Microsoft StreamInsight™

A Hitchhiker's Guide to Microsoft StreamInsight Queries

Ramkumar (Ram) Krishnan, Jonathan Goldstein, Alex Raizman (Version: Jun 12 2012)[1]

Abstract: This paper is a developer's introduction to the Microsoft StreamInsight Queries. The paper has two goals: (1) To help you think through stream processing in simple, layered levels of understanding, complementing the product documentation. (2) To reinforce this learning through examples of various use cases, so that you can design the query plan for a particular problem and compose the LINQ query. This ability for top-down composition of a LINQ query, combined with bottom-up understanding of the query model, will help you build rich and powerful streaming applications. The more advanced sections of this paper provide an overview of a query plan, the operators that constitute the query, and, where appropriate, the foundational query algebra itself. The paper is not a feature reference or a substitute for MSDN documentation.

1. Introduction

Enterprises have always been under pressure to reduce the lag between data acquisition and *acting* on the acquired data. Over the past decade, the expansion of free markets has created a massive user base for consumer and business technology – everyone from a billionaire in Seattle to a street hawker in Bangalore has a mobile phone. The Web owns our eyeballs – in six short years, online advertising and its adjoining business models have overtaken traditional media spend. The next generation of productivity and media products will be a seamless mash-up between a user's office and personal life with ubiquitous access to any content from any device. Competitive mandates in manufacturing and worldwide initiatives like Green IT have led to massive volumes of sensor-driven, *machine-born* data. Utility computing from platforms like Windows Azure aim to eliminate technology barriers of scale and economics. These irreversible trends portend a scenario where the success of an enterprise depends on its capability to efficiently respond to every stimulus from its customers and partners. Such a competitive enterprise that seeks a 24x7, 360^{o} view of its *opportunities* will face a torrent of high volume, high velocity business and consumer data.

The singular impact that these trends will have on traditional IT applications is to elevate *the awareness of time*. In this attention-deficit world described above, every arriving data point is *fungible* – *i.e.*, it represents a customer touch-point, an abnormal sensor reading, or a trend point *whose business value rapidly diminishes with time*. For a growing number of applications, this necessitates a shift in mindset where we want to *respond to an opportunity as soon as it presents itself*, based on insight built in an *incremental* manner over time, with the response having an *optimal*, if not always the best, fidelity or returns. In other words, it necessitates a shift towards *event-driven* processing, where you deal with each arriving data point as an event – as something that happened at a point, or over a period, in time – and apply your business logic on this event to respond in a time-sensitive manner to the opportunity that this event represents.

This model suits a growing number of applications with demands of low latency (sub-second response time) and high throughput (handling 100x K events/sec). This includes financial services (risk analytics, algorithmic trading), industrial automation (process control and monitoring, historian based process analysis), security (anomaly/fraud detection), web analytics (behavioral targeting, clickstream analytics, customer relationship management), and business intelligence (reporting, predictive analytics, data cleansing) and more.

StreamInsight is a .NET-based platform for the continuous and incremental processing of unending sequences of such events from multiple sources with near-zero latency. It is a temporal query processing engine that enables an application paradigm where a *standing* (*i.e.*, potentially infinitely running) query processes these *moving* events over windows of time. Processing can range from simple aggregations, to correlation of events across multiple streams, to detection of event patterns, to building complex time series and

analytical models over streaming data. The StreamInsight programming model enables you to define these queries in declarative LINQ, along with the ability to seamlessly integrate the results into your procedural logic written in C#. The goal of this paper is to help you learn, hands-on, how to write declarative queries in LINQ with a clear understanding of reasoning with windows of time.

In this paper, we provide a hands-on, developer's introduction to the Microsoft StreamInsight Queries. The paper has two goals: (1) To help you think through stream processing in simple, layered levels of understanding, complementing product documentation. (2) To reinforce this learning through examples of various use cases – so that you can design the query plan for a particular problem, and compose the LINQ query. This ability for top-down composition of a LINQ query, combined with a bottom-up understanding of the query model, will help you build rich and powerful streaming applications. As you read this paper, please study the actual code examples provided to you in the form of a Visual Studio 2010 solution (HitchHiker.sln).

StreamInsight is an in-memory event processing engine that ships as part SQL Server 2012 and is available separately in the Microsoft Download Center. It requires .NET 4.0 Full Edition, a C# 3.0 compiler, and Windows 7. The engine leverages CLR for its type system and .NET for runtime, application packaging, and deployment. The choice of Visual Studio gives you the benefit of productivity features like IntelliSense. An event flow debugger provides a complete developer experience. You are assumed to have a working knowledge of C# and LINQ.

Part I Understanding Query Development

2. Tutorial

We will start this section with a tutorial example of an application that shows the basic scaffolding required around a StreamInsight query. Keeping this application relatively constant, we will progress through query capabilities – from simple to the more complex ones. To get started, install/enable .NET 4.0 Full Edition, StreamInsight, and Visual Studio 2010 on your Windows 7 machine. Download the code from the same location as this paper, and load up HitchHiker.sln in Visual Studio. Besides the code, keep the documentation on StreamInsight, C#, and LINQ within reach. Keep a checkered notepad and a pencil handy to draw time plots.

2.1 "Hello, Toll!" – Programming a StreamInsight application

A tolling station is a common phenomenon – we encounter them in many expressways, bridges, and tunnels across the world. Each toll station has multiple toll booths, which may be manual – meaning that you stop to pay the toll to an attendant, or automated – where a sensor placed on top of the booth scans an RFID card affixed to the windshield of your vehicle as you pass the toll booth. It is easy to visualize the passage of vehicles through these toll stations as an *event stream* over which interesting operations can be performed.

In `HelloToll.cs`, we show an application built around a set of demo queries which we will go over in more details later in this paper. The "Program plumbing" region demonstrates basic steps needed to run a query in StreamInsight. We will walk through this example step by step. Some code regions that are not immediately relevant to understand how a query fits into an application are collapsed for brevity and appear as `...` in the example below.

```
...
static void TumblingCount(Application app)
{
    var inputStream = app.GetTollReadings();
    var query = from win in inputStream.TumblingWindow(TimeSpan.FromMinutes(3))
                select win.Count();  ❸
    app.DisplayIntervalResults(query);
}
...

#region Program plumbing
...
static IQStreamable<TollReading> GetTollReadings(this Application app)
{
    return app.DefineEnumerable(() =>
        // Simulated readings data defined as an array.
        // IntervalEvent objects are constructed directly to avoid copying.
        new[]
        {
            IntervalEvent.CreateInsert(
                new DateTime(2009, 06, 25, 12, 01, 0),
                new DateTime(2009, 06, 25, 12, 03, 0),
                new TollReading  ❶
                {
                    TollId = "1",
                    LicensePlate = "JNB 7001",
                    State = "NY",
                    Make = "Honda",
                    Model = "CRV",
                    VehicleType = 1,
                    VehicleWeight = 0,
                    Toll = 7.0f,
                    Tag = ""
                }),
            ...
        })
        // Predefined AdvanceTimeSettings.IncreasingStartTime is used
        // to insert CTIs after each event, which occurs later than the previous one.
        .ToIntervalStreamable(e => e, AdvanceTimeSettings.IncreasingStartTime);  ❷
    }

static void DisplayPointResults<TPayload>(
    this Application app,
    IQStreamable<TPayload> resultStream)
{
    // Define observer that formats arriving events as points to the console window.
    var consoleObserver = app.DefineObserver(() =>
        Observer.Create<PointEvent<TPayload>>(ConsoleWritePoint));

    // Bind resultStream stream to consoleObserver.
    var binding = resultStream.Bind(consoleObserver);  ❹

    // Run example query by creating a process from the binding we've built above.
    using (binding.Run("ExampleProcess"))  ❺
```

```
        {
            ...
        }
    }
```

We implement a running StreamInsight query essentially in five logical steps. For now, please focus on the code fragments – we will explain the rationale behind the various choices made in this application as a next step.

