Oracle Complex Event Processing Exalogic Performance Study

An Oracle White Paper
September 2011

ORACLE

Oracle Complex Event Processing
Exalogic Performance Study

Oracle Complex Event Processing
Exalogic Performance Study

INTRODUCTION

Oracle Complex Event Processing (Oracle CEP) provides a modular platform for building applications based on an event-driven architecture. At the heart of the Oracle CEP platform is the Continuous Query Language (CQL) which allows applications to filter, query, and perform pattern matching operations on streams of data using a declarative, SQL-like language. Developers use CQL in conjunction with a lightweight Java programming model to write applications. Other platform modules include a feature-rich IDE, management console, clustering, distributed caching, event repository, and monitoring, to name a few.

As event-driven architecture and complex event processing have become prominent features of the enterprise computing landscape, more and more enterprises have begun to build mission-critical applications using CEP technology. Today, mission-critical CEP applications can be found in many different industries. For example, CEP technology is being used in the power industry to make utilities more efficient by allowing them to react instantaneously to changes in demand for electricity, in the credit card industry to detect potentially fraudulent transactions as they occur in real time, and in capital markets for applications like order routing and algorithmic trading. The list of mission-critical CEP applications continues to grow.

Oracle Exalogic Elastic Cloud is a complete hardware and software platform for enterprise applications consisting of best-of-breed compute, storage, network, operating system, and software products that are engineered, integrated, tested, and optimized together. Exalogic is designed to provide extreme reliability, scalability, and performance while retaining the benefits of an open, standards-based platform that supports thousands of existing applications.

The benchmark study described in this whitepaper demonstrates Oracle CEP's ability to provide both very high throughput and low latency on a single Exalogic compute node with a use case that is very typical of financial front office applications in capital markets. The benchmark application implements a Signal Generation scenario in which the application is monitoring multiple incoming streams of market data, watching for the occurrence of certain conditions that will then trigger some action.

For this application, Oracle CEP was able to sustain an event injection rate of up to one million events per second on a single Exalogic compute node, while maintaining low average and peak latencies. At an injection rate of one million events per second the average event latency for the full processing path on the server was 32 microseconds with 99.99 percent of the events processed in less than 2 milliseconds.

The remainder of this paper includes a discussion of the product features that enable this level of performance, and a detailed description of the benchmark and its results.

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 and high performance runtime.

Server Architecture

Figure 1 shows the high level software architecture of an Oracle CEP server instance. At the lowest level is the Java Virtual Machine (JVM). The JVM used for this benchmark was the JRockit Real Time JVM, which provides deterministic garbage collection (small and bounded GC pause times) and is designed for applications requiring the lowest latencies and highest level of determinism.

