CoherentPaaS

Coherent and Rich Peas with a Common Programming Model

ICT FP7-611068

Table Oriented Cloud CEP Design

D7.1

March 2014

Document Information

Scheduled delivery	31.03.2014
Actual delivery	11.04.2014
Version	1.0
Responsible Partner	UPM

Dissemination Level:

PU Public

Revision History

Date	Editor	Status	Version	Changes
14.03.2014	Ricardo Jiménez	Draft	0.1	table of contents
19.03.2014	Valerio Vianello	Draft	0.2	First internal draft
21.03.2014	Ricardo Jimenez	Draft	0.3	Internal draft
24.03.2014	Marta Patiño	Draft	0.4	Internal draft
25.03.2014	Valerio Vianello	Draft	0.5	Version for peer review
01.04.2014	Ricardo Jimenez	Draft	0.6	Revision incorporating reviewer comments
02.04.2014	Valerio Vianello	Draft	0.7	Revision incorporating reviewer comments
02.04.2014	Ricardo Jimenez	Draft	0.8	Internal review
02.04.2014	Valerio Vianello	Draft	0.9	Final Draft
03.04.2014	Marta Patiño	Draft	0.91	Final Draft
10.04.2014	Valerio Vianello	Draft	0.95	Revision incorporating reviewer comments
11.04.2014	Ricardo Jimenez	Final	1.0	Final Delivery

Contributors

Ricardo Jiménez, Valerio Vianello, Marta Patiño

Internal Reviewers

Martin Kersten(MonetDB), Raquel Pau (Sparsity), Vassilis Spitadakis (Neurocom)

Acknowledgements

Research partially funded by EC 7th Framework Programme FP7/2007-2013 under grant agreement n° 611068.

More information

Additional information and public deliverables of CoherentPaaS can be found at: http://coherentpaas.eu

Glossary of Acronyms

Acronym	Definition
D	Deliverable
DoW	Description of Work
EC	European Commission
PM	Project Manager
PO	Project Officer
WP	Work Package
EPL	Eclipse Public License
SPE	Stream Processing Engine
TCP	Transmission Control Protocol
CEP	Complex Event Processing
DBMS	Database Management System
SQL	Structured Query Language

Table of Contents

List of Figures

List of Tables

1. Executive Summary

This deliverable presents the architecture of the CEP subsystem and its integration with the holistic transactions and the other data stores of the CoherentPaaS architecture. The main issue solved in this deliverable is the impedance mismatch between the CEP system and the point-in-time queries used in the data stores.

This impedance is solved by a combination of two techniques. The first technique lies in the introduction of CEP database operators that enable to access CoherentPaaS data stores. Operators allow correlating events with data stored in any cloud data store (e.g. enrich the event with stored information, or check whether there is a corresponding stored record in a data store). The second technique lies in enabling the use of the output of a CEP query by the data stores queries. This technique lies in materialization operators that can store the output of a CEP query in a CoherentPaaS data store.

The CEP system consists of several components: query compiler (generates the code of a query from a SQL-like query description), query deployer (deploys a query in a running system) and CEP engine (processes the events by means of the deployed queries). In logical terms, it has a set of operators that are classified as regular CEP operators, and the additional operators we mentioned earlier: database operators and materialization operators.

In order to ease the use of the CEP, instead of forcing the application developers to learn to use them, we have created a SQL-like language to formulate the CEP queries. The CEP compiler takes care of transforming queries written in the SQL-like language into CEP queries using the CEP operators. Since the operators by the underlying CEP system, Storm are too low level, relational algebra operators will be developed on top of storm, and SQL-like queries will be translated into these operators.

The deployer then takes care of deploying the generated query on a particular setup.

Finally, another important integration point with the project contributed by this deliverable lies in how to integrate with the holistic transactions. Two mechanisms have been introduced for this. The first mechanism provides read consistency for events when no transactional context is specified by the developer. This read consistency ensures that the events stemming from an event injected by an event source will read a consistent snapshot when they read from the CoherentPaaS data stores. The second mechanism enables to associate a transaction to a batch of events, so each time a batch of events stemming from these events reach a database operator they are written to the database as part of a transaction to guarantee all-or-nothing semantics for the set of related events (e.g. several tuples coming from a set of windmills in a wind farm that we want to store in an all-or-nothing manner).

Finally, the technical architectural solution to integrate with the transactional system is discussed and one alternative is chosen lying in using a JDBC driver that has as main advantage that failures in the CEP system do not force recovery on the transactional system.

2. Storm Overview

Storm is the distributed, reliable and fault-tolerant computation system currently integrated into Twitter architecture and several other companies [1]. Twitter acquired the social media analytics company that developed Storm and they decided to release it as an open source project on GitHub [2] under the Eclipse Public License (EPL) [3] in 2011. Storm is a stream processing engine (SPE) that can process on-the-fly data coming from different data sources to produce new streams of data as output.
Storm is implemented in Java. Out of the box Storm provides several artefacts and functionalities, in the following of this Section these concepts are introduced:

- Stream
- Spout
- Bolt
- Topology
- Storm Cluster
- Parallelism of Topologies
- Communication Mechanisms

2.1. Stream

A stream is an infinite sequence of tuples, ordered on time, with the same schema (Figure 1). A tuple represents an instantaneous occurrence of an event of interest in a system at a certain point of time. Each and every tuple of a stream has a pre-defined schema (A_1, A_2, … , A_N), where A_j defines the type of the j-th attribute of the tuple. Storm allows the user to define custom types for the schemas having the constraint that these new types must be serializable. Furthermore, Storm makes available several built in types for stream schemas such as: integer, long, short, byte, string, doubles, float, boolean and byte array.

Figure 1 – Stream of tuples.

2.2. Spout

Spouts are the Storm components in charge of feeding the initial streams with tuples. These tuples can be either generated by the spout itself or fetched/received from external data sources. In Storm there are two types of spouts: reliable and unreliable spouts. The former is a spout able to replay a tuple if Storm detects a failure during its processing. At the contrary, unreliable spouts forget about tuples as soon as they leave the spout. Spouts can emit tuples on any number of streams (Figure 2).

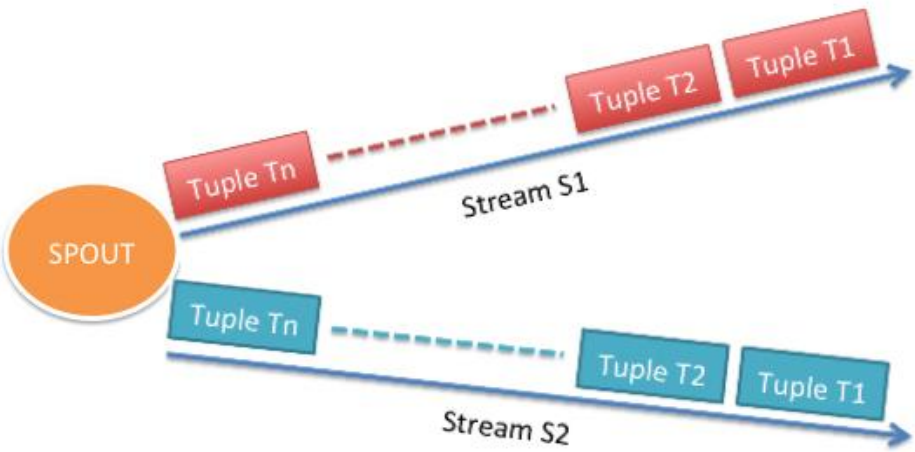

Figure 2 – Storm Spout with two streams.

2.3. Bolt

In the Storm architecture, bolts are the basic unit of processing. They provide all the mechanisms needed for managing any number of input and output streams but it is up to the user to add the business logic (Figure 3).

