Oracle Complex Event Processing: Lightweight Modular Application Event Stream Processing in the Real World

An Oracle White Paper
Updated June 2009

ORACLE

Oracle Complex Event Processing: Lightweight Modular Application Event Processing in the Real World

Oracle Complex Event Processing: Lightweight Modular Application 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). GPS (Global Positioning Systems) data continuously flows indicating the immediate location of critical resources and millions of dollars are lost or gained in microseconds as stock information streams along a market data feed. 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 leveraged in any standard enterprise application. These new complex event platforms bring the power of event-driven insight to any industry and any business user.

Oracle Complex Event Processing (CEP) provides a rich, declarative environment for the development and deployment of event processing applications that can process and act on hundreds of thousands of events per second. Key CEP features include a lightweight EDA Java Application Server and an industry leading CEP engine providing pattern matching, user-defined windows for event evaluation (including time windows, row windows, predicate windows and landmark windows) and the contextual enrichment of events.

The integrated Oracle Complex Event Processing engine uses extensions to the SQL language, called Oracle Continuous Query Language or CQL, to enable anyone with standard SQL skills to quickly develop CEP-based applications.

Oracle Complex Event Processing employs a lightweight, modular software architecture that runs on top of a Java virtual machine (JVM) featuring deterministic garbage collection to provide the reliability and performance needed by real-time, event-driven applications.

The Oracle Complex Event Processing EDA Java application server is designed to support real-time, event-driven applications that can require extremely high throughput and deterministic latency which represents a new frontier for Java-based middleware, in large part due to the challenges posed by garbage collection. To satisfy the demands of these applications, Oracle Complex Event Processing employs a lightweight, modular software architecture that runs on top of a Java virtual machine (JVM) featuring deterministic garbage collection. This architecture facilitates a service-oriented approach in which modules implement interdependent

services that together provide the functionality needed by real-time, event-driven applications.

Oracle Complex Event Processing provides application developers with a hybrid programming model featuring both Java and the Continuous Query Language (CQL). This allows application developers to mix and match Java and CQL seamlessly within the same application to deliver benefits of both complex event processing technology—including incremental evaluation of CQL queries—and the ability to write business logic in a high level programming language with built-in memory management.

Oracle Complex Event Processing applications are composed of reusable components that are configured using dependency injection techniques. Because these applications are configuration-driven, arbitrary configuration changes may be made to the server without interrupting running applications. For example, an CQL rule set can be changed in real time resulting in a change in program behavior without interrupting the running application. Such dynamic support results in a new level of flexibility for enterprise applications.

The remainder of this paper expands on these capabilities and investigates CQL concepts with the introduction of real world customer use cases.

ORACLE COMPLEX EVENT PROCESSING ARCHITECTURE

At its core, Oracle Complex Event Processing is a Java container implemented with a lightweight, modular architecture based on the Open Services Gateway initiative (OSGi™). It provides a complex event processing (CEP) engine and Continuous Query Language (CQL), as well as a rich development platform without sacrificing performance.

Server Architecture

Oracle Complex Event Processing provides a complex event processing (CEP) engine and Continuous Query Language (CQL), as well as a rich development platform without sacrificing performance.

Oracle Complex Event Processing is architected as a set of relatively fine-grained software modules. Each module contains a set of Java classes that implement some aspect of the server's functionality. A module may act as a library for other modules by exporting classes for client modules to use. A module may also register services in a service registry for use by other modules.

Figure 1 shows the high level software architecture of an Oracle Complex Event Processing instance. At the lowest level there is a Java virtual machine [13] with deterministic garbage collection (DGC) [3]. The JVM provides the foundation for support of applications that demand deterministic latency by limiting the length of garbage collection pauses. The next layer is composed of the modularity framework [15] which allows modules to control the import and export of Java classes. The modularity layer also handles class loading and provides versioning support for classes. The service framework [16] is responsible for instantiating the classes that implement a service and for resolving dependencies between services.

The service framework uses dependency injection (DI) [7] to provide service instances with configuration data and references to other services that they need.

