

Complex Event Processing in the Real World

*An Oracle White Paper
September 2007*

Complex Event Processing in the Real World

INTRODUCTION

Today, probes and sensors are deployed in everything from IT networks to enterprise software systems and physical world devices (through RFID readers, bar code scanners, manufacturing equipment sensors, and others). As these systems continue to proliferate, they generate events at a growing rate. Significant improvements in operational business decisions await those organizations that can capture and process these events into meaningful business insight.

A new class of event-processing solutions has entered the market that integrates into standard middleware architectures and enables event processing to be embedded in any standard enterprise application. These new complex event processors bring the power of event-driven insight to any industry and any business user.

The Oracle Complex Event Processor (CEP) provides a rich, declarative environment for the development of event processing applications that can process and act on hundreds of thousands of events per second. Key CEP features include pattern matching, user-defined windows for event evaluation (including time windows, row windows, predicate windows and landmark windows) and the contextual enrichment of events.

CEP's use of extensions to the SQL language, called Oracle Continuous Query Language or CQL, enable anyone with standard SQL skills to quickly develop CEP-based applications. CEP can be deployed as a stand-alone offering on third party application servers or as an integrated service engine within the Service Infrastructure of the Oracle Application Server 11g.

CONTINUOUS QUERY LANGUAGE (CQL)

CEP's use of extensions to the SQL language, called Oracle Continuous Query Language or CQL, enable anyone with standard SQL skills to quickly develop CEP-based applications.

Databases are best equipped to run queries over finite stored data sets. However, many modern applications require long-running queries over continuous unbounded sets of data. By design, a stored data set is appropriate when significant portions of the data are queried repeatedly and updates are relatively infrequent. In contrast, data streams represent data that is changing constantly, often exclusively

through insertions of new elements. It is either unnecessary or impractical to operate on large portions of the data multiple times.

Many types of applications generate data streams as opposed to data sets, including sensor data applications, financial tickers, network performance measuring tools, network monitoring and traffic management applications, and click stream analysis tools. Managing and processing data for these types of applications involves building data management and querying capabilities with a strong temporal focus.

To address this requirement, Oracle has introduced CEP, a data management infrastructure that supports the notion of streams of structured data records together with stored relations. Streams can include multiple tuples. The term tuple refers to the data portion of a stream, excluding the timestamp. Streams can be of the following three types:

- Time-ordered collection of tuples
- Bag of tuple, timestamp pairs
- Mapped time to set of tuples

To provide a uniform declarative framework for processing streams and tuples, Oracle offers Oracle Continuous Query Language (Oracle CQL), a query language based on SQL with added constructs that support streaming data. Oracle CQL is designed to be:

- Scalable with support for a large number of queries over continuous streams of data and traditional stored data sets
- Adaptive and capable of dealing with cases where abrupt changes in data stream rates and system resource limitations

KEY CQL CONCEPTS

The major concepts within CQL include streams, relations, operators, functions, and patterns.

Streams

Streams feed raw data to the Oracle Complex Event Processor. Stream sources can be any database, file, or JMS topic. CEP can access data from stream sources through either a push-based mechanism whereby the source raises data to CEP or through a pull based mechanism, whereby CEP pulls or queries data from the source system.

Streams are created in CQL by using the CREATE STREAM DDL. Every stream created to feed data into CEP has two elements, the data itself, which are called tuples and a timestamp. Timestamps in any stream reflect an application's notion of time, not particularly system or wall-clock time. The timestamp is part of the

schema of a stream, and there could be zero, one, or multiple elements with the same timestamp in a stream.

There are two classes of streams: base streams, which are the source data streams that arrive at the CEP engine, and derived streams, which are intermediate streams produced by operators in CQL.

Relations

Relations identify the relationships between incoming data elements in CEP. In the standard relational model a relation is simply a set (or bag) of tuples, with no notion of time as far as the semantics of relational query languages are concerned.

In CQL the term instantaneous relation is used to denote a relation in the traditional bag-of-tuples sense, and relation to denote a time-varying bag of tuples. The term base relation is used for input relations and derived relation for relations produced by CQL query operators.