❶ Each event in StreamInsight consists of two components – the *event payload*, which reflects the data values carried by the event, and the *event shape* – *i.e.*, the temporal nature of the event as it models an actual occurrence. The event shape defines the lifetime of the event – *i.e.*, the duration for which the values in the payload last along the forward progression of time. In this example, `TollReading` is a C# class that represents the payload for events input into the query. The query output is of primitive event type `long`.

❷ Define the input stream of events as a function of event payload and shape. We defined the payload component in the above step. Next, we model the act of a vehicle crossing the toll booth as an *interval* of time (via combination of `ToIntervalStreamable` function and `IntervalEvent` type). We complete the input stream definition by wrapping sample data array into enumerable sequence using `DefineEnumerable` function. StreamInsight allows the query engine to interface with practically any input or output device using `IEnumerable`, `IObservable`, or StreamInsight specific units called *adapters*.

❸ Define the query logic itself, which is expressed as a LINQ statement that declaratively describes the processing on events flowing into the engine through `inputStream`. The query produces an output stream of events with payload of type `long`. We will discuss the query logic in the next section. Learning how to write such queries is the central focus of this paper. For now, the key takeaway is that query logic allows you to define the event processing intent purely based on your knowledge of input and output event payload and shape definitions – isolated from the physical aspects of event delivery and output.

❹ "Bind" the query logic to the consumer. The consumer in our example just outputs formatted events to the console using the simple observer `consoleObserver` defined using `DefineObserver` method. The query logic result is then bound to it using the `Bind` method. The output events are interpreted as point events due to usage of `PointEvent` type in the `DefineObserver` and `ConsoleWritePoint` methods. The `ConsoleWritePoint` method provides actual implementation for outputting the events to the console. All that is remaining is to create a process.

❺ Running the query. In order to start evaluation of the query logic we need to create a process by calling the `Run` method on the binding. From this point on, unless it is stopped by a subsequent call to `Dispose` on this process, or abnormally terminated by some runtime condition, this *standing query* will continue to run in perpetuity.

This programming model enables a flexible and rapid Dev-Deploy cycle. First, the concept of a query template allows you to *declaratively* express your business logic, isolated from concerns of event delivery and output. The combination of LINQ with C# allows you to code both the declarative and procedural aspects in a single programmatic environment. The Visual Studio IDE gives you the convenience of IntelliSense for code completion, code analysis, refactoring and other features for rapid development. All the steps leading up to query start are resolved at compile time, minimizing the number of surprises to deal with at runtime.

While this simple example shows you anonymous entities, the full Object Model API allows you to register each discrete entity that is part of a process in the StreamInsight metadata. Given a shared knowledge of event payload and shape, query and source/sink developers can work independently to build and ship their modules as .NET assemblies. The product is based on a .NET platform, so deployment simply entails locating these assemblies with the StreamInsight libraries. An application integrator can write a wrapper program like `HelloToll.cs` that connects to a server, binds a given query logic with sources and sinks, and creates a process.

2.2 Query Development Process

In this section, we present a five step process to be used as a general guideline to implement your StreamInsight queries. The key goals of this process are to help you develop the right mindset when approaching a real-world problem, and to illustrate the use of StreamInsight query features in arriving at the solution. The steps are:

Step 1 – Model the payload and shape of input and output events from the application's perspective
Step 2 – Understand the required query semantics by building sample input and output event tables
Step 3 – Gather elements of the query logic to develop an event flow graph and compute the output
Step 4 – Compose the query as a streaming transformation of the input to output
Step 5 – Specify the timeliness of query output in consideration of its correctness

We will repeat this process for each query we develop in this example.

Let's try to give a definition for the first query in the `HelloToll.cs` example:

[Tumbling Count] Every 3 minutes, count the number of vehicles processed at the toll station since the last result. Report the result at a point in time, at the end of the 3 minute window.

2.2.1 Step 1 – Model input and output events from the application's perspective

StreamInsight allows you to model the event *shape* to reflect happenings in real-life – as required by the application. As we discussed in ❷, we model the input event as an *interval* of time based on the assumption that many toll stations are manned, and we want to account for the few seconds to a minute[2] of interaction between an attendant and a motorist. The StartTime and EndTime of each event mark a vehicle's arrival at, and departure from, a toll booth respectively.

The output events containing the counts are modeled as *Point events*, reflecting the fact that we would know the count value only at the end of each 3 minute interval.

2.2.2 Step 2 – Understand required query semantics by building sample input and output event tables

StreamInsight queries function based on a strong foundational query algebra that treats every event as an interval event, irrespective of the shape defined at the programmatic level. In this step, we tabulate a representative sample of input events *as the query engine would see it* in order to understand the query semantics.

A *canonical history table* (or CHT) is a simple tabulation of a set of sample events, with each event showing the StartTime and the EndTime of the event, along with the relevant payload fields. Here, given that the output is just a simple count of events (rather than some aggregate computation based on payload values), we skip showing the payload values. The variety of input interval durations helps in building an understanding of the query's semantics. To further simplify our example we will consider only events generated by toll booth number 1.

Table 1. Input events canonical history table for example [Tumbling Count] for toll booth 1.

Event	StartTime	EndTime	Payload TollId	Payload VehicleId
e1	12:01	12:03		
e2	12:02	12:03		

[2] For simplicity we constructed a sample dataset to be rounded to minutes, *i.e.*, StartTime and EndTime always have 0 seconds.

e3	12:03	12:08		
e4	12:07	12:10		
e5	12:10	12:14		
e6	12:11	12:13		
e7	12:20	12:22		
e8	12:22	12:25		

It is useful to graph CHTs on a time plot. A time plot is a simple X/Y plot where the X axis marks the progression of time, and Y is simply a space to place the events as they arrive. The markings on the X axis denote minutes past 12:00, *e.g.*, X=4 corresponds to 12:04.