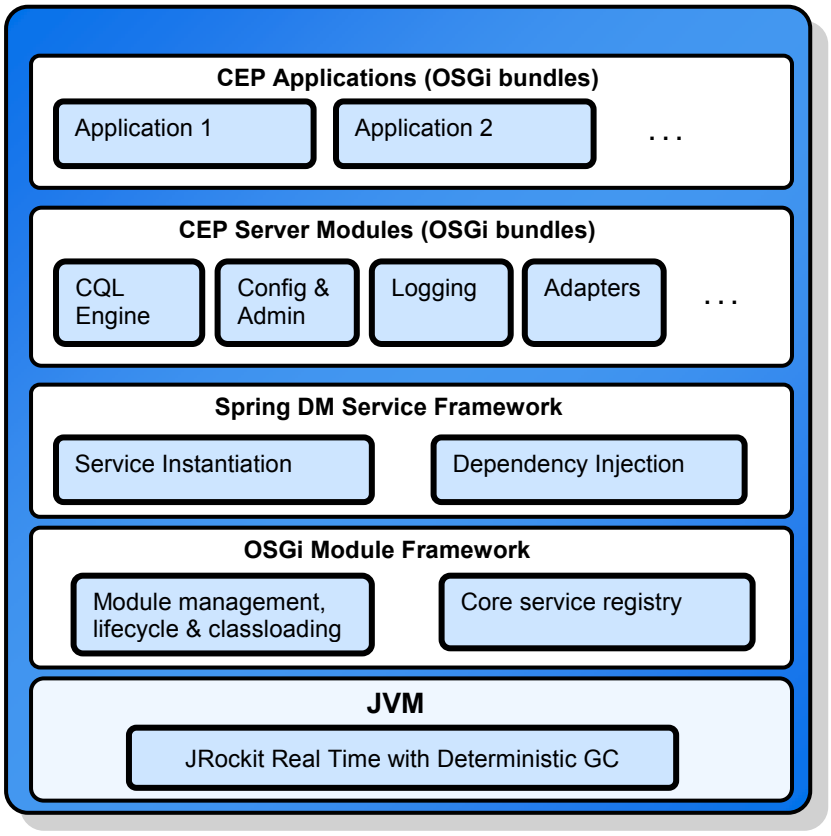

Figure 1 – Oracle CEP Software Stack

The next layer is composed of the modularity framework (OSGi framework), which controls the visibility of Java packages between modules and handles classloading and versioning of classes. The service framework is based on Spring Dynamic Modules and is responsible for instantiating services and resolving dependencies. The service framework uses dependency injection to provide service instances with configuration data and references to other required services.

The server modules layer is implemented as a collection of OSGi-based modules ("bundles" in the OSGi parlance) that provide the core CEP server functionality. Examples of modules implemented at this layer include the CQL engine (which executes queries written in the CQL language against streaming data), server configuration and administration, security, logging, and a core set of adapters for handling transport and marshalling/unmarshalling of input and output data. Finally there is the application layer consisting of CEP applications. The applications themselves are packaged as OSGi-based modules. The architecture within the server modules and application layers is uniform in that server and application modules are packaged identically and they are treated uniformly by the lower levels of the architecture (OSGi and Spring DM frameworks).

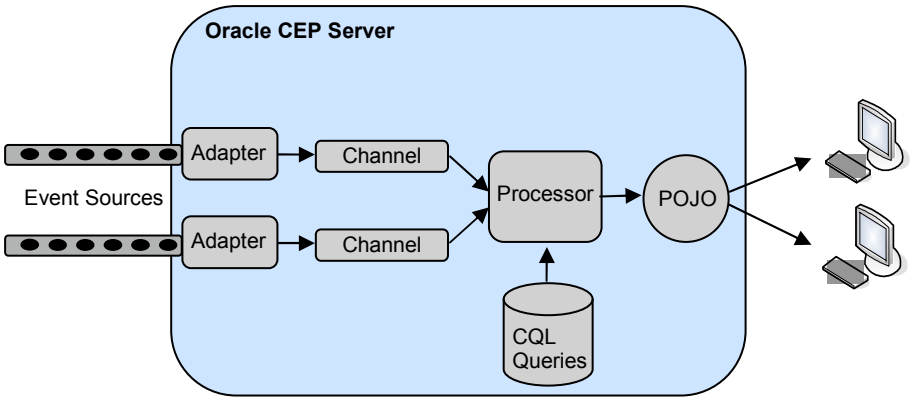

Figure 2 – Typical Oracle CEP Server Data Flow

Figure 2 illustrates the typical data flow through an Oracle CEP application. On the inbound (left) side are event data streams from one or more event sources. The incoming data is received, unmarshalled, and converted into an internal event representation within an adapter module. As the adapter creates event objects it sends them downstream to any components that are registered to listen on the adapter. In Figure 2, the listening components are "channel" components. A channel component is essentially a queue with an associated thread pool that allows the upstream and downstream components to operate asynchronously from each other. Channels can be very useful in increasing concurrency for applications that might otherwise have limited concurrency (for example, a data feed coming in over a single connection). The next component in the Figure 2 data flow is the processor component. A processor represents

an instance of the Complex Event Processing engine and hosts a set of queries written in Continuous Query Language (CQL). CQL queries support filtering, aggregation, pattern matching, and joining of event streams and other data sources. The output of the configured CQL queries is sent to any downstream listeners. In this example a POJO is configured to listen to the processor output. The POJO may perform additional processing on the events output from the queries and may trigger actions or send the output data to external systems via standard or proprietary messaging protocols.