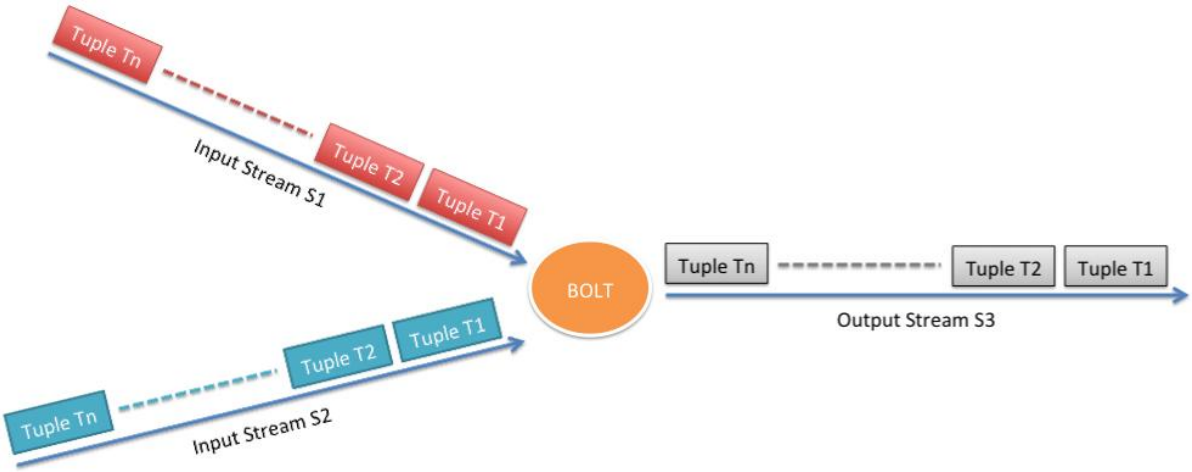

Figure 3 – Storm Bolt with two input streams and one output stream.

2.4. Topology

A topology is the top-level abstraction to do computation on Storm. Topologies are direct graphs with input and output edges representing input and output streams. Each node of the graph is either a spout or a bolt. An edge of the graph between the nodes “n” and “m” means that the output stream of node “n” is the input stream of node “m”.

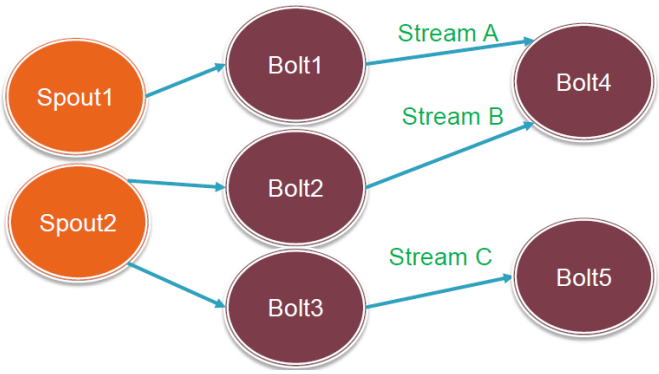

Figure 4 – Example of Storm Topology

Figure 4 shows an example of a topology where the spout Spout1 sends tuples to the bolt "Bolt1" and the spout "Spout2" to the bolts "Bolt2" and "Bolt3". Furthermore bolt "Bolt4" is subscribing to "Stream A" and "Stream B", "Bolt5" is subscribing to "Stream C". Storm Topologies can be seen as continuous queries over the data coming from data sources (spouts). Topologies run continuously over Storm producing results each time the input data is processed by the bolts of the topology.

2.5. Storm Cluster

The Storm Cluster is defined as the whole set of nodes and processes used to execute topologies. The nodes of a Storm Cluster can be of three types: (i) Master node, (ii) Worker node and (iii) Zookeeper node.

In a Storm Cluster there can be one and only one *Master node* with the *Nimbus* daemon running on it. The *Nimbus* daemon (a Storm process) is in charge of distributing the topologies code around the cluster, assigning tasks to the *Worker nodes* and monitoring the failures.

Unlike the Master node, there must be at least one Worker node per Storm cluster. Each Worker node is controlled by a *Supervisor* daemon (a Storm process). The *Supervisor* daemon can run several *worker* processes and it assigns work to them based on what the *Nimbus* daemon has established. A *worker* process belongs to a specific topology and it can run some of its components (spouts and bolts).

Zookeeper nodes are used to deploy a Zookeeper cluster [4] which maintains all the configuration information and naming conventions. Zookeeper provides the *Nimbus* and the *Supervisors* with distributed synchronization and group services. Additionally, *Nimbus* and *Supervisor* daemons are fail-fast and stateless. In fact, they immediately report any failure and the whole Storm Cluster state is kept in Zookeeper. If they die, they will restart like nothing happened.

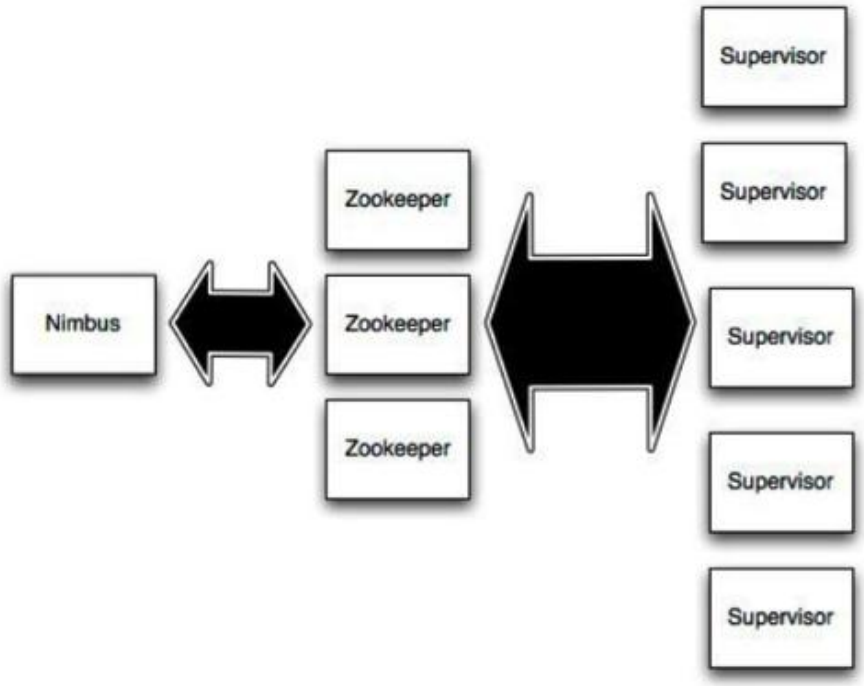

Figure 5 – Storm Cluster [5]

2.6. Parallelism of Topologies

Storm architecture allows spouts and bolts to be executed in parallel. This means that when the tuple rate on the input streams overcome the processing power of a single node, a topology can be deployed on multiple nodes taking advantage of the distributed processing power. Storm gives support for both inter-topology and inter-operator parallelism. With the *inter-topology* mechanism, the operators (built on top of spouts and bolts) of a topology can run on different machines. With the *inter-operator* mechanism, a single operator can be executed on multiple nodes. In any case, the result of the parallel-distributed execution would be the same result produced by the same topology executed standalone in only one node.
The parallelization management is accomplished in Storm with three entities:

- *Worker process*: this process runs in a Worker node controlled by the Supervisor. Each worker can run one or more executors of a topology.
- *Executor*: an executor is a thread spawned by the worker process. Each executor runs one or more tasks of the same operator.
- *Task*: A task performs the actual data processing.