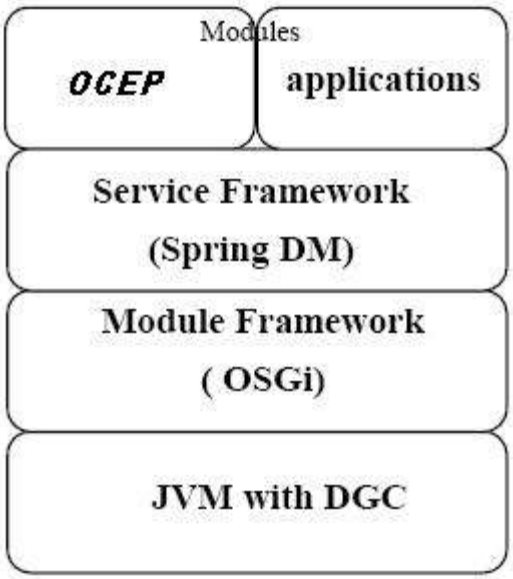

Figure 1. Oracle Complex Event Processing software stack.

The final layer represents the modules themselves. There are two fundamental types of modules--modules that are part of the server implementation and modules that make up applications deployed to a server instance. Typically, one or more modules will implement a logical subsystem in the server, such as the configuration or deployment subsystems. Applications may also be composed of one or more modules. The architecture is uniform in that there is no physical distinction between server and application modules. Application modules use services provided by server modules. In addition, application modules can extend the functionality of the server by providing services that are invoked by server modules.

Figure 2 shows a sampling of the server modules that make up Oracle Complex Event Processing. Most of the subsystems implemented by these modules, such as configuration, deployment, security, and logging would also be found in more traditional application server implementations.

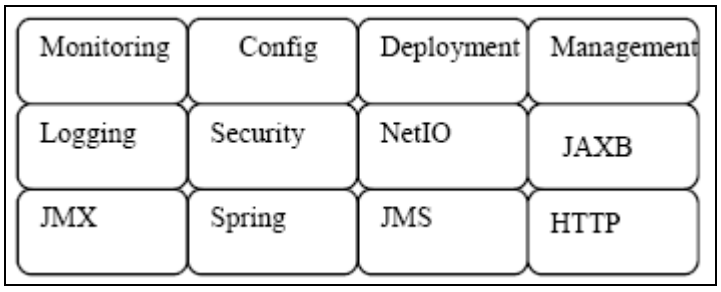

Figure 2. Examples of Oracle Complex Event Processing server modules.

An example of a library module is JAXB [12] which is used by the configuration module for processing XML configuration. The configuration module in turn exports a service that is used by the management module to read and update server-wide and application-level configuration. These relationships are illustrated in Figure 3.

To avoid cyclical dependencies Oracle Complex Event Processing uses a layered approach in which server services are implemented as components using a generic component framework [16] that provides capabilities such as dependency injection, and service lifecycle management.

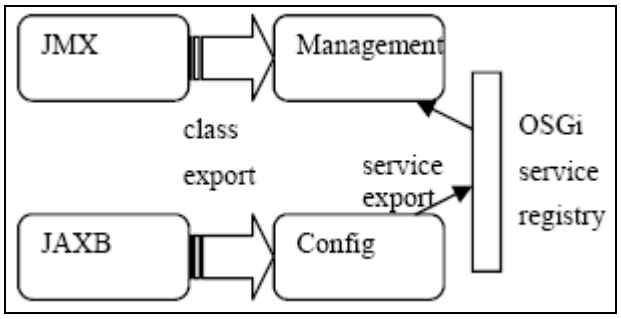

Figure 3. Module class and service dependencies.

Given the large number of modules in the server, the dependency relationships could become quite complex. To avoid cyclical dependencies Oracle Complex Event Processing uses a layered approach in which server services are implemented as components using a generic component framework [16] that provides capabilities such as dependency injection, and service lifecycle management. All of these services are implemented using the Java programming language.

Distributed Architecture

Event-driven architecture is a distributed architectural style composed of decoupled applications that interact by exchanging events. Event-driven applications either initiate or process events through sense-and-respond.

Due to its asynchronous and decoupled nature, event-driven architecture has an intrinsic quality for scaling and supporting real-time constraints.

