

New Event-Processing Design Patterns using CEP

Alexandre de Castro Alves

Oracle Corporation, Redwood Shores, CA, USA,
alex.alves@oracle.com

Abstract. Complex Event Processing (CEP) is a powerful technology for supporting advanced event-processing scenarios at a higher level of abstraction. Because of its expressiveness, CEP allows prompt creation and classification of new event-processing design patterns, some of which have been implemented in the past in a non-reusable form. This paper documents a set of new patterns for event processing, describing their problem domain and providing a solution template implemented using CEP, which is both succinct and highly re-usable.

Key words: CEP, stream management, stream processing, SQL, CQL, design pattern

1 Introduction

CEP [1] is an emerging technology that allows the implementation of advanced event-processing scenarios. One of the main advantages of CEP is the usage of a domain-specific declarative language to perform the event processing, which is commonly referenced to as the event processing language (EPL) [6].

Because of the expressiveness of EPLs, scenarios that were in the past laboriously implemented over large extend of time can be now supported with a few lines of code, thus allowing us to largely re-use the solutions as design patterns [2] for event-processing. This easiness of implementation also allow us to rapidly create and document new patterns, hence contributing to the library of techniques that can be used to solve event-processing scenarios. Before describing the event-processing design patterns we have worked on we provide some basic background on the building blocks of CEP. This is needed so that the reader can better understand the solution template provided for each pattern.

In this paper we will describe four event-processing design patterns. We will start by describing event filtering, which is arguably the most common and simple event-processing pattern. We choose this pattern as a way to familiarize the reader with the building blocks and concepts used by CEP. Following, we will describe three patterns that, although some users may know them in one way or the other, have not been documented using CEP in a manner that they can be re-used as design patterns for solving event-processing problems. These are the new event detection pattern, the old event detection pattern, and the missing event detection pattern.

The patterns are described by first stating their problem domain, that is, by describing the problem that the event-processing application is trying to solve. Next, we provide a semi-formal structure of the problem. The intent of the structure is to help users understand the family of problems that can be solved by applying the pattern. Finally, a solution template using our EPL language of choice, called CQL, is provided. We conclude the paper with observations of the results we have seen so far by systematically employing these patterns and describing on-going work on the matter.

2 Preliminaries

The work in this paper is based upon the Stanford STREAMS project [3] and in particular uses the Continuous Query Language (CQL) [7], an extension to SQL, as the event processing language of choice. Granted that any work on design patterns should be demonstrated using more than a single programming language, otherwise there is a chance that the pattern is a reflection of a programming idiom rather than the solution of a real use-case. Nevertheless, the author feels that CQL is generic enough that the problem is avoided. In particular, the CQL extensions to SQL are presented in isolation and details, facilitating porting to other languages. Furthermore, future work on this subject should include examples on other language, such as a logic programming language.

Before we begin, one high level clarification is needed. In this paper we opt for the term CEP, even though there exists other terms in the industry and academia, such as stream processing and business event processing, that are also attributed to event-processing technologies. As this paper is driven from use-cases, the author hopes that details around the implementation technology is immaterial and that the solution is generic enough to be applicable in different event-processing technologies.

3 Building Blocks

The author strongly encourages the reader to go through the cited references; nevertheless in this section we revise a few important concepts that are extensively used through out the rest of this paper.

An event is a tuple of event properties defined by a schema. The event schema is also commonly referenced as the event type. For example, a stock event type may define three event properties: symbol of type string, lastBid of type float, and lastAsk of type float. An example of an event of the stock event type is:

```
{symbol: AAA, lastBid: 10.0, lastAsk: 12.0}
```

Streams and relations are two types of collections of events. Streams are time-ordered sequence of events. Streams support only the append operation, that is, one can append an event to the end of the stream, but events cannot be deleted

from the stream. Streams are unbounded by nature, that is, have (conceptually) no fixed limiting size. An example of a stream of stock events is:

```
{1t, AAA, 10.0, 12.0}, {2t, BBB, 11.0, 12.5},
{3t, AAA, 10.5, 11.5}, ...}
```