According to the Storm documentation the number of tasks of an operator is always the same during the lifetime of a topology, hence the condition "*# executors ≤ # tasks*" is always true. The default configuration says to Storm to run one task per executor. Tasks are mainly used to assign and rebalance the load of a topology on the Storm cluster [6].
Figure 6 shows a topology with tasks set for each component. In particular there is a spout with 2 tasks and there are three bolts with respectively 4, 3 and 2 tasks.

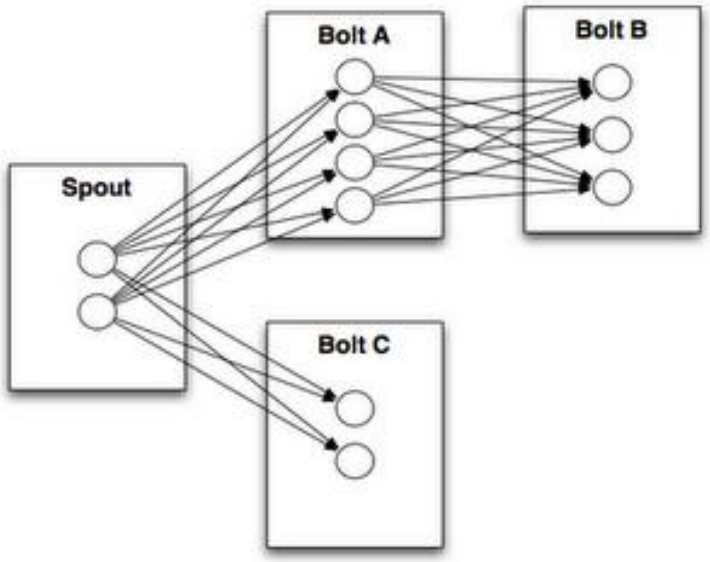

Figure 6 – Storm Topology at the task level [5]

Storm uses the "*stream grouping*" protocols in order to route the tuples when the components of a topology are defined with multiple tasks. The stream grouping protocols are used to instruct the Storm cluster on how a specific stream should be partitioned among the operator's task. There are several built-in stream groupings in Storm [7] and it also allows users to define their own stream grouping protocols. The most interesting protocols are:

- Shuffle grouping: tuples are evenly distributed across all the tasks of an operator.
- Fields grouping: the stream is partitioned by the fields specified in the grouping and tuples with the same value in these fields are always sent to the same task.
- All grouping: the tuples of the stream are replicated and sent to all the tasks.

2.7. Communication Mechanisms

In a Storm Cluster there are three types of communication mechanisms. The most common one is the *Intra-worker* communication also named internal messaging. This is the messaging mechanism used by the executors of a worker process to communicate among them. Storm implements this communication by means of the LMAX Disruptor [8], which is a high performance inter-thread messaging library.
Figure 7 from [9] describes the internal messaging communication of a worker process giving emphasis to the management of queues.

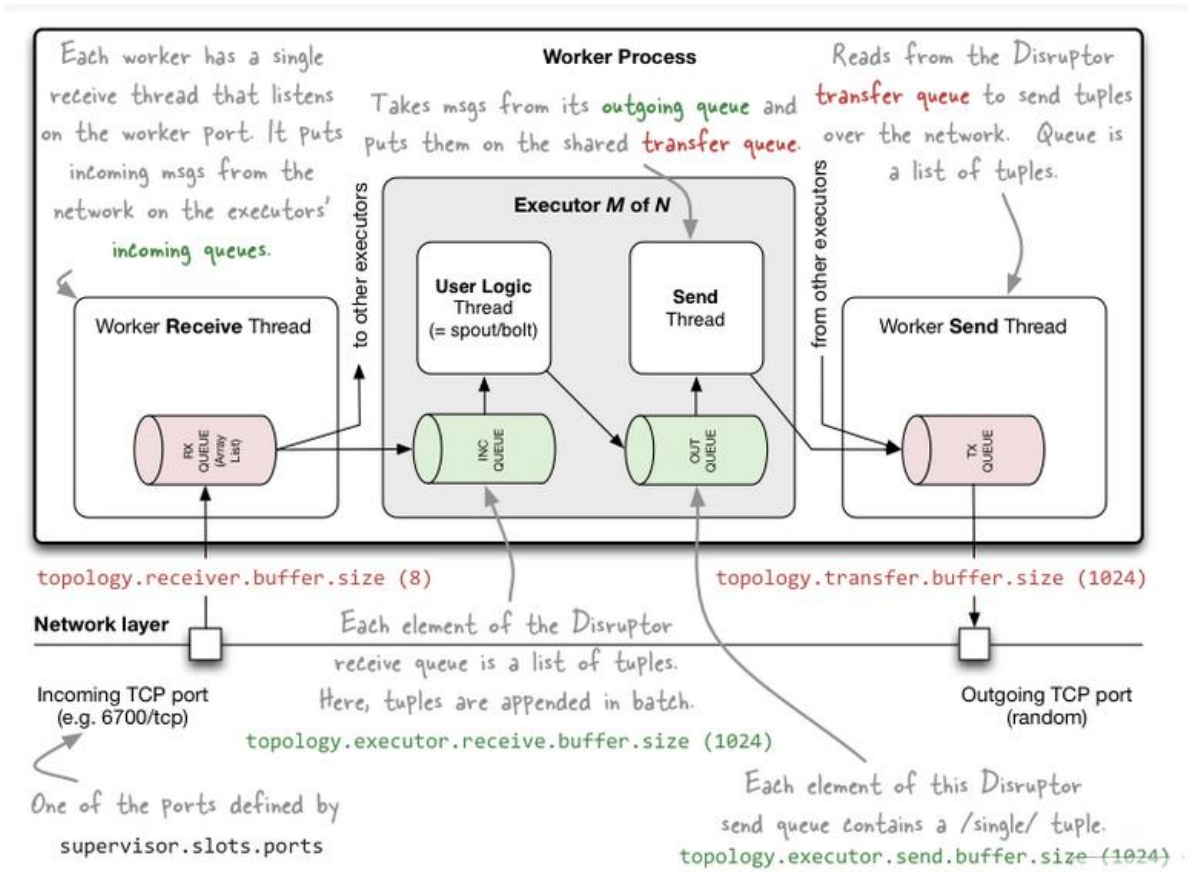

Figure 7 – Overview of a worker's internal message queues in Storm [9]

In Figure 7 two types of queues can be distinguished, the worker process queues are coloured in red and the ones belonging to the executors are coloured in green. Each worker has two threads for managing its incoming and outgoing tuples. The *Receive thread* listens on a TCP port and puts the incoming tuples in the corresponding queue. On the other hand, the *Send thread* reads messages from the worker transfer queue and sends them over the network to downstream workers.
Each executor controlled by a worker has its own incoming and outgoing queues. The Receive worker thread is also responsible of moving tuples from the worker input queue to the corresponding executor incoming queue. It is worth recalling that a specific worker can control multiple executors. Finally, the executor has its own *Sending thread* that sends the executor's outgoing tuple from the outgoing queue to the parent worker output queue.
The other communication mechanisms used in the Storm Cluster are:

- *Inter-worker* communication: this form of communication goes through the network because normally happens across machines. Storm can be configured either with ZeroMQ [12] or Netty [13] from version 0.9.0. These protocols are used when a task in a certain worker wants to send data to a task running in worker process of a different machine in the Storm Cluster.

- *Inter-topology* communication: this type of communication refers to the communication among topologies that is, an operator belonging to a specific topology sends a tuple to an operator running in other topology. Currently Storm

does not provide any implementation for this communication scenario and it is up to the user its implementation.

