Microsoft® SQL Server®2008

Introducing Microsoft StreamInsight

Technical Article

Writer: Torsten Grabs, Roman Schindlauer, Ramkumar Krishnan, Jonathan Goldstein

Published: September 2009

Applies to: StreamInsight

Summary: While typical relational database applications are query-driven, event-driven applications have become increasingly important. Event-driven applications are characterized by high event data rates, continuous queries, and millisecond latency requirements that make it impractical to persist the data in a relational database for processing. These requirements are shared by vertical markets such as manufacturing, oil and gas, utilities, financial services, health care, web analytics, and IT and data center monitoring. Event-driven applications use complex event processing (CEP) technology with the goal of identifying meaningful patterns, relationships and data abstractions from among seemingly unrelated events and trigger immediate response actions.

This paper provides an overview of the Microsoft SQL Server StreamInsight platform for complex event processing . StreamInsight allows software developers to create complex and innovative CEP solutions along two scenarios: (1) building packaged event-driven applications for low latency processing and (2) developing custom event-driven applications for businesses with high throughput, low latency needs. The paper describes the StreamInsight architecture and major components and demonstrates how to write continuous queries in a declarative way to analyze and process the events flowing through the system.

Copyright

Contents

Introduction

Microsoft StreamInsight is a powerful platform for developing and deploying complex event processing (CEP) applications. Its high-throughput stream processing architecture and .NET-based development platform enable developers to quickly implement robust and highly efficient event processing applications. Typical event stream sources include data from manufacturing applications, financial trading applications, Web analytics, or operational analytics. StreamInsight enables you to develop CEP applications that derive immediate business value from this raw data by lowering the cost to extract, analyze, and correlate the data and by allowing you to monitor, manage, and mine the data for conditions, opportunities, and defects almost instantly.

You can achieve the following tactical and strategic goals for your enterprise by developing your CEP applications using StreamInsight.

- Monitor your data from multiple sources for meaningful patterns, trends, exceptions, and opportunities.

  Analyze and correlate data incrementally while the data is in-flight -- that is, without first storing it-- yielding very low latency. Aggregate seemingly unrelated events from multiple sources and perform highly complex analyses over time.

- Manage your business by performing low-latency analytics on the events and triggering response actions that are defined on your business key performance indicators (KPIs).

  Respond quickly to areas of opportunity or threat by incorporating your KPI definitions into the logic of the CEP application, thereby improving operational efficiency and your ability to respond quickly to business opportunities.

- Mine events for new KPIs.

  Move toward a predictive business model by mining historical data to continuously refine and improve your KPI definitions.

Key Benefits

StreamInsight has the following key benefits:

- Highly optimized performance and data throughput

  StreamInsight implements a lightweight streaming architecture that supports highly parallel execution of continuous queries over high-speed data. The use of in-memory caches and incremental result computation provide excellent performance with high data throughout and low latency. Low latency is achieved because the events are processed without costly data load or

storage operations in the critical processing path. With StreamInsight, all processing is automatically triggered by incoming events. In particular, applications do not have to incur any overhead for event polling. The platform provides the functionality for handling out-of-order events. In addition, static reference or historical data can be accessed and included in the low-latency analysis.

- .NET development environment

  Developers can write their CEP applications using Microsoft's .NET language such as Visual C#, leveraging the advanced language platform LINQ (Language Integrated Query) as an embedded query language. Given the large community of developers already familiar with these technologies, this capability reduces development costs and the time from application development to production.

  By using LINQ, developers familiar with SQL will be able to quickly write queries in a declarative fashion that process and correlate data from multiple streams into meaningful results. The optimizer and scheduler of the CEP server in turn ensure optimal query performance.

- Flexible deployment capability

  StreamInsight supports two deployment scenarios. It can be fully integrated into the application as a hosted (embedded) DLL or deployed as a stand-alone server with multiple applications and users sharing the server. In its stand-alone configuration, the CEP server runs in a wrapper such as an executable, or the server could be packaged as a Windows Service.

- Manageability

  The monitoring and manageability features built into the CEP server provide for low total cost of ownership (TCO) of CEP applications. The management interface and diagnostic views that are provided in the CEP server allow the administrator to monitor and manage the CEP application. The manageability framework also allows for ISVs and system integrators to remotely monitor and support CEP-deployed systems at manufacturing and other scale-out installations.

  StreamInsight ships with a stand-alone event flow debugger that can be used to analyze, diagnosis, and troubleshoot queries used in CEP applications.

CEP Scenarios

The need for high-throughput, low-latency processing of event streams is common to the following business scenarios:

- Manufacturing process monitoring and control
- Clickstream analysis
- Financial services
- Power utilities
- Health care

- IT monitoring
- Logistics
- Telecom

The following sections discuss some of these scenarios and investigate their requirements for event processing.

Manufacturing Process Monitoring and Control

To ensure that products and processes are running optimally and with the least amount of downtime, manufacturing companies require low-latency data collection and analysis of plant-floor devices and sensors. The typical manufacturing scenario includes the following requirements:

- Asset-based monitoring and aggregation of machine-born data.
- Sensor-based observation of plant floor activities and output.
- Observation and reaction through device controllers.
- Ability to handle up to 10,000 data events per second.
- Event and alert generation the moment something goes wrong.
- Proactive, condition-based maintenance on key equipment.
- Low-latency analysis of aggregated data (windowed and log-scales).

Clickstream Analysis

An optimal customer experience from a commercial Web site requires low-latency processing of user behavior and interactions at the site. The typical click stream analysis application includes the following requirements:

- Ability to drive page layout, navigation, and presentation based on low-latency click stream analysis.
- Ability to handle up to 100,000 data events per second during peak traffic times.
- Immediate click-stream pattern detection and response with targeted advertising.