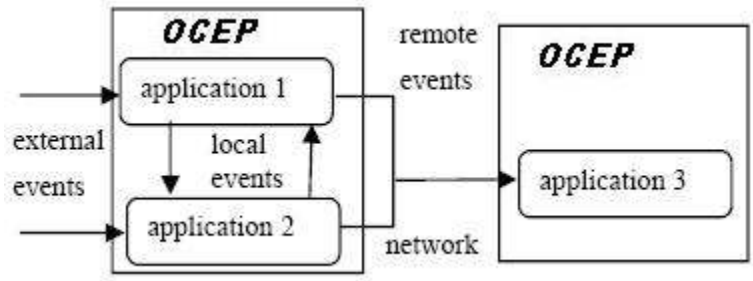

Figure 4. Event-driven applications deployed in Oracle Complex Event Processing.

Figure 4 illustrates Oracle Complex Event Processing as part of a distributed, event driven architecture. In this example application modules application1 and application2 are deployed in one instance of Oracle Complex Event Processing and send events to a third application deployed in a separate instance. In addition, both application1 and application2 receive events from an external event source, such as a market data feed in a financial application.

Event-driven architecture is important, because the real-world is event-driven. One example is the financial services industry in which trader applications react to events (or changes) made to the financial exchange market. Event-driven situations should be modeled by event-driven architecture. Due to its asynchronous and decoupled nature, event-driven architecture has an intrinsic quality for scaling and supporting real-time constraints.

PROGRAMMING MODEL

Oracle Complex Event Processing provides a native programming model for authoring event-driven applications. The model supports applications that are a mixture of reusable Java components and CQL.

Event-driven Components

In Oracle Complex Event Processing, a user defines an event-driven application by assembling a set of event-driven components, including:

- **event source**: a component that generates events
- **event sink**: a component that consumes events
- **channel**: which can be a Stream or Relation
    - **stream**: a component through which events flow, provides queuing and concurrency
    - **relation**: identifies the relationship between incoming data elements
- **processors**: a component capable of processing events
- **event types**: metadata defining the properties of events

Within an application, events flow starting at event sources. Events then flow through a set of processors and finally reach the event sink(s). Event sources and sinks are also termed adapters because they are primarily responsible for converting events from their external wire format to the Java format understood by the application and vice versa. Channels link components together forming an event flow graph, which is called an Event Processing Network (EPN). Formally, an EPN is a non-rooted directed graph, in which vertices/nodes represent instances of event sources, event sinks, or processors, and arcs represent streams.

An EPN is a formal model, based upon Petri Networks [17], which allows:

- the specification of the concurrency between the EPN nodes
- the prioritization of EPN paths
- the composition of applications

Figure 5 provides an example of an EPN for a simple financial market pricing application. This event-driven application contains two event sources, each

receiving stock tick events from two different exchange markets. The application further defines a processor that is configured to calculate and output the price of a stock symbol as being the average price received from the exchange market event sources. Finally, there is a single event sink that publishes the calculated average stock price to a well known JMS [19] destination.

Event sources and sinks are also termed adapters because they are primarily responsible for converting events from their external wire format to the Java format understood by the application and vice versa. Channels link components together forming an event flow graph, which is called an Event Processing Network (EPN).

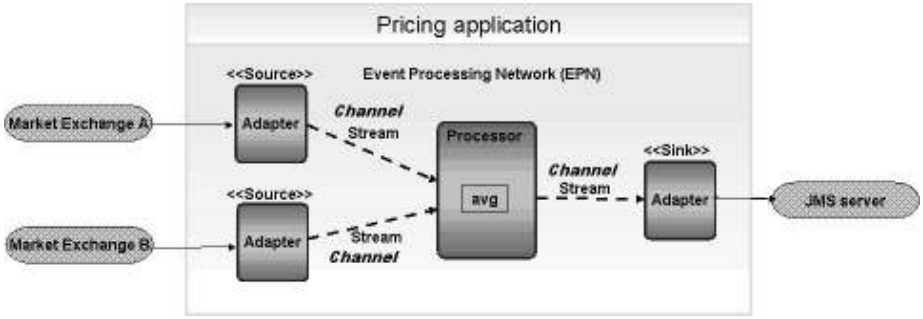

Figure 5. Example data flow through an Oracle Complex Event Processing pricing application.