The collection of interconnected adapter, channel, processor, and POJO components is collectively referred to as the Event Processing Network (EPN). Although the example in Figure 2 shows a common topology, arbitrary EPN graphs may be wired together consisting of any number of components of each type in any order.

Real Time Event Processing Infrastructure

Meeting the stringent latency and throughput requirements of typical event driven applications requires specialized support in the areas of thread scheduling, synchronization, and I/O. Specific performance features implemented by Oracle CEP include:

- Thread scheduling that attempts to minimize blocking and context switching in the latency critical path. Whenever possible, a given event will be carried through its full execution path on the same thread with no context switch. This approach is optimal for latency and also ensures in-order processing of events for applications that require this. However, in some cases handoff of an event between threads may be desirable. For example, an application may wish to handle data from a single incoming network connection concurrently in multiple threads. The runtime provides flexible thread pooling and handoff mechanisms that allow concurrency to be introduced wherever it is desired in the processing path with minimal impact on overall latency.
- Synchronization strategies that minimize lock contention which could otherwise be a major contributor to latency when running with highly parallel workloads. Significant enhancements have been made in the parallelism support in the CEP 11.1.1.6 release, including the ability to specify UNORDERED or PARTITON_ORDERED constraints on CQL queries, making it easier for applications to configure the CQL engine for the maximum degree of parallelism while meeting application requirements for ordered event processing.
- Careful management of memory including object reuse and optimized management of retain windows within the CQL query processing engine. The memory optimizations benefit latency by reducing both the allocation rate and the degree of heap fragmentation, both of which help the garbage collector achieve minimal pause times.
- A pluggable adapter framework that allows high performance adapters to be created for a variety of network protocols, with support for multiple threading and I/O handler dispatch policies.
- On the Exalogic platform, the ability of CEP to scale to both large numbers of threads combined with its scalable clustering support ensure that it can scale to take advantage of the massive compute capacity offered by Exalogic. CEP also benefits directly from

Exalogic's integrated Infiniband support, particularly for cluster configurations and in the common cases where CEP applications are tightly integrated with Oracle Coherence and WebLogic Server.

- Use of the JRockit Real Time JVM described in the following section.

JRockit Real Time JVM

While Oracle CEP is certified on both HotSpot and JRockit JVMs, JRockit Real Time was used for this benchmark in order to minimize latencies and ensure the highest level of determinism. JRockit Real Time consists of the JRockit JVM with enhancements for low latency and deterministic garbage collection. Typical throughput-oriented garbage collection algorithms stop all of the threads of an application when they perform a collection. The resulting "GC pause time" can be very long (several seconds or longer) in some environments and is a major contributor to latency spikes and jitter. JRockit Real Time's deterministic garbage collector uses a different approach designed to make GC pause times both shorter and more predictable. The deterministic collector handles much of the collection while the application is running and pauses only briefly during critical phases of the GC. In addition the JRockit Real Time collector will monitor the duration of individual pauses to ensure that the amount of time spent in a given GC pause doesn't exceed a user specified pause target. For example with a user specified pause target of 10 milliseconds, the deterministic collector will limit the duration of individual GC pauses to no more that 10 milliseconds providing a high degree of predictability compared to traditional GC algorithms.

BENCHMARK APPLICATION

The application used for this benchmark study implements a Signal Generation scenario in which the application is monitoring multiple incoming streams of market data watching for the occurrence of certain conditions that will then trigger some action. This is a very common scenario in front office trading environments. Figure 3 shows the overall structure of the benchmark including the components of the Event Processing Network (EPN).