Algorithmic Trading in a Financial Services Environment

Algorithmic trading, with its high volume data processing needs, typically has the following requirements:

- Ability to handle up to 100,000 data events per second.
- Time-critical query processing.
- Monitoring and capitalizing on current market conditions with very short windows of opportunity.
- Smart filtering of input data.
- Ability to define patterns over multiple data sources and over time to automatically trigger buy/sell/hold decisions for assets in a portfolio.

Power Utilities

The utility sector requires an efficient infrastructure for managing electric grids and other utilities. These systems typically have the following requirements.

- Immediate response to variations in energy or water consumption, to minimize or avoid outages or other disruptions of service.
- Gaining operational and environmental efficiencies by moving to smart grids.
- Multiple levels of aggregation along the grid.
- Ability to handle up to 100,000 events per second from millions of data sources.

StreamInsight Architecture

The StreamInsight runtime is the CEP server. It consists of the core engine and the adapter framework. The adapter framework allows developers to create interfaces to event stores such as Web servers, devices or sensors, and stock tickers or news feeds and event sinks such as pagers, monitoring devices, KPI dashboards, trading stations, or databases. Incoming events are continuously streamed into standing queries in the CEP server, which processes and transforms the data according to the logic defined in each query. The query result at the output can then be used to trigger specific actions.

The following illustration presents a high-level overview of the StreamInsight architecture.
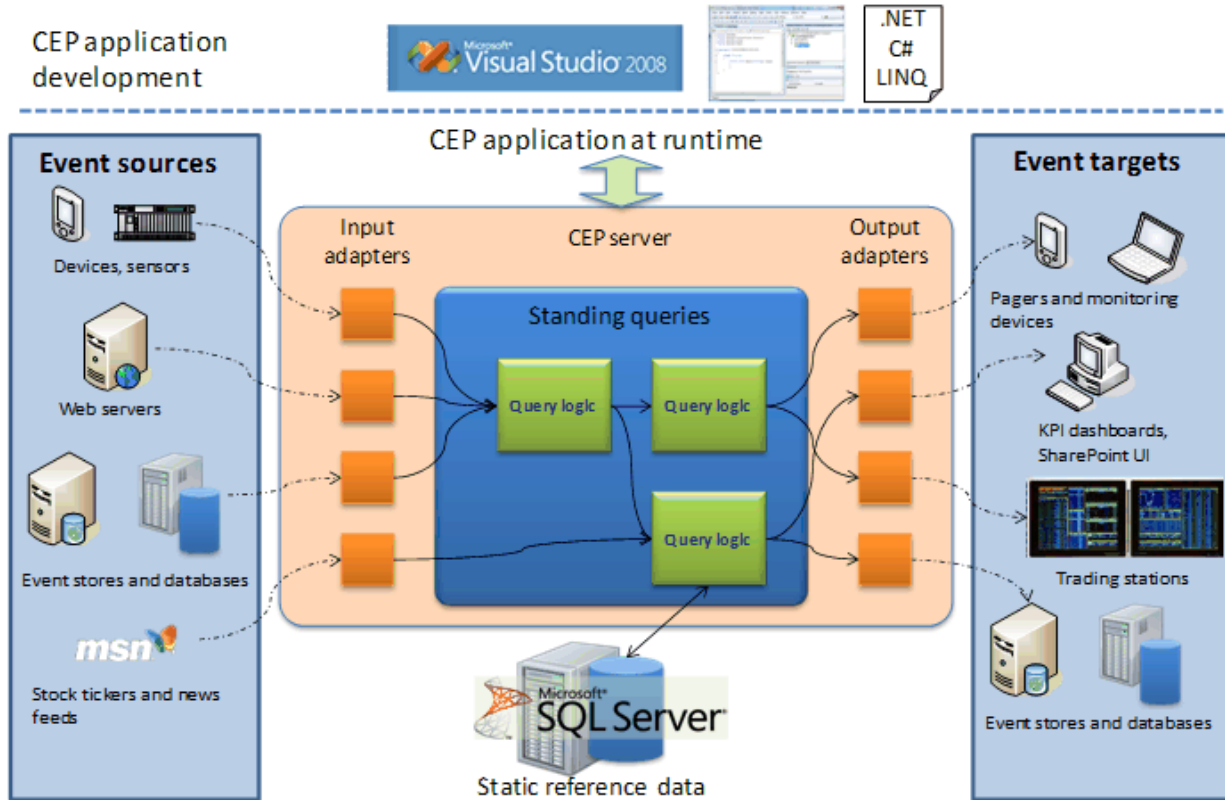
Figure 1: StreamInsight Architectural Overview

CEP Application Development

As shown in Figure 1, developing a CEP application using StreamInsight consists of the following tasks:

1. Defining event sources and event targets (sinks).

2. Creating an input adapter to read the events from the source into the CEP server for processing and an output adapter to consume the processed events for submission to the event targets.

3. Creating the query logic required to meet your business objectives and binding the query to the adapters at runtime to instantiate the query in the CEP server.

Development Models

StreamInsight provides three development models that support CEP application development. While the tasks defined above remain the same, actual implementation of the tasks varies with the development model . Sample applications demonstrating each of these models is available in the StreamInsight Samples Software Development Kit (SDK) available at this Microsoft Web site.

Explicit Server Development Model

The explicit server development model provides a full-service CEP application environment by allowing the application developer to explicitly create and register all of the objects required to transform and process events coming into and going out of the CEP server. This gives the developer complete control of his or her CEP application and development environment by using the client-side object model API.

Typical use cases for the explicit server development model include CEP applications that require:

- Full control and access to the CEP server.

- Reusing queries through static or dynamic query composition, or reusing adapters, event types, and query templates defined by a third party.

