



# Complex Event Processing: Ten Design Patterns

---

---

Copyright © 2006 Coral8, Inc. All rights reserved worldwide.

**Worldwide Headquarters:**

Coral8, Inc.  
82 Pioneer Way, Suite 106  
Mountain View, CA 94041  
*Phone:* 650-210-3810  
*Fax:* 650-210-3811

Visit us on the web at [www.coral8.com](http://www.coral8.com) or e-mail us at [info@coral8.com](mailto:info@coral8.com)

---

# Table of Contents

INTRODUCTION.....	III
1 FILTERING.....	1
2 IN-MEMORY CACHING .....	2
3 AGGREGATION OVER WINDOWS .....	4
4 DATABASE LOOKUPS .....	6
5 DATABASE WRITES.....	8
6 CORRELATION (JOINS).....	9
7 EVENT PATTERN MATCHING .....	11
8 STATE MACHINES .....	13
9 HIERARCHICAL EVENTS .....	15
10 DYNAMIC QUERIES .....	17
CONCLUSION.....	19

# Introduction

Complex Event Processing (CEP) engines have received a lot of attention lately. Understanding CEP on a high level is easy: a CEP engine is a platform for building and running applications, used to process and analyze large numbers of real-time events. These applications come up in a number of domains: algorithmic trading and execution, risk management, compliance monitoring, supply chain management, click-stream analysis, network intrusion detection, business process monitoring, logistics, power grid monitoring, infrastructure monitoring, military and intelligence, and others.

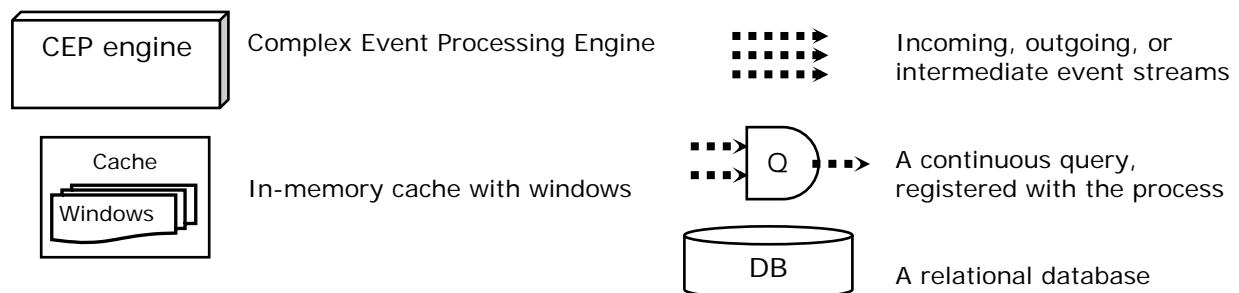
CEP engines are so new, however, that many application developers are struggling to understand the new capabilities. There is a certain disconnect between the producers and consumers of CEP technology. While the potential for the implementation of CEP engines seems perfectly obvious to those who build and market them, the most frequent question from potential users is: "What can I do with a CEP engine?" This paper is intended to provide the answers to this question.

This document describes ten fundamental CEP design patterns<sup>1</sup> that appear repeatedly in CEP applications, listed below in order of complexity, from the simplest to the most sophisticated. These basic patterns may be thought of as building blocks that can be combined to create complete applications. They are:

1. Filtering
2. In-memory caching
3. Aggregation over windows
4. Database lookups
5. Database Writes
6. Correlation (Joins)
7. Event pattern matching
8. State machines
9. Hierarchical Events
10. Dynamic Queries

For each design pattern, we provide:

- The name of the design pattern, and a brief description.
- Three sample areas of applicability. (Most design patterns can be used in many more than three areas, however).
- A simple Continuous Computation Language (CCL) example. CCL is an SQL-based language for creating *continuous queries*, that is queries registered with a CEP engine, that subscribe to data from data streams, and continuously publish output when the query criteria are met. For more information about CCL, please see <http://www.coral8.com/developers/index.html>.
- A conceptual diagram, using the following symbols:



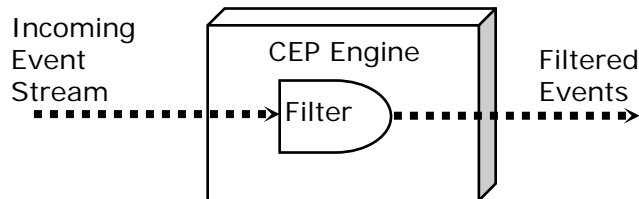
<sup>1</sup> The term *design patterns* has several meanings in software development. In this document, it refers to the fundamental reusable building blocks that form the foundation of CEP applications.

# 1 Filtering

---

## Filtering Events Based on Event Attributes

We begin our exploration of CEP design patterns with basic filtering. While filtering is easy to implement with a number of non-CEP products, or with custom-built applications, this example will help us set up a framework for describing other, more complex patterns.