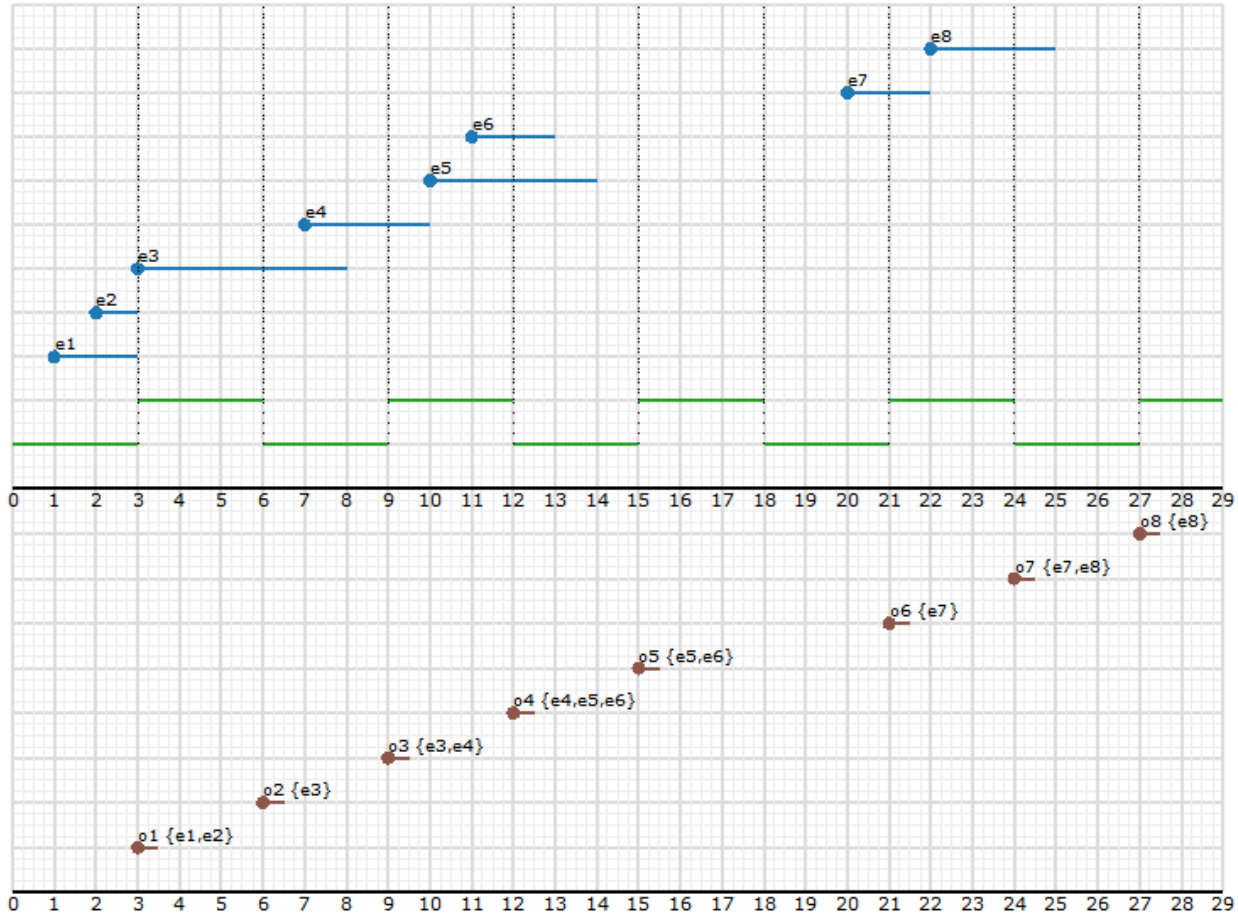


Figure 1. Time plot for input events, a tumbling window of size 3, and result counting vehicles being processed

With the input events mapped on the time plot, we will build an output CHT that reflects the query requirements stated in **[Tumbling Count]** above. The problem requires a count to be reported once every three minutes. We therefore partition time into 3 minute intervals as shown on the time plot above. The green lines depict a *tumbling window* of size 3 minutes. Now, the problem statement is a bit ambiguous. Are we counting the number of vehicles which *arrived during this period*? Or perhaps the number of vehicles *which were being processed at some point during that period*? Or is it the number of vehicles *which completed processing during that period*? Such subtle ambiguities will be brought to light when we construct example output CHTs.

All intervals and windows in StreamInsight are closed at the left, open on the right – meaning that an event's value at the start time of the interval or window is included in every computation, but the value at the end time is excluded. Since the result of a computation over a window can be known only at the end of the window interval we want the output as of a point in time, showing the events processed over the window interval. From the query engine's perspective, a point event is nothing but an interval event with a lifetime of one chronon (one .NET tick, equivalent to 100 nanoseconds, which we will represent as ε). In other words, assume we had input events as of a point in time. Then we can represent them as interval events in the CHT with each event having an EndTime that is StartTime + ε. Results depicted on Figure 1 below the axis are points. We depict ε as a small line smaller than a unit of time represented on the time plot. The output events are annotated with a set of events over which this output is computed for easier cross referencing.

Let's revisit the earlier problem definition to remove the ambiguity (note the emphasized part).

[Tumbling Count] Every 3 minutes, report the number of vehicles processed *that were being processed at some point during that period* at the toll station since the last result. Report the result at a point in time, at the end of the 3 minute window.

The comments in the table on the right hand side show the events that contributed to the result during the associated window of time.

Table 2. Output CHT for [Tumbling Count]

Event	StartTime	EndTime	Payload Count		Events	Comment
o1	12:03	12:03 +ε	2		e1,e2	e3 excluded because window interval is open at the right
o2	12:06	12:06 +ε	1		e3	e1, e2 are excluded, since event interval is open at the right
o3	12:09	12:09 +ε	2		e3, e4	
o4	12:12	12:12 +ε	3		e4, e5, e6	
o5	12:15	12:15 +ε	2		e5, e6	
	12:18	12:18+ε				Note that no event is produced for this window.
o6	12:21	12:21+ε	1		e7	
o7	12:24	12:24+ε	2		e7, e8	
o8	12:27	12:27+ε	1		e8	

To provide some perspective, if we had decided to output the number of vehicles *which arrived during* a given 3 minute period, output events would have the following count values, taking only the event start times into account. The query required to generate this output will look very different.

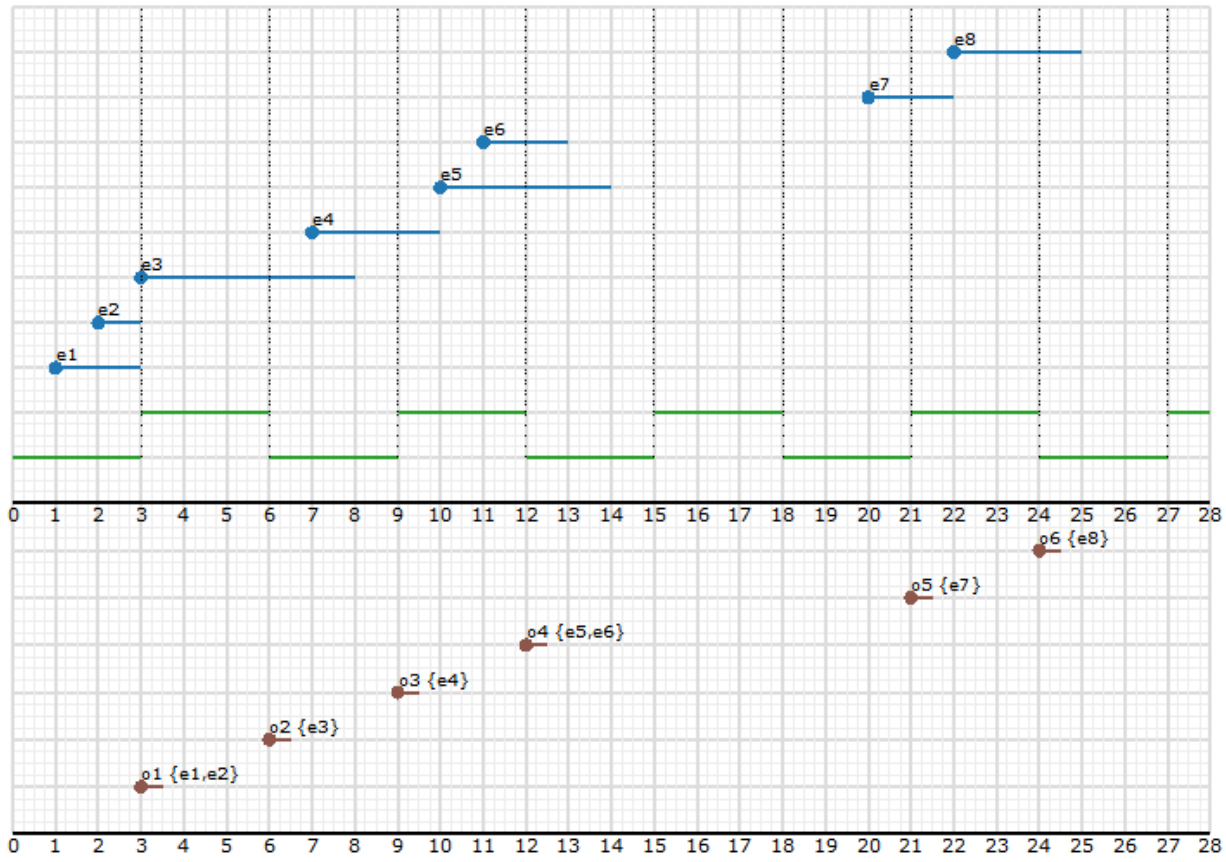


Figure 2. Time plot for input, a tumbling window of size 3, and result counting arrival of the vehicles

Note that even though e3's StartTime aligns with the window's end time, since the window interval is open-ended, e3 cannot be added to this window count. Also, result o3 does not include e3, even though e3 overlaps with this window, because this window doesn't contain e3's StartTime.

If our interest was in the count of events that *completed processing* during a window, the output CHT would have the following counts. Again, the query that needs to be composed to get this output would be very different. In fact let's consider two variations here.