Implicit Server Development Model

The implicit server development model provides an easy-to-use environment that hides much of the complexity associated with the explicit server development model . It does this by allowing the CEP server to act as the implicit host creating and registering most of the objects required to transform and process events coming into and going out of the CEP server. This allows the developer to focus his or her efforts on developing the query logic needed to process and analyze the events within the CEP server. The server object itself is "anonymous" and cannot be accessed directly through the object model.

Typical use cases for the implicit server development model include:

- Embedding the CEP server in an application in which only a single query is required.

- As a test environment for query developers.

IObserverable/IObserver Development Model

The IObservable/IObserver development model provides an alternative method of implementing input and output adapters as the producers and consumers of event sources and sinks. This model is based on the IObservable/IObserver design pattern in which an observer is any object that wishes to be notified when the state of another object changes, and an observable is any object whose state may be of interest, and in whom another object may register an interest. For example, in a publication-subscription application, the observable is the publisher, and the observer is the subscriber object. For more information, see Exploring the Observer Design Pattern on MSDN.

In (CEP) applications, the observable is the event source. The query acts as an observer of this source and presents its result as an observable to the event target, which is also an observer.

Typical use cases for the IObservable/IObserver development model include:

- You want to tie the CEP event production and consumption tighter into the CLR model of querying over enumerations of events by using LINQ.

- You want your .NET applications to continue working on data elements without a major learning or programming investment to define the temporal properties of the events. At the same time, if the application is actually event-driven, a CEP application developer can view these data elements as events and process them using the CEP server for real-time analysis and insights.

CEP Application Components

In this section, we describe the components and objects that are required in a CEP application and provide details and examples for the application development tasks.

Defining Event Sources and Event Targets

When defining the event sources and targets, it is important to understand the structure of the event data (for example, the number of fields and data types) as well as the temporal characteristics of the event (for example, the time period that an event is valid). In this section, we describe the components of an event and the temporal characteristics of events. You will use this information to create event types for your CEP application.

The underlying data represented in the event stream is packaged into events. An event is the basic unit of data processed by the CEP server. Each event consists of the following parts:

- Header

  An event header contains metadata that defines the event kind and one or more timestamps that define the time interval for the event. The timestamps are application-based and supplied by the data source rather than a system time supplied by the CEP server. Note that the timestamps use the **datetime** data type, which has time zone awareness and is based on a 24-hour clock. The CEP server normalizes all times to UTC and verifies on input that the UTC flag is set on the timestamp fields.

- Payload

  The payload of an event is a .NET data structure that contains the data associated with the event. The fields in the payload are user-defined and their types are based on the .NET type system. CLR scalar and elementary types are supported for payload fields. Nested types are not supported.

Event Header
The header of an event defines the event kind and event model.

Event Kind
The event kind indicates whether the event is a new event in the stream or the event is declaring the completeness of the existing events in the stream. StreamInsight supports two event kinds: INSERT and CTI (current time increment).

The INSERT event kind adds an event with its payload into the event stream. In addition, the header of the INSERT event identifies the start and end time for the event.

The CTI event kind is a special punctuation event that indicates the completeness of the existing events in the stream. The CTI event structure consists of a single field that provides a current timestamp. It is used to manage out-of-order events or latency in the event stream. The CTI event indicates to the CEP server that no subsequent incoming INSERT events will revise the event history before the CTI timestamp. After a CTI event has been issued, no INSERT event can have a start time earlier than the timestamp of the CTI event. By indicating completeness, the CEP server can release the results of windowing or other aggregating operators that have accumulated state, thus ensuring that events flow efficiently through the system.

Event Model

The event model defines the event shape based on its temporal characteristics. StreamInsight supports three event models: interval, point, and edge.

Interval Model

The interval event model represents an event whose payload is valid for a given period of time. The interval event model requires that both the start and end time of the event be provided in the event metadata. Interval events are valid only for this specific time interval. Examples of interval events include the width of an electronic pulse, the duration of (validity of) an auction bid, or a stock ticker activity in which the bid price for the stock is valid for a specific time period. In a utility power monitoring scenario, a power meter event stream may be represented with the following interval events, in which the payload is a single field containing power consumption for a given meter for the given time period.

Event Kind	Start Time	End Time	Payload (Power Consumption)
INSERT	2009-07-15 09:13:33.317	2009-07-15 09:14:09.270	100
INSERT	2009-07-15 09:14:09.270	2009-07-15 09:14:22.253	200
INSERT	2009-07-15 09:14:22.255	2009-07-15 09:15:04.987	100

Point Model

A point event model represents an event occurrence as of a single point in time. It is a subclass of the interval event model. The point event model requires only the start time for the event. The CEP server infers the valid end time by adding a tick (the smallest unit of time in the underlying time data type) to the start time to set the valid time interval for the event. Point events are valid only for this single instant of time.

Examples of point events include a meter reading, the arrival of an email, a user Web click, a stock tick, or an entry into the Windows Event Log. In the power monitoring example described above, the power meter event stream may be represented with the following point events. Note that the end time is calculated as the start time plus 1 tick.

11

Event Kind	Start Time	End Time	Payload (Consumption)
INSERT	2009-07-15 09:13:33.317	2009-07-15 09:13:33.317 + t	100
INSERT	2009-07-15 09:14:09.270	2009-07-15 09:14:09.270 + t	200
INSERT	2009-07-15 09:14:22.255	2009-07-15 09:14:22.255 + t	100

Edge model

An edge event model represents an event occurrence whose payload is valid for a given interval of time, however, only the start time is known upon arrival to the CEP server. The end time of the event is known later and updated. The edge event model contains two properties: time and an edge type. Together, these properties define either the start or end point of the edge event.