The above diagram depicts a simple filter query. . The query subscribes to one stream, evaluates a specified logical condition based on event attributes, and, if the condition is true, publishes the event to the destination stream. For example, an application monitoring a stream of purchase orders may filter out all orders where the condition is **Priority != 'High' and Amount < 100000**.

This example presents the simplest kind of filter, where events are evaluated one by one, and where the query condition only involves the attributes of one event. It is also possible to construct many other more complex filters, for example, filters that compare events to other events in the same stream, or in another stream, or compare events to a computed metric. For instance, a filter might capture orders where the purchase amount is larger than the previous purchase amount, or purchase amounts that are larger than the average for the previous day. Such relatively more complex queries are discussed later in this document.

## Sample Areas of Applicability

Filtering is ubiquitous in CEP applications. Here are some examples:

- Trading: a filter may be used to filter out all trades where the volume is too small, or all trades that do not refer to particular stock symbols.
- Click-stream analysis: a filter may be used to capture the trades that originate from a certain set of IP addresses.
- Sensor network: a filter may be used to capture sensor readings where values fall outside of the normal range.

## CCL Examples

Filter queries in CCL look much like SQL queries. Unlike SQL queries, however, CCL queries execute continuously. The query shown below subscribes to StreamAllPOs, continuously evaluates the condition for all events, and sends those events that match the condition to StreamImportantPOs.

```
INSERT INTO StreamImportantPOs
SELECT *
FROM StreamAllPOs
WHERE PRIORITY = 'High' OR Amount >= 100000;
```

## 2 In-memory Caching

---

### Caching and Accessing Streaming and Database Data in Memory

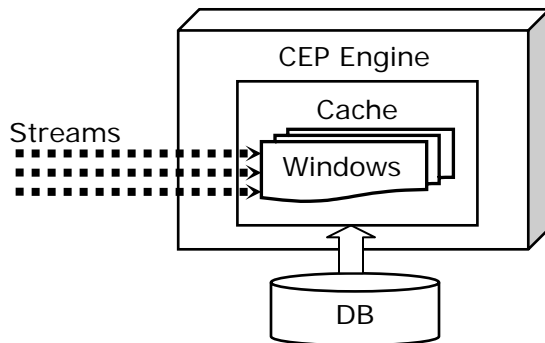
This is the first of the design patterns considered in this document, where multiple events are kept in memory. In-memory data caching is the foundation of most CEP design patterns. The cache typically stores two kinds of data:

- **Recent events from one or more streams**

Recent events are typically stored in *windows*. A window is an object, similar to an in-memory database table. However, a window can manage its state automatically, by keeping and evicting certain events according to its policy. For example a window policy might specify: KEEP 1000 ROWS PER Id. This window maintains 1000 rows for each ID value, and expires old rows, as necessary.

- **Data from one or more database tables**

Just as streaming events can be cached in memory, it often makes sense to cache data from a relational database, so that different kinds of operations may be performed on this data more efficiently. This cache is typically managed according to the Least Recently Used (LRU) algorithm, or by explicit invalidation.



Note that, although we are describing an in-memory cache here, many applications require this cache to be persistent. This means that, if a machine that hosts the CEP engine fails, the data kept in windows is not lost. This functionality is even more important when the window holds not just the last few seconds', but minutes', hours', days', and even weeks' worth of events.

### Sample Areas of Applicability

In-memory caching is used in every non-trivial CEP application. Here are just a few examples:

- **Trading:** in a trading application, the cache may hold the values of recent trades, recent orders, or recent news events, coupled with the relevant historical and reference information.
- **Click-stream analysis:** a typical application may hold the recent clicks and searches performed by the users, coupled with the relevant historical and reference information.
- **Network security:** A typical application may hold recent events from firewalls, intrusion detection systems, and other devices, coupled with the relevant historical and reference information.

## CCL Examples

In this example, we will focus on the first form of caching, namely, caching events from a data stream. The Database Lookup design pattern, presented later in this document, discusses the caching of database data.

### Windows

-- A window to keep data for ten minutes, populated by stream StreamClicks

```
CREATE WINDOW RecentHistory(IPAddress LONG, URLvisite STRING, FileSize LONG)
KEEP 10 MINUTES;
```

```
INSERT INTO RecentHistory
SELECT *
FROM StreamClicks;
```

-- A window to keep the last value for each IPAddress:

```
CREATE WINDOW LastValues(IPAddress LONG, URLvisite STRING, FileSize LONG)
KEEP LAST PER IPAddress;
```

-- A window that keeps the 1000 largest downloads. Each download is kept for one hour, or until a larger download displaces it from the window:

```
CREATE WINDOW LargestDownloads(IPAddress LONG, URLvisite STRING, FileSize LONG)
KEEP LARGEST BY FileSize KEEP 1 HOUR;
```