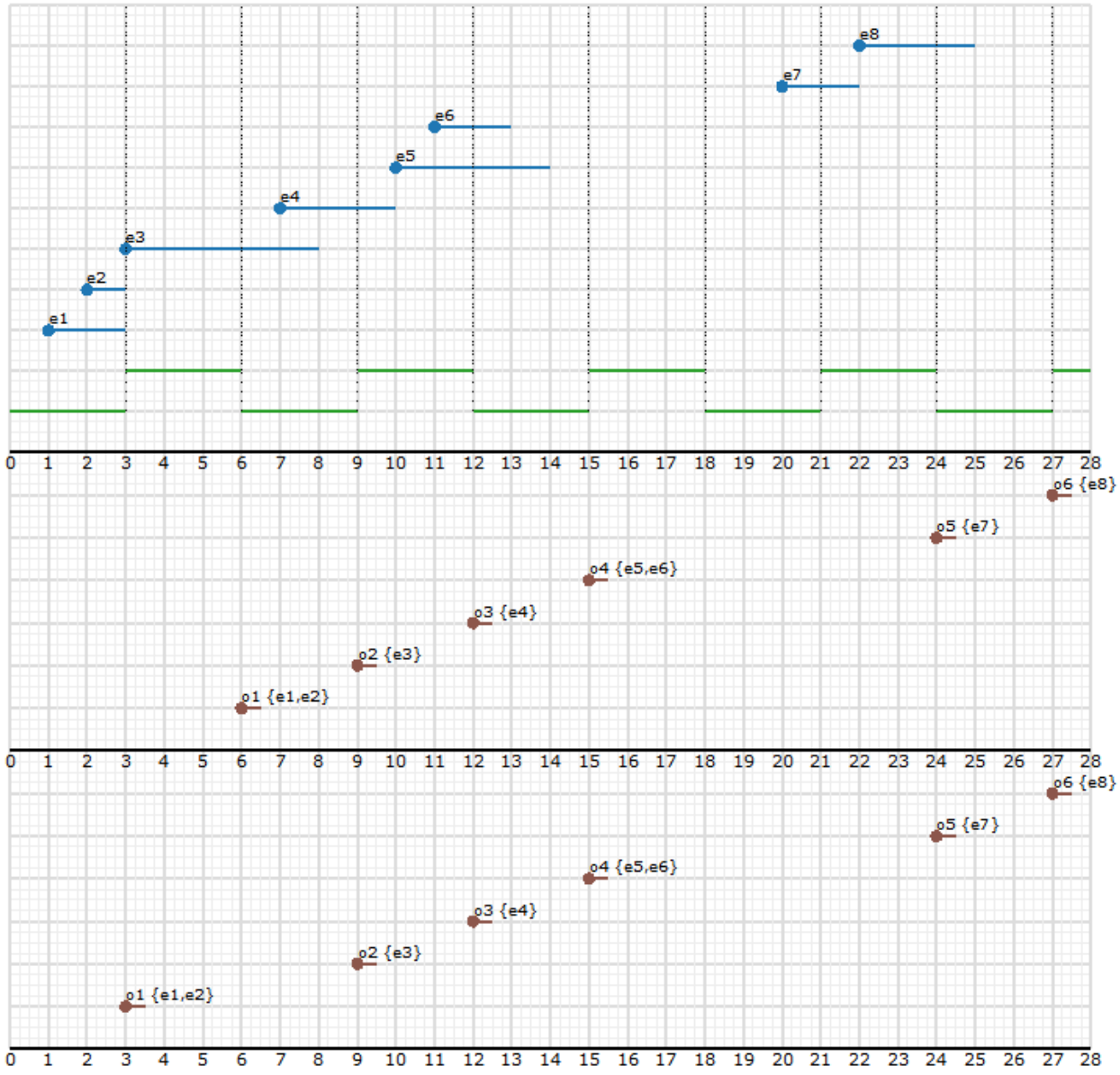


Figure 3. Time plot for input, a tumbling window of size 3, and two variations of results counting completion of the vehicle processing

If we are to consider the EndTime of each event as a point and count these points, we end up with the result shown right below the input events on the time plot, which may seem odd for the problem definition. The vehicle processing represented by e1 and e2 are accounted for only at time 12:06. The more plausible results can be obtained if EndTimes were to be shifted back by ε. The latter result is shown on the bottom-most track on the time plot. Notice that the only difference is in the very first result o1. Let's trace what happens to e1 and e2 here: If we consider the EndTime as a point representing the event, then we consider the StartTime of these points to be 12:03. As we have discussed, these will not be accounted for in the first window, but rather they will be picked up in the second window causing the result to appear at 12:06. On the other hand, if the EndTime is pushed back by ε, these events would be accounted for in the first window. Shifting event time by ε only affects computation for events that align to the window boundaries.

Now that we have decided how we want the output to look, the next step is …

2.2.3 Step 3 – Gather elements of query logic to compute the output and develop an event flow graph

There are three main elements for this query:

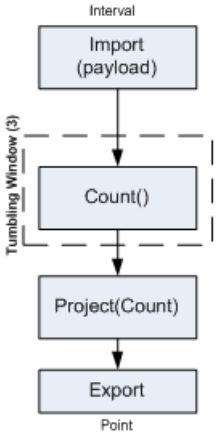


Figure 4. Query Graph for the [Tumbling Count] query

Window – Windows are the mechanism by which events are grouped together into sets. In this case, we chose a 3 minute tumbling window, which means that our windows consist of consecutive non-overlapping 3 minute periods of time, as described above.

Count() – Computation done within a window – we count the number of events in each window using this built-in aggregate.

Project – We project the results out to the output stream, in this case, without any further transformations to the value.

The query graph is a simple sketch of the event flow through these operators. Chart the query graph *from top to bottom* to align yourself with the query graphs displayed by the Event Flow Debugger.

2.2.4 Step 4 – Compose the query as a streaming transformation of input into the output

Here is how the query reads: For each tumbling window win of size 3 minutes defined on `inputStream`, compute the count of events and project it out as an event with payload of type `long`.

```
var query = from win in inputStream.TumblingWindow(TimeSpan.FromMinutes(3))
            select win.Count();
```

2.2.5 Step 5 – Consider the timeliness balanced against correctness of query output

This is a critical step which requires specification on the output delivery based on an understanding of the behavior of the input event stream(s). We will discuss this step in detail under the section “13. Step 5 – Time and Commitment” later in the document. For now, please overlook the event entries marked “CTI” in the output display in the examples below.

2.3 Understanding the Query Output

The input events streamed into the `HelloToll` query are shown in Figure 5. Each line has StartTime and EndTime values, followed by the values for the six payload fields. In the example program, input data is emulated as an array. The output events generated from an actual run are shown in Figure 6. On the output, each INSERT signifies an insertion of an event from the query into the output stream. Let's analyze the output: The first INSERT event signifies the output for the first tumbling window, reported with the EndTime of the window (12:03), and reflects a count value of 3, computed based on the inputs:

INSERT	6/25/2009 12:01:00	06/25/2009 12:03:00	1	JNB 7001	NY	Honda	CRV	1	0	7.0	
INSERT	6/25/2009 12:02:00	06/25/2009 12:03:00	1	YXZ 1001	NY	Toyota	Camry	1	0	4.0	123456789
INSERT	6/25/2009 12:02:00	06/25/2009 12:04:00	3	ABC 1004	CT	Ford	Taurus	1	0	5.0	456789123

The second INSERT event with a count of 4 is the output for the second tumbling window, and includes the following events:

INSERT	6/25/2009 12:02:00	06/25/2009 12:04:00	3	ABC 1004	CT	Ford	Taurus	1	0	5.0	456789123
INSERT	6/25/2009 12:03:00	06/25/2009 12:07:00	2	XYZ 1003	CT	Toyota	Corolla	1	0	4.0	
INSERT	6/25/2009 12:03:00	06/25/2009 12:08:00	1	BNJ 1007	NY	Honda	CRV	1	0	5.0	789123456
INSERT	6/25/2009 12:05:00	06/25/2009 12:07:00	2	CDE 1007	NJ	Toyota	4x4	1	0	6.0	321987654