Relations are also collections of events, or more precisely, a bag of events. However, relations (also known as instantaneous relations) are bounded and are always tied to some instantaneous time t . Relations support insert, delete, and update operations. Lets consider relation R that also contains stock events.

```
At time  $t = 0$ ,  $R = \{\{AAA, 10.0, 12.0\}, \{BBB, 11.0, 12.5\}\}$ 
```

```
At time  $t = 1$ ,  $R = \{\{BBB, 11.0, 12.5\}, \{CCC, 5.0, 5.5\}\}$ 
```

That is, initially relation R contained stock price for symbols AAA and BBB. Next, stock AAA is deleted and stock CCC is inserted.

Event processing agents (EPA) are entities whose input and output are streams and relations and whose role is to process events by executing rules (or queries) specified using some EPL. The execution of a rule can be potentially divided into the execution of separate operators, which may convert streams into relations and vice-versa. One of the goals of CEP is to leverage operators from existing technologies, such as from database systems.

4 Event-Processing Design Patterns

4.1 Event Filtering

Problem Description The simplest and most common design pattern for event processing is the idea of event filtering. In this case, the problem an application is trying to solve is to discard events that do not meet some criteria.

Some examples of applications performing event filtering are:

- A financial application receiving stock events from the market exchange is looking for a set of stock symbols (e.g. AAA, BBB) and discarding the rest (e.g. CCC, DDD).
- A medical application receiving health status from medical devices is interested when some threshold is crossed (e.g. high blood pressure) and discarding the statuses otherwise.

Problem Structure Before describing the solution, we must model the problem in terms of event processing concepts [2]. Event filtering consist of an event-processing agent (EPA) whose input is a stream $S1$ and output is a stream $S2$, where $S2$ contains a subset of events of $S1$:

$$S2 \subseteq S1 \tag{1}$$

Lets consider the scenario of a stock filtering application. This application is receiving events from a stream called stockstream, which is defined by the StockTick event type:

- symbol: string
- lastBid: float
- lastAsk: float

The stock filtering application is looking for events whose symbol is equal to AAA. The following table provides an example of event filtering for three input events coming from a stream where time is progressing in 1t units. The first two events meet the criteria and are not filtered out. The last event does not meet the criteria and is discarded.

Table 1. Example of input and output events for event filtering

Time	Input Stream	Output Stream
1	{AAA, 10.0, 10.5}	{AAA, 10.0, 10.5}
2	{AAA, 10.0, 10.5}	{AAA, 10.0, 10.5}
3	{BBB, 10.0, 10.5}	-

Solution Template Using the event processing language CQL, a solution to the problem consists of the following query:

```
SELECT * FROM stockstream [NOW] WHERE symbol = AAA
```

To understand this query, let's break it into a set of steps or operations. Firstly, we specify the event source, this is done by means of a FROM clause. In this case, the source is a stream named stockstream. Secondly, we must convert the stream into a relation. This is needed because the filter operator, which is used afterwards, works off a relation and not a stream. The reason why the filter operator uses relations instead of streams is that the filter operator is leveraged from the well-known relational model of database systems. To convert the stream stockstream into a relation we make use of the window operator NOW. This stream-to-relation operator outputs a relation containing the event as of the current time from its input. With a relation, we are now able to apply the filter operator described by the predicate symbol = AAA in the WHERE clause.

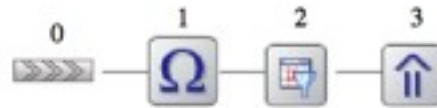


Fig. 1. Query plan for event filtering query

Finally, the output of the filter operator is projected out as the overall output of the query. In summary, a total of four operators are applied, as depicted in Figure 1.

