular, the *ON SQL* clause is used to execute a classical SQL query against an external data store. The syntax for *ON SQL* clause is:

ON SQL queryName, tableName, fieldname1 [,fieldname2][,..],outfieldname1[, outfieldname2][,..]

queryName is a variable with the SQL statement to be executed. *tableName* is the name to give to the result set of the SQL query. *fieldnames* are fields belonging to the input stream used as parameter for the SQL statement.*outfieldname* is used to identify the fields of the result set in order to be used in other clauses such as *SELECT* or *WHERE.*

As an example of a whole CQL query let us consider a simple scenario where there are two streams with Call Description Records (CDR) events into the CEP and we want to calculate the daily bill of each phone number. We assume that:

- A CDR event has the following fields: *idcaller* (id of the caller), *idreceiver* (id of the receiver), *duration* (duration of the phone call in seconds), *timestamp* (timestamp of the phone call).
- Calls with a duration less than 10 seconds are free.
- The cost of the call is calculated with the formula: *cost=duration*10 +10*



```
CREATE STREAMSCHEMA stream1 WITH idcaller VARCHAR, idreceiver VARCHAR,
duration INTEGER, timestamp BIGINT
CREATE STREAMSCHEMA stream2 WITH idcaller VARCHAR, idreceiver VARCHAR,
duration INTEGER, timestamp BIGINT

INSERT INTO unionout
SELECT *
FROM stream1, stream2

INSERT INTO mapout
SELECT idcaller, (duration * 10 + 10) AS cost, duration, timestamp, idreceiver
WHERE duration > 10
FROM unionout

INSERT INTO aggrout
SELECT lastval(idcaller) AS idcaller, sum(cost) AS bill, lastval(timestamp) AS timestamp
FROM mapout
USING TIMEWINDOW WITH (86400, 43200)
GROUP BY idcaller
```

Figure 4: Query written using the CEP query language.

The query for billing clients is shown in Figure 4. First, the schema of the two input streams is declared. Then a first statement (*INSERT*) is used to merge the two streams into one output stream named *unionout*. A *SELECT* statement is used to calculate the cost of each phone call and to filter out all these calls whose duration in shorter than 10 seconds. Finally, the last *SELECT* statement calculates the daily bill for each user.

5.1 Compiler

The query compiler is the component in charge of translating queries written using the declarative language (SLQ like) into continuous queries made by algebraic operators. As an example the query compiler can transform the CQL query from Figure 4 into the graph of algebraic operator depicted in Figure 5

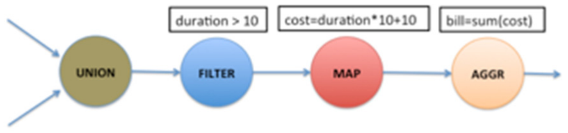


Figure 5: Query translated into graph of algebraic operators.

6 CONCLUSIONS

This paper presented PaaS-CEP, a complex event-processing engine that provides a SQL-like language for programming queries over streaming data. The main features PaaS-CEP provides are:

1. Algebraic operators.
2. Integration the CEP with external data stores allowing the correlation of streaming data with historical data in an ACID way.
3. Definition of a query language to ease the definition of continuous CEP query.

ACKNOWLEDGEMENTS

This research has been partially funded by the European Commission under projects CoherentPaaS and LeanBigData (grants FP7-611068, FP7-619606), the Madrid Regional Council, FSE and FEDER, project Cloud4BigData (grant S2013TIC-2894), and the Spanish Research Agency MICIN project BigDataPaaS (grant TIN2013-46883)

REFERENCES

CoherentPaaS, 2013. *CoherentPaaS project web site* http://www.coherentpaas.eu Last visited 20/01/2016.

Storm, 2015. *Aoache Strom web page* http://www.coherentpaas.eu Last visited 20/01/2016.

Esper, 2006. *Esper EQL documentation*. http://www.espertech.com/esper/documentation.php Last visited 20/01/2016.

LeanBigData, 2014. *LeanBigData project web site* http://leanbigdata.eu/ Last visited 20/01/2016.

Borealis, 2007. *The Borealis project*, http://cs.brown.edu/research/borealis/public/ Last visited 20/01/2016.

Flink, 2015. *Apache Flink web page*, https://flink.apache.org/ Last visited 20/01/2016.

Spark, 2016. *Apache Spark streaming web page*, http://spark.apache.org/streaming/