3. Architecture Overview

3.1. Global view

Section 2 presented Storm, a stream processing engine (SPE) that provides an infrastructure for doing high performance and reliable parallel distributed computation. The goal of this Section is to present the architecture of a Table-oriented Complex Event Processing (CEP) engine that can be built on top of a SPE infrastructure such as Storm. A Table-oriented CEPs are defined as Complex Event Processing engine able to read and write raw data from/to external data storages and to materialize the results of continuous queries in such data storages.

One of the main issues we have faced is the impedance mismatch[1] between CEP queries that are continuous and SQL queries that are point-in-time. A CEP query is deployed and then is delivering results continuously till is decommissioned. However a SQL query (or any other kind of query like the ones in NoSQL data stores) is a point-in-time query that processes existing data and delivers the result.

In order to solve this impedance mismatch we have integrated two new mechanisms. The first one enables CEP queries to correlate events in real-time with data stored in any of the CoherentPaaS data stores. The second one enables CEP engines provide the users with the capability of easily defining continuous queries for the detection of meaningful events over raw data. These queries, combining data from multiple data sources, can infer patterns of events with much higher semantic information than the simple events generated by the data sources.
Operators are the building blocks of continuous query processing and they can be clustered in the following categories:

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

The first two categories are described in the rest of this Section and the CEP materialization operators are described in Section 5.

3.2. CEP Query Operators

The standard CEP operators are: Map, Filter, Union, Aggregate and Join, according to [10][11]. These operators, along with the MOFilter operator, which is a generalization of the Filter operator, can be divided into two categories: stateless and stateful. In what follows we provide the specification of the operators that we will develop for our table-oriented CEP.

[1]Impedance mismatch is a common term in computer science to denote a mismatch between two paradigms. It has been heavily used for instance to refer to the different scopes of the object and relational models. See http://www.agiledata.org/essays/impedanceMismatch.html

3.2.1. Stateless Operators

Stateless operators process each tuple independently. The operator output tuples, if any, only based on the information in the current tuple. Stateless operators´ logic enables to perform transformations on individual tuples, they provide basic processing functionalities such as filtering and projection transformations.

- *Map*: it is a generalized projection operator defined as:

  $Map(S) = \{ A'_1 = f_1 (t), A'_2 = f_2 (t), \ldots , A'_n = f_n (t) , O\}$

  It requires one input stream and one output stream. The schema of these two streams may be different. The Map transforms each tuple *t* on the input stream *S* by applying a logical and/or arithmetic expression (f_i) . The resulting tuple with attributes $A'_1, \ldots , A'_n$ where, $A'_i = f_i (t)$, is sent through the output stream *O*. Figure 8 shows an example of Map operator using three expressions to transform tuples from the input stream *S* into tuples of the output stream *O*. The operator transforms the schema of the tuple from (*In1, In2, In3, TS_In*) to (*Out1, Out2, TS_OUT*) according to the expressions: *Out=In1+In2*, *Out2=In3* and *TS_Out=TS_In*.

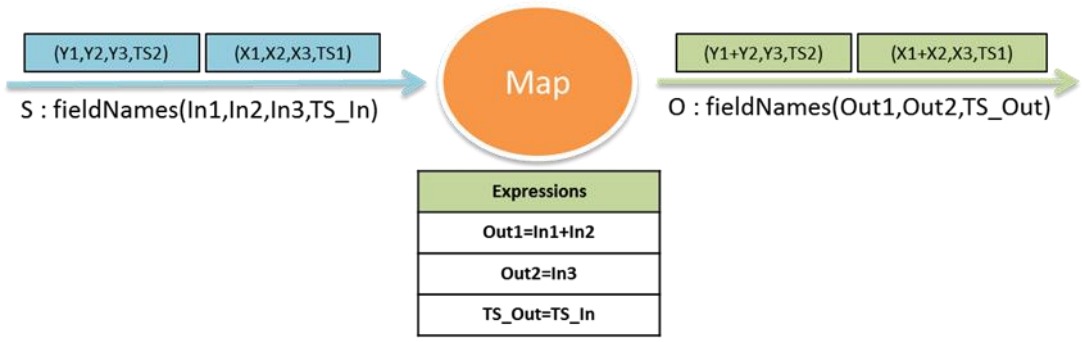

Figure 8 – Example of Map Operator

- *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*. Figure 9 shows an example of Filter operator. In this example the operator emit on the output stream *O* only the tuples (whose schema is *In1, In2, In3, TS_In*) that satisfy the condition *In1==X1 OR In2==Z2*.

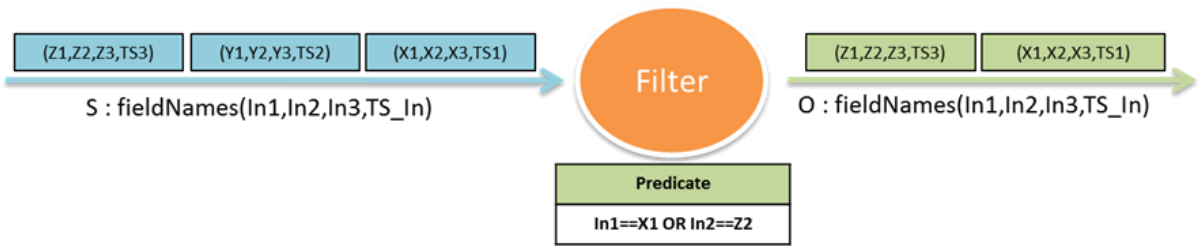

Figure 9 – Example of Filter Operator

- *MOFilter*: it is a selection and semantic routing operator defined as:

  $MOFilter(S) = \{(P_1(t), O_1), (P_2(t), O_2), \ldots, (P_n(t), O_n)\}$

  The MOFilter operator requires one input stream and at least one output stream, all with the same schema. The MOFilter emits a tuple t on all the output streams O_i for which the user defined condition $P_i(t)$ is satisfied. Figure 10 depicts the scenario where a MoFilter operator receives tuples from the stream S with schema *(In1, In2, In3, TS_In)* and it routes on the output stream O_1 those tuples for which the predicate *In1==X1* is true and on the output stream O_2 those one which verify the condition *In1==Z1*.

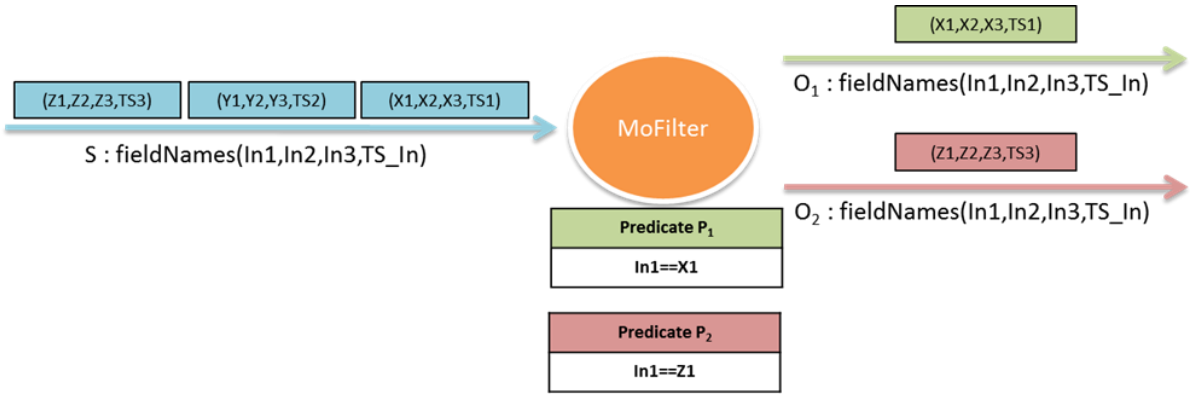

Figure 10 – Example of MoFilter Operator

- *Union*: it is a merger operator defined as:

  $Union(S_1, S_2, \ldots, S_n)\{O\}$

  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. Figure 11 shows the Union operator in example scenario. The operator takes two input stream S_1 and S_2, with the same schema *(In1, In2, In3, TS_In)*, and it inserts the received tuples in the output stream O according to their arrival order.