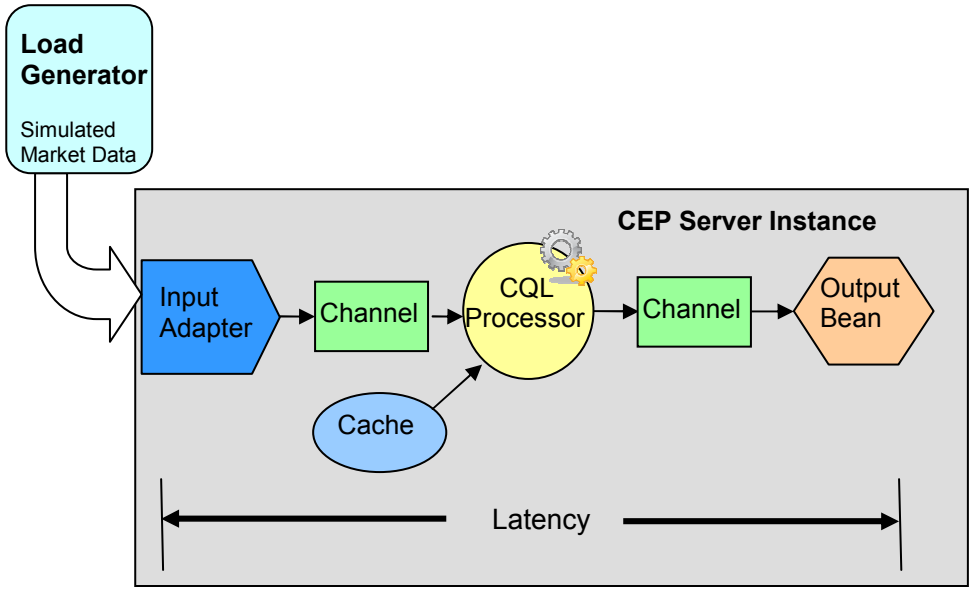

Figure 3 – Benchmark Application Structure and EPN

The incoming data is generated by a load generator which creates simulated stock market data and sends it to the server over one or more TCP connections at a configured, metered rate. The format of the data on the wire is specific to the implementation of the load generator and adapter and is designed for compactness. Within the event server the adapter reads the incoming data from the socket, unmarshalls it, creates an event instance (a Java object conforming to certain conventions) for each incoming stock tick, and forwards the events to the CQL processor via the processor's input channel. The second input to the CQL processor is a local cache instance (Oracle Coherence) which holds a "watch list" of stock symbols indicating the specific subset of input stocks that should be monitored by the CQL processor. In the benchmark configuration, there were a total of 1470 distinct stock symbols in the load generator input data and 300 symbols in the cached watch list. The CQL processor is configured with the queries shown in Figure 4.

```
<view id="S" schema="symbol lastPrice"
      ordering-constraint="PARTITION_ORDERED"
      partition-expression="T.symbol">
  <![CDATA[
    RStream(select T.symbol, T.lastPrice
    from StockTickStream[now] as T, SymbolsRelation as R
    where T.symbol = R.symbol)
  ]]>
</view>



<query id="perc"
      ordering-constraint="PARTITION_ORDERED"
      partition-expression="symbol">
  <![CDATA[
    select symbol, lastPrice, percLastPrice
    from S MATCH_RECOGNIZE (
      PARTITION BY symbol
      MEASURES
        B.symbol as symbol,
        B.lastPrice as lastPrice,
        100*(B.lastPrice - A.lastPrice)/A.lastPrice
            as percLastPrice
      ALL MATCHES
      PATTERN (A B)
      DEFINE
        B AS (100*(B.lastPrice
              - A.lastPrice)/A.lastPrice > 2.0
         or   100*(B.lastPrice
              - A.lastPrice)/A.lastPrice < -2.0)
      ) as T
  ]]>
</query>
```

Figure 4 – CQL Queries

There are two main stages to the processing done by the CQL engine. In the initial stage, implemented in CQL by the view with id "S", the incoming stream of market data (StockTickStream) is joined against the symbol watch list pulled from the cache (SymbolsRelation), and an output event is generated for each input event that matches a symbol on the watch list. The output from this initial filtering stage is fed into a subsequent pattern matching stage implemented by the query with id "perc". The pattern matching query produces an output event whenever it detects that the price of a given stock has increased or decreased by more than 2 percent from its immediately previous price. The output from the pattern match query is sent to any downstream listeners in the EPN, which in this case consists of an output bean (Java POJO) which computes aggregate statistics and latency data for the benchmark based on the output events it receives.