Examples of edge events are Windows processes, trace events from Event Tracing for Windows (ETW), a Web user session, or quantization of an analog signal. An event stream containing edge events may be represented with the following events. The combination of the two insert events that define the start and stop points of the edge are used to define the valid time interval for that payload. That is, the valid time interval for an edge event is the difference between the timestamp of the START event and the timestamp of the STOP event. Notice that event 5, with a payload value of 'd', does not have a known end date at this point in time.

Event Kind	Edge Type	Start Time	End Time	Payload
INSERT	Start	t0	∞	a
INSERT	End	t0	t1	a
INSERT	Start	t1	∞	b
INSERT	End	t1	t3	b
INSERT	Start	t3	∞	c
… and so on				

The following illustration shows the quantization of an analog signal using edge events based on the start and end times defined in the table above.
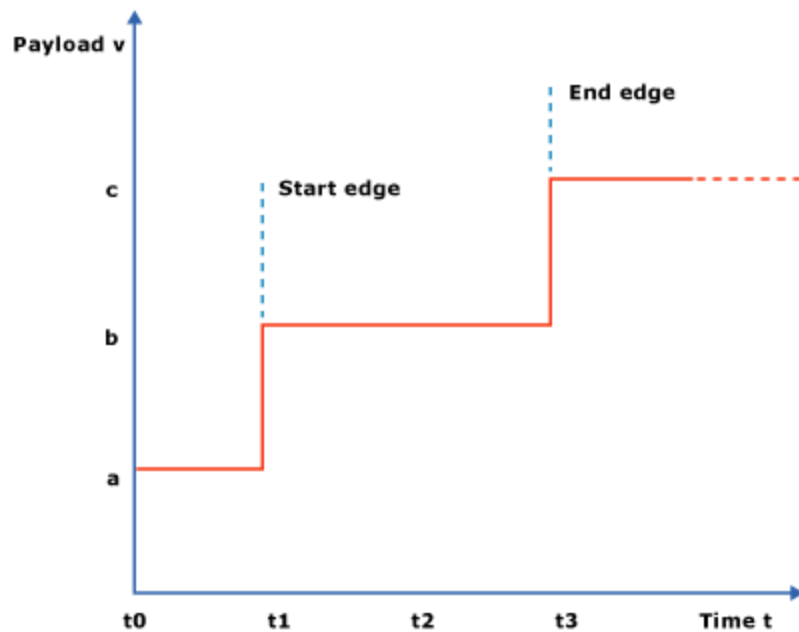
Creating Input and Output Adapters

The next task is to create the adapters that submit events to the CEP server for processing or consume the processed events and submit them to the event target. StreamInsight provides a highly flexible adapter SDK that enables you to build adapters for your domain-specific event sources and event targets.

Input Adapters

An input adapter instance accepts incoming event streams from external sources such as databases, files, ticker feeds, network ports, and so on. The input adapter reads the incoming events in the format in which they are supplied and translates this data into the event format that is consumable by the CEP server.

You create an input adapter template to handle the specific event sources for your data source. If the event source only produces a single event type, your adapter can be typed--that is, it is implemented to emit a single event type. With a typed adapter, all instances of the adapter produce the same fixed payload format in which the number of fields and their types are known in advance. Examples of such events are ticker feed data or sensor data emitted by a specific device. If your event source emits different types under different circumstances, that is, the events may contain different payload formats or the payload format may not be known in advance, your adapter must be untyped. The event payload format is provided to the adapter at the time of binding the adapter to the query, as part of a configuration specification. Examples of such events include CSV files that contain a varying number of fields where the type of data stored in the file is not known until query instantiation time, or an adapter for SQL Server tables where the events produced depends on the schema of the table that the adapter instance is pointed to by the user. It is important to note that, at runtime, a single adapter instance, whether typed or untyped, always emits events of one specific type. Untyped adapters provide a flexible

13

implementation to accept events types at query bind time, rather than defining the event type at the time the adapter is implemented.

You create an output adapter template to receive the events processed by the CEP server, translate the events into a format expected by the event target, and emit the data to that device. Designing and creating an output adapter is similar to designing and creating an input adapter.

Based on the event type and model, you select the appropriate adapter base class.

Adapter type	Input adapter base class	Output adapter base class
Typed point	**TypedPointInputAdapter**	**TypedPointOutputAdapter**
Untyped point	**PointInputAdapter**	**PointOutputAdapter**
Typed interval	**TypedIntervalInputAdapter**	**TypedIntervalOutputAdapter**
Untyped interval	**IntervalInputAdapter**	**IntervalOutputAdapter**
Typed edge	**TypedEdgeInputAdapter**	**TypedEdgeOutputAdapter**
Untyped edge	**EdgeInputAdapter**	**EdgeOutputAdapter**

For more information about creating adapters, see Creating Adapters on MSDN.

Creating Queries to Process and Analyze Events

With StreamInsight, event processing is organized into queries based on query logic that you define. These queries take a potentially infinite feed of time-sensitive input data (either logged or real time), perform some computation on the data, and output the result in an appropriate manner. Developing the query logic you need to process and analyze incoming events is the core task in developing your CEP application.