Event	StartTime	EndTime	Toll Id	License Plate	State	Make	Model	Vehicle Type	Vehicle Weight	Toll	Tag
INSERT	6/25/2009 12:01:00	06/25/2009 12:03:00	1	JNB 7001	NY	Honda	CRV	1	0	7.0	
INSERT	6/25/2009 12:02:00	06/25/2009 12:03:00	1	YXZ 1001	NY	Toyota	Camry	1	0	4.0	123456789
INSERT	6/25/2009 12:02:00	06/25/2009 12:04:00	3	ABC 1004	CT	Ford	Taurus	1	0	5.0	456789123
INSERT	6/25/2009 12:03:00	06/25/2009 12:07:00	2	XYZ 1003	CT	Toyota	Corolla	1	0	4.0	
INSERT	6/25/2009 12:03:00	06/25/2009 12:08:00	1	BNJ 1007	NY	Honda	CRV	1	0	5.0	789123456
INSERT	6/25/2009 12:05:00	06/25/2009 12:07:00	2	CDE 1007	NJ	Toyota	4x4	1	0	6.0	321987654
INSERT	6/25/2009 12:06:00	06/25/2009 12:09:00	2	BAC 1005	NY	Toyota	Camry	1	0	5.5	567891234
INSERT	6/25/2009 12:07:00	06/25/2009 12:10:00	1	ZYX 1002	NY	Honda	Accord	1	0	6.0	234567891
INSERT	6/25/2009 12:07:00	06/25/2009 12:10:00	2	ZXY 1001	PA	Toyota	Camry	1	0	4.0	987654321
INSERT	6/25/2009 12:08:00	06/25/2009 12:10:00	3	CBA 1008	PA	Ford	Mustang	1	0	4.5	891234567
INSERT	6/25/2009 12:09:00	06/25/2009 12:11:00	2	DCB 1004	NY	Volvo	S80	1	0	5.5	654321987
INSERT	6/25/2009 12:09:00	06/25/2009 12:16:00	2	CDB 1003	PA	Volvo	C30	1	0	5.0	765432198
INSERT	6/25/2009 12:09:00	06/25/2009 12:10:00	3	YZX 1009	NY	Volvo	V70	1	0	4.5	912345678
INSERT	6/25/2009 12:10:00	06/25/2009 12:12:00	3	BCD 1002	NY	Toyota	Rav4	1	0	5.5	876543219
INSERT	6/25/2009 12:10:00	06/25/2009 12:14:00	1	CBD 1005	NY	Toyota	Camry	1	0	4.0	543219876
INSERT	6/25/2009 12:11:00	06/25/2009 12:13:00	1	NJB 1006	CT	Ford	Focus	1	0	4.5	678912345
INSERT	6/25/2009 12:12:00	06/25/2009 12:15:00	3	PAC 1209	NJ	Chevy	Malibu	1	0	6.0	219876543
INSERT	6/25/2009 12:15:00	06/25/2009 12:22:00	2	BAC 1005	PA	Peterbilt	389	2	2.675	15.5	567891234
INSERT	6/25/2009 12:15:00	06/25/2009 12:18:00	3	EDC 3109	NJ	Ford	Focus	1	0	4.0	198765432
INSERT	6/25/2009 12:18:00	06/25/2009 12:20:00	2	DEC 1008	NY	Toyota	Corolla	1	0	4.0	
INSERT	6/25/2009 12:20:00	06/25/2009 12:22:00	1	DBC 1006	NY	Honda	Civic	1	0	5.0	432198765
INSERT	6/25/2009 12:20:00	06/25/2009 12:23:00	2	APC 2019	NJ	Honda	Civic	1	0	4.0	345678912
INSERT	6/25/2009 12:22:00	06/25/2009 12:25:00	1	EDC 1019	NJ	Honda	Accord	1	0	4.0	

Figure 5. Input events in HelloToll.cs

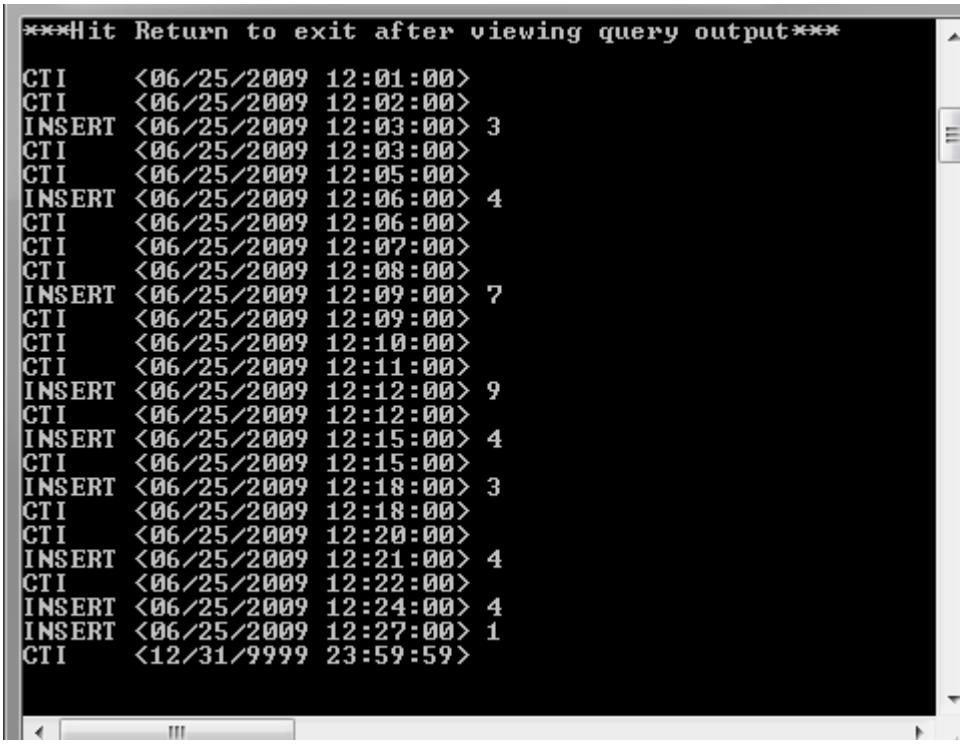


Figure 6. Output events from [Tumbling Count] in HelloToll.cs

Summary: In this section, we covered the following concepts:

1. The anatomy of a StreamInsight application and simple query.
2. A five step process for query development.
3. The use of canonical history tables to visualize the input and output, clarify a problem statement, and write the correct query.
4. An application can model an event as an interval or point event.
5. The query engine internally treats every event as an interval event with a start and end time.
6. Understanding the output of a simple query.

3. More Windows – Hop, Skip and Jump

A tumbling window has the advantage of simplicity – it splits time into buckets of equal width over which aggregates can be easily computed, and it is easy to reason about the events "belonging" to a particular bucket. But it is good to be aware of some of the limitations of this windowing construct.

There is an inherent latency in reporting of the results, equal to the size of the window. In the above example, assume that a burst of 50 vehicles arrives at the tolls at 12:09. This indicator of impending congestion will be reported at time 12:12 – three minutes later.

To track the movement of vehicles at a finer granularity of time, we can use a generalized version of tumbling window, called **Hopping Window.** It is a fixed sized window which hops along the timeline with a particular

duration. The tumbling window is a special case of a hopping window where the hop size is equal to the window size.

Let's recast the first query in the `HelloToll.cs` example, [Tumbling Count], to use hopping windows instead. We will incorporate some precision into the problem statement (emphasized) based on what we have learned from Step 2 of the query writing process above. Rather than show the result every 3 minutes, we will show the result every minute, but still compute the count over a 3 minute window. So the new problem statement becomes:

[Hopping Count] Report the count of vehicles *being* processed *at some time over a* 3 minute window, with the window moving in one minute hops. Provide the counts as of the last reported result as of a point in time, reflecting the vehicles processed *over the last 3 minutes*.

Let's walk through the query development process again.

Step 1 Model input and output events from the application's perspective. No change from [Tumbling Count].
Step 2 Understand the desired query semantics by constructing sample input and output CHTs.

The sample input events are the same as for [Tumbling Count]. The time plot for this input is shown below. The green lines above the time axis show the progression of a hopping window of size 3 minutes, with a hop size of 1 minute. For convenience, each window is annotated with the output event name associated with the window. The annotation is placed right above the window start.
For the output, we want the query to determine the output as of a point in time, showing the events processed *over the last 3 minutes*. We will specify this interval-to-point transformation in the query logic.
The output events for this sample input are shown in the bottom track of the time plot. Just like in previous example, the output events are annotated with a set of events over which this output is computed.