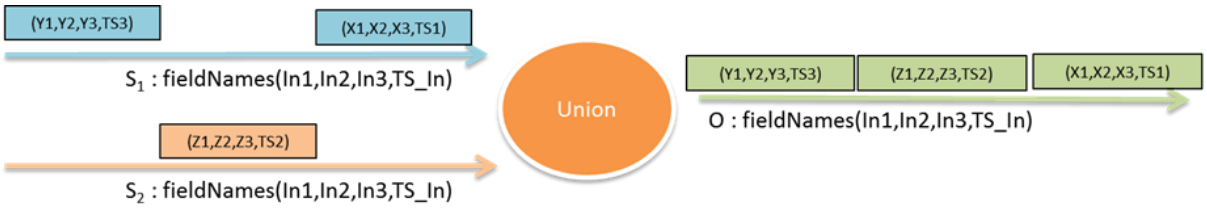

Figure 11 – Example of Union Operator

3.2.2. Stateful Operators

Stateful operators keep into sliding windows the tuples received from one or more input streams during a specific amount of time. These operators are the ones that are more distant from Storm functionality, which does not provide any support for sliding windows. Their output is a function of all the tuples stored in the window. Sliding windows are volatile memory structures defined by three parameters:

- *Size*: defines the capacity of the window.
- *Advance*: defines how much slide the window when this becomes full.
- *Type*: defines how the window must slide, either based on time or on the number of the stored tuples.

For tuple based sliding windows, the *size* parameter is expressed as the maximum number of tuples that can be kept in the window. For time based sliding window, the *size* parameter sets the size of the window in terms of seconds. The window is slid any time the difference in time between the newest tuple and the oldest one is greater than the window size.

There are two types of stateful operators: Aggregate and Join.

- *Aggregate*: it computes aggregate functions (e.g., sum, average, min, count, ...) on a window of tuples. It is defined as:

  $Aggregate(S) = \{ A'_1 = f_1(t,W), \ldots, A'_n = f_n(t,W), s, adv, t, Group\text{-}by(A_1, \ldots, A_m), O\}$

  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 tuples; that is, the operator keeps a separate window for each of cluster defined by the attributes $(A_1, \ldots, A_m)$. Any time a new tuple *t* arrives on the input stream and the sliding window of the corresponding cluster is full, the set of aggregate functions $\{f_i\}_{i1 \le i \le n}$ are computed over the tuples in that sliding window *W* and on the current tuple *t*. The resulting tuple with attributes $A'_1, \ldots, A'_n$ 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. Figure 12 shows an example of the Aggregate operator with t time based sliding window of size 3 and advance 3 (This means that any time the window is full a new tuple is emitted and the window is emptied). Tuples on the input stream *S* have the schema *(In1, In2, TS_In)* while output tuples are produced according with the

schema *(Out1, Out2, TS_Out).* The operator inserts the first 3 tuples in the window without producing any output because the difference in time between the first and last received tuple is smaller than the window size (size=3 and difference = 2). When the operator receives the forth tuple, it inserts the tuple in the window and detects that the windows has become full. At this point, the operator applies the functions *(Out1=Sum(In2), Out2=Count() and TS_Out=LastVal(TS_In)*) over all the tuples kept in the window and finally it emit the resulting tuple on the output stream *O*.

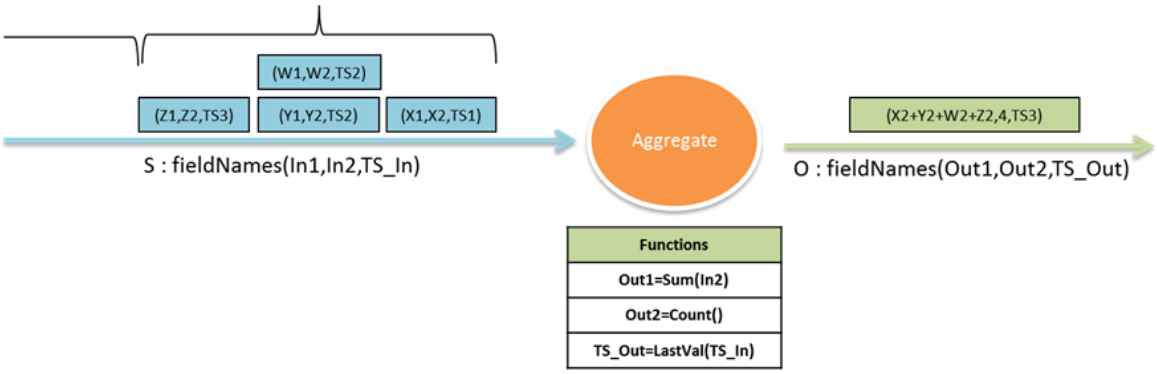

Figure 12 – Example of Aggregate Operator with time based Sliding Window, size 3 and advance 3

- *Join*: it correlates tuples coming from two input streams. It is defined as:

  $Join(S_l,S_r) = \{A'_1 = f_1(t, W_l, W_r), .., A'_n = f_n(t, W_l, W_r), P, w_l, w_r, Group\text{-}by(A_1, .., A_m), O\}$

  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 tuples 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 tuple 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 $A'_1, \ldots, A'_n$ 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 13 shows an example of Join operator. The operator is configured with a time based sliding window with size 3. The input streams S_1 has the schema *(In1, In2, TS_In)* and S_2 *(In3, In4, TS_In)* while the output stream *O* has the schema *(Out1, Out2, TS_Out).* When the first event is received from stream S_1, the operator does not produce any output because there is no match among the tuple received on the *left stream* and the window defined

on the *right stream* (actually it is still empty). Then, the operator inserts this tuple in the *left window*. The same happen to the second tuple received from S_1 with timestamp TS2. Instead, when the first tuple arrives from stream S_2 , with timestamp TS2 , it is matched with one of the tuples kept in the *left window* and the first output event is emitted. The same happens when the tuple with timestamp TS5 arrives from stream S_2. This time the tuple matches with 2 events kept on the left window (with timestamp TS3 and TS4) hence 2 events are emitted on the stream *O.*

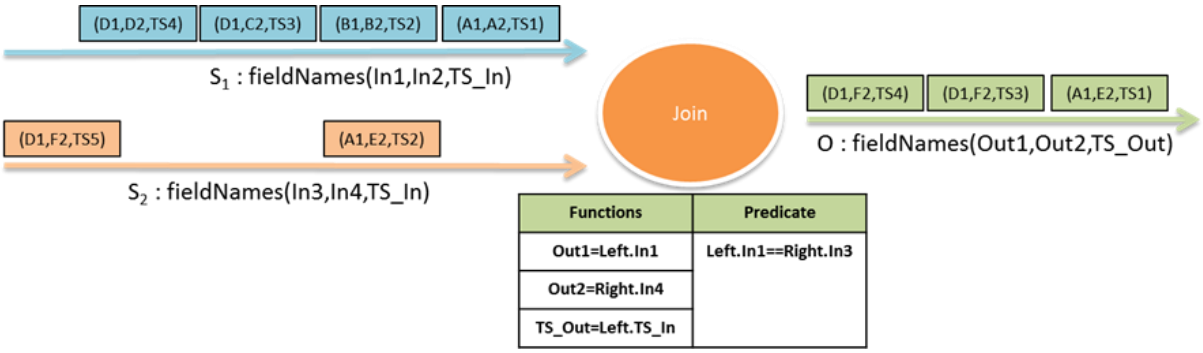

Figure 13 – Example of Join Operator with time based Sliding Window of size 3

3.3. CEP database operators

Database operators are able to perform operations on any CoherentPaaS data store using data received from continuous streams of tuples. These operators offer basic functionalities such as persisting and retrieving tuples to/from an external storage system. The cost of these operators in terms of latency of the operation is expected to be greater than the CEP query operator cost due to the response time of traditional DBMS that is known to be greater than the time needed for in memory processing operations. All the database operators must be configured at least with the driver needed to reach the remote instance of the CoherentPaaS data store and with the table name used for reading/writing tuples. The following operators will be available:

- *Insert*: This operator accepts one input stream and does not define any output stream. The table schema in the remote storage system and the schema of the tuples on the input stream must match. The operator uses a common SQL expression to insert the incoming tuples as new rows on the destination database table.

