## TABLE OF CONTENTS

Executive Summary 4

**CEP - Introduction** 4

CEP at Work 4

CEP for Capital Markets 5

Related Architecture 6

**CEP Design Considerations** 6

Pattern-Matching Techniques 7

In-Memory Caches 9

Volume and Latency 9

Scalability 10

Availability 11

**Developer Considerations for Building a CEP Solution** 12

Ease of Development – Tools and Languages 13

Ease of Partitioning 13

Calculating Aggregates 14

Writing Input and Output Adapters 15

Inspecting, Debugging and Testing 16

Deployment Considerations 16

Analyzing for Performance 17

Conclusion 18

Glossary 18

References 19
EXECUTIVE SUMMARY

In capital markets, business operators need to be able to identify and respond to threats and opportunities with tremendous speed and precision. Real-time data streams, such as market feeds, are on the rise—increasing the need for complex event processing to support such rapid analysis and identification.

According to Forrester\(^1\), complex event processing, or CEP, “provides a software infrastructure that can detect patterns of events by filtering, correlating, contextualizing and analyzing data captured from disparate live data sources to respond as defined using the platform’s development tools.” In other words, CEP combines data from multiple sources to infer events or patterns that suggest more complicated circumstances. The ultimate goal: to identify meaningful events, such as opportunities or threats, and respond to them as quickly as possible.

Though the use of CEP was initially limited to algorithmic trading, capital markets are now applying CEP to many more use cases—including smart order routing, insider trading and compliance monitoring. This paper explains CEP as a concept, discusses applications and scenarios relevant to capital markets, and highlights key considerations for architects and developers as they design and build CEP applications.

INTRODUCTION

CEP AT WORK

CEP is used along with other similar technologies, such as business activity monitoring, business process management, operational intelligence and message-oriented middleware. While CEP shares some features with these other event processing technologies, CEP is distinguished by its agility. Used alone or as part of the enterprise architecture, CEP is capable of operating on live data arising from diverse sources producing homogeneous or heterogeneous events.

In processing millions of events coming into the event-processing pipeline, CEP uses several event correlation techniques, such as temporal correlation (based on event time) and logical or causal correlation (based on certain event attributes), as well as spatial- and dimension-based correlation techniques. Events satisfying the correlation criteria are called event windows. A pattern-matching algorithm is then applied over these windows to determine a specific business situation. Thus, capital markets can use CEP to:

- Detect and inform, with intelligent information filters to identify:
  - **Rogue trading.** Rogue trading occurs when one or more traders deliberately try to subvert or manipulate a market for an unfair gain. CEP is being deployed in trading rooms and exchanges and by regulators. These solutions monitor trading activity in real time and can immediately detect instances of market abuse.
  - **Insider trading.** Analyzing market data in fractions of a second, CEP helps financial firms accurately monitor trades and adapt in real time. That, in turn, makes it easier for firms to generate alerts on suspicious transactions and behaviors as they occur.

\(^1\) The Forrester Wave™: Complex Event Processing (CEP) Platforms, Q3 2009
• Detect and act, with the ability to recognize patterns of events and respond to:
  › **Algorithmic trading.** Strategies, such as index arbitrages, market making, data mining and pattern recognition are used for algorithmic trading. CEP is able to meet the need for real-time pattern recognition and computation over multiple streams.
  › **High-frequency trading.** In a high-frequency trading strategy, high-performance computers transact a large number of orders at very fast speeds with the goal of earning profit through alpha seeking or market making. This kind of trading is very sensitive to the latency across markets; thus, CEP applications’ low-latency characteristics are well suited to high-frequency trading.
  › **Smart order routing (SOR).** Smart order routing systems automate the selection of execution venue and methodology in order to assure best execution, systematize the selection process and reduce execution costs. SOR systems require sophisticated, complex event processing detection mechanisms, such as liquidity detection and optimization, in real time.
  › **Hedge strategy executions.** Portfolio managers submit hedge strategies to trading systems with the objective of having a neutral strategy execution—that is, both legs of the hedge trades should be in a balance based on the chosen balancing algorithm, such as Cheapest to Deliver (CTD) neutral or market neutral. CEP plays a role by detecting a strategy execution on one leg, determining quantities for the deficient legs of the strategy, and placing the orders in the market to maintain neutrality in execution.

• Enrichment scenarios, to enrich the data from incoming events in:
  › Post-trade compliance reporting and market/trade event data enrichment are situations that require aggregation of data from multiple sources. CEP is also used for cleansing, continuously evaluating the source and computing additional fields on the fly.
  › Real-time risk evaluation requires the ability to analyze numerous and disparate streams of live data in near-real time. This practice is taking center stage in banks’ and brokers’ risk management strategies. CEP is also gaining traction with stock exchanges, where some regulators should be monitoring potential market abuse. The technology enables financial services firms to take a holistic view of counterparty, market and credit risk. Previously, number crunching to calculate a firm’s positions would have been done overnight.

**CEP FOR CAPITAL MARKETS**

Generally speaking, CEP is applied in the reactive realm of the business. A pattern of business events over a time period results in a business situation that could be critical to the organization. The situation could be a threat (such as fraud or insider trading) or an opportunity (such as a buy-versus-sell decision).