Operators

There are three classes of operators used with streams and relations in CEP:

- *Relation-to-Relation* – These operators take 1 or more relations as inputs (depending on the specific operator) and produce a relation as output. Examples are Join, Select (Filter), and Project etc A CQL stream-to-relation operator takes a stream as input and produces a relation as output.
- *Stream-to-Relation* – These operators take a stream as input and produce a relation as output. The concept of a window over a stream can be used to define operators belonging to this class. Unlike relational database tables, data streams typically do not end. It is therefore useful to be able to define windows, or subsets of the streams. CQL supports 6 types of windows: Time, Row, Partition, Predicate, Extensible and Landmark.
- *Relation-to-Stream* – These operators take a relation as input and produce a stream as output. CQL supports 3 Relation-to-Stream operators: Insert Stream (IStream), Delete Stream (DStream), and Relation Stream (RStream).

Functions

Oracle CEP supports a library of functions including standard SQL functions, mathematical and statistical functions. Examples of supported SQL functions are in the following table.

CONCAT	LOG_NOT	RAWTOHEX
HEXTORAW	LOG_OR	SYSTIMESTAMP
IS_NULL	LOG_XOR	TO_BIGINT
LENGTH	NVL	TO_FLOAT
LOG_AND	PREV	TO_TIMESTAMP

Oracle CEP also enables users to “plug” in off-the-shelf software into the CEP engine to reference pre-defined algorithms while defining CQL. These include existing open source providers of Java algorithms, off-the-shelf mathematical packages and user-defined implementations of algorithms. Oracle CEP provides an extensibility framework through which users can register functions (with Java implementations) with the CEP system and also reference these external functions in their CQL. Oracle CQL supports `CREATE`, `ALTER`, and `DROP` with user-defined functions.

Patterns

Oracle CEP also has the capability to detect regular expression patterns in real-time. This is supported through the pattern recognition extensions to SQL in CEP. In CEP, Pattern recognition comes in two flavors:

1. Recognize patterns in streams
2. Recognize patterns in relations

The pattern recognition extensions, add a number of sub-clauses to SQL for the specifications of patterns. We add a `MATCH_RECOGNIZE` sub-clause which takes a regular expression pattern defined over a regular expression of alphabets as argument. This can be followed by an optional `PARTITION BY` over an attribute of the incoming stream or relation. In addition to these, we allow a user to specify expressions through a `MEASURES` sub-clause, followed by a choice of `ONE` or `ALL ROWS PER MATCH`. The latter dictates whether or not overlapping patterns are matched or not. All this is then followed by a `DEFINE` sub-clause, which defines the alphabets in the pattern. For example:

```
SELECT <select-list> FROM <stream|relation-name>
MATCH_RECOGNIZE (PARTITION BY)
MEASURES (...)
ONE | ALL ROWS PER MATCH
PATTERN <pattern-expression>
DEFINE <define-alphabets>)
```

APPLICATIONS OF CEP AND CQL

CEP and CQL are technology components offered by Oracle for application in a variety of industry use cases. The following examples showcase business applications in which CEP can have a significant impact on business insight. Each example is presented in conjunction with a series of sample CQL statements that showcase how these examples could be executed in the CQL language.

Financial Services: Banking or Stock Transaction SLAs

The volume, speed, and complexity of automated financial transactions offer a host of well suited examples for complex event processing. One of the more complex sets of transactions in financial markets is stock transactions between institutions. A financial services institution sets strict Service Level Agreements (SLAs) with its partners to ensure timeliness and accuracy of stock trades. Yet, as you can imagine, these SLAs have traditionally been difficult to monitor and execute. It is here that complex event processors can help.

Here is an example of a financial services institution that has a complex SLA defined. The elapsed time from when a single trade is first identified to the institutions' transaction processing system to the time that the system gets the first status update back from the trading system needs to be no greater than 200 seconds for 95% of the trades received in the last 60 minutes. And measurements are taken every 5 minutes.