Concurrency

The EPN allows the specification of different concurrency models. The paths initiated at the roots of the EPN, that is the event sources, are executed concurrent to each other. Adjacent nodes in the EPN establish an ordering dependency, so in the context of an event, two adjacent nodes process the event sequentially. Conversely, nodes that are not adjacent, for example, two nodes that are the fan-out of a common parent, may execute the same event concurrently, as determined by the runtime infrastructure.

Thus the application developer has the means to model the application logic and restrictions in the EPN. For example, if an application has to first cleanse an event and then calculate a property value, it should model these two processing functions into adjacent processor nodes, as illustrated in Figure 6.

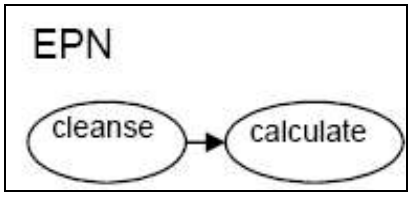

Figure 6. Sequential processing of events.

However, if there are several different ways of calculating the property value and these calculations can happen concurrently, then each calculation should be placed into parallel nodes in the EPN, as illustrated in Figure 7.

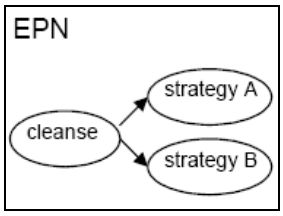

Figure 7. Concurrent processing of events.

Another concurrency design pattern is to partition events into separate categories that can be processed concurrently and place these as separate roots at the EPN. For example, if events from different clients can be processed concurrently, one can create a separate event source for each client, or for a set of clients.

Prioritization

Oracle Complex Event Processing is implemented using several real-time design patterns and practices, such as avoiding unbounded data structures, including linked lists.

Oracle Complex Event Processing is designed and implemented with the goal of providing a deterministic execution environment for event-driven applications. Consider the pricing application discussed previously. It is important that the application calculate the price of the stock symbol within the bounds of a guaranteed worst-case time, otherwise, considering the fast pace of the stock market, the price could be stale and not usable. In particular, the guaranteed worst-case time must hold even when the load on the system is high, which generally happens during peak stock trading hours.

To accomplish this, Oracle Complex Event Processing is implemented using several real-time design patterns and practices, such as avoiding unbounded data structures, including linked lists. For example, a user may specify the maximum number of events that a stream component in the EPN is able to hold at a given time. The user may also specify different discard policies to handle situations in which the maximum is reached. By assigning a maximum size to the streams in the EPN, the user can model the worst-case behavior of an application. The user specifies which paths in the EPN have more resources to process events and thus have higher priority, particularly when the load in the system is high.

Application Composition

A user authors an Oracle Complex Event Processing application by wiring EPN components together. The components perform distinct functions, collaborating to fulfill the goal of the application. In this way, a user can decompose the application logic into separate stages to better manage application complexity and optimize component re-use. Processor components can be implemented using either Java or CQL, which allows the user to seamlessly integrate a domain-specific language with a general purpose programming language.

Component implementations are hosted by an event-driven container that is implemented as an extension of a general purpose container [16] supporting

dependency injection, Plain-Old-Java-Object (POJO) components, aspect-oriented programming (AOP), and declarative XML configuration. This gives application developers access to Oracle Complex Event Processing extensions as well as the full power of the underlying container. An EPN is specified declaratively using a Oracle Event Processing-specific extension of the generic container's XML configuration file format. The generic XML configuration has been extended to support Oracle Complex Event Processing's programming model, in which users create the EPN graph formed of event sources, event sinks, streams, and processors.

Oracle Complex Event Processing provides a pre-packaged processor implementation that supports CQL. The CQL processor implementation delegates to the configuration module to handle the persistence of CQL queries, and supports dynamic changes to its configuration, such as the addition and removal of CQL queries. A user may also write a custom POJO component to act in the role of a processor. In another words, the POJO is an intermediate node in the EPN graph and performs event processing functions. Generally, event processing functions performed by POJO components are geared towards functions such as event passing, event routing, and event mediation, or tasks that are simply better suited to being written in Java. CQL processors handle complex query processing, such as aggregation, and pattern matching.