In this task, you create a query template to store the required query logic and bind the query template to specific input and output adapters to register a query instance in the CEP server. This is the final run-time manifestation of the query. Once data has been brought into the CEP server via input adapters, computation may be continuously performed over the data. In other words, as individual events arrive in the server, these events are processed by standing queries, which emit output events in response to the arrival of input events. The queries can be started, stopped, and managed. The following illustration shows the CEP query and adapter ecosystem. The CEP server consumes and processes the event when the instance of the input adapter is bound to an instance of a query. The processed data is then pushed to the instance of the output adapter that is bound to the same query instance.
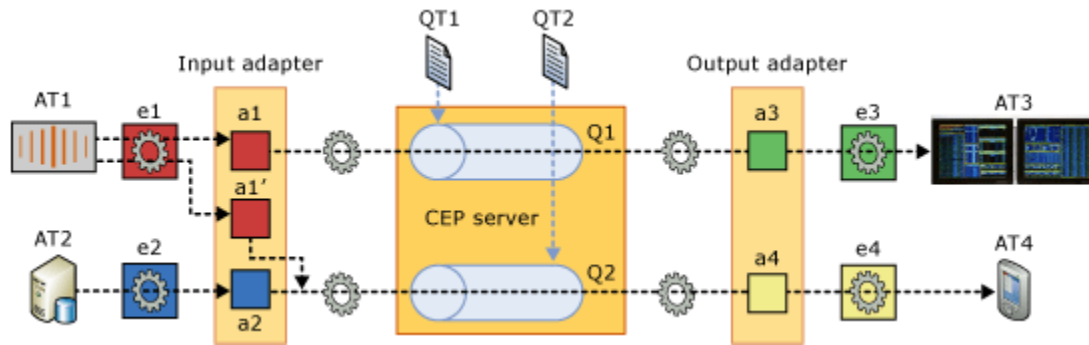
Query Templates

A query template is the fundamental unit of query composition. It is the structure the defines the business logic required to continuously analyze and process events submitted to the CEP server from the input adapter and generate an event stream that is consumed by the output adapter. For example, you may want to evaluate incoming power consumption events for maximum or minimum values over a given time period that exceed certain thresholds that you establish.

Query templates can be written to perform specific units of work and then composed into more complex query templates. Query templates are written in LINQ combined with a .NET language. LINQ is a language platform that enables you to express declarative computation over sets in a manner that is fully integrated into the host language. This gives you the power to combine declarative processing of events with the flexibility of procedural programming in the same development platform, without the concern of impedance mismatch between these two programming paradigms. For more information about LINQ, see Language-Integrated Query (LINQ) on MSDN.

Creating an Event Stream Object

In the query template, a LINQ expression is always defined over an object of type **CEPStream<PayloadType>**, which represents the event stream source for the query template. An event stream can be created in one of the following ways:

- From an unbound stream.

  The static method **Create()** of the class **EventStream** can produce a stream with only a shape defined and no binding information, as shown in the following example. This approach suggests the development of an unbound query template, for instance to be explicitly registered into a CEP server.

  ```
  CepStream<PayloadType> inputStream =
                  CepStream.CreateInputStream<PayloadType>("inputStream");
  ```

- From a user-defined input adapter factory.

  The **CepStream** type implements static methods that take an input adapter factory and an input configuration, and produces an input stream of a given shape as shown in the following example. The adapter factory must provide for the instantiation of the adapter with the given stream shape. This approach corresponds to the implicit server development model in which the registration of adapters is not needed as a separate

15

step. The example below shows how to define stream based on a factory for generically typed adapters. A corresponding method exists to define a stream from a typed adapter factory.

```
CEPStream<PayloadType> inputStream1 =
     CepStream<PayloadType>.CreateFromUnTypedFactory<InputAdapterFactory,
                                               Config>(
                                   streamName, eventShape, myconfig);
```

- From an **IObservable** object.

  The **IObservable** interface provides the method **ToCepStream()**, which converts an **IObservable** data source into a CEP stream as shown in the following example. The **ToCepStream()** method takes as a single argument: a lambda expression that defines the timestamps of the events. In this example the timestamps are specified as coming from a payload field of the events. Note that only a single timestamp is defined, resulting in point events.

```
CEPStream<PayloadType> inputStream1 =
          myObservableSource.ToCepStream<PayloadType>(e => e.myTimeStamp);
```

Create the Query Template Definition

The query logic is specified in terms of CEP operators through LINQ statements built on top of **CepStream** objects, as shown in the following example.

```
CepStream<PayloadType> outputStream = from e in inputStream
                                      where e.value < 10
                                      select e;
```

In the explicit server development model, in which an explicit CEP application is created, the specified query logic can be registered as a **QueryTemplate** object into the application, as shown in the following example.

```
QueryTemplate qt = application.CreateQueryTemplate("samplequerytemplate",
outputStream);
```

The registered query template can now be reused in multiple bindings and, hence, instantiated in multiple queries each bound to potentially different input and output adapters. These bindings for registered query templates are defined through the **QueryBinder** object, which is described in the following section.

Binding Query Templates

How a runnable query is obtained from a query template depends on the development model.

Specifying the Event Producers and Consumers Through Binding

When using the explicit server development model, explicit adapter objects are created and the output stream is bound through the query binder as part of the process of defining a query, as shown in the following example.

```
// Create input adapter object from existing factory
```

16

```
InputAdapter myInputAdapter =
application.CreateInputAdapter<TextFileInputFactory>("CSV Input", "Reading
tuples from a CSV file");
// Create output adapter object from existing factory
OutputAdapter myOutputAdapter =
application.CreateOutputAdapter<TextFileOutputFactory>("CSV Output", "Writing
result events to a file");


// create a query binder, wrapping the defined query template object
QueryBinder myQuerybinder = new QueryBinder(qt);


// Bind input adapter to the query template's stream that was
// created with the name "querysource"
// and apply the runtime configuration.
myQuerybinder.BindInputStream<PayloadType, InConfig>("querysource",
myInputAdapter, EventShape.Point, inputConf);
// Bind output adapter to query and apply the runtime configuration.
myQuerybinder.AddQueryConsumer<OutConfig>("queryresult", myOutputAdapter,
outputConf, EventShape.Point, StreamEventOrder.FullyOrdered);


// Create query in the application, from the query binder.
Query query = application.CreateQuery("query", myQuerybinder, "description
for query");
```

The bindings of input and output require instances of the **InputAdapter** and **OutputAdapter** objects, which are obtained from their respective factories, as shown above.