### Looking up Data in a Window

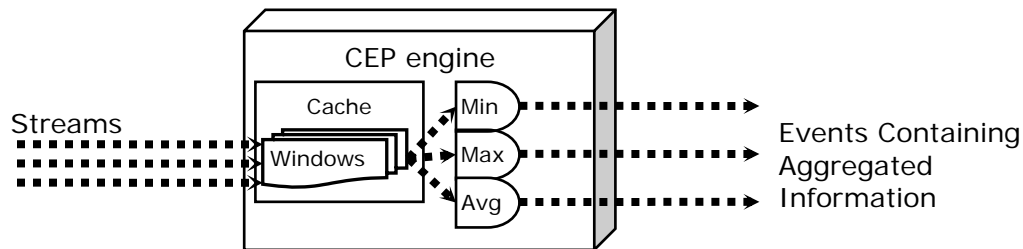
-- Use the RecentHistory window defined above to check how many other records with  
-- the same IP address have been seen recently

```
INSERT INTO StreamNewIP
SELECT S.*, Count(H)
FROM StreamClicks AS S,
     RecentHistory as H
WHERE S.IP = H.IP
```

## 3 Aggregation over Windows

### Computing Statistical Metrics over Various Kinds of Moving Windows

Unlike the previous design pattern, this pattern does not merely keep events in memory, but uses the stored values to compute various statistics. A typical example here would involve computing a running average over a *sliding window*. (As we have seen in the previous example, a window is an object that holds a set of events in memory):



This design pattern comes in quite a few flavors, differing along the following dimensions:

- **The kinds of aggregators computed**  
These include running averages, sums, counts, minimum, maximum, standard deviation, user-defined aggregators, and so on.
- **The kinds of windows used**  
These include time-based and count-based windows, sliding and jumping (tumbling) windows, windows that keep the specified number of largest or smallest elements, and so on.
- **Output frequency: continuous vs. periodic**  
In the case of continuous output (also called “tick-by-tick” output) each incoming event updates the calculated expression, and an output event is produced. With periodic output, the calculated expression is updated continuously, but is published only periodically, for example, every ten seconds. Note that, in both cases, the expression is computed *incrementally*, that is, the entire window is *not* rescanned on each incoming event.

### Sample Areas of Applicability

Windows-based computations are used in a wide variety of applications. For example:

- **Trading:** it is often necessary to compute “one minute bars” the average, maximum, minimum, and/or closing price within each one-minute interval.
- **Click-stream analysis:** it is often useful to compute the number of visitors who click on a particular link within a specified time interval.
- **System management:** applications may compute maximum and minimum CPU usage, memory, and Disk I/O utilization for each machine, within a specified time interval.



## CCL Examples

-- Compute one-minute "bars" (Avg, Max, Min, Closing) and VWAP (Volume-Weighted Average Price). Output results continuously, as soon as results change.

```
INSERT INTO OneMinuteBarView
SELECT Symbol, AVG(Price), MAX(Price), MIN(Price), SUM(Price*Volume)/SUM(Volume)
FROM StreamFeed KEEP 1 MINUTE;
```

-- Determine the total bandwidth consumed by each IP address for the last 1000 downloads. Output results every 15 second.

```
INSERT INTO StreamConsumedBandwidth
SELECT IP, SUM(FileSize)
FROM StreamDownloadLog KEEP 1000 ROWS
GROUP BY IP
OUTPUT EVERY 15 SECONDS;
```

## 4 Database Lookups

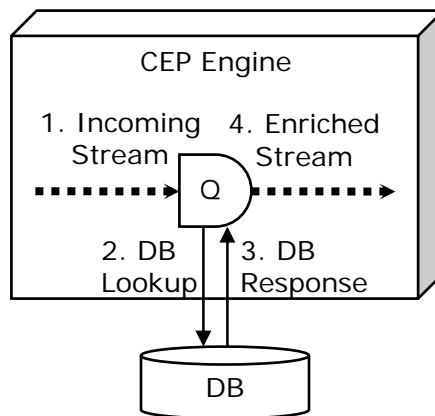
---

### Accessing Databases to Retrieve Historical or Reference Context for Incoming Events

While there are applications that deal exclusively with real-time events, most useful applications refer to historical data or reference data to enrich the incoming events.

The following diagram shows how:

1. An event comes into the system.
2. The engine issues an SQL request to the database and passes a key (from the event) as a parameter to the database query.
3. The database returns a result.
4. The engine combines the result with data from the event, and forwards the enriched event to the next query for further processing.



In SQL terminology, this design pattern implements a *join* between an incoming data stream and a database table.