Note that both the view and the subsequent query specify an ordering constraint of PARTITION_ORDERED. This instructs the CQL engine that the arrival order of the events

(as indicated by their timestamps) must be maintained within a partition, where a partition in this case consists of a series of events with the same symbol. With this ordering constraint, the engine will allow parallel (unordered) execution of events from different partitions. For the common case where there are a large number of distinct partition keys (symbols in this case), this results in a high degree of parallelism within the engine which is evident in the results presented later.

Latency data for the benchmark is computed based on timestamps taken in the adapter and output bean, as depicted in Figure 3. The adapter takes the initial timestamp after reading the data from the socket and prior to unmarshalling. This initial timestamp is inserted into each event created by the adapter, is passed through the event processor and inserted into any output events generated by the pattern matching query. When the output bean receives an output event it takes an end timestamp and subtracts the timestamp generated by the adapter to compute the processing latency for that event. These latencies are then aggregated to produce overall latency data for the duration of the benchmark run.

BENCHMARK CONFIGURATION AND METHODOLOGY

Load Injection

The load generator can be configured to specify the number of connections it should open to the CEP server and the rate at which it should send data over each connection. We will refer to the aggregate send rate across all connections as the aggregate *injection rate*. For this benchmark the data sent by the load generator for each event consists of a stock symbol, simulated price, and timestamp data. The average size of the data on the wire is 20 bytes per event not including TCP/IP header overhead. The stock symbols are generated by repeatedly cycling through a list of 1470 distinct stock symbols. If the load generator is configured to open multiple connections to the server, the symbol list is partitioned evenly across the set of connections. The price data is generated dynamically and the price for a given symbol is updated each time the symbol is sent.

CEP Server Configuration

The CEP server configuration for the benchmark consisted of a cluster of two CEP server instances (two JVMs) sharing a single Exalogic compute node. A single Exalogic compute node contains two Intel Xeon processors with six cores each at 2.93GHz (12 total cores). Note that a full Exalogic rack contains 30 compute nodes, so this benchmark consumed only 1/30th of the compute resources of a full Exalogic rack.

An Exalogic compute node features a NUMA memory architecture in which each of the two processors has an associated pool of "local" memory which can be accessed with lower latency compared to "remote" memory. The use of two CEP server instances allowed each instance to be bound to one of the two processors and its associated pool of local memory using the Linux numactl command. This is a best practice for optimum memory performance when running on a NUMA architecture.

The application was deployed to the cluster, resulting in an identical copy of the application, with EPN and queries as described above, running on each of the CEP server instances. The load generator was run on a separate machine, and partitioned the input data across the two CEP server instances based on symbol. The input adapter in the CEP application was configured to use a blocking thread-per-connection model for reading the incoming data and dispatching the events within the server – this results in one active processing thread in the server for each connection opened by the load generator.

Methodology

The benchmark data was collected as follows:

- An initial 15 minute warmup run was done with the load generator opening two connections to each CEP instance and sending data at a rate of 50,000 events per second per connection.
- The warmup was followed by a series of 10 runs scaling the number of connections per CEP instance from one to 10 with the load generator sending 50,000 events per second per connection in all cases. The maximum injection rate achieved was 1,000,000 events/second (500,000 events/second/instance). The duration of each run was 10 minutes.
- An additional series of 10 runs was done holding the number of connections fixed at 10 per CEP instance and scaling the injection rate per connection from 5,000 to 50,000 events. The maximum injection rate achieved was 1,000,000 events/second (500,000 events/second/instance). The duration of each run was 10 minutes.
- The injection rate, output event rate, average latency, and latency distributions were collected for all runs.

BENCHMARK RESULTS

Table 1 and Figures 5 and 6 show the results scaling from one to 10 connections per instance at 50,000 events per second per connection. The average and 99.99 percentile latencies are reported separately for the two CEP instances. As discussed earlier, the latency values are collected only for those events that are forwarded to the output bean as a result of a match, and represent the latency from an initial timestamp in the adapter (prior to unmarshalling and creation of the internal event object) and a timestamp when the event is received by the output bean.