Note that the explicit server development model allows for the separation of the following steps:

1. Defining the query template and storing it on the server.
2. Binding an existing query template to specific input and output adapters (using the binder), thus creating a query that can be run.

The second step can be carried out for a single query template many times potentially, creating multiple, distinct run-time instances of the same query logic.

A query can have one output adapter as the consumer, but multiple input adapters as event producers. The binding of the input streams, therefore, must refer to the stream names that were used when creating the **CepStream** objects in their respective query template, as shown in the following example. The example uses the same input adapter object in the input binding, but with different run-time configuration objects. When starting a query based on this binder, the CEP server creates two different runtime instances of the input adapter, each with a different event type and runtime configuration.

17

Note that the actual query template, the creation of the binder, and the output adapter object are not shown in this example.

```
CepStream<SensorReading> sensorStream =
CepStream.CreateInputStream<SensorReading>("sensorInput");
CepStream<LocationData> locationStream =
CepStream.CreateInputStream<LocationData>("locationInput");


// Define query template in LINQ on top of sensorStream and locationStream
// ...
// Create query binder like above
// ...


InputAdapter csvInput =
application.CreateInputAdapter<TextFileInputFactory>("CSV Input", "Reading
tuples from a CSV file");


qb.BindInputStream<SensorReading, InConfig>("sensorInput", csvInput,
EventShape.Interval, sensorInputConf);
qb.BindInputStream<LocationData, InConfig>("locationInput", csvInput,
EventShape.Interval, locationInputConf);
```

Consuming an Event Stream Directly from the Query Template

When using the implicit server development model, once the query logic is defined, the output adapter can consume the event stream directly from the query, as shown in the following example. The **CepStream.ToQuery()** method takes a **CepStreamConsumer** object, which can be created from an output adapter factory through the **CepStreamConsumer.Create()** method.

```
// Create a stream consumer directly from the output adapter factory.
CepStreamConsumer<ResultType> streamConsumer =
CepStreamConsumer<ResultType>.Create<MyOutputAdapterFactory,
                           MyOutputConfig>("adapterConsumer",
                           EventShape.Interval,
                           outputConf,
                           StreamEventOrder.FullyOrdered);
// define the query logic in LINQ
CepStream<PayloadType> outputStream = from e in inputStream
                                    where e.value < 10
```

```
                                    select e;


// Create a query from the query template and feed it into the consumer.
Query query = outputStream.ToQuery<ResultType>(streamConsumer);
```

Using the IObservable/IObserver Development Model

You can use the **IObservable/IObserver** interface as the producer and consumer of event sources and targets. The IObservable/IObserver development model ties the CEP event production and consumption tighter into the CLR model of querying over enumerations of events by using LINQ. The event source is defined as an **IObservable** object. The CEP query acts as an **IObserver** object of this source and presents its result as an **IObservable** object to the event sink, which is again an **IObserver** object.

In the following example, the process for using the **IObservable/IObserver** interface is demonstrated. We assume the definition of the following event type:

```
public class MyPayload
{
    public double value;
    public MyPayload() { }
    public MyPayload(double i) { value = i; }
}
```

First, the input source is modeled as an **IObservable** object. The namespace **Microsoft.ComplexEventProcessing.Adapters.Observable** provides various extension methods to accomplish this task, for example, the object can be created from an **IEnumerable** object such as a **System.Collections.Generic.List**.

```
List<MyPayload> numberSeries = new List<MyPayload>();
for (int i = 0; i <= 20; i++)
    numberSeries.Add(new MyPayload(i));


IObservable<MyPayload> observableInput =
numberseries.ToObservable<MyPayload>();


IObservable<MyPayload> observableInput =
numberseries.ToObservable<MyPayload>();
```

Here, the extension method **ToObservable()** wraps the **IEnumerable** in an **IObservable** object that can now be used as a CEP stream.

19

```
var input = observableInput.ToCepStream<MyPayload>(e =>
DateTime.Now.Add(TimeSpan.FromSeconds(e.value)));


var result = from e in input
             select new MyPayload { value = e.value % 2 };
```
The resulting stream is then encapsulated in an **IObservable** object.

```
var queryOutput = result.ToObservable<MyPayload>();

IDisposable disposable = queryOutput.Subscribe(new
ObserveToConsole<MyPayload>());
```
The query is implicitly started as soon as the first observer subscribes to it. In this example, the observer simply dumps the result events to the console window:

```
public class ObserveToConsole<T> : IObserver<T>
{
    private StreamWriter _writer;
    FieldInfo[] _fieldInfos = typeof(T).GetFields();

    public ObserveToConsole()
    {
        _writer = new StreamWriter(Console.OpenStandardOutput());
    }


    public void OnCompleted()
    {
        _writer.Flush();
        _writer.Close();
    }


    public void OnError(Exception error)
    { }


    public void OnNext(T value)
    {
        for (int fieldNo = 0; fieldNo < _fieldInfos.Length; fieldNo++)
        {
            _writer.Write(fieldInfos[fieldNo].GetValue(value).ToString());
```

```
            if (fieldNo != _fieldInfos.Length - 1)

            {

                _writer.Write(", ");

            }

        }

        _writer.WriteLine();

    }

}
```

Defining Query Logic using LINQ

In the previous section, using LINQ to write declarative queries in your CEP application was discussed in terms of implementing the query logic in the query template. In this section, we describe in more detail some of the functionality available in StreamInsight to write expressive queries and analytics.