The more advanced implementations of this design pattern often require:

- **Caching**  
A high throughput is difficult to achieve if the CEP engine must refer to the database on every incoming event. Thus, a good caching layer is critical to performance.
- **Granular caching and access**  
Sometimes it is acceptable to cache entire database tables in memory. Such a paradigm works for small database tables, but does not scale well if many large tables need to be accessed. For example, if an RFID application is trying to look up information about a specific RFID tag, caching the entire database table is often not possible. Accessing and caching specific rows is required for these applications.
- **Concurrent lookups**  
For some applications, it is acceptable to issue serial calls to the database, blocking the processing of events until the database returns results. For others, concurrent lookups that do not block the entire system are a must.

### Sample Areas of Applicability

This design pattern is widely applicable. For example:

- Trading: a trading application may look up historical price for a stock, or certain information about an order, or certain rules and regulations stored in a database.
- RFID application: an application may look up information about a palette or case, identified by its tag ID, or information about the reader that reported the tag. An application may also check where the object should be located, according to the plan stored in the database, and compare this location to the actual location of the object.
- Network security: when deciding how serious an alert is, it may be necessary to refer to other alerts related to the same IP address.

## CCL Examples

In these examples, the SQL lookup code (delimited by [[ ... ]]) is executed in the database, not in the Coral8 Engine.

-- Look up the SKU and the name of the product for each RFID event, via the Tag ID.

```
INSERT INTO
    StreamRFIDEventsEnriched
SELECT
    StreamRFIDEvents.TagID, DbResult.SKU, DbResult.ProductName
FROM StreamRFIDEvents,
    (DATABASE "OracleDb"
    SCHEMA "schemas/product_info.ccs"
    -- SQL Code to perform lookup:
    [[SELECT sku, product_name
    FROM product_info pi
    WHERE pi.tag_id = StreamRFIDEvents.TagID]]) AS DbResult;
```

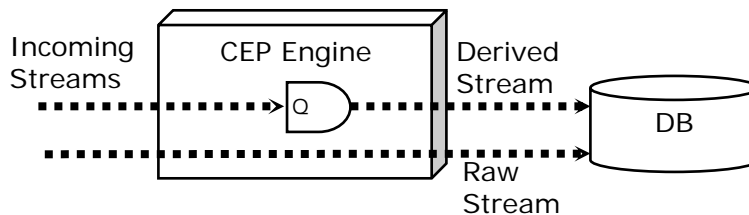
-- Look up yesterday's closing prices for each incoming trade.

```
INSERT INTO
    StreamTradesEnriched
SELECT
    InTrades.Symbol, InTrades.Price, DbResult.ClosingPrice
FROM StreamTrades,
    (DATABASE "OracleDb"
    SCHEMA "schemas/closing-price.ccs"
    -- SQL Code to perform lookup:
    [[SELECT closing_price
    FROM price_history ph
    WHERE ph.symbol = InTrades.Symbol
    AND ph.closing_date = current_
```

## 5 Database Writes

### Sending Raw or Derived Events to a Relational Database

While the CEP Engine can store large volumes of events in windows, it is often necessary to write raw or processed (filtered, aggregated, correlated, and so on) events into a traditional relational database. A relational database can manage very large volumes of data for very long periods of time, and it also supports a number of interfaces that other applications can use to retrieve the data. This design pattern illustrates the complementary nature of databases and CEP engines.



Note that, if the database must store large volumes of events, this design pattern may call for a number of advanced techniques, such as batching, asynchronous writing (to avoid blocking), queuing (to handle spikes), concurrent writes, writing via native database interfaces, and so on.

### Sample Areas of Applicability

This design pattern cuts across a wide range of applications, such as:

- Trading: writing 1 minute bars (maximums, minimums and the closing price for each oneminute interval) into the database.
- Click-stream analysis: storing the raw click-stream history, together with derived data, in the database.
- Network security: storing new relevant security events in the database.

### CCL Examples

```
-- Store aggregated information, calculated every minute, based on a 10-minute
-- sliding window, into an Oracle database table called one_minute_summary.
```

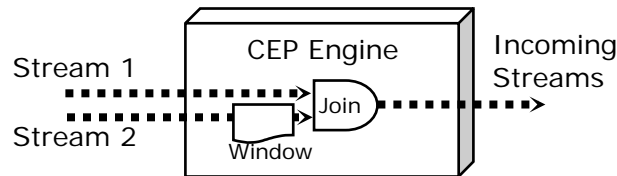
```
EXECUTE STATEMENT
    DATABASE "MyOracleDb"
[[
    -- SQL code to perform an INSERT. Can be SQL or PL/SQL or T-SQL
    INSERT INTO one_minute_summary
    VALUES(?symbol, ?closing_price, ?avg_price, ?max_price, ?min_price, ?vwap);
]]
SELECT Symbol AS symbol,
    Price as closing_price
    AVG(Price) AS avg_price,
    MAX(Price) AS max_price,
    MIN(Price) AS min_price,
    SUM(Price*Volume)/SUM(Volume) AS vwap
FROM StreamFeed KEEP 10 MINUTE
OUTPUT EVERY 1 MINUTE;
```

## 6 Correlation (Joins)

---

### Joining Multiple Event Streams

While simple applications often look just at one stream at a time, most advanced applications must look at and correlate events across multiple streams. A join in a CEP application shares many characteristics with a join in SQL.