As Table 1 shows, the output event rate was a fixed percentage (5.2 %) of the injection rate as the load increased. There was a slight increase in average latencies as the number of connections and overall injection rate increased. The 99.99 percentile latencies remained flat at 200 microseconds or less from 100,000 through 600,000 events/second and then increased slightly as the total injection rate approached 1,000,000 events/second. At the maximum benchmarked load of 1,000,000 events/second the average and 99.99% latencies are still quite low.

Conn. per Instance	Injection Rate per Instance (events/ sec)	Total Injection Rate (events/ sec)	Output Rate (events/ sec)	Average Latency (microsecs)		99.99% Latency (millisecs)		Overall Max Latency (ms)
				CEP 1	CEP 2	CEP 1	CEP 2	
1	50,000	100,000	5275	20.1	20.1	0.2	0.2	7.4
2	100,000	200,000	10,540	20.7	20.2	0.1	0.1	8.6
3	150,000	300,000	15,733	21.1	20.5	0.2	0.1	8.8
4	200,000	400,000	20,949	21.5	20.7	0.2	0.1	9.3
5	250,000	500,000	26,201	21.9	20.8	0.2	0.2	16.8
6	300,000	600,000	31,410	22.5	21.2	0.2	0.2	23.2
7	350,000	700,000	36,652	25.5	24.5	0.3	0.5	17.9
8	400,000	800,000	41,889	28.1	26.8	0.6	0.7	22.7
9	450,000	900,000	47,100	30.3	29.1	1.2	1.3	25.9
10	500,000	1,000,000	52,173	32.5	31.0	1.8	1.8	34.5

Table 1 – Scaling from 1 through 10 connections per instance at 50,000 events per second per connection

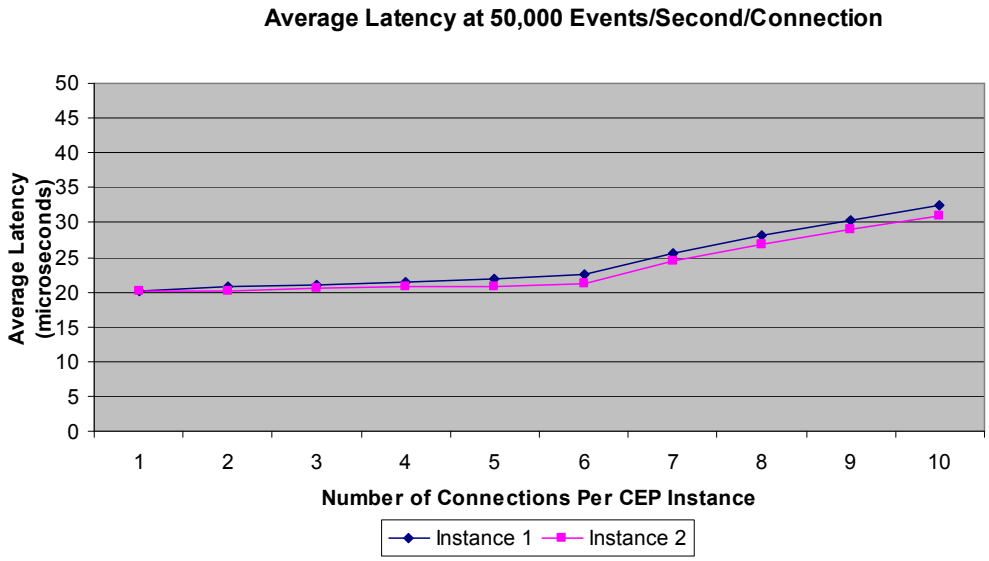

Figure 5 – Average latency scaling from 1 through 10 connections/instance (100,000 – 1,000,000 events/second)