Calculations to introduce additional event properties

Use cases such as unit conversions require you to perform calculations on top of the events that you receive. Using the projection operation in the CEP server, you can add additional fields to the payload and perform calculations over the fields in the input event. For example, every event in the stream `MeterReading` will be projected into a new event in the stream `realValueStream`, using the explicitly declared event type `MeterWattage`. The `Consumption` field for each event in the `MeterReading` stream is converted to a double CLR type and divided by 10 and then assigned to the wattage field of `MeterWattage`.

```
var realValueStream = from e in InputStream
                      select new MeterWattage {wattage =
(double)e.Consumption / 10};
```

Filtering of events

In use cases such as alert notifications, you may want to check whether a certain payload field exceeds the operating thresholds for the piece of equipment that you are monitoring. In general, only a subset of events that satisfy certain characteristics is relevant for these use cases. Events that do not have these characteristics do not need to be processed and can be discarded. The filter operation allows you to express Boolean predicates over the event payload and discard events that do not satisfy the predicates.

In the following example, the events in the event stream `someStream` are limited to events in which the value in field `i` is greater than 10. Events that do not meet that criterion are removed from the stream.

```
var queryFilter = from c in someStream
                  where c.i > 10
                  select c;
```

Grouping events

Consider an event stream that gives you temperature readings from all of your temperature sensors. If all the events are provided through a single event stream, you may want to partition the incoming events based on the sensor location or the sensor ID. The CEP server provides a grouping operation that allows you to partition the incoming stream based on event properties such as location or ID and then apply other operations or complete query fragments to each group separately.

The following example groups events by the specified `modulo` function. It then applies a snapshot window to each group and computes the average over a payload column on each group separately.

```
var avgCount = from v in inputStream
                    group v by v.i % 4 into eachGroup
                    from window in eachGroup.Snapshot()
                    select new { avgNumber = window.Avg(e => e.number) };
```

Windows over time

Grouping events over time is a powerful concept that enables many scenarios. For instance, you may want to check the number of failures that occur during a fixed period of time and raise an alarm if they exceed a threshold. Hopping and sliding windows allow you to define windows over your event streams to perform this kind of analysis.

A sliding window contains events within the last X time units, at each point in time. The following example shows the summation of a payload field within the last one hour. This statement first assigns as a new event duration the original one (end - start) plus one hour, so that each event now also occupies snapshots for one full hour after the event's original lifetime. Subsequently, the **snapshot** operator creates those windows, so that the aggregation can be applied.

```
var slidingSum = from window in inputStream.AlterEventDuration(e =>
e.EndTime.Subtract(e.StartTime).Add(TimeSpan.FromMinutes(60))).Snapshot()
                    select new { sum = window.Sum(e => e.i) };
```

Aggregation

When you do not care about each single event, you might want to look into aggregate values such as averages, sums, or counts instead. The CEP server provides built-in aggregations for sum, count, and average that typically operate on time windows.

In the following example, the from clause applies a snapshot window on the stream `inputStream`, returning an event stream **CepWindowStream<T>**. Each element `w` in this stream represents a window that contains events. Aggregation functions are defined as methods of the window object and take lambda expressions as their argument. In the example, the `sum` aggregation results go to the payload field `e.i` and the `average` aggregation results go to the payload field `e.f`. The exception is the `Count` aggregate, which is not defined over a specific expression, but counts events as a whole. The example also shows how to combine several aggregations in the same statement. They are all computed within the same window.

```
// Multiple aggregations on top of snapshot window.
var snapshotAgg = from w in inputStream.Snapshot()
                select new { sum = w.Sum(e => e.i),
                            avg = w.Avg(e => e.f),
                            count = w.Count() };
```

Identifying top *N* candidates

A special kind of aggregation operation is needed in use cases where you want to identify heavy hitters in an event stream. The TopK operations allows you to check for those based on an order that you establish over the event fields in the stream.

The following example takes the top five events from each snapshot window defined for the input stream `inputStream` and generates a new event stream. The events in each window are ordered by payload fields `e.f` and `e.i`.

```
var topfive = (from window in inputStream.Snapshot()
            from e in window
            orderby e.f ascending, e.i descending
            select e).Take(5);
```

Matching events from different streams

A common use case is the need to reason about events received from multiple streams. For example, because event sources provide timestamps in their event data, you may want to make sure that you only match events in one stream with an event in the second stream if they are closely related in time. In addition, you may have additional constraints on which events to match, and when to match them. The CEP server provides a powerful join operation that performs both tasks: first, it matches events from the two sources if their times overlap and second, it execute the join predicate specified on the payload fields. The result of such a match contains both the payloads from the first and the second event. In the following example, events in stream `stream1` are compared with events in stream `stream2`. Events in the stream that meet the equality criteria defined in the `on` clause are joined and output into a new event that contains the payload fields `i` and `j` from event `e1` and field `j` from event `e2`.

```
var equiJoin = from e1 in stream1
            join e2 in stream2
            on e1.i equals e2.i
            select new { e1.i, e1.j, e2.j };
```

Combining events from different streams in one

Multiple data sources may provide events of the same type that you may want to feed into the same query. The union operation provided by the CEP server allows you to multiplex several input streams

23

into a single output stream. The following example, combines all events from `stream1` with the events in `stream2` into a single event stream.

```
var unioned = stream1.Union(stream2);
```

User defined functions

The built-in query functionality of the CEP server may not be sufficient in all cases. To allow for domain-specific extensions, queries in the CEP server can invoke user-defined functions that are provided as static functions in .NET assemblies. In the following example, the user-defined function `MyFunctions.valThreshold` is specified in the filter predicate.

```
var filteredStream = from e in stream
                     where e.value < MyFunctions.valThreshold(e.Id)
                     select e;
```

Deployment Models

StreamInsight supports two deployment scenarios for the complex event processing (CEP) server:

- Full integration into the CEP application as a hosted (embedded) DLL.