- *Update*: This operator accepts one input stream and does not define any output stream. The operator uses a common SQL expression parameterized with the fields of the incoming tuples to update rows on the destination database table.

- *Delete*: This operator accepts one input stream and defines one output stream. The table schema in the remote storage system and the schema of the tuples on the output stream must be matchable. The operator uses a common SQL expression parameterized with the fields of the incoming tuples to delete rows on the destination database table. Deleted rows are inserted in the output stream.

- *Select*: This operator accepts one input stream and defines one output stream. The result of the select query run against the remote table and the schema of the

tuples on the output stream must match. The operator uses a common SQL expression parameterized with the fields of the incoming tuples to run a select query on the destination database table. The rows belonging to the result set of the query are inserted in the output stream.

Database operators can be used against any data store providing a JDBC driver as it is the case with a couple of data stores in CoherentPaaS, Derby and MonetDB, but they can also access any of the other data stores integrated in CoherentPaaS by means of the common query language engine that it is also integrated with this operator.

3.4. CEP query compiler

The query compiler is the CEP component in charge of transforming a query written in a human-friendly language with no information about distribution, parallelization or deployment into an artefact runnable in the used SPE infrastructure. In particular, in the case of using Storm as SPE infrastructure the CEP query compiler converts a query written in the language presented in Section 4 into a Storm topology.
An en example, Figure 14 shows how the CEP Query Compiler would transform a query written in the SQL-like language into a CEP Query.

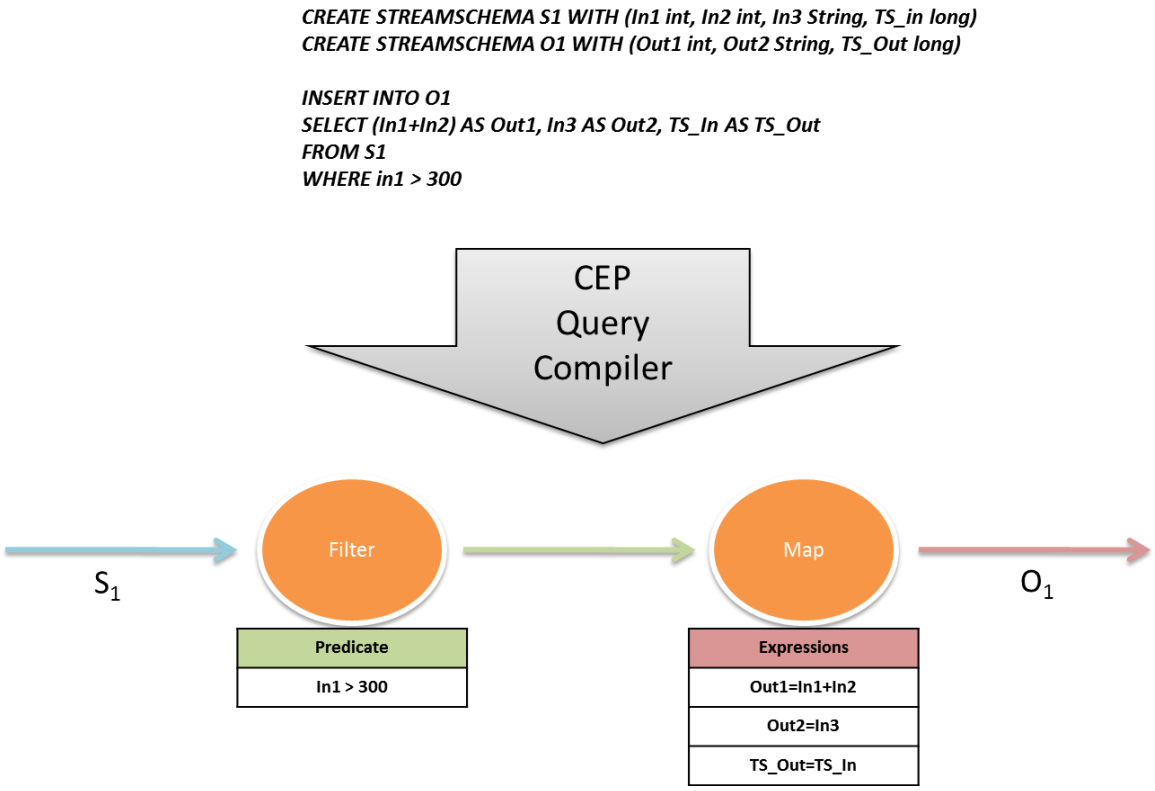

Figure 14 – An Example of transformation from a SQL-like query to the corresponding CEP Query

3.5. CEP query deployer

The task of the query deployer is to deploy the runnable query produced by the query compiler in the SPE infrastructure. To this end, the query deployer is configured with the necessary knowledge of the available resources in the infrastructure, and the desired

parallelization and distribution factor for the runnable query. Using Storm as SPE infrastructure, the query deployer sends the compiled topology enriched with the information, provided by either the user or configuration files, about the number of workers, executors and tasks to the Nimbus daemon running in the Master node. The Nimbus daemon takes care of deploying and start running the topology in the Storm Cluster.

4. CEP Query Language

The query language for CEP allows users to easily formulate those statements able to infer, aggregate and correlate information from one or more continuous streams of events. Our CEP query language is defined taking into account the traditional SQL language where the main differences are the replacement of tables with continuous streams and rows with events. Events are the basic unit of information for CEP queries and as for rows in a relational table all the events belonging to a specific stream must share the same schema. In order to declare the schema of a stream the CEP query language provides the statement *CREATE STREAMSCHEMA* which can be used as:

(1) *CREATE STREAMSCHEMA streamname WITH (attribute1 type1, attribute2 type2, … attributeN typeN)*

The supported attribute types are *int, long, short, byte, string, double, float* and *boolean*. In the same way, it is possible to create table schemas on remote data storages using:

(2) *CREATE TABLESCHEMA tablename WITH (attribute1 type1, attribute2 type2, … attributeN typeN)*

Creating a schema for the input streams of a query also define the structure of the events that a specific data source can send to this query.
The CEP query language being SQL-like supports the three SQL main clauses: *SELECT*, *FROM* and *WHERE*. The SELECT clause specifies the list of attributes to be taken from events. The FROM clause indicates the streams to be used in the query. Finally, the WHERE clause is used to define the predicate that must be satisfied in order to select the events. The comparison operators that can be used in predicates are **"=, < , > , >=, <=, !="**, furthermore the WHERE clause also supports logical combination via *AND* & *OR*.