4.2 New Event Detection

Problem Description In this case, an application is interested on detecting if an event is new in some defined context. Generally the context is specified by some predicate. Examples of applications performing new event detection are:

- A financial application receiving stock events from the market exchange is interested on detecting if a particular stock has its price changed, but otherwise if the price has not changed then the application discards the event. It is important to observe that the market exchange, as it is the generally the case of any stream, outputs events constantly even when these events are not changing. This is so because consumers of the events in the stream may join and leave at any time.
- A medical application receiving health status from medical devices is interested on detecting if some health metric has changed (e.g. blood pressure has gone up or down). The context in this case maybe a particular patient, that is, the medical application is only interested on health status changes for a patient.

Problem Structure The problem of detecting new events can be modeled as an EPA whose input is a stream S1 and output is a stream S2, where the event e is only present in S2 if the event e is also present in S1 at time t , but is not present in S1 at time $t - 1$; or more precisely, at time $t - \text{delta}$, where $t - \text{delta}$ is the time of the last event at S1 that shares the same context as event e .

$$S2 = \{e \in S1 \mid e \in S1(t) \wedge e \notin S1(t - \text{delta})\} \quad (2)$$

Lets continue with the stock application we used in the previous section. The following table illustrates the input and output to the stock application when it detects if the last bid or the last asking price changed for a particular stock symbol:

Table 2. Example of input and output events for new event detection

Time	Input Stream	Output Stream
1	{AAA, 10.0, 10.5}	{AAA, 10.0, 10.5}
2	{AAA, 10.0, 10.5}	-
3	{BBB, 10.0, 10.5}	{BBB, 10.0, 10.5}
4	{BBB, 10.0, 10.5}	-
5	{AAA, 10.0, 10.5}	-
6	{BBB, 10.1, 10.6}	{BBB, 10.1, 10.6}

The event at time 1, being the first one, is outputted normally. However, the same event at time 2 is not outputted, as it is similar to the previous event. This pattern repeats itself for events 3 and 4. Event at time 5 is not outputted, however it is not outputted because it is similar to the event at time 2, which also shares the same stock symbol of AAA. If we were to compare event 5 to event 4, we would have outputted event 5, as it is different than event 4. Finally, event 6 is outputted because the prices differ from event 4.

Solution Template The new event detection pattern for the stock application scenario can be solved with the following CQL query: query:

```
ISTREAM(SELECT * FROM stockstream [PARTITION BY symbol ROWS 1])
```

This query receives events from the stream `stockstream`, and converts the stream into a relation by using the window operator `[PARTITION BY symbol ROWS 1]`. The clause `PARTITION BY symbol` partitions the stream into separate relations where the event in the relation has the same value for the event property `symbol`. The clause `ROWS 1` keeps the last event per partition. Finally, the `ISTREAM` (i.e. `INSERT STREAM`) operator is a relation-to-stream operator that only outputs an event if the event exists in the relation at time t and does not exist at time $t - 1$, that is, the event is new in the relation.

4.3 Old Event Detection

Problem Description This is the opposite case of the new event detection pattern, where the application is interested on detecting if an event is no longer valid, that is, the event has become stale. Examples of application performing old event detection are:

- A financial application receiving stock events from the market exchange is interested on detecting if the price for a particular stock is no longer valid.
- A medical application receiving health status from medical devices is interested on detecting if some condition (e.g. high blood pressure) is no longer holding.

Problem Structure The problem of detecting old events can be modeled as an EPA whose input is a stream $S1$ and output is a stream $S2$, where the event e is only present in $S2$ if the event e is also present in $S1$ at time $t - 1$, or more precisely, at time $t - \text{delta}$, where $t - \text{delta}$ is the time of the last event at $S1$ that shares the same context as event e , but is not present in $S1$ at time t .

$$S2 = \{e \in S1 \mid \neg S1(t) \wedge e \in S1(t - \text{delta})\} \quad (3)$$

The following table illustrates the input and output to our stock application as it detects if the last bid or the last asking price is no longer valid for a particular stock symbol:

In this case, no event is outputted until the application receives event 5, which replaces event 2. Note that at this time, event 2 and not event 5 is outputted. Likewise, when the application receives event 6, it outputs event 4.