CEP platforms are used to analyze these events as they arrive in very high volumes from disparate sources, such as market feeds or transactional systems. Over a time period, a pattern of interest is detected from these streams. Further workflows can then be triggered for responding to the complex event, which is a summarization of the pattern of events.

Because a complex event could indicate an opportunity or threat for the business, reaction time is critical. Thus, it is very important that the processing time is much lower for the response to be faster. Because CEP allows the business to make decisions in near-real time, it has found substantial use in many segments within capital markets.
CEP is used for event processing within an event-driven architecture. Unlike a simple event processing or event stream processing mechanism, CEP uses event-correlation and pattern-matching techniques to “infer” complex events. It augments business activity monitoring with inferences from source events treated as business events and sent for further action or analysis via dashboards. In business process management, CEP is used as an enrichment system that reports business events as they occur. Threats and opportunities are reported to further workflows. In a service-oriented architecture, CEP systems can be used to determine the business trigger to launch further services in the processing pipeline.

**DESIGN CONSIDERATIONS**

In building CEP applications, designers can employ a set of common building blocks. A typical CEP architecture would be as follows:

The event sources are responsible for producing or sending the events. The sources could be pushing events (such as tickers); in some event sources, a pull (such as database or web services) can be employed.

Within the CEP engine, some pre-processing is generally required. It may include converting the source-specific event format to a format understandable by the CEP engine. Once pre-processing is complete, events are washed over pre-defined queries. A matching pattern is then interpreted as a complex event and forwarded to post-processing.

The post-processing could be a reverse of the pre-processing in which the CEP-specific format is undone and a sink- or target-specific format is created. The event could then be made available to the target as a push or a pull.

Given the architecture above—and regardless of the CEP platform—there are some common patterns across CEP applications. The following sections will provide a brief overview on such patterns.
**PATTERN-MATCHING TECHNIQUES**

Pattern matching is an integral element of complex event processing. Pattern matching lets a business situation be inferred or identified. It involves combining several methods, such as grouping and correlating, as well as filtering and aggregation to identify a specific pattern to events within or across streams.

**Correlations**

The first step of applying a pattern is to group relevant events, forming a “window.” These events are correlated using a common set of techniques called window policies:

1. Temporal windows, also known as time windows, can be used to do a stateful event correlation based on the event occurrence. Based on the time, a “peephole” is created on the event stream, and the state of the previous events in the stream is used with the current event’s state to determine a pattern.
   
   *Example:* Stock value declined by 5 percent within one hour of buying the stock

2. Spatial correlation or dimension-based windows are similar to temporal windows. The difference is that the peephole focuses on number of events rather than time. This technique is also called count windows, as the count of events determines the window.
   
   *Example:* Three consecutive high stock prices in the stock ticker

3. Direct filters can be applied on the attributes of the event or on aggregated events.
   
   *Example:* Event.CurrencyPair == EURUSD

Temporal windows and dimension windows could be started and stopped based on set conditions. Events within windows, known as “tuples,” are evicted based on an eviction policy. When all the tuples are evicted from the existing window before a new window is created, the window is called a tumbling window. Sometimes only the oldest tuple or tuples get evicted so that the window condition is met; this type of window is called a sliding window.

To illustrate the tumbling window and sliding window, consider the following input stream:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 .. ..
```

In this stream, each box represents a new event. The order of the events is strictly increasing by time. Applying a dimension of 3 and specifying a tumbling window will result in the events being grouped as follows:

```
1 2 3 | 4 5 6 | 7 8 9 | 10 11 12 13 .. ..
```

Each alternate color indicates a new window. No events are repeated in the next window. Applying the same dimension of 3, if the window is specified as a sliding window, the events will be grouped as follows:

```
1 2 3
2 3 4
3 4 5
4 5 6
5 6 7
6 7 8
7 8 9
8 9 10
.. .. ..
```

Here, too, the alternate color indicates a new window. However, as the new event comes in, the old event is evicted to retain the number of events to the specified dimension of 3.
Direct filters based on predicate expressions over event attributes are another important way of partitioning events. Typically, high-volume, no-value or low-value events should be filtered out so that the residues of low-volume, high-value events are subjected to further processing. For example, if the interest is only on how the Euro fares against the USD, the filter would eliminate all other currency pairs. The remaining stream would contain only events with EURUSD as the currency pair.

Note that all of the above could happen on one stream or on multiple streams from multiple event sources.

**Computing Aggregates**

As events are correlated, aggregates should be calculated for further computation or filtering.

To calculate an aggregate, events should be grouped into a set. Grouping events can be achieved through window partitioning techniques as discussed above.

Aggregates can be calculated on any type of window—including temporal, dimension, tumbling or sliding windows.

For example, consider the following stream of events:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 ...
```

Assuming the number in the box indicates the event weight, to calculate an average weight for 3 events, the first step would be to partition the window based on the dimension (count = 3).