To solve this example with the Oracle CEP, we first parse the data feeds that the financial institution is receiving into two data streams for analysis, TradeInputs and TradeUpdates.

```
CREATE STREAM TradeInputs (tradeId integer)
CREATE STREAM TradeUpdates (tradeId integer)
```

Then we create a view on those streams. Remember that a view here is just like a database view but instead of database tables, this view operates off of our data streams in real-time. Our view for this sample tracks our cut-off window of 200 seconds described in the use case above.

```
CREATE VIEW CutOffTrades(tradeId, tradeVolume) AS
DStream(SELECT * from TradeInputs[RANGE 200 SECONDS]);
```

We use this view and build on top of it a secondary view to now track the trades that satisfy the SLA of getting an update back from the trading system within the defined 200 second time window.

```

CREATE VIEW OKTrades(tradeId) AS
IStream(SELECT T.tradeId FROM CutOffTrades[NOW] AS R,
TradeUpdates[RANGE 200 seconds] AS T
WHERE R.tradeId = T.tradeId);

```

To complete our CQL queries we then want to automate the calculations that determine whether we have met our SLAs or not. Thus, we create a query that is a count of Total Trades, another query that is a Count of OK Trades, e.g. those trades that were successfully executed within the 200 second time window, and a final third SLA query that determines whether we were in violation of our SLA.

Count of Total Trades

```

CREATE VIEW TotalTrades(tradeCount) AS
SELECT COUNT(*) from CutOffTrades[RANGE 1 hour];

```

Count of Total OK Trades

```

CREATE VIEW TotalOKTrades(tradeCount) AS
SELECT COUNT(*) from OKTrades[RANGE 1 HOUR];

```

SLA Query

```

CREATE QUERY Q AS SELECT T.totalCount, F.totalCount FROM
TotalTrades as T, TotalOKTrades as F
WHERE F.tradeCount < 0.95*T.tradeCount;

```

With these CQL queries defined and deployed in Oracle CEP, our system is up and running. Note that we used only simple CQL constructs in this example, CREATE STREAM, CREATE VIEW and CREATE QUERY. Other examples in this whitepaper will showcase the application of more advanced constructs.

The output of the CEP processing can then be fed to components of Oracle's SOA Suite to deliver real-time alerts, notifications and real-time dashboards through Oracle BAM or orchestrate exception handling business processes through the Oracle BPEL Process Manager.

Financial Services: Algorithmic Trading

A typical complex event processing use case can be found in automated stock trading using algorithmic trading. Prior to the advent of complex event processing engines, stock traders often sat at a desk manually monitoring 5-8 screens to identify patterns in the market that indicated an opportune moment for a trade. Analytic information and patterns were often manually tracked in a spreadsheet. Today, this type of tracking can be entirely done in the complex event processing engine and a trade automatically triggered when the patterns are right. While the system is monitoring real-time market conditions, the traders gain time and mindshare to evaluate the performance of their algorithms and refine them.

For example, a trading desk may want to follow movement in a particular market segment as an indicator to buy or sell a stock. For example, a stock trader who is following the software market may be interested in tracking Microsoft and IBM's stock. By comparing the movement of both stocks to the S&P Index, a trader may want to gauge when Microsoft or IBM is defecting from the pattern of stock trades within the S&P Index, and thus signal an appropriate time to buy or sell. This simplified, algorithmic trading logic may be similar to this:

```
IF MSFT price moves outside 2% of MSFT-15-minute-VWAP (volume-  
weighted average price) FOLLOWED BY S&P moving by 0.5% AND  
IBM's price moves up by 5%  
OR MSFT's price moves down by 2%  
ALL within a 2 minute time window  
THEN BUY MSFT and SELL IBM;
```

For the sake of constructing a realistic example, assume that VWAP = volume weighted average price is tracked here over a 15 minute window. That means that $VWAP_MSFT_15 = \frac{\sum(V_i C_i)}{\sum(V_i)}$ where C_i is the cost of stock and V_i the volume for each time the stock trades in a 15 minute window.

Previous technologies would tackle this problem with a complex series of 10-20 nested rules to support this simple use case. More complex cases are likely to require 50-100 rules.