In CEP, a join between two data streams necessarily involves one or more windows. Streams do not store events, but pass events to, from, and between queries. To perform a join, it is necessary to store some events in memory, to wait for events on the corresponding stream. This is what a window does. The above diagram depicts a stream to window join. Events arriving in Stream 2 are stored in the window. Events arriving in Stream 1 are joined with events stored in the window, and the matching pairs are published by the join. See the CCL Examples section for specific examples.

Other kinds of interesting CEP joins include:

- **Window to Window joins**  
Joins where data from multiple streams is retained by the windows and is incrementally joined.
- **Outer joins**  
These joins are similar to SQL outer joins, and are surprisingly useful in CEP applications. For example, when joining a stream with a window, it may be necessary to produce results, regardless of whether or not a match is found. An outer join is needed to perform this function.
- **Stream to database joins**  
These joins relate event streams to data stored in a database. Such joins generalize the ideas presented in the Database Lookup design pattern.

Even at low data rates, joins are very CPU-intensive. Thus, most CPE joins require heavy indexing. The indices used in CEP applications are similar to the ones used in relational databases, but there is at least one important difference: CEP indices must be highly dynamic. For example, if a window that stores the last 10 minutes' worth of data is indexed, the index must be updated, both when new events enter the window, and when old events expire from the window. At 10,000 events per second, that's 20,000 index updates per second! A good CEP engine creates proper indices for windows without requiring the user to specify them.

### Sample Areas of Applicability

Most sophisticated CEP applications employ joins. Here are some examples:

- Trading: correlating information from multiple exchanges to find arbitrage opportunities.
- Business process monitoring: correlating information from multiple systems that participate in a business process to manage and track exceptions.
- Network security: correlating information across different security devices and applications for sophisticated intrusion detection and response.

## CCL Examples

-- Stream to Window Join: keep a 10 second window on stream StreamFeed2,  
-- and join all the events from StreamFeed1 with this window.

```
INSERT INTO StreamMatches
SELECT StreamFeed1.*, StreamFeed2.*
FROM StreamFeed1,
     StreamFeed2 KEEP 10 SECONDS
WHERE StreamFeed1.Symbol = StreamFeed2.Symbol;
```

-- Same as above, but perform a left outer join: produces outputs  
-- (containing some NULLS) even if no matches are found.

```
INSERT INTO StreamMatches
SELECT StreamFeed1.*, StreamFeed2.*
FROM
     StreamFeed1
  LEFT OUTER JOIN
     StreamFeed2 KEEP 10 SECONDS
ON
     StreamFeed1.Symbol = StreamFeed2.Symbol;
```

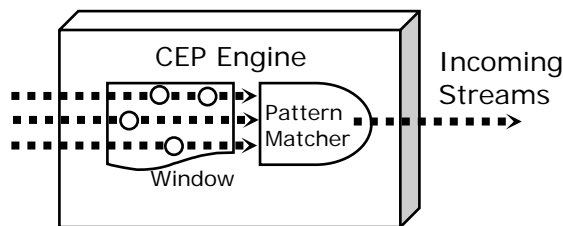
-- Window to Window join: keep windows on both streams. This example correlates  
alerts from a Virus Checker System and an Intrusion Detection System.

```
INSERT INTO StreamAlertCommonIP
SELECT StreamIDSAlerts.IP
FROM
     StreamIDSAlerts KEEP 10 MINUTES,
     StreamVirusCheckAlerts KEEP 10 MINUTES,
WHERE
     StreamIDSAlerts.IP = StreamVirusCheckAlerts.IP;
```

## 7 Event Pattern Matching

### Complex Time-Based Patterns of Events across Multiple Streams

The continuous joins, discussed in the Correlation pattern, are quite powerful, but some tasks make the use of multiple joins very cumbersome. Suppose we want to be notified if, within a 10 minute interval, event A occurs, followed by event B, followed by either event C or D, followed by the absence of event E, with all events relating to each other in some way. While such an event pattern can be tracked with a combination of inner and outer joins, it is often desirable to have a more direct way of expressing such time-based relationships. The diagram below depicts a similar pattern with four events in three streams:



Most interesting event patterns involve a number of relationships among events:

- **A followed by B**  
Event B occurs after event A.
- **A and B**  
Both events A and B occur, in either order.
- **A or B**  
Either A or B (or both) occur.
- **Not A**  
Event A does *not* occur. Some of the most interesting patterns involve “negative” events, in which the pattern matching criteria are met when a specified event does *not* occur within the specified time interval. For example, when tracking a process based on requests and responses, it may be important to know when a response does not occur within a specified time period from the request, or when a response occurs without first being preceded by a request.