If the window is a tumbling window, the average for each window would be calculated as follows:

```
<table>
<thead>
<tr>
<th>Window</th>
<th>Avg0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
<td>2</td>
</tr>
<tr>
<td>4 5 6</td>
<td>5</td>
</tr>
<tr>
<td>7 8 9</td>
<td>8</td>
</tr>
</tbody>
</table>
```

If the window is specified as a sliding window, the aggregate value and the window would be as follows:

```
<table>
<thead>
<tr>
<th>Window</th>
<th>Avg0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
<td>2</td>
</tr>
<tr>
<td>2 3 4</td>
<td>3</td>
</tr>
<tr>
<td>3 4 5</td>
<td>4</td>
</tr>
<tr>
<td>4 5 6</td>
<td>5</td>
</tr>
<tr>
<td>5 6 7</td>
<td>6</td>
</tr>
<tr>
<td>6 7 8</td>
<td>7</td>
</tr>
<tr>
<td>7 8 9</td>
<td>8</td>
</tr>
</tbody>
</table>
```

Most CEP platforms have simple, built-in aggregate functions, such as `sum()`, `avg()`, `min()`, `max()`, `count()`.

Some platforms allow running statistical calculations—such as `stddev()`, `median()`, `variance()`—and support retrieving values from the window—such as `first()`, `last()`, `firstn()`, `lastn()`—for writing aggregate expressions on the window.

These platforms also offer extension points so that user-defined aggregate functions can be written in an external language, such as Java, C++ or C#, and called for calculating the aggregate.

Care should be exercised when writing a custom or a user-defined aggregate function; performance of the function could be a key factor. Proper impact analysis and testing should be carried out to check if this would constrain the event-processing pipeline.

As each event arrives, it participates in the aggregate expression. The state of the event stream is remembered in further computations—that is, the events within a window that participated in the previous computation will not be used on the next computations.

Thus, the aggregate values computed can be emitted when the window is closed or when the tuples are emitted. The emission of the aggregate value is periodic and calculated within the working set.
IN-MEMORY CACHES

Most CEP platforms provide some sort of in-memory caches as part of the runtime engine.

In-memory caches are used by the CEP engine primarily to store events and windows. Caches are also used for other purposes, such as sharing data between streams, maintaining state and storing external data (from relational databases and data from web services, for example) for ultra-fast processing. These data types are stored outside the windows and event streams but can participate in computation when required. It should be noted that data stored in this cache is not persistent unless specified otherwise; when a server is shut down, the cached data goes down with it.

CEP engines implicitly manage evictions while caching events and windows using policies such as Least Frequently Used (LFU), Least Recently Used (LRU), and First in First out (FIFO) when the maximum cache size is reached. Eviction of user-defined cache objects should be managed explicitly.

In case an application requires a persistent cache wherein data is not lost across the runs or when one node fails, CEP engines can be integrated to use disk-based cache or distributed cache that is out of process. Some platforms bundle these alternative caching mechanisms as a feature; others can be extended to use an off-the-shelf product. Depending on the platform, these alternative mechanisms can be used to store events received or reference data.

A persistent cache plays an important role in high availability. From a performance point of view, however, in-memory caches are much faster. If the reads are high and the invalidation happens rarely for a relatively small data volume, the distributed cache objects could be replicated in memory to form a near-cache. A tradeoff between high availability and speed would determine caching design.

VOLUME AND LATENCY

In their present state, CEP engines work well for high-velocity applications. Scenarios that involve a certain volume of events that need to be processed with a very low latency are strong candidates for CEP engines.

The high volume of event data (measured in terabytes) processing capability by CEP platforms is being questioned by the advent of big data platforms (that is, data volumes measured in petabytes and above). While the velocity and scale of incoming events and data would undergo minimal changes, the scale of data to which incoming events have to be related and computed would increase over time.

Some CEP platforms have already started adapting to the change. For example, Fujitsu has Big Data CEP Server software, a hybrid system that is interoperable with Hadoop. It should be noted that the interoperability is limited to sending the real-time processed or input event data to Hadoop for further parallel data processing. Meanwhile, Twitter’s Storm and Yahoo’s S4 are capable of using stream processing to handle big data volumes through distributed stream processing.

The overall velocity of the event-processing pipeline would be limited—not only by the CEP engine’s processing power or hardware, but also by the volume of data to which events must be related. Several CEP platforms offer profiling tools to identify the time and CPU-consuming operations in order to enhance the velocity. Once the cause is identified, other complementing techniques, such as caching and scaling, can be used to improve latency and increase event-processing velocity.
SCALABILITY

High volume and low latency are key characteristics of CEP scenarios. When data volumes increase, CEP engines should still have an acceptable throughput.

Almost all CEP engines support scaling up. These engines are designed to take advantage of increasing cores to have an extra degree of parallelism. Further, the in-memory caches also benefit while scaling up. No specific design considerations would be required for this purpose.

While scaling up can take care of most of the performance bottlenecks, more options would be required as it soon hits its limit. Also, the events might be distributed across the geography in certain scenarios. To attain greater parallelism, scale-out clusters have to be employed.

Unlike support for scaling up, CEP platforms offer limited support for scaling out. In fact, only a few platforms, such as Oracle CEP, Twitter’s Storm and Yahoo’s S4, provide native support for distributing the work across the nodes in the cluster and making them work in parallel. There are some references to Storm’s topology and the Esper engine being combined to offer the best of both worlds.