By defining the relations between streams that can be used for real-time analysis of complex patterns within time windows Oracle CEP can use 4 queries to produce the same output as the 10-20 nested rules required in older solutions. To solve this use case, CEP would create 3 derived data streams or stream views. One derived data stream would cover the vwap_stream over a 15 minute window using a user defined function to compute VWAP. Note that this use case also highlights the application of the CQL syntax for pattern matching. Pattern matching is accomplished here through use of the advanced CEP construct PATTERN.

```
CREATE VIEW vwap_stream (vwap_price) AS  
RStream(SELECT symbol, VWAP(price) FROM ticker [RANGE 15  
minutes]);
```

To ensure that we are tracking MSFT stock in relationship to the VWAP stream, we take the above derived stream s and join it with the ticker stream for symbol MSFT.

This is simple too as I create a view on the streams (similar to a database view but with CQL this view is applied to data streams rather than database tables) by joining the ticker with vwap_stream and looking at the modulus of the two prices for symbol = "MSFT".

```

CREATE VIEW vwap_outside_price(vwap_outside_count) AS
SELECT COUNT(*) AS price_outside_vwap FROM ticker, vwap_stream
[range 15 minutes]
WHERE |price - vwap_price| > 0.02*price AND symbol = "MSFT";

```

The final step is to create a final view that uses the power of pattern matching to look at the complex condition that we have specified in our algorithmic trading logic. This is to identify when the conditions of relative movement between the S&P Index and both the MSFT and on IBM stocks has matched our expectations.

```

CREATE VIEW trade_cond_stream (matching_row_count) AS
SELECT COUNT(*) FROM ticker [RANGE 2 minute]
RECOGNIZE ONE ROW PER MATCH
PATTERN [S T]
DEFINE S AS |price - PREV(price)| <= .05*PREV(price) AND symbol =
"S&P"
DEFINE T AS (price >= 1.05*PREV(price) AND symbol = "IBM") OR
(price <= 1.02*PREV(price) AND symbol = "MSFT");

```

Our trade will execute when the answer to this query is true, which occurs only when the outputs of Sample 2 and Sample 3 have a nonzero count.

Transportation: Toll System Management

Municipalities worldwide are evaluating new business models for toll roadways. The standard flat-fee toll way is no longer sufficient. Charging a single price for every vehicle, regardless of time of day or congestion is not an efficient model. It creates the potential for congested toll ways for drivers and does not maximize revenue for municipalities. And with today's vehicle transponder technologies, much more advanced toll schemes are possible. Recently, a good deal of research has been prepared on systems where tolls can be computed dynamically based on congestion, currently known accidents, and strategies for optimizing traffic. What has been holding many municipalities back is the software to consistently evaluate and price tolls based on these factors in real-time. Complex event processing provides a solution.

This example is based on a research study called the Linear Road Benchmark model. It describes a variable tolling system for Linear City where tolls are computed based on surrounding road congestion. Each car on the expressway is equipped with a responder that emits a position every 30 seconds. This enables the system to generate real-time statistics about traffic conditions on every second of every expressway once per minute. These statistics include average vehicle speed, number of vehicles, and the existence of any accidents. All of these input variables are used to determine toll charges for the given segment and control

traffic flow.

Oracle CEP solves this optimization problem by first parsing the responder data feeds by creating an input stream.

CarLocStr: Stream of car location reports

```
CREATE STREAM CarLocStr(car_id, /* unique car identifier */
                        speed, /* speed of the car */
                        exp_way, /* expressway: 0..10 */
                        lane, /* lane: 0,1,2,3 */ dir, /*
                        dir: 0(east), 1(west) */
                        x-pos); /* coordinate in express way */
```

Then CQL is used to define relationships between the data elements in the input stream.

```
CREATE RELATION AllSeg (exp_way integer,
                       lane integer,
                       dir integer,
                       seg integer /* segment */);
```

On top of the input streams and relations, Oracle CEP is then instructed to use Derived Streams to compute the segment of toll way in which a car is currently traveling. Derived streams in CQL are defined as views.