### Sample Areas of Applicability

Event patterns occur naturally in situations where complex behavior is tracked, such as:

- **Fraud detection:** fraud patterns are often described as a sequence of events, in one or more streams. For example, in financial services, many fraud patterns involving traders and brokers include events, such as the broker taking an order from the customer and emailing the trader, the trader issuing a certain trade and, perhaps, calling another trader, the other trader waiting for certain market events then issuing another transaction, and so on.
- **Business process monitoring:** many instances of business process failures may be described as patterns. For example, an application may initiate a certain sequence of steps, some of which complete normally, while others encounter problems because of another application.
- **Network security:** network attacks are often sophisticated, and involve a number of events. For example, an attacker may send a certain sequence of packets to certain ports, then try to authenticate across a number of servers and applications, send another sequence of packets, and so on. Tracking and preventing such attacks, especially distributed denial of service attacks, involves the monitoring of a large number of patterns.

## CCL Examples

```
-- CCL uses the following operators for event patterns:
-- "," (comma) to mean "followed by"
-- "&&" to mean "AND"
-- "||" to mean "OR"
-- "!" to mean "NOT" (event did not happen)

-- Here is a sample pattern that uses all of these operators.
-- This example tracks three streams: Stream1, Stream2, Stream3, and looks for five
-- events (a-e) defined within these streams, where the following sequence of
-- events occurs: first events a and b occur, then c or d occurs, and then e does NOT
-- occur, all within a 10-second interval. Other conditions are also specified in the
-- WHERE clause.
```

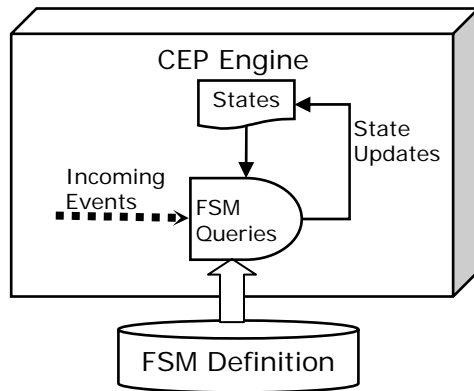
```
INSERT INTO
    StreamAlerts
SELECT
    Stream1.id
FROM
    Stream1 a, Stream1 b, Stream2 c, Stream2 d, Stream3 e
MATCHING
    [10 SECONDS: a && b, c || d, !e]
ON
    a.id = b.id = c.id = d.id = e.id
WHERE
    a.size > 1000 AND b.age < 10;
```



## 8 State Machines

### Modeling Complex Behavior and Processes via State Machines

State machines are used in a wide variety of applications, where complex behavior and processes need to be modeled and tracked. A simple finite state machine (FSM) defines a set of states for a process, together with events that define transitions from one state to another.



A few important considerations when designing a finite state machine in a CEP environment are:

- **Metadata is in a database**  
It is a good design practice not to hard-code the definition (the possible states and transition rules) of the finite state machine, but to keep it in a relational database, where it may be easily inspected and modified.
- **Tracking multiple processes**  
A finite state machine typically tracks not one, but many processes, each identified by a process ID. The state of each process is stored in memory, and backed up on disk, if state persistence is enabled.
- **Coping with imperfections**  
While in the ideal world all processes would always be in a valid state, events would never be lost, and no illegal transitions would occur, the real world is far more complex. A production finite state machine is usually designed to cope with and recover from these failures.

### Sample Areas of Applicability

Finite state machines are applicable wherever an application tracks complex behavior and processes. Here are some examples:

- **Order processing:** in many businesses, order processing is highly complex, as the order goes through many stages, or states. For example a request for a price quote is issued, the order is received, then approved and fulfilled and so on. The number of states sometimes reaches into hundreds and thousands.
- **Tracking business processes:** many other business processes involve a number of complex interactions among applications. These interactions may be tracked with an FSM.
- **Tracking user behavior on a web site:** user behavior on a web site may be described as a series of transitions between states (First-time user, Researching Products, Looking for a product, Trying to place an order, Lost, Order completed, and so on). Tracking and responding to these transitions can often be accomplished with an FSM.

## CCL Examples

For a detailed example, please see the Finite State Machine sample available at <http://coral8.com/developers/samples.html>. A simplified main query is depicted below. In this example, the StateEventStream triggers transitions to a new state. The new state, the action corresponding to the current state and the incoming event are retrieved again from Oracle's table xdfsm\_state\_transitions.

Of course, the engine would not have to go to the database to perform the lookup on every event due to caching. Without caching, the performance of this FSM would be poor.