As a first step, the application should be designed to partition the event streams. The partitioned event stream would become a unit that can be processed independently by a node in the CEP architecture. The partitioning logic should be based on the event’s attribute (for example, a direct filter event-partitioning technique) so that a filter (that is, a causal correlation) can be applied to create the event-stream partition. Spatial and temporal correlation would then occur within this partition.

The following figure shows the Oracle CEP’s approach to scalability:
The event partitioner channel partitions the event partitions based on event properties. A hash key is generated and forwarded to the appropriate listener (processor) for further processing.

Where the platform does not support native horizontal scaling, multiple CEP engines can be used to create a load-balanced environment. Events from producers will be routed to a central processing engine. The central CEP engine applies a direct filter and creates a context-based partition. These partitions are then sent to the worker CEP engines deployed in a parallel topology, which is based on pre-set balancing algorithms determined by the segmented events.

Further correlation, pattern matching, aggregation and filtering happen in these worker engines. The results are then merged and forwarded to appropriate consumers.

These patterns ensure that for every event consumed, there is exactly 0 or 1 message produced. Also, they are stateless. Because there are multiple points of failure, additional care should be taken to devise a proper failover mechanism. In any case, if the event stream cannot be partitioned using a filter or if the correlation happens across the event stream partitions, scaling out is not possible.

**AVAILABILITY**

As with any software system, CEP systems are subject to software and hardware limitations. Since CEP systems are used in quick decision-making scenarios, they must be operational and highly available. Also, CEP systems are stateful. The inference process would involve temporal techniques or spatial correlation techniques; hence, it is important that no event gets lost.

To maintain the statefulness of the CEP systems, the platforms offer one or more of the following:

- **Active/Active** – replicate both the state and behavior of one node in another
- **Active/Passive** – replicate the state of one node in another
- **Checkpointing** – save the state for later replays

Each option has its own characteristics. Active/Active offers very short failover time and better performance. Active/Passive has lower resource utilization since one node is not doing any processing; thus, there is greater choice for sharing the resources with other applications. Checkpointing is relatively simple to implement.

Note that not all platforms support all of the above described options. For instance, StreamInsight supports only checkpointing; StreamBase supports Active/Passive; and Oracle CEP and Twitter’s Storm support Active/Active.

Generally, to design highly available CEP systems, start by preserving the state that is of significance rather than preserving everything. Doing so puts fewer loads on the resources regardless of the option given by the CEP platform.
The selected window size also plays a role in recovering from failure; the smaller the window, the faster the recovery time when a failure happens.

Maintaining or sharing state in or between nodes should be avoided. If the node where the state is residing fails, then the recovery will leave the state inconsistent. If the state has to be shared or maintained, then state synchronization should be turned on if the CEP platform supports it. If there is no platform support, share the state in a centralized cache or data store.

**DEVELOPER CONSIDERATIONS FOR BUILDING A CEP SOLUTION**

To understand the considerations while implementing a CEP solution, the following scenario was implemented using three different CEP platforms:

- StreamInsight
- Esper
- StreamBase

In the last 30 minutes of the trading day, any price movements accompanied by volumes higher than one-minute peak for the day generate a buy/sell call. Meanwhile, the system will monitor incoming ticks from a market data source for stock and derivative prices for all securities that are traded in a derivatives market. Breaking down the problem, the following steps would have to be carried out from incoming ticks:

- Calculate one-minute peak volume for the day for each symbol
- Calculate average price for each symbol
- Calculate price movements in the last 30 minutes
- Calculate volume movements in the last 30 minutes
- If traded volume is more than one-minute peak and the Ask/Bid prices are deviating from the average price, then induce a buy/sell signal

If the platform supports only checkpointing, adequate measures should be taken to store and replay the events that occur after the checkpoint but before or during the outage. Since replay of events could happen, there is a high probability of duplicate processing. Hence, the downstream systems should also be designed to be tolerant of these duplicate messages.

Some CEP engines, such as Esper and StreamInsight, offer embedded hosting mode wherein the CEP engine can be bundled within the application. The CEP engine’s availability is then limited to the hosting application’s availability.

First, the team had to decide on the market event data structure. To identify the peak volume, a one-minute sliding window (temporal window) was applied on the event stream. Aggregate operations were applied on the window to find the peak volume for each symbol.

To keep the average as current as possible, the average price for each symbol was calculated for the last 10 events.

To find the price and volume movements in the last 30 minutes, the team created a temporal window that will be active only for the last 30 minutes of the trading day. This will monitor every market event within that period and compare the price and volume against the peak volume and average price.

If deviations are found, then a buy or sell signal is triggered.

The following sections share a developer’s perspective on CEP platform considerations based on the implementation experience from this scenario.
**EASE OF DEVELOPMENT – TOOLS AND LANGUAGES**

Depending on the CEP platform, the availability of tools and support for languages may vary.

For instance, StreamBase has modeling support over Eclipse IDE. Adequate extension options are available for writing operators and aggregation functions for extending the event-processing model created. These extensions are done using Java.

StreamInsight uses Visual Studio IDE. Writing a CEP application in StreamInsight involves writing C# code. Enough support exists for writing user-defined aggregates and operators in C#.

Application development in Esper can be done using Eclipse IDE and involves writing Event Processing Language (EPL), an SQL-like construct. Any scripting language that supports JSR 223 and also the MVEL scripting language can be specified in EPL.