- As a stand-alone server with multiple CEP applications and users sharing the server. In its stand-alone configuration, the CEP server runs in a wrapper such as an executable, or the CEP server could be packaged as a Windows Service.

Hosted DLL Deployment

The hosted deployment model allows applications to transparently embed the CEP server into their solutions. The embedding application controls all access to the CEP server and thus can prevent others from accessing metadata and the data being processed by the CEP server. The following examples represent scenarios in which the hosted deployment model might be a good fit for your CEP solution:

- You want to use the implicit server development model to develop your application or rely on the IObservable/IObserver programming paradigm to develop your CEP application.

- You want to minimize the memory footprint of your CEP application on the system to which you are planning to deploy your solution.

- You are in the process of developing an application and you prefer to use a single process for both the application you are writing and the CEP server. Switching to the stand-alone server model is easy when you use the explicit server development model with the CEP server.

- Shared access to the metadata and the streaming event data that is being processed by the CEP server is not a requirement.

- You want to tightly control access to the CEP server through your application logic that wraps the CEP server.

Stand-alone Server Deployment

A stand-alone CEP server is preferable when multiple applications need to share the same event data sources and can benefit from access to mutual metadata objects. The following examples are scenarios in which the stand-alone deployment model might be a good fit for your CEP solution:

- You want to share metadata objects such as event types, adapter types, or query templates among multiple applications. The stand-alone server makes it easy for you to keep the metadata consistent between these applications because only a single copy of it is registered in the CEP server.
- A data source is registered with the CEP server and already provides an event stream for an existing application. Using the stand-alone server will make it easy for you to re-use the already registered adapters for the data source and to share the incoming data stream between the applications.

The stand-alone server deployment can use the StreamInsightHost.exe as the server host. Applications written against a stand-alone CEP server must follow the explicit server development model and must connect to the CEP server by using the Web service URI of the CEP server host process.

Monitoring and Troubleshooting

Monitoring the state of a CEP server involves tracking the overall health of the system and query performance. The state of a CEP server is captured by monitoring the CEP queries running on the server; evaluating how the entities that compose a CEP query are utilizing system resources.

Monitoring Diagnostic Views

Monitoring information can be obtained by creating diagnostic views using the **ManagementService** API. You can create diagnostics views that return attributes at the sever level and at the query level. Query-level diagnostics are available for the query template definition and the query instance itself. Server-level diagnostics are available for two server components by using the Event Manager and Plan Manager. All objects in the server are accessed by using Uniform Resource Identifiers (URI) that have a hierarchical naming schema starting with the server and progressing down to query operators and event streams. These values are obtained by calling the **GetDiagnosticView()** method. The **SetDiagnosticsSettings()** and **ClearDiagnosticSettings()** methods can be used to set or clear specific settings for a given query. For more information, see Monitoring the CEP Server and Queries on MSDN.

Query Analysis and Debugging

StreamInsight ships with a stand-alone debugging tool, the Event Flow Debugger. This tool enables you, as a developer or administrator of a complex event processing (CEP) application, to inspect, debug, and reason about the flow of events through a CEP query.

Event flow debugging involves analyzing an event through the passage of time, as it proceeds from one stage of the CEP query to the next; and within a query stage, from one operator to the next. Here,

debugging involves understanding the effects an event has on a stream as it enters and exits from a given operator over time, and how new events are generated as a result of computations on events input into an operator. The emphasis in event flow debugging is on how the operator's semantics (FILTER, PROJECT, JOIN, AGGREGATE, MULTICAST and so on) affect the event, rather than on the (control flow) execution of the operators themselves. As a consequence, the debugger helps understand the impact that a given event has on other events, and the impact of other events on the event being analyzed.

The Event Flow Debugger provides the following key functionalities.

- Ability to view the query plan for a given query; the query operators and the event streams. This can help you understand the performance of the query.

- Ability to inspect all events produced on all streams and search for specific events.

- Ability to step through the trace of a query execution and understand how events propagate through a streaming query.

- Ability to analyze events and understand how they reached a given state - that is, how other events or operators impacted their event times and payloads.

- Ability to analyze the impact that any given event has on events that are downstream from the current operator - essentially to look ahead into the future processing of the events until the event finally affects the output.

- View the query execution statistics in terms of the events that are consumed and produced, their latency and throughput characteristics and memory requirements.

To implement these functionalities, the debugger provides three features for analysis:

- **Step Through Time** - Using this feature, you can step through the event stream one event at a time and watch its progress from one operator to the next.

- **Root Cause Analysis** - Using this feature, you can "look back" at the "root cause" - or the sequence of operations or changes that caused the event to reach its present condition.

- **Event Propagation Analysis –** Using this feature, you can analyze the effects of this event down the stream either in terms of the changes the particular event itself goes through, or in terms of how it impacts other events, or causes the generation of new events. This feature is the reverse of Root Cause Analysis.

In addition, the monitoring dashboard allows you to select various query execution statistics to analyze the latency and throughout characteristics of the events that are consumed and produced and to understand their memory requirements. For more information, see the white paper included in the StreamInsight Samples and Documentation download.

Conclusion

Using Microsoft's StreamInsight platform for complex event processing, you can develop robust event-driven applications with high-performance and scalability.

Bibliography

Barga, Roger S., Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. "Consistent Streaming Through Time: A Vision for Event Stream Processing." *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings.* 2007. 363-374.

For more information:

http://www.microsoft.com/sqlserver/: SQL Server Web site

http://technet.microsoft.com/en-us/sqlserver/: SQL Server TechCenter

http://msdn.microsoft.com/en-us/sqlserver/: SQL Server DevCenter

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

- Are you rating it high due to having good examples, excellent screen shots, clear writing, or another reason?
- Are you rating it low due to poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of white papers we release.

Send feedback.