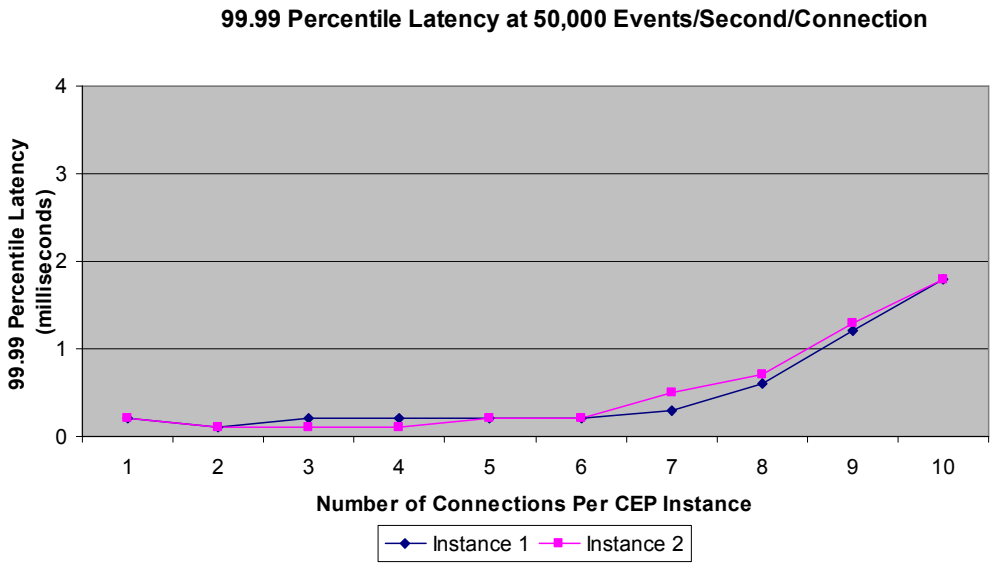

Figure 6 – 99.99 percentile latency scaling from 1 through 10 connections/instance (100,000 – 1,000,000 events/second)

Table 2 and Figures 7 and 8 show the results when holding the number of connections per instance fixed at 10 and scaling the injection rate per connection.

Rate. per Connect	Injection Rate per Instance (events/ sec)	Total Injection Rate (events/ sec)	Output Rate (events/ sec)	Average Latency (microsecs)		99.99% Latency (millisecs)		Overall Max Latency (ms)
				CEP 1	CEP 2	CEP 1	CEP 2	
5,000	50,000	100,000	5,239	27.9	30.7	0.3	0.2	8.1
10,000	100,000	200,000	10,479	29.6	31.1	0.3	0.2	7.5
15,000	150,000	300,000	15,716	29.2	29.3	0.3	0.2	17.1
20,000	200,000	400,000	20,945	29.4	29.2	0.3	0.4	11.8
25,000	250,000	500,000	26,180	29.7	28.9	0.6	0.6	19.4
30,000	300,000	600,000	31,426	30.1	29.0	0.9	0.9	23.8
35,000	350,000	700,000	36,663	30.9	29.4	1.3	1.4	23.5
40,000	400,000	800,000	41,893	32.0	30.0	1.4	1.6	20.3
45,000	450,000	900,000	47,133	33.0	31.5	1.6	1.8	22.8
50,000	500,000	1,000,000	52,173	32.5	31.0	1.8	1.8	34.5