CarSegStr: Derived stream to compute in which segment is the car

```
CREATE VIEW CarSegStr (car_id integer, speed integer, exp_way integer,
                      lane integer, dir integer, seg integer) AS
SELECT car_id, speed, exp_way, lane, dir, divint( x_pos, 5280) AS seg
from CarLocStr;
```

CurCarSeg: Derived relation to compute the current car segment

```
CREATE VIEW CurCarSeg(car_id, exp_way, lane, dir, seg) AS
SELECT car_id, exp_way, lane, dir, seg from CarSegStr
[PARTITION BY car_id ROWS 1 RANGE 35 SECONDS];
```

SegAvgSpeed: Derived relation to compute segments having average speed less than 50

```
CREATE VIEW SegAvgSpeed (exp_way integer, lane integer, dir integer,
                        seg integer, avg_speed float) AS
SELECT exp_way, lane, dir, seg, avg(speed) AS avg_speed
FROM CarSegStr [RANGE 5 MINUTES] GROUP BY exp_way, lane, dir,
seg
HAVING AVG(speed) < 50;
```

SegVol: Derived relation to compute density of cars in a segment
 CREATE VIEW SegVol (exp_way integer, lane integer, dir integer, seg integer, volume integer) AS
 SELECT exp_way, lane, dir, seg, count(*) AS volume FROM
 CurCarSeg GROUP BY exp_way, lane, dir, seg having count(*) > 50;

AccCars: Derived relation to compute accidented cars
 CREATE VIEW AccCars(car_id integer, acc_loc float) AS
 SELECT car_id, avg(x_pos) as acc_loc FROM CarLocStr
 [PARTITION BY car_id ROWS 4 RANGE 120 SECONDS] GROUP BY
 car_id
 HAVING max(x_pos) = min(x_pos) and count(car_id) = 4;

AccSeg: Derived relation to compute accident segment
 CREATE VIEW AccSeg(exp_way integer, lane integer, dir integer, seg integer, acc_loc integer) AS
 SELECT DISTINCT exp_way, lane, dir, seg, toINT(acc_loc) FROM
 CurCarSeg, AccCars
 WHERE CurCarSeg.car_id = AccCars.car_id;

The final CQL statement in the CEP solution defines the query that uses all of the previously defined streams and relations to derive what toll to charge per segment.

SegToll: Derived Relation to compute per segment toll
 CREATE VIEW SegToll (exp_way integer, lane integer, dir integer, seg integer, toll integer) AS
 SELECT SegAvgSpeed.exp_way, SegAvgSpeed.lane, SegAvgSpeed.dir,
 SegAvgSpeed.seg, 2*(SegVol.volume-50)*(SegVol.volume-50) FROM
 SegAvgSpeed, SegVol
 WHERE SegAvgSpeed.exp_way = SegVol.exp_way AND
 SegAvgSpeed.lane = SegVol.lane
 AND SegAvgSpeed.dir = SegVol.dir AND SegAvgSpeed.seg =
 SegVol.seg;

The toll output would be raised by CEP to a transactional application that in real-time would then execute a charge communicated to the drivers' transponder for the cost of the toll way service used during that session. This application could be an Oracle application delivered through Oracle Fusion Middleware or with the hot-pluggable capabilities of CEP, an application delivered through a third party application development environment.

CONCLUSION

Oracle CEP and CQL are new capabilities offered in Oracle Fusion Middleware to support the unique needs modern, event-driven architectures and applications that require long-running queries over continuous unbounded sets of data. Oracle CEP

has been built to support data streams represent data that is changing constantly, often exclusively through insertions of new elements.

Oracle CEP is an integrated component of the Fusion Middleware suite and can be deployed in conjunction with all of the other components of Oracle's Event Driven Architecture and Service-Oriented Architecture offerings. These include (i) a Business Activity Monitoring (BAM) solution to define and monitor events and event patterns that occur throughout an organization; (ii) a Business Rules engine to capture, automate and flexibly change business policies; (iii) Enterprise Messaging to reliably deliver event messages with configurable qualities-of-service; (iv) a multi-protocol Enterprise Service Bus (ESB) to connect applications and route messages; (v) a BPEL Process Manager (BPEL PM) for orchestrating business processes and (vi) a Sensor Edge Server (SES) to enable connectivity to RFID readers and other physical devices to capture and filter events.

FOR MORE INFORMATION

For more information on any Oracle Fusion Middleware components, including CEP and CQL, please visit www.oracle.com/goto/eda.



Complex Event Processing in the Real World
September 2007
Author: Stephanie McReynolds

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Copyright © 2007, Oracle Corporation and/or its affiliates. All rights reserved.
This document is provided for information purposes only and the contents hereof are subject to change without notice.
This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.