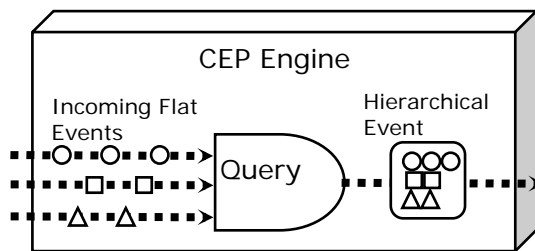
```
INSERT INTO CurrentStateStream
SELECT StateEventStream.Id, DbFsm.ToState, DbFsm.Action,
       StateEventStream.FromState, StateEventStream.Event
FROM StateEventStream,
     (DATABASE "OracleDb"
      SCHEMA "../FiniteStateMachine/schemas/FsmDod.ccs"
      [[SELECT to_state, action
        FROM xdfsm_state_transitions d
        WHERE d.from_state = ?StateEventStream.FromState
           AND d.event = ?StateEventStream.Event
      ]])
) AS DbFsm;
```

## 9 Hierarchical Events

### Processing, Analyzing, and Composing Hierarchical (XML) Events

Most simple CEP applications analyze flat events. A flat event is similar to a row in a database table: it has a fixed number of fields, corresponding to the columns of the table. Flat events provide sufficient functionality for many applications, but other applications deal with events that are more complex. For example, a Purchase Order event may contain a list of the ordered items. Such hierarchical events appear more often in CEP applications, especially with the rise of XML and SOA.

This diagram depicts the design pattern where a complex event is created out of a number of simple events:



The following operations may be performed on hierarchical events:

- **Decomposing hierarchical events**  
A complex, hierarchical event may need to be decomposed into simpler events. For example, it may be necessary to know the items that make up a purchase order.
- **Correlation across hierarchical events**  
It may be necessary to know, for example, if two or more purchase orders contain the *same* item.
- **Composing a hierarchical event**  
For example, it may be necessary to compose a purchase order from a list of items.

While all these operations may, theoretically, be modeled with flat events, it is often surprisingly difficult to do so in practice. For example, if one hierarchical event is represented as a number of smaller flat events, it becomes necessary to make sure that all the “component events” travel together through the queries that analyze them, which is difficult to achieve. Direct support for hierarchical events is needed.

### Sample Areas of Applicability

Finite state machines are applicable wherever complex behavior and processes are tracked. For example:

- Order processing: see the previous section for examples.
- RFID applications: an RFID-tagged pallet may contain a number of RFID-tagged cases, each of which may contain a number of RFID-tagged items. Hierarchical events are typically necessary to model such containment directly.
- News feed and RSS monitoring: hierarchical events often arise in these applications.

### CCL Examples

This example demonstrates the use of CCL XML functions to create a hierarchical event that contains information about a pallet and all the cases that belong to it. This example performs a join between

a stream called Pallets and a window that holds records of all the cases seen during the past one minute.

The resulting hierarchical events correspond contain multiple <case\_id> tags and may look like this:

```
<pallet>
  <pallet_id>RSP436090</pallet_id>
  <scan_time>2005-12-22 12:04:27</scan_time>
  <case_id>QTB53294</case_id>
  <case_id>DXQ63860</case_id>
  <case_id>MQD960761</case_id>
  <case_id>LIY358183</case_id>
</pallet>
```

```
INSERT INTO
  Pallet_out
SELECT
  XMLELEMENT(          -- XMLELEMENT is a basic constructor for XML elements
    "pallet",
    XmlElement(
      "pallet_id",
      p.tag_id
    ),
    XMLELEMENT(
      "scan_time",
      TO_STRING(dp.ts, "YYYY-MM-DD MM:HH:SS")
    ),
    XMLAGG(            -- XMLAGG aggregates all the matching cases together
      XMLELEMENT(
        "case_id",
        c.case_id
      )
    )
  )
FROM
  Pallets AS p
  LEFT OUTER JOIN
  Cases AS c KEEP 1 MINUTE
  ON p.tag_id = c.pallet_id;
```

The full version of this example is available at <http://www.eng.coral8.com/developers/samples.html>.

## 10 Dynamic Queries

### Submitting Parameterized Queries, Requests, and Subscriptions Dynamically

Many CEP discussions revolve around the subject of continuous queries, that is, queries registered with a CEP engine and triggered by the arrival of data. This does not mean, however, that CEP queries cannot be submitted dynamically, or explicitly triggered by the users. Dynamic queries come in several flavors:

- **Dynamic registration of continuous queries**

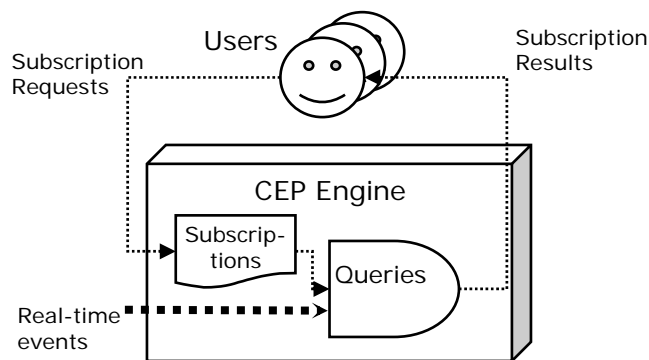
In many applications, the ability to register continuous queries programmatically, without restarting the server, is important. Note that this applies not just to queries, but also to data streams, publishers, and subscribers. All of these often need to be added dynamically.