It can be generally assumed that .NET and/or Java platform knowledge can be leveraged for CEP application development if the chosen CEP product supports these platforms. Generic language skills would be much needed for writing input adapters, output adapters, event modeling, custom operators and user-defined functions and for creating a pipeline in these CEP products.

In a modeling environment such as StreamBase, most of the tasks can be achieved without knowing a programming language such as C# or Java. The processing pipeline is created visually, and IDE provides adequate support to parameterize or customize constructs within the pipeline. For writing custom operators and functions, Java language could be used with StreamBase to define the same.

For matching patterns, each CEP platform would have its own specifics. StreamInsight uses LINQ providers with which a pattern can be specified. Esper uses EQL to specify patterns. StreamBase uses a pattern-matching language for the same purpose, with most of the matching done within the event-processing constructs.

**EASE OF PARTITIONING**

To implement the scenario discussed above, multiple partitioning was applied to the incoming market data stream.

The first partitioning was done to create the peak volume per minute based on the temporal characteristics of the events. A sliding window of one-minute duration was created, with a window starting every event and ending a minute after the event arrived.

The second partitioning was done to calculate the average price based on the number of incoming events to apply an aggregate condition. A window is started on every event arrival and closed and emitted every 10 events.

All three platforms—Esper, StreamBase and StreamInsight—were able to support both types of partitions.

Esper has a strong set of partitioning options. The time (sliding) and time_batch (tumbling) windows create windows based on the temporal properties of the events, while the length window is created based on the dimensions. To support data partitions (direct filters), Esper allows for creating context partitions for segmenting incoming events. Partitions could also be created without such a context declaration using the standard EPL constructs, such as group by, join and partition. This would mean that direct filters can be mixed and matched with time and length windows.

StreamBase allows the event stream to be partitioned and aggregated using the Aggregate Operator. This construct provides tuple-based dimensions, time-based dimensions, event field-based dimensions and a predicate-based dimension. Sliding and tumbling windows are supported for tuple-based, time-based and field-based dimensions. Although the predicate expression can be used as a direct filter, it cannot be combined with other dimensions.
StreamInsight supports count window (based on number of events), tumbling window (based on temporal dimensions) and snapshot window (based on event properties). In its current state (v 2.1), StreamInsight offers no direct support for creating sliding windows. A snapshot window is combined with AlterEventDuration to provide an almost equivalent sliding window. This was a concern, as the one-minute peak volume may not be as accurate as when calculated in other platforms. Data partitioning the event stream can also be achieved by powerful LINQ constructs where filters, groupings and joins can be applied.

In essence, strong partitioning support will save a lot of time in working around those areas where the CEP platform may be lacking. It is important to choose the platform with the right set of partitioning support based on current and future requirements.

CALCULATING AGGREGATES

On the first window partition (one-minute sliding window), two aggregate functions were applied. The sum() aggregate function was applied to the events in the window to calculate the sum of volumes per minute. The resultant set was then subjected to a max() aggregate function to arrive at the one-minute peak volume per symbol.

To identify the average price for each symbol, the avg() aggregate function was applied on the spatial window of 10 market data events.

Each of the CEP platforms had varying degree of aggregate functions support:

- StreamInsight’s built-in aggregate functions are limited to average, summation, minimum, maximum and count. Hence, a custom user-defined aggregate (UDA) function was written in C# to check if the incoming event group had three consecutive highs or lows. StreamInsight also allows for writing a user-defined operator (UDO). The difference between the UDA and UDO is that UDA returns a single value [a Boolean, in this case] while a UDO could return one or more resultant events.

Esper also has a limited set of built-in, SQL-like aggregate functions, such as avg, sum, min, max, count, median, stdev and avedev. It also provides aggregate functions that operate on a data window, such as first, last, window, firstever, lastever and nth. To check if incoming events have consecutive highs/lows, a custom aggregate class [extending AggregationSupport class] containing the aggregate function was written in Java. Unlike StreamInsight—where the name of the class/method is used as the Aggregate Function name—the name of the UDA has to be configured in Esper. Esper also provides single-row functions, such as prev(), which can be used in the where clause to achieve the desired result. To compare three events, the prev() should be used at least three times. The statement would become tangled if more such events have to be compared; thus, it is advisable to stick to UDA functions for such cases.

- StreamBase, a Map operator was used to investigate the window as a list and generate the desired signal. Similar to Esper’s prev, StreamBase can convert the window events to a list and can be accessed through indexes (like an array) but would become cumbersome if the list of events grows. A custom Java aggregate function [extending AggregateWindow class] was written for better maintainability. Of the three platforms, StreamBase offers an extensive list of aggregate functions ranging from the simple avg, min, max, count and stdev to functions specific to financial markets (such as alpha, exp_moving_avg, covariance and vwap).

Aggregates play a vital role in inferring event data. While out-of-the-box support would be ideal for a specific operation, having necessary extensibility options is a must in the chosen CEP platform. It should also be noted that there could be more than one way of solving a specific problem. Adequate attention should be devoted to identifying the “most right” solution.
WRITING INPUT AND OUTPUT ADAPTERS
A classic adapter’s intent is to convert the interface of a class into another interface that clients expect. Doing so allows classes to work together that otherwise could not due to incompatible interfaces.