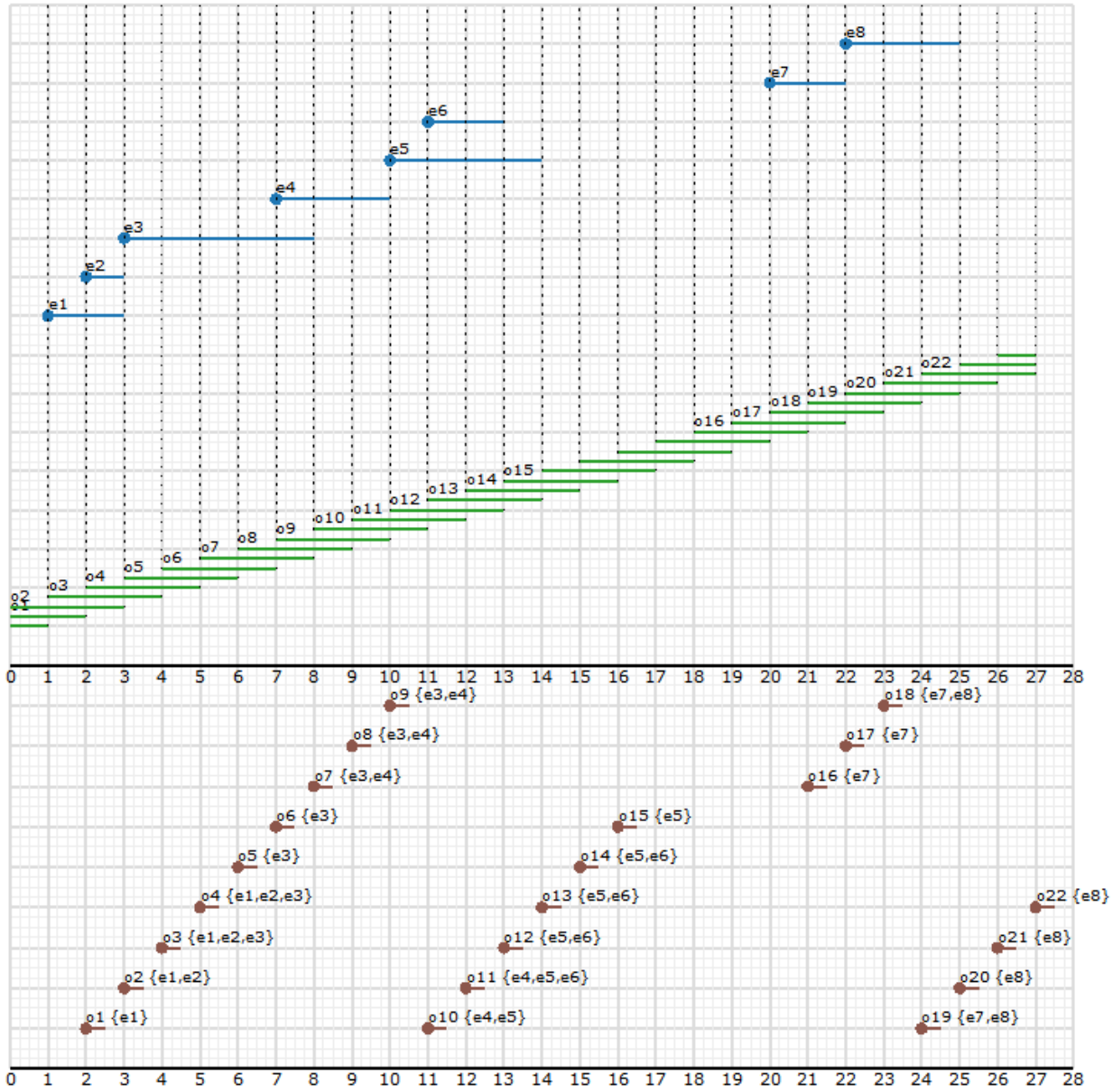


Figure 7. Time Plot for Hopping Window of size 3 and hop size 1 [Hopping Count]

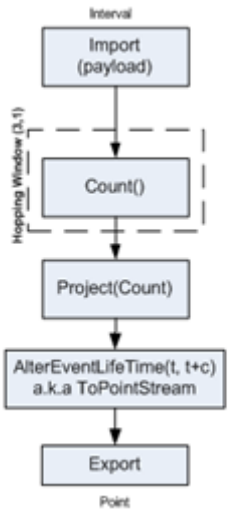


Figure 8. Query Graph for the [Hopping Count] query

The result can be reasoned about in the same terms as we did for the tumbling window. Let's reiterate a few observations: There are no events produced for windows ending at 12:17 to 12:20, since there are no input events overlapping those and we do not generate events for 0 counts. Just like in the case of tumbling windows, e1 and e2 are not included in the window starting at 12:03, since intervals are open at the end.

Step 3 Gather elements of query logic to compute the output and develop an event flow graph.

The new elements in the query compared to [Tumbling Count] are:

- Window – we choose a hopping window of size 3 minutes, with a hop size of 1 minute – as per requirement.
- Interval to Point Event Conversion – Recall that the query engine always deals with interval events. We are required to report the output as of a point in time at the end of a window.

Step 4 Compose the query as a streaming transformation of input to output.

The resulting query is an example of compile time query composition in StreamInsight. First, **countStream** is composed over events provided by **inputStream** – it defines a Count() computation over hopping windows of size of 3 minutes, and a hop size of 1 minute. Next, the result is composed over **countStream** by applying `ToPointEventStream` which converts each event in the stream from an interval to a point event. This has the effect of truncating the EndTime of the output event to StartTime + ε (one chronon).

```
var countStream = from win in inputStream.HoppingWindow(TimeSpan.FromMinutes(3), TimeSpan.FromMinutes(1)
                  select win.Count();
var query = countStream.ToPointEventStream();
```

`ToPointEventStream` is syntactic sugar on top of an operator called `AlterEventDuration`, which is based on a more general operator called `AlterEventLifeTime`. We will see these operators in the upcoming examples.

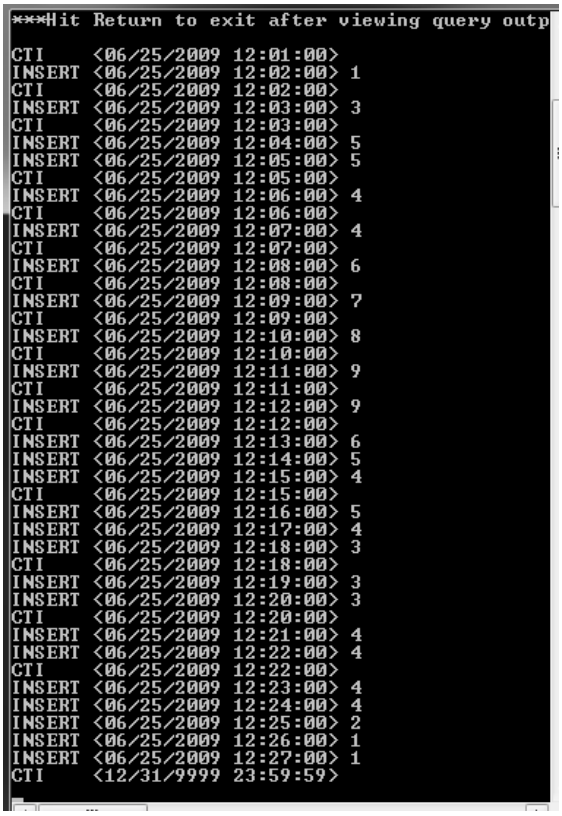


Figure 9. Output from [Hopping Count]

The output from the complete query in HelloToll.cs is shown in Figure 9.

For now, it is clear that a hopping window finds use in scenarios where you need to segment time into equal size windows, determine the membership of events in these windows, and output the result in granules of time (hops) that are less than the window size. It should also be clear that a tumbling window is a special case of hopping window – where the window size and the hop size are the same.