(3) *SELECT attribute1, attribute2,…, attributeN*
FROM streamname
WHERE predicate

This query produces an unnamed stream where the attribute's names correspond with the names of the attributes taken from *streamname.* The name of the attributes can be set using the clause *AS*:

(4) *SELECT attribute1 AS name1, attribute2 AS name2,…, attributeN AS nameN*
FROM streamname
WHERE predicate

Furthermore, to declare the name of the output stream produced by the query, the *INSERT INTO* clause can be used:

(5) *INSERT INTO streamname_out*
SELECT attribute1 AS name1, attribute2 AS name2,…, attributeN AS nameN
*FROM streamname*_in
WHERE predicate

The token * is used to select the entire event's attribute set for example, it can be used to persist the events from a continuous stream into a data storage table:

(6) *INSERT INTO tablename*
*SELECT **
FROM streamname
WHERE predicate

Streams are potentially infinite sequence of tuples hence in order to perform aggregation or correlation among streams CEP queries need to define one or more windows. *Windows* define a finite portion of a stream over which the query is executed. Windows can be defined either over time or over events. Both types of windows have to be configured with the parameters size and advance.

(7) *CREATE TIMEWINDOW windowname WITH (SIZE size, ADVANCE adv)*

(8) *CREATE EVENTWINDOW windowname* WITH (SIZE *size*, ADVANCE *adv)*

Table 1 lists the aggregate operations which can be performed on one of the attributes defined by the event schema kept in the windows. To engage a window with SELECT queries, the CEP query language defines the *USING* clause.

Operation Name	Description
sum	Sum of values
max	Maximum values
min	Minimum values
avg	Average of values
count	Number of values
lastval	Most recent value
firstval	Most older value

Table 1 – Aggregate operations available on windows.

(9) *INSERT INTO streamname_out*
SELECT op1(*attribute1) AS name1, op2(attribute2) AS name2,…, opN(attributeN) AS nameN*
*FROM streamname*_in
WHERE predicate
USING windowname

The CEP query language allows running *JOIN* queries between 2 streams with time windows. Tuple windows are not supported because the join results would not be deterministic.

(10) *INSERT INTO streamname_out*
SELECT left_*attribute1 AS name1, right_attribute2 AS name2,…, right_attributeN AS nameN*
FROM leftstreamname, rightstreamname
WHERE predicate
USING leftwindowname, *rightwindowname*

It is worth noting that when running a join query the following syntax assumptions always hold:

- In the FROM clause, the first stream is identified as left stream and the second as right stream.
- In the USING clause, the first window keeps tuples belonging to the left stream and the second keeps tuples belonging to the right one.
- In the SELECT clause, the attribute's name must have either the suffix *"left_"* or *"right_"* to specify, in case of streams containing attributes with the same name, from which stream read the attribute.

Queries using windows can also define the *GROUP BY* clause to group the events in sub windows according to the values of the attributes in the clause.

(11) *INSERT INTO streamname_out*
SELECT op1(*attribute1) AS name1, op2(attribute2) AS name2,..., opN(attributeN) AS nameN*
*FROM streamname*_in
WHERE predicate
USING windowname
GROUP BY attribute1, attribute2,..., attributeN

It is worth noting that complex queries can be defined filling the FROM clause of a given query with the *streamname* defined in the INSERT clause of another one.

5. Materialization Support

As aforementioned in order to solve the impedance mismatch between the CEP and point-in-time queries one of the mechanisms that we propose is materialization operators that enable to see the output of CEP queries as relational tables.

To this end a new CEP operator is defined by extending the behaviour of the Insert operator.

- *MaterializeStream*: This operator accepts one input stream and does not define any output stream. The table schema in the remote storage system and the schema of the tuples on the input stream should match. The user has to configure how the operator inserts tuples in the table choosing between two write profiles: *append* and *circular*.

On the one hand, with the *append* profile the operator keeps adding new rows in the table as new tuples are received, on the other hand using the *circular* profile, a persistent circular buffer is used and any time the operator inserts a new row also deletes all the rows older than the size of the circular buffer. The size of the circular buffer is based on the number of rows currently stored in the table. To use this functionality the CEP query language provides the *PERSIST* clause. This clause can be configured either with the *append* token to use the append profile or with a natural number *n* to use the *circular* profile with a circular buffer of size *n*.

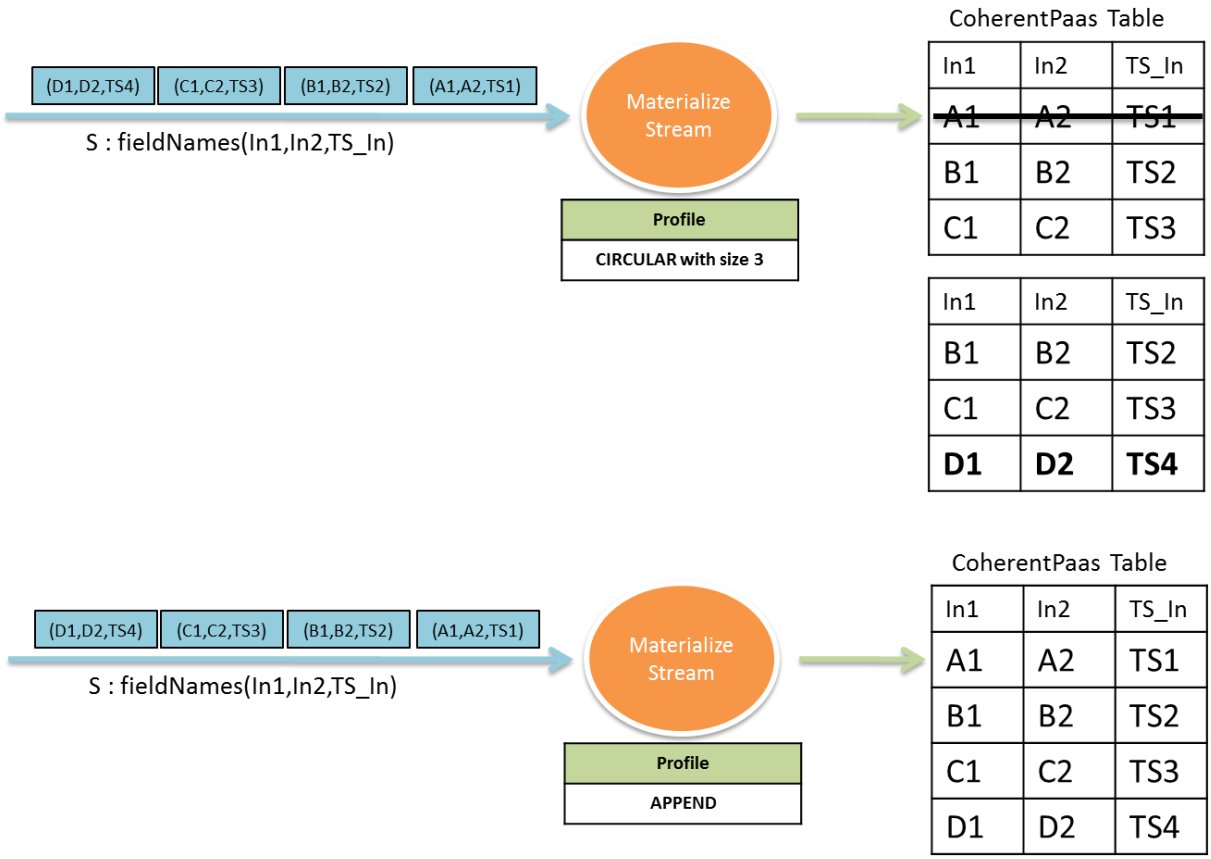

Figure 15 – Example of MaterializeStream Operator

(12) *INSERT INTO tablename*
*SELECT **
FROM streamname
WHERE predicate
PERSIST append | n

6. Integration with Holistic Transactions

CEP systems since are in-memory processing systems they do not have the notion of transactions. Additionally, tuples are handled fully independently which makes difficult to define the notion of a transaction.