Adapters in the CEP world are no different—except that they have an additional responsibility. Depending on whether an adapter is an input or output adapter, the primary responsibilities of a CEP adapter are to:

1. Read from source or write to sink
   - Events can be delivered to the consumer from the provider using a push or a pull.
   - Adapters should know how to connect [access mechanism] with the source or sink.

2. Transform
   - An event could be transformed from source event type to CEP known type or from CEP known type to destination event type.

3. Enqueue or Dequeue
   - Once the transformation is done, the event will be either placed or removed in/from the CEP engine.

Typically, an input adapter would perform the Read, Transform and Enqueue while an output adapter would do a Dequeue, Transform and Write in the same order.

In the trade processing scenario described above, market feeds should be continuously monitored. The input adapter has to continuously read the incoming tuples and events, transform them, and feed them to the CEP engine so that events can be partitioned and aggregate results deduced. Similarly, the output adapter will wait on the decisions inferred from the CEP engine, transform them to target- or sink-specific type, and write to the sink.

Esper has a very basic set of pre-built adapters [CSV, Spring JMS, HTTP, Socket, JDBC and XML]. It also bundles the OpenTick input adapter specifically for capital market scenarios.

StreamInsight does not ship with any pre-built adapters because of its emphasis on ease of adapter creation using IEnumerable and IObservable. Some basic adapter sets that exist in the web include CSV, SQL Server, MSMQ and WCF.

Esper and StreamInsight come with an adapter kit that can be used to write specific adapters fitting the purposes.

StreamBase has adapters built for most of the common data and connectivity tasks. Immediately available adapters include CSV, SQL Server, Oracle, IBM MQ, TIBCO, e-mail and Hadoop. Apart from these general adapters, it also offers an exhaustive list of capital market–specific adapters for FX, FIX, market feeds from various sources, and financial news feeds. Its adapter kit supports .NET languages as well as Java.

It should be noted that depending on the architecture chosen, these adapters can be built embedded within the CEP platform or built externally. Some platforms support this inherently; in others, it is not difficult to model. Regardless of this design choice, the CEP engine has to work with a known type so that it can run the pattern-matching queries.

For rapid application development, off-the-shelf adapters would be very useful. If the number of input stream and output stream types are relatively low, writing an adapter should not be time consuming. However, complexity would grow as the number of incoming and outgoing event types increases.
INSPECTING, DEBUGGING AND TESTING

Given the complex correlation techniques and aggregates that could be present in a CEP application, it is essential to have sound inspecting and debugging capabilities within the CEP platform.

Once the trade-processing CEP application was built using the CEP platform (in this case, Esper, StreamInsight and StreamBase), the application had to be tested using an event stream.

To determine whether or not the event pipeline works at every construct, events must be fed in a controlled manner to enable inspection at every construct. StreamBase offers very rich, on-the-spot inspection and event-injecting capabilities. StreamInsight’s event-flow debugger can be used to debug and perform query analysis showing the state of each query as it runs. Though basic, Esper’s inspection support offers comprehensive runtime metrics and is available directly in a rich web-based GUI using EsperHQ package. All of the above had pause, step-through and breakpoint support through their own IDEs.

A basic step to fake an event stream is to write a random generator that generates events based on the event type and feed them to the input adapter, preferably in the same programming language as the adapter. Complications would slowly arise when more control is needed over the event data or over event rate.

Esper and StreamInsight lacked any immediate support for generating a stream. Event publishers were written in Java (Esper) or C# (StreamInsight). While rapid and controlled randomization with a specified number of events was achievable, the rest of the options involved a significant implementation effort. On the other hand, StreamBase supports input-feed generation by guessing the input event type and offers extensive options to control the input and its speed.

Where more event types would be needed, writing such event generators would be redundant. The feed generator would then have to be changed as the application adapts to a new event source. Also, conditioning the input and applying a randomness or uniformity to it—so that all conditions within the pipeline could be met—would require some work, as well.

What follows are some ways to control or condition input event data being generated for testing:

1. Controlled randomization—randomize within a set between a minimum and maximum value
2. Input files—a .csv or an .xml file from which data can be read
3. Database—data that is controlled from a database table and read using JDBC or ODBC

All of these options can very useful at various points of testing. Should the chosen platform not possess an automatic feed-generation feature, a generic event generator (for the given event type) with the minimum set of options listed above is recommended for testing purposes.

Fincos is an OpenSource tool for benchmarking performance for JMS AP-supported platforms or through custom adapters. Although it is a performance benchmarking tool, many of its features revolve around generating inputs synthetically or using a .csv file. Input schema can be defined and the values can be controlled. The output thus generated could be saved to a .csv file or fed directly into CEP engines, such as Esper.

For unit testing of custom operators, as well as filters and aggregate functions, unit-testing tools and frameworks of the same language (such as nunit or junit) could be used.

DEPLOYMENT CONSIDERATIONS

Depending on the hosting options provided by the CEP platform, CEP applications could either bundle the CEP engine with the application (self-hosted) or connect to the central CEP server (remote hosted).