Hopping windows have the advantage of predictable memory consumption over the stream's lifetime, given that output is released at a fixed frequency. Hopping windows are the principal mechanism in the StreamInsight release for time-weighted computations (discussed later). `HoppingWindow` supports more overloads to specify alignment of the window to a reference timeline, and input and output policies for treating event membership in the windows.

On the flip side, hopping windows, like tumbling windows, are chatty in situations where the window or event intervals are long, and/or when events arrive in a sporadic manner. This is because it is the window's properties – rather than that of the incoming event – that controls the output behavior. For example, in the output above, {o5, o6} and {o12, o13, o14} output the occurrence of the same event. A smaller hop size only exacerbates the problem.

More importantly, at high event rates, hopping windows defined for aggregations over *groups of events* can tend to be expensive in terms of state maintenance and computation time. In StreamInsight, only the built-in aggregates such as Sum, Count, and Average are incremental in nature. But any custom aggregation (via a feature called UDA – user defined aggregate) defined by you could be expensive, by virtue of having to repeat the computation over the same set of events that have already been seen in a previous window. Future support for incremental aggregates will remove this limitation – but for StreamInsight, it is good to be aware of this constraint.

4. Partitioning Event Streams for Scale out computations

So far, we have based our examples on a single toll booth. A toll station has several toll booths, and most highways or urban areas will have multiple toll stations. For example, there are 20 such toll stations in a 20 mile radius around New York City that collectively handle 1.4 million crossings a day. (((1.4x10^6/20)/24)/3600) implies an average of a vehicle per minute at a toll station – which is not much. But if you assume that most toll stations are integrated with sensor-based traffic monitoring systems on main arteries and roadways, the volume of events and the kinds of interesting queries that you can ask will demand a scalable querying mechanism. Some examples:

- Which toll stations are congested – now, over the last N minutes/hours? (integration with traffic management)
- Which toll booths are most active in a given station? (this helps in planning road work, shifts, routing)
- What is the inbound (from suburbs to city) versus outbound (city to suburbs) volume of vehicles at a toll station?
- What is the count of transgressions (vehicles with no/expired tags) for a given automated booth at a given station?

These questions require grouping and partitioning of vehicles across different toll stations and possibly other logical hierarchies of related objects in the application. The computations themselves may be far more sophisticated than the simple count aggregate we have seen so far. The Group and Apply operator in StreamInsight addresses this need for partitioned, scaled-out processing. We will explain this feature and introduce a new windowing construct using a simple grouping query:

[Partitioned Hopping window] Find the toll generated from vehicles being processed at each toll booth at some time over a 3 minute window, with the window advancing in one minute hops. Provide the value as of the last reported result.

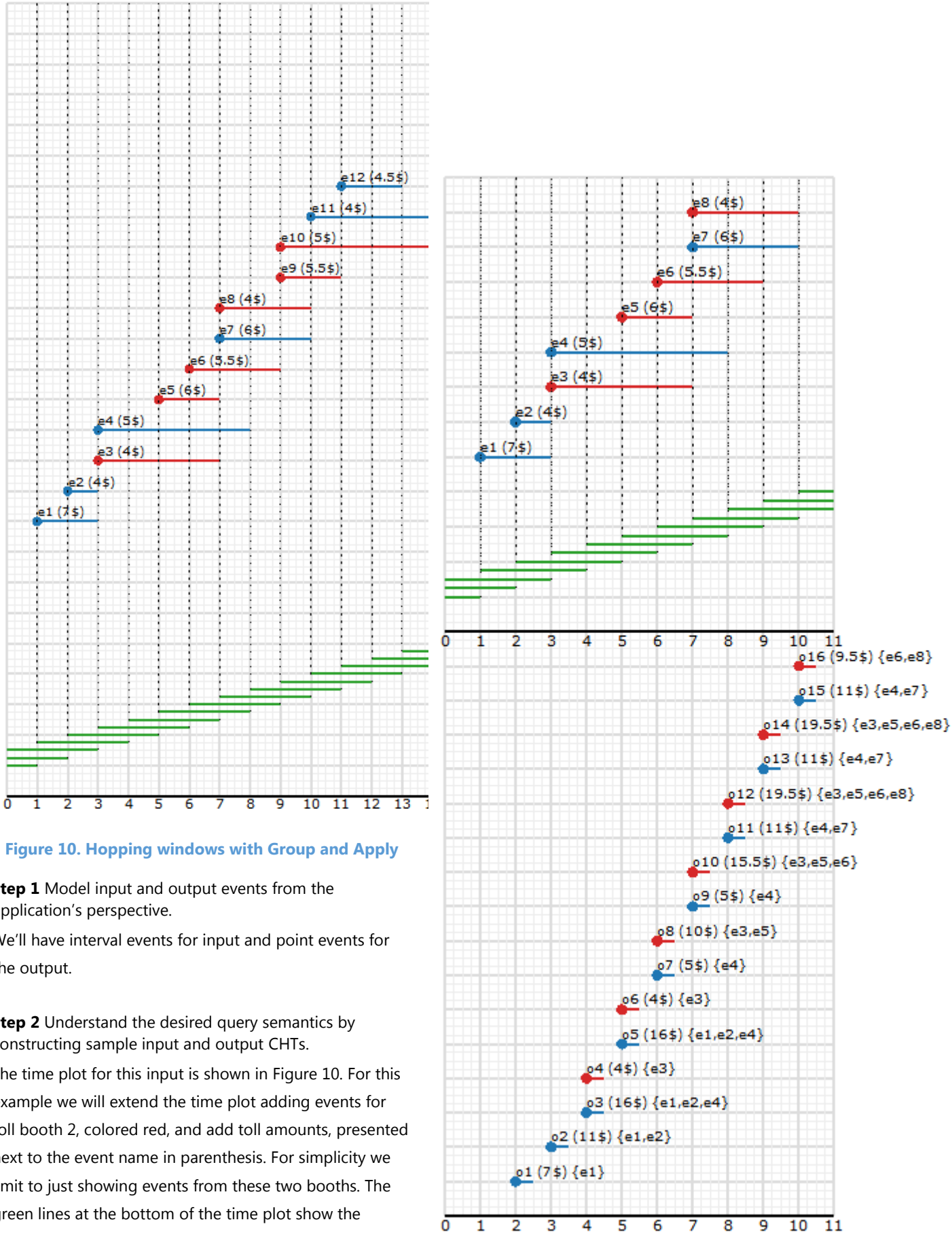


Figure 10. Hopping windows with Group and Apply

Step 1 Model input and output events from the application's perspective.

We'll have interval events for input and point events for the output.

Step 2 Understand the desired query semantics by constructing sample input and output CHTs.

The time plot for this input is shown in Figure 10. For this example we will extend the time plot adding events for toll booth 2, colored red, and add toll amounts, presented next to the event name in parenthesis. For simplicity we limit to just showing events from these two booths. The green lines at the bottom of the time plot show the

progress of a hopping window of size 3 minutes, with a hop size of 1 minute.

In this query, the output is the sum of the toll generated in each hopping window and is computed on a per-booth basis. As we seen earlier, hopping windows are very chatty and generate a lot of output, so for brevity, we will not show the complete result here. A partial result is shown on the bottom track of the time plot in Figure 12. For clarity we colored events according to the booth group from which they were produced. They are also annotated with the resulting toll sum and the set of events that contributed to it, as we did in previous examples.

The important takeaways from the time plot are:

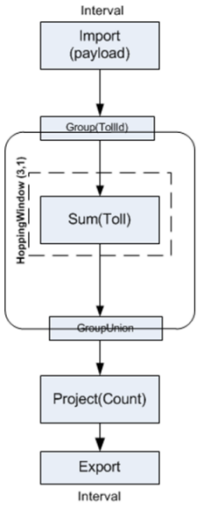


Figure 11. Query Graph for [Partitioned Hopping window] query

- An event output is generated for each hopping window for each group of events (in this case 2 groups).
- The aggregate operation (Sum) is repeated for each hopping window.
- The same set of events repeat themselves in some windows. For example, events o11, o13, and o15 compute the Sum() operation over the same set of two events { e4, e7 }.

Figure 12. Partial output events from [Partitioned Hopping window] query

Step 3 Gather elements of query logic to compute the output and develop an event flow graph.