Oracle Complex Event Processing is a domain-specific application server; that is, an application server targeted for the development and execution of event-driven applications. Due to its modular and service oriented architecture, it is highly customizable.

Lastly, Oracle Complex Event Processing is a domain-specific application server; that is, an application server targeted for the development and execution of event-driven applications. Due to its modular and service oriented architecture, it is highly customizable. The extensible architecture of the programming model allows for other processor implementations that support additional query languages to be plugged into the server.

CONTINUOUS QUERY LANGUAGE

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 complete EDA Platform 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 include 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 use-defined implementations of algorithms. Oracle CEP provides an extensibility framework through which users can registers 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 userdefined functions.

Patterns

Oracle CEP also has the capability to detect regular expression patterns in realtime. 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>)

CQL extends SQL with a number of features needed to query streams of events. CQL was inspired by many of the ideas that have come out of the research and industrial CEP Standards focused communities.

EDA PLATFORM CONFIGURATION

Like most commercial application servers, Oracle Complex Event Processing supports dynamic configuration changes for server-level configuration data, such as connection pool and logging configuration. In addition, it goes beyond traditional configuration and component models [6] by allowing dynamic configuration changes to be made to individual application components. The ability to dynamically change the behavior of application components through configuration updates gives administrators and end-users of Oracle Complex Event Processing applications increased flexibility and control over applications. It allows applications to be modified in predefined ways without the need to take the application off-line or requiring that a new version of the application be deployed.

The ability to dynamically change the behavior of application components through configuration updates gives administrators and end-users of Oracle Complex Event Processing applications increased flexibility and control over applications.

Configuration changes can span multiple components and even application, they are atomic in that either all components involved in a configuration change will be updated, or none of them will. A standard two-phase algorithm is used notify components of a dynamic configuration change. Figure 8 shows how dynamic configuration changes are implemented within Oracle Complex Event Processing.

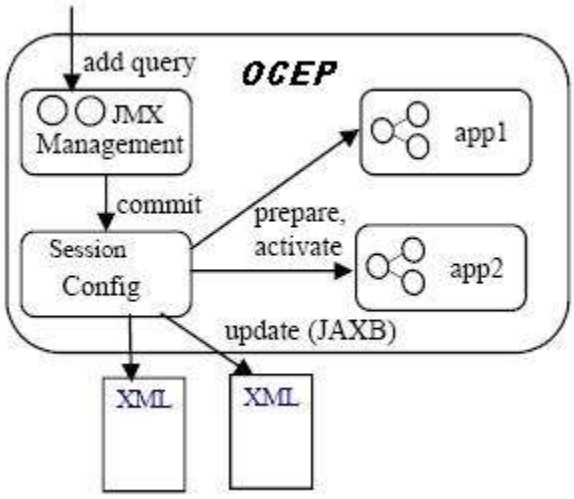

Figure 8. Dynamic configuration change notification

In this example, two separate applications have been deployed to the Oracle Complex Event Processing instance. Each application is packaged as a separate OSGi module. The configuration change being made is the addition of a new CQL query to a processor component in each application. The configuration update begins in the management module. A remote JMX client invokes the "add query" operation on the management components for the two processors. Internally, the management module begins a session with the configuration subsystem, uses the session to retrieve and update Java objects that represent the processors' configuration data, and then commits the session.

Once the configuration session is committed, the configuration module invokes prepare and activate operations on the affected application components, passing each component the Java representation of its updated configuration data. Components may reject a change during the prepare operation which will cause a rollback operation to be invoked on each component instead of these subsequent activate operation. Once all components have accepted the configuration change, the configuration module persists the updated configuration data to XML documents stored in the file system. JAXB is used to map between the Java and XML representations.

Oracle Complex Event Processing's configuration system is extensible. Application developers can define custom configuration data for the components that they write. Configuration data for custom components can be updated dynamically,

using the same techniques as those employed for built-in components such as processors and streams.

APPLICATIONS OF CEP AND CQL

CEP and CQL is technology 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.