Table 3. Example of input and output events for old event detection

Time	Input Stream	Output Stream
1	{AAA, 10.0, 10.5}	-
2	{AAA, 10.0, 10.5}	-
3	{BBB, 10.0, 10.5}	-
4	{BBB, 10.0, 10.5}	-
5	{AAA, 10.0, 10.6}	{AAA, 10.0, 10.5}
6	{BBB, 10.1, 10.6}	{BBB, 10.1, 10.6}

Solution Template Very similarly to the new event detection pattern, the old event detection pattern for the stock application scenario can be solved with the following CQL query:

```
DSTREAM(SELECT * FROM stockstream [PARTITION BY symbol ROWS 1])
```

The main difference is the usage of the relation-to-stream operator `DSTREAM` (i.e. delete stream). `DSTREAM` only outputs an event if the event exists in the relation at time $t - 1$ and does not exist at time t , that is, the event is no longer in the latest relation.

4.4 Missing Event Detection

Problem Description In the missing event detection pattern, an application is interested on being alerted if some expected event is not received within some amount of time.

Examples of this pattern being employed are:

- A retail application handling order requests needs to verify if shipment of the goods is executed within some time of the request of the order.
- A service monitoring application needs to verify if every service request issued by a client receives a response within some maximum amount of time.

Problem Structure The problem of detecting missing event can be modeled as an EPA whose input is a stream $S1$, and output is a stream $S2$, where the event $e1$ is only present in $S2$ if the event $e2$ is not present in $S1$ after t time.

$$S2 = \{e \in S1 \mid e2 \notin S1(t) \Rightarrow t_s < t < t_e\} \quad (4)$$

For this pattern, lets consider a stream `salesstream`, whose events are of the event type `SalesRequest`:

- `requestId`: string
- `type`: order, shipment, or delayed

A sales request can be an order request or a shipment request. An order request for a `requestId` must be followed by a shipment request within 10 minutes; otherwise a delay event must be outputted by the application. In the context of

Table 4. Example of input and output events for missing event detection pattern

Time	Input Stream	Output Stream
1m	{1, order}	-
5m	{2, order}	-
10m3	{1, shipment}	-
15m	{3, order}	-
15m+t	-	{2, delayed}
20m	{3, shipment}	-

this scenario, the table 4 illustrates the missing event detection pattern for a set of input samples:

The first order request is received at time 1 minute, and its shipment happens at time 10 minutes, that is, within our service agreement of 10 minutes. The second order is received at time 5 minutes, but by the time 15 minutes + 1t, where t is the smallest accountable time increment for the application, no shipment has been processed yet, hence a delayed event is outputted.

Solution Template The missing event detection scenario can be solved by the following query:

```
SELECT request.requestId, "DELAYED" as type,
FROM salestream MATCH_RECOGNIZE (
    PARTITION BY requestId
    MEASURES Order.requestId AS requestId
    PATTERN (Order NotTheShipment*) DURATION 10 MINUTES
    DEFINE
        Order AS (type = ORDER),
        NotTheShipment AS ((NOT (type = SHIPMENT)))
) AS request
```

As usual, the query receives events from a stream, in this case the salesstream. However, differently than all previous cases, we don't need to convert the stream into a relation because MATCH RECOGNIZE is a stream-to-stream operator.

Let's investigate MATCH RECOGNIZE, which is a very useful tool for performing complex pattern matching. Firstly, we define the objects we are interested in matching. This is done with the DEFINE clause, where we specify an order object and a not-a-shipment object. The not-a-shipment object is defined as being any event that is not of type SHIPMENT. Secondly, we define the actual pattern to match with respect to those objects defined in the previous step. In this case, the pattern is defined as an order object followed by zero or more numbers of not-a-shipment objects. The pattern is open-ended; hence we specify duration of 10 minutes to close the pattern. Finally, if the pattern is matched, the MEASURE clause defines the event being outputted to the final projection (i.e. SELECT clause) operator.