Table 2 – Scaling injection rate with connections fixed at 10 per instance

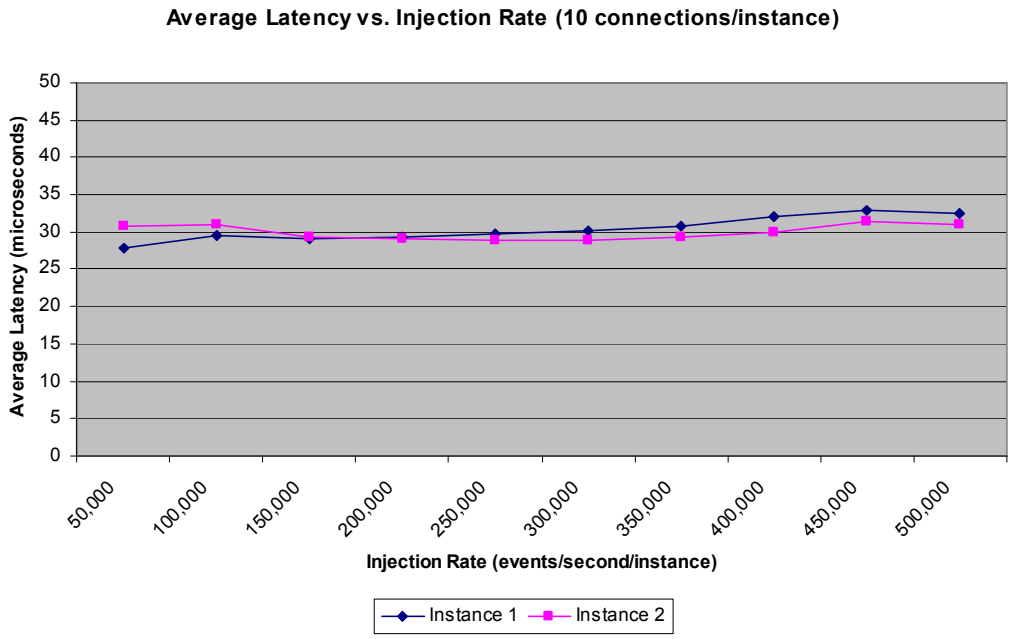

Figure 7 – Average latency with number of connections fixed at 10 per instance

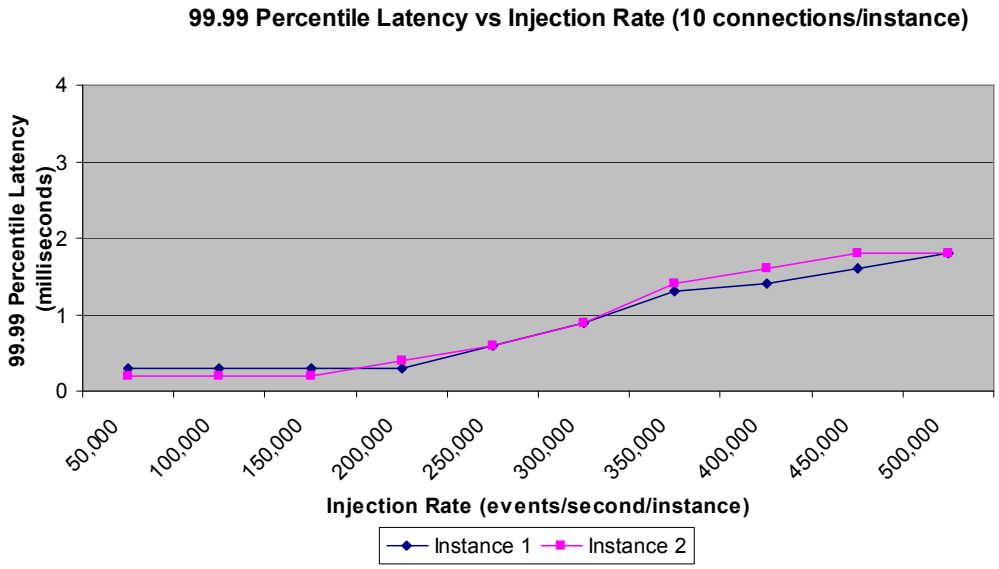

Figure 8 – 99.99 percentile latency with number of connections fixed at 10 per instance

With the number of connections fixed at 10 per CEP instance, the average latencies were fairly flat with increasing load. There was some modest increase in 99.99 percentile latency with increasing load when scaling with a fixed number of connections. When running at lower data rates, the latencies at a given load for this application were slightly better when using a smaller number of connections with a higher rate per connection (vs. using a larger number of connections to achieve the same rate).

CONCLUSION

This paper has reviewed the overall architecture of Oracle CEP, along with some of the specific features and design characteristics that allow it to provide high performance for event driven applications on the Exalogic platform. The performance characteristics of Oracle CEP were studied using a very common event processing use case. For this application a single Exalogic node (1/30th of a full Exalogic rack) achieved CEP throughput of 1 million events per second while maintaining very low and predictable latencies. The results demonstrate that Oracle CEP on Exalogic is capable of delivering extreme throughput and latency performance to satisfy the most demanding event processing requirements in the enterprise.

Oracle Complex Event Processing Exalogic Performance Study
September 2011

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