Since in CoherentPaaS, the CEP system is being integrated with other data stores that are or will become transactional, we have to find a way to provide transactional semantics for the CEP queries. We have split the problem into two different concerns, read consistency and write atomicity.

6.1. Transactional Read Consistency for CEP

Transactional semantics in CoherentPaaS relies on snapshot isolation. This means that a transaction should observe a consistent snapshot as of start time. In the case of the CEP system we can provide a similar kind of consistency for the database reads perform by databases operators. Unfortunately, there is no notion of transaction, so how can this read consistency be provided and based on what?

Since the unit of processing in the CEP system are tuples, we have studied what can be done so this processing of an individual tuple is read consistent. One of the challenges is that each time a tuple traverses a CEP operator can be transformed or contribute to a state that will be later propagated by another tuple. This is where read consistency becomes important.

In Storm, tuples are injected in the CEP system via spouts. We have decided to integrate transactional management with each spout. In this way, we can insert information in tuples injected in the CEP that can help us to provide consistent reads. Basically, we have added a new function accessible to applications to get the current snapshot. With this, the spout adds to each tuple the current snapshot. Whenever a tuple is processed by a stateless operator and generates another tuple, the snapshot value is propagated. Stateful operators are more complex since the tuples that they output typically reflect the contribution of several input tuples. In this case, consistency lies in that the resulting tuple observe in any database read a state that is consistent with all tuples that contributed to the tuple. Since tuples can be aggregated and correlated together, we also need to have a way to propagate a consistent snapshot value. In the case of joins, a tuple resulting from the join of two tuples will carry the highest snapshot from the two. In the case of aggregates, a tuple resulting from the aggregation of the tuples in a sliding window will carry the highest snapshot value among all the tuples. Figure 16 shows 2 examples of snapshot value propagation, one with a stateless operator (such as Map) and one with a stateful operator (such as Aggregate). For simplicity, we only show the snapshot value filed of the tuple hiding the rest of fields. With the *Map* operator each output tuple has the same snapshot value as the input tuple that triggered the output. The *Aggregate* operator in Figure 16 is defined with a tuple based sliding window with size 3 and advance 1. Any time the window is full the operator emit a tuple using a snapshot value the highest value available among the tuples in the window.

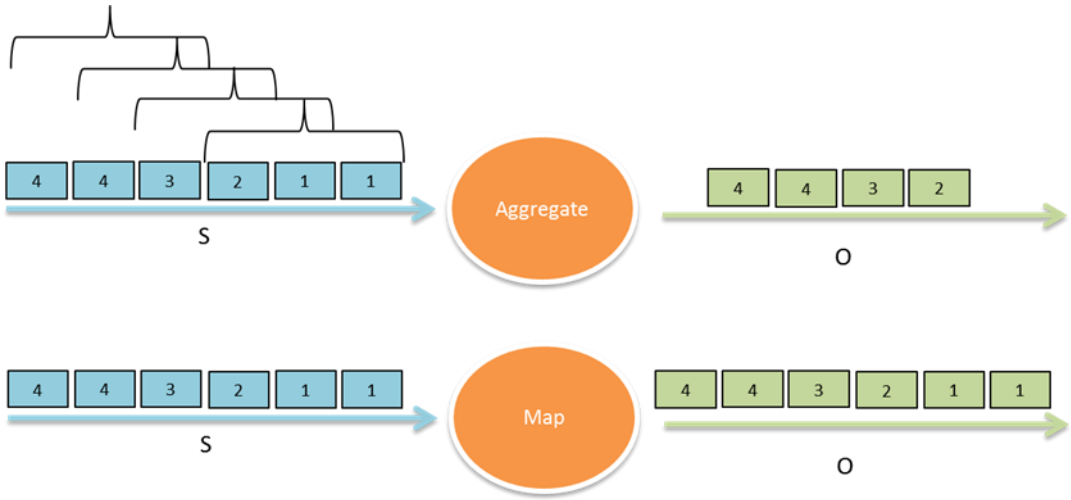

Figure 16 – Example of snapshot value propagation

With the proposed mechanism, we can provide a monotonically increasing snapshot consistency to the CEP system so when a tuple is correlated with a data store state it observes a consistent state across the whole CEP query.

6.2. Transactional Write Consistency for CEP

In the case of updates, as it happens with regular transactional databases, write consistency can only be specified by the user by bracketing transactions. The alternative that we have chosen for this is to allow spouts to bracket transactions. There are two modes, the auto-commit mode and the bracketing mode. In auto-commit mode any update performed by a database operator on behalf of a tuple will be a single transaction. In the bracketing mode, a spout can decide to process a batch of tuples (in the storm sense) as a transaction.

The batch of tuples will be associated with a start timestamp as a regular transaction in any CoherentPaaS data store. The propagation of this information follows the same strategy as explained in the previous section for providing read consistency in the absence of transaction bracketing. However, these batches of tuples that are initially batched together can follow different paths within a CEP query. Since a global transaction of them would require too much coordination, we have decided that all tuples belonging to the same batch that are processed by a given database operator they are dealt with by that database operator as a single transaction. The database operator will open a transaction, then process the batch performing as many updates as tuples, and finally commit the transaction.

6.3. Integrating Transactional Management

Since integrating local transactional managers with spouts might have an impact when there is a failure in provoking a recovery process that would be unneeded we have decided to integrate the transactional processing through a JDBC driver of our SQL data store. In this way, failures of a spout node do not have a negative impact in the transactional and persistent data management. For this purpose, the JDBC driver API will be extended to enable to get a start timestamp and force it at other JDBC drivers.

Updates require the integration of database operators with a JDBC driver as it happens with the spouts. Using a JDBC driver instead of a local transactional manager has the same motivation as before.

7. References

[1]. Storm Powered By. Web Page: https://github.com/nathanmarz/storm/wiki/powered-by Last visited (19/03/2014)
[2]. Storm Project – GitHub. Web Page: https://github.com/nathanmarz/storm Last visited (19/03/2014)
[3]. Eclipse Public License 1.0. Web Page: http://opensource.org/licenses/eclipse-1.0.php Last visited (19/03/2014)
[4]. Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. . In Proceedings of the 2010 USENIX conference on USENIX annual technical conference (USENIXATC'10). USENIX Association, Berkeley, CA, USA, 11-11
[5]. Storm Tutorial. Web Page: https://github.com/nathanmarz/storm/wiki/Tutorial Last visited (19/03/2014)
[6]. Understanding the parallelism of a Storm topology. Web Page https://github.com/nathanmarz/storm/wiki/understanding-the-parallelism-of-a-storm-topology Last visited (19/03/2014)
[7]. Storm Concepts. Web Page https://github.com/nathanmarz/storm/wiki/concepts Last visited (19/03/2014)
[8]. LMAX Disruptor. Web Page http://lmax-exchange.github.io/disruptor/ Last visited (19/03/2014)
[9]. M. Noll. Web Page http://www.michael-noll.com/blog/2013/06/21/understandingstorm-internal-message-buffers Last visited (19/03/2014)
[10]. Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring streams - a new class of data management applications. In 28th International Conference on Very Large Data Bases (VLDB'02), pages 215-226, 2002
[11]. Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. VLDB J., 15(2):121-142, 2006.
[12]. ZeroMQ Project. Web Page http://zeromq.org/ Last visited (19/03/2014)
[13]. Netty Project. Web Page http://netty.io/ Last visited (19/03/2014)