Hosted/embedded mode is useful when working with
one client application, when doing a quick prototype or when there is no event-type sharing between client applications. Access to the CEP engine can be controlled within the application logic. In a hosted-mode deployment, since the application is hosting the CEP engine, there are no specific steps apart from compiling and running the application within the platform-supported runtime.

If the event types, pattern queries or adapters need to be shared across multiple client applications, the CEP platform should be used in server mode. Clients can then connect to the server and use the shared entities. Also, a central, server-based deployment allows for deploying queries dynamically. The server should be secured in accordance with platform recommendations. All sensitive information in the configuration files should be encrypted. Deploying CEP applications to a server would involve compiling, bundling and deploying using several deployment mechanisms based on the platform, such as administrative APIs and configuration files. The server platform should also support "undeployment" to remove all associated entities from the server.

StreamInsight and Esper support a hosted deployment model where they run in engine mode. StreamInsight also offers a remote server mode, wherein the engine can be installed as a win32 service and acts as a server host. Esper offers a remote server mode as part of the Esper Enterprise Edition. StreamBase offers a server mode wherein CEP application bundles can be deployed.

ANALYZING FOR PERFORMANCE

As CEP applications aim to achieve high-velocity, low-latency processing, these applications need to be analyzed to ensure that they meet performance goals. Tools should be used to load, measure, evaluate and identify any performance issues.

Although performance testing suites for CEP applications are a rare find, Fincos, which was introduced earlier, is one such suite. Apart from controlling the data, it also provides options to control the data rate.

Some CEP platforms, such as StreamBase and Esper, provide such tools for generating the necessary load and setting up the performance tests. For other platforms, the event-feed generators discussed in Inspecting, Debugging and Testing can be enhanced to control the event rate as follows:

1. Throughput—Number of events to be generated in a given interval
2. MaxLoad—Send as many events as possible; time to start and stop event generation would be a “nice to have”
3. Velocity—Events generated per second
4. Latency—Lead time between two events

Monitoring the CEP application performance statistics for these factors and other hardware counters, such as CPU time and memory, is necessary to identify bottlenecks. Individual platforms could provide tools for measuring; for example, StreamInsight offers Event Flow Debugger and Diagnostics API, which reports on various statistics for operators, adapters and queries. For its part, StreamBase offers StreamBase Monitor for monitoring the performance counters.

Generally, bottlenecks could occur in a CEP application in the following constructs:

1. Adapters—input and output adapters struggling to enqueue/dequeue from the CEP engine
2. Operators—improper use of aggregations and/or custom operators that are not able to provide adequate performance
3. Heavy Pipeline—large memory consumption of event windows and/or too many joins across the windows

Once bottlenecks are identified using the monitoring tools, multiple techniques can be applied for performance tuning. Techniques, such as filtering early, narrowing results, caching and increasing computing power, can be used to fine-tune CEP application performance.
This paper has described the concepts of CEP, some applications of CEP, and factors to consider when designing and developing a CEP application. A sampling implementation approach—spanning three CEP products on diverse platforms—formed the input to the paper and helped in understanding commonalities across CEP platforms and applications.

Some of the key design factors, including scalability, availability, pattern matching and caching, were discussed, including various aspects of CEP application development and recommendations for each. Architects and developers can leverage these insights when designing and building CEP applications.

In general, a holistic approach should be taken while designing and developing CEP applications. By giving an overview of the array of choices and highlighting some key recommendations, this paper will help designers and developers bring such a view to their work in CEP applications.

GLOSSARY

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>An immutable record of a past occurrence of an action or state change</td>
</tr>
<tr>
<td>Stream</td>
<td>A set of associated events</td>
</tr>
<tr>
<td>Event Stream Processing</td>
<td>A technology stack designed to support implementation of event-driven architectures</td>
</tr>
<tr>
<td>Volume</td>
<td>Total number of events being processed</td>
</tr>
<tr>
<td>Velocity</td>
<td>Number of events processed in a given interval (such as one second)</td>
</tr>
<tr>
<td>Latency</td>
<td>Elapsed time from the event arriving in the CEP engine and being processed</td>
</tr>
<tr>
<td>BAM</td>
<td>Business Activity Monitoring</td>
</tr>
<tr>
<td>BPM</td>
<td>Business Process Management</td>
</tr>
<tr>
<td>CTD</td>
<td>Contribution To Duration</td>
</tr>
</tbody>
</table>
REFERENCES

1. Complex Event Processing – 10 Design Patterns – Sybase
2. Big Data vs Event Processing – The TIBCO Blog
3. Understanding Scalability – Oracle CEP Guide
4. The Forrester Wave™: Complex Event Processing (CEP) Platforms, Q3 2009
5. Storm & Esper
6. The Esper CEP Ecosystem
7. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions - Gregor Hohpe, Bobby Wool; Addison-Wesley Professional
10. StreamBase: Expression Language Functions
11. Esper: Packaging and Deploying
12. StreamBase: Administration Guide
13. StreamInsight: Planning and Architecture
14. StreamInsight: Resiliency
15. Oracle Complex Event Processing High Availability
16. StreamBase tuning tips
17. Fincos – Benchmarking tool for CEP systems

ABOUT SAPIENT GLOBAL MARKETS

Sapient Global Markets, a division of Sapient® [NASDAQ: SAPE], is a leading provider of services to today’s evolving financial and commodity markets. We provide a full range of capabilities to help our clients grow and enhance their businesses, create robust and transparent infrastructure, manage operating costs, and foster innovation throughout their organizations. We offer services across Advisory, Analytics, Technology, and Process, as well as unique methodologies in program management, technology development, and process outsourcing. Sapient Global Markets operates in key financial and commodity centers worldwide, including Boston, Chicago, Houston, New York, Washington, D.C., Calgary, Toronto, London, Düsseldorf, Frankfurt, Geneva, Munich, Zurich, and Singapore, as well as in large technology development and operations outsourcing centers in Bangalore, Delhi, and Noida, India.