Emergency Response Resource Proximity/Location Tracking

In terms of Event-driven Architecture solutions nothing can be more mission critical, with a need to be highly available, than when the handling the response to multiple fire emergencies.

When fire events are triggered, many simultaneous activities are initiated, such as to immediately isolate the fire location and define proximity/exclusion zones around the incident. In parallel, the identification of available fire resources, that are best equipped and in nearest vicinity is vital. As these events are occurring, the command center must be able to monitor the situation in Real Time with relevant resource movements highlighted on a Dashboard indicating, second by second, the location and zone proximity status of each resource (see figure 9), with the solution pro-actively and dynamically responding, averting dangerous zone transgressions.

Key factors to consider with this type of solution is the dynamic definition and management of "exclusion" zones which are areas where resources *"should not"* be, and are defined as complex shapes (polygons) which can move instantly, due to weather conditions or the fire spreading. Each identified moving emergency resource is monitored constantly using real time streaming GPS events providing continuous location updates, with the command center and command personnel being informed when Resources are "near" or "in" a Zone and, perhaps more importantly, when Resources have not left a Zone within a defined time period.

Figure 9. Lightweight Monitoring Dashboard showing resource movement/Zones

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 of 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 (volumeweighted 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 = Sigma(ViCi)/Sigma(Vi) where Ci is the cost of stock and Vi 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 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 realtime 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 Complex Event Processing is a next-generation application server designed specifically for event processing applications that require high throughput and deterministic latency. Oracle Complex Event Processing leverages an innovative server design to create compelling advantages, including:

- a modular, service-oriented architecture. The modular design helps to keep the server lightweight since modules are pluggable.
- a hybrid application programming model that supports both the Java programming language and the CQL query language. The programming model allows application developers to seamlessly mix-and-match languages. Application developers can take advantage of their existing Java skill set while also leveraging the power of a stream query language.
- dynamic configuration changes for application components which makes applications more flexible and manageable by business and end users.
- deterministic performance by addressing determinism throughout the software stack. This includes running on a Java virtual machine that provides deterministic garbage collection, as well as employing specialized algorithms and design principles within the application server itself.

Consequently, Oracle CEP and CQL are new capabilities offered in Oracle Fusion Middleware to support the unique needs of modern, event-driven architectures and applications that require long-running queries over continuous unbounded sets of data. Oracle CEP has been built to support event data streams that are changing constantly, often exclusively through insertions of new elements. Oracle CEP is an integrated platform provided with both the EDA and SOA suites, and can be introduced, 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 reliable deliver event messages with configurable qualities-of-service; (iv) a multi-protocol Enterprise Service Bus (ESB) to connect applications and route messages; and (v) a BPEL Process Manager (BPEL PM) for orchestrating business processes.

FOR MORE INFORMATION

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

APPENDIX 1: REFERENCES

[3] Oracle JRockit RealTime product documentation, Oracle Corporation, 2007.

[6] DeMichiel, L. and Keith, M., Enterprise JavaBeans,Version 3.0, Sun Microsystems, May 2006.

[7] Dependency Injection, Wikipedia, http://www.wikipedia.org, 2008.

[9] Gosling, J., Joy, B., Steele, G., Bracha, G., "The Java Language Specification, 3rd Edition", Prentice Hall, June 2005.

[12] Kawaguchi, Kohsuke, "Java Architecture for XML Binding (JAXB) 2.0 ", Sun Microsystems, 2006.

[13] Lindholm, T. and Yellin, F., "Java Virtual Machine Specification, 2nd Edition", Prentice Hall, April 1999.

[15] OSGi Service Platform Core Specification, Release 4, Version 4.1, OSGi Alliance, April 2007.

[16] Spring Dynamic Modules for OSGi Service Platforms product documentation, SpringSource, January 2008.

[17] Peterson, James Lyle, "Petri Net Theory and the Modeling of Systems", Prentice Hall, 1981.

[19] Tharakan, George, "Java Message Service (JMS) API", Sun Microsystems, 2003.

Oracle Complex Event Processing: Lightweight Modular Application Event Processing in the Real World
Updated June 2009
Authors: Oracle Product Management and Development Teams

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