4.5 Discussions and Related Work

In [12], Tsimelzon documents ten design patterns for CEP, including a filtering design pattern. In his work, there is a mixture of concerns around the selected design patterns. The filtering design pattern, similar to the patterns aforementioned in this paper, is centered on the semantic of the events. However other scenarios, such as database lookup and dynamic queries, are centered on the processing agent, that is, they can be seen as engine capabilities. Furthermore, the paper does an excellent job of explaining the scenarios and applicability, but does not provide a framework for describing the problem, nor details the underpinnings of the solution. In this paper, we have tried to identify the abstractions for the problem domain, such as events, streams, and relations; as well as provide implementation tools for the solution, for example, by explaining relation-to-stream, stream-to-relation, and stream-to-stream operators.

In Paschke [11] and Ammon, et al [12], a pattern language is presented for CEP. A pattern language is a template for the definition of instances of concrete design patterns. Paschke also documents several categories for the CEP design patterns, including the categories workflow patterns and coordination patterns, which seem the most applicable to the patterns provided in this paper. The pattern language is a useful one, and we should attempt to adhere to its template in future works.

4.6 Conclusion and Future Work

Developers are used to working with events and tables, however the concept of streams and (instantaneous) relations and the high expressiveness of EPLs may overwhelm new adopters of CEP. The documentation of event-processing design patterns implemented using CEP is able to lower the entry barrier and bring users up-to-speed with CEP. Furthermore, it has allowed us to identify new event-processing design patterns, some which have being known in the past, but were never documented as a design pattern thus facilitating their re-use. In the future, we hope to continue to increase the library of event-processing design patterns, by including other patterns, such as event batching, and the W pattern detection, and also documenting commonly known patterns, such as event enrichment, event aggregation, and event correlation.

Acknowledgements The author would like to thank the members of the Oracle CEP group Anand Srinivasan, Andy Piper, Eric Shiao, Parul Jain, Manju James, Mohit Thattle, Seth White, Shailendra Mishra and others for their invaluable work in implementing the software that supports these CEP design patterns.

References

1. Luckham, D. The Power of Events, An Introduction to Complex Event Processing in Distributed Enterprise Systems (2002).

2. Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software.
3. Arasu, A. and Babcock, B. and Babu, S. and Cieslewicz, J. and Datar, M. and Ito, K. and Motwani, R and Srivastava, U. and Widom, J. (2004) STREAM: The Stanford Data Stream Management System.
4. Schulte, R., Bradely, A.: A Gartner Reference Architecture for Event Processing Networks, ID G00162454, 2009.
5. Etzion, O., EDA Conceptual Model (work in progress) IBM, 3rd EPTS Event Processing Symposium, Orlando, September 2007
6. Luckham, D. and Schulte, R. Event Processing Glossary Version 1.1 (2008).
7. A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical report, Stanford University, Oct. 2003. <http://dbpubs.stanford.edu/pub/2003-67>.
8. U. Srivastava and J. Widom. Flexible time management in data stream systems. In Proc. of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2004.
9. White, S., Alves, A., Rorke, D. WebLogic event server: a lightweight, modular application server for event processing. Proceedings of the second international conference on Distributed event-based systems (2008).
10. Jain, N., Mishra, S., Srinivasan, A., Gehrke, J., Balakrishnan, H., Cetintemel, U., Cherniack, M., Tibbetts, R., Zdonik, S. Towards a streaming SQL standard. Proceedings of the VLDB Endowment (2008).
11. Paschke, A. Design Patterns for Complex Event Processing. Proceedings from Distributed Event-Based Systems Symposium, 2008.
12. Tsimelzon, M., Coral8 Design Patterns. <http://www.coral8.com/blogs/blog-entry/cep-design-patterns>, 2006.
13. Ammon, R, Silberbauer, C., Wolff, C. Domain Specific Reference Models for Event Patterns (2007)
14. Paschke, A. EuroPLop'08 Focus Group on CEP Patterns and Rule Patterns, http://www.biotec.tu-dresden.de/~adrianp/europlop08_cep/EuroPLOP_CEP_Focus.pdf
15. Anonymous, Pattern matching in sequence of rows (2007), <http://asktom.oracle.com/tkyte/row-pattern-recogniton-11-public.pdf>