- **Request/response queries**

Request/response queries analyze streaming data, but these queries return results only upon explicit request from a user. For example, a request/response query may keep a running average over a window, but only return the results when the user issues a request. Often the query itself is pre-registered with the engine, and its execution is triggered by a message on a separate "request" stream.

- **Subscription queries**

Subscription queries are similar to request/response queries, as these queries are instantiated by an explicit command. However, while request/response queries produce a single response immediately, subscription queries register interest in certain events, and the responses are streamed to subscribers. The diagram illustrates how the engine keeps a list of subscriptions from users, and dispatches the results of queries to the right subscriber:



Request/response and subscription queries often require parameterization. Typically, this means that the query developer registers a single subscription template, but business users supply their own values for parameters. For example, the query template may include specification to notify the user when the running average of the price of Stock X grows by more than Y%, starting from time Z. One trader may issue a subscription with X=MSFT, Y=.1%, Z=10 am. Another may issue a subscription with X=IBM, Y=.2, and Z=1 pm. It is not uncommon for thousands of instances of a single query to be registered, all with different parameters.

### Sample Areas of Applicability

Dynamic queries come up in many applications, especially those that involve large numbers of business users. Here are some examples:

- Trading environments: every trader can enter their subscriptions, as described above.

- Enterprise portals: every user of an enterprise portal, from the CEO down, can subscribe to different queries. Again, parameterization is important here: while the CEO may register interest in incoming purchase orders over one million dollars, a sales manager may want to know about all purchase orders for their territory.
- Fraud detection and other machine learning applications: machine learning applications in a CEP environment must dynamically adjust both the queries and parameters, in response to ever-changing external conditions.

## CCL Examples

The queries described in this section are extremely varied, and cannot be illustrated with a single example. Some tasks, such as submitting a continuous query dynamically, are performed via a special programmatic interface, similar to the mechanism used to submit SQL queries via ODBC or JDBC.

The example below illustrates a simple set of queries, used to compute whether stock “Symbol” has risen over Y% above the average price for the past 10 minutes.

```
-- A Subscriptions window to maintain subscriptions
CREATE WINDOW Subscriptions(SubId INTEGER, Symbol STRING, Threshold FLOAT)
KEEP 1 DAY; -- Expire at the end of the day

-- Populate the window from StreamSubscriptionRequests.
INSERT INTO Subscriptions
SELECT *
FROM StreamSubscriptionRequests;

-- Compute all average prices for a 10-minute period, based on data from StreamFeed.
INSERT INTO StreamAvgs
SELECT Symbol, Price, AVG(Price),
FROM StreamFeed KEEP 10 MINUTES;

-- Now, join StreamAvgs with the Subscriptions window, to find all the results
-- that must be sent to the subscriber.

INSERT INTO Subscription Response
SELECT sa.SubId, sa.Symbol, sa.Price
FROM StreamAvgs AS sa,
     Subscriptions AS sub
WHERE Sa.Symbol = sub.Symbol AND -- symbols match
      (sa.Price - sa.AvgPrice) / sa.AvgPrice -- crossed the threshold
      * 100 > sub.Threshold;
```

More examples of request/response and subscription queries are found in the examples shipped with the Coral8 Engine, such as Finance/finance-pubsub (basic request/response and subscriptions), Finance/vwap (implements Interval VWAP) and others.

## Conclusion

---

We have come to the end of our discussion of the ten fundamental CEP design patterns. We hope that this paper has clarified how a CEP engine can be used to implement filtering, aggregation, caching, correlation, event pattern matching, interfacing with databases, XML processing, finite state machines, and dynamic queries. These patterns can be easily combined to create a large variety of CEP applications from many domains.

Finally, a few words on what this paper did *not* cover. While we chose to focus on the fundamental CEP design patterns, we have not talked about CEP developer tools (development environments, compilers, debuggers, visualization tools, etc.), enterprise features (scalability, high-availability, deployment, integration, management, security, etc.), and other important subjects. We encourage the reader to contact the relevant product documentation for a full discussion of these topics. The latest Coral8 Engine documentation, and the Corla8 Engine itself, can be downloaded from <http://www.coral8.com/developers/>