For more information, visit www.sapientglobalmarkets.com.
# GLOBAL OFFICES

<table>
<thead>
<tr>
<th>City</th>
<th>Address</th>
<th>Phone Number</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bangalore</strong></td>
<td>Salarpuria GR Tech Park 6th Floor, &quot;VAYU&quot; Block #137, Bengaluru 560066</td>
<td>Tel: +91 (080) 410 47 000</td>
</tr>
<tr>
<td><strong>Boston</strong></td>
<td>131 Dartmouth Street 3rd Floor Boston, MA 02116</td>
<td>Tel: +1 (617) 621 0200</td>
</tr>
<tr>
<td><strong>Calgary</strong></td>
<td>888 3rd Street SW Suite 1000 Calgary, Alberta T2P 5C5 Canada</td>
<td>Tel: +1 (403) 444 5574</td>
</tr>
<tr>
<td><strong>Chicago</strong></td>
<td>30 West Monroe, 12th floor Chicago, IL 60603</td>
<td>Tel: +1 (312) 458 1800</td>
</tr>
<tr>
<td><strong>Delhi</strong></td>
<td>Unitech Infospace Ground Floor, Tower A Building 2, Sector 21 Old Delhi - Gurgaon Road Dundahera, Gurgaon 122016 Haryana India</td>
<td>Tel: +91 (124) 416 7000</td>
</tr>
<tr>
<td><strong>Düsseldorf</strong></td>
<td>Speditionstraße 21 40221 Düsseldorf Germany</td>
<td>Tel: +49 (0) 211 540 34 0</td>
</tr>
<tr>
<td><strong>Frankfurt</strong></td>
<td>Skyper Villa Taunusanlage 1 60329 Frankfurt Germany</td>
<td>Tel: +49 (0)69 505060594</td>
</tr>
<tr>
<td><strong>Geneva</strong></td>
<td>Succursale Genève c/o Florence Thiébaud, avenue du Cendrier 15 1201 Geneva Switzerland</td>
<td>Tel: +41 (0) 58 206 06 00</td>
</tr>
<tr>
<td><strong>Houston</strong></td>
<td>Heritage Plaza 1111 Bagby Street Suite 1950 Houston, TX 77002</td>
<td>Tel: +1 (713) 493 6880</td>
</tr>
<tr>
<td><strong>London</strong></td>
<td>Eden House 8 Spital Square London, E1 6DU United Kingdom</td>
<td>Tel: +44 (0) 207 786 4500</td>
</tr>
<tr>
<td><strong>Los Angeles</strong></td>
<td>1601 Cloverfield Blvd. Suite 400 South Santa Monica, CA 90404</td>
<td>Tel: +1 (310) 264 6900</td>
</tr>
<tr>
<td><strong>Milan</strong></td>
<td>Sapient Italy S.r.l Viale Bianca Maria 23 20122 Milan Italy</td>
<td>Tel: +39-02-36269529</td>
</tr>
<tr>
<td><strong>Munich</strong></td>
<td>Arnulfstraße 60 80335 München Germany</td>
<td>Tel: +49 (0) 89 552 987 0</td>
</tr>
<tr>
<td><strong>Noida</strong></td>
<td>(NCR of Delhi) &quot;Oxygen&quot;, Tower C, Ground - 3rd floor Plot No. 7 Sector 144 Expressway Noida 201304 Uttar Pradesh India</td>
<td>Tel: +91 (120) 479 5000</td>
</tr>
<tr>
<td><strong>New York</strong></td>
<td>40 Fulton Street 22nd Floor New York, NY 10038</td>
<td>Tel: +1 (212) 206 1005</td>
</tr>
<tr>
<td><strong>Singapore</strong></td>
<td>158 Cecil Street, #03-01 Singapore 069545</td>
<td>Tel: +65 6671 4933</td>
</tr>
<tr>
<td><strong>Toronto</strong></td>
<td>129 Spadina Avenue Suite 500 Toronto, Ontario M5V 2L3 Canada</td>
<td>Tel: +1 (416) 645 1500</td>
</tr>
<tr>
<td><strong>Washington DC</strong></td>
<td>1515 North Courthouse Road 4th Floor Arlington, VA 22201-2909</td>
<td>Tel: +1 (703) 908 2400</td>
</tr>
<tr>
<td><strong>Zürich</strong></td>
<td>Seefeldstrasse 35 8008 Zurich Switzerland</td>
<td>Tel: +41 (58) 206 06 00</td>
</tr>
</tbody>
</table>