- Group and Apply – we need a grouping operator to group the events in the stream based on Booth ID.
- Window – we choose a hopping window of size 3 minutes, with a hop size of 1 minute, as per the requirements.
- Sum() – we use this aggregation operator to return the toll amount corresponding to a window

Step 4 Compose the query as a streaming transformation of input to output.

In **[Partitioned Hopping window]**, the grouping clause groups events by TollId into groups per toll booth. A hopping window is defined on each group, and within each hopping window a Sum is computed against the value of the Toll. As you can see, the query formulation is very intuitive and follows the English description to a tee.

```
var query = from e in inputStream
            group e by e.TollId into perTollBooth
            from win in perTollBooth.HoppingWindow(TimeSpan.FromMinutes(3), TimeSpan.FromMinutes(1))
            select new Toll
            {
                TollId = perTollBooth.Key,
                TollAmount = win.Sum(e => e.Toll)
            };
```

In this example, we use Sum as the aggregate. Built-in aggregates are incremental, so this is not much of a problem. While this may seem trivial in this example, aggregates in real-life scenarios can be fairly complex. Each computation has to load up the same set of events in its state. High volume streams in applications like trading platforms can have 10's of 1000's, if not millions, of such groups. The choice of hopping window in such scenarios can become expensive.

4.1 Snapshot Window and AlterEventDuration to form Sliding Windows

A windowing construct whose size and forward progression is truly influenced by the incoming events, rather than a static window definition, is the **Snapshot Window**. Rather than partitioning time into fixed-size buckets, a snapshot window segments time according to the start and end timestamps of all occurring changes. The duration of a snapshot window is defined by a pair of closest event end points – in terms of the event StartTime and EndTime timestamps – from the same or different events. To clarify, consider two events with start and end timestamps {A, 12:10, 12:30} and {B, 12:15, 12:25}. This yields three snapshot windows {w1, 12:10, 12:15}, {w2, 12:15, 12:25} and {w3, 12:25, 12:30}. Thus, unlike a hopping window, the time span of a snapshot window will not contain the start time or end time stamps for any individual event – the events themselves define the window.

To explain this, we will recast [Partitioned Hopping Window] to use the snapshot window instead.

[Partitioned Sliding window] Find the most recent toll generated from vehicles being processed at each station over a one minute window reporting the result *every time a change occurs in the input*.

Step 1 Model input and output events from the application's perspective. No change from [Partitioned Hopping window].
Step 2 Understand the desired query semantics by constructing sample input and output CHTs

The phrase "every time a change occurs" in the above statement points us towards snapshot, rather than hopping, windows. Along with this, we are required to report the most recent toll amount collected over a one minute window. In other words, we need to find the aggregate toll amount *over the last snapshot window lasting one minute from any given point in time along the time axis*. What we have just done is to essentially request a **Sliding Window** - a window that slides along the time axis, but with the output being reported whenever an event arrives into the system.

To compute the aggregate, you want to pick a point on the time axis - say, "Now" - and look back one minute. But recall that it is the events that define the snapshot. So how do we simulate the idea of "look for events back one minute"? The answer is simple – for each event that arrives, you make the *events look forward one minute*. In other words, you change the duration (*i.e.*, the lifetime) of the event ahead by one minute.

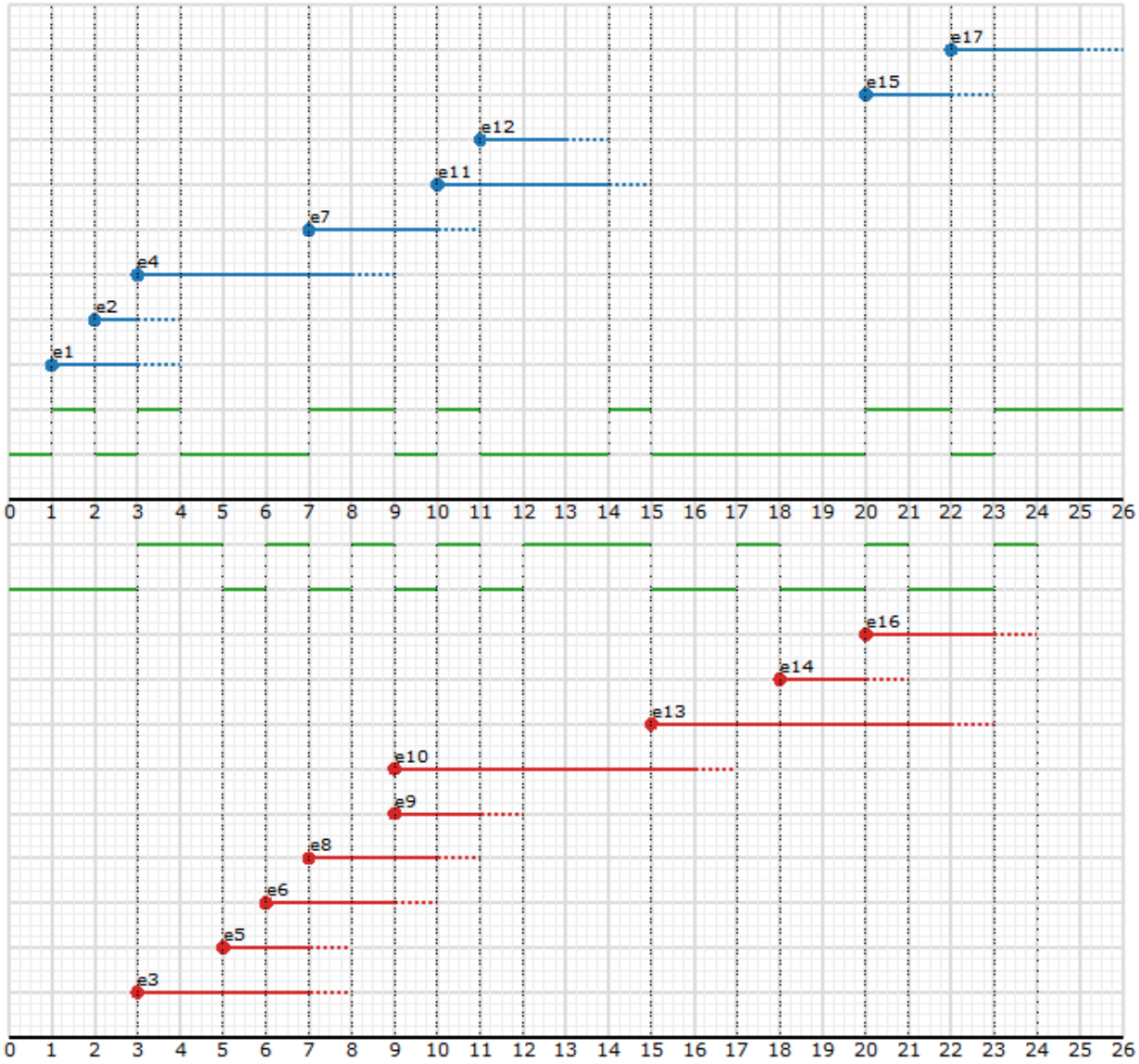


Figure 13. Group and Apply with Snapshot windows

The time plot for this input is shown in Figure 13, based on the same input considered for hopping windows. We have split the input stream into two groups based on the TollId as a partitioning key, and they are shown in blue (for TollId 1) and red (for TollId 2) on separate tracks. As before, windows are depicted as green line segments, but this time we had to draw windows for each group separately since they constitute a snapshot in time for a particular group and so are different across groups. The dotted blue and red extensions of the original events show how the event duration (*i.e.*, the lifetime of the event) has been altered or extended by one minute into the future. The intended results of the query are shown in the time plot in Figure 14. And just like before, result events are color coded according to the toll booth to which they belong, as well as annotated with the resulting toll sum and set of events contributing to it.

Some observations:

- The first thing you'll notice is that the number of output events is reduced by a third compared to hopping windows. The smaller window size of 1 minute compared to 3 minutes in hopping windows does not matter since this event duration is being uniformly applied to all events. The output stream is event driven – *i.e.*, generated only at changes in the input. This implies that at any point in time, you see the most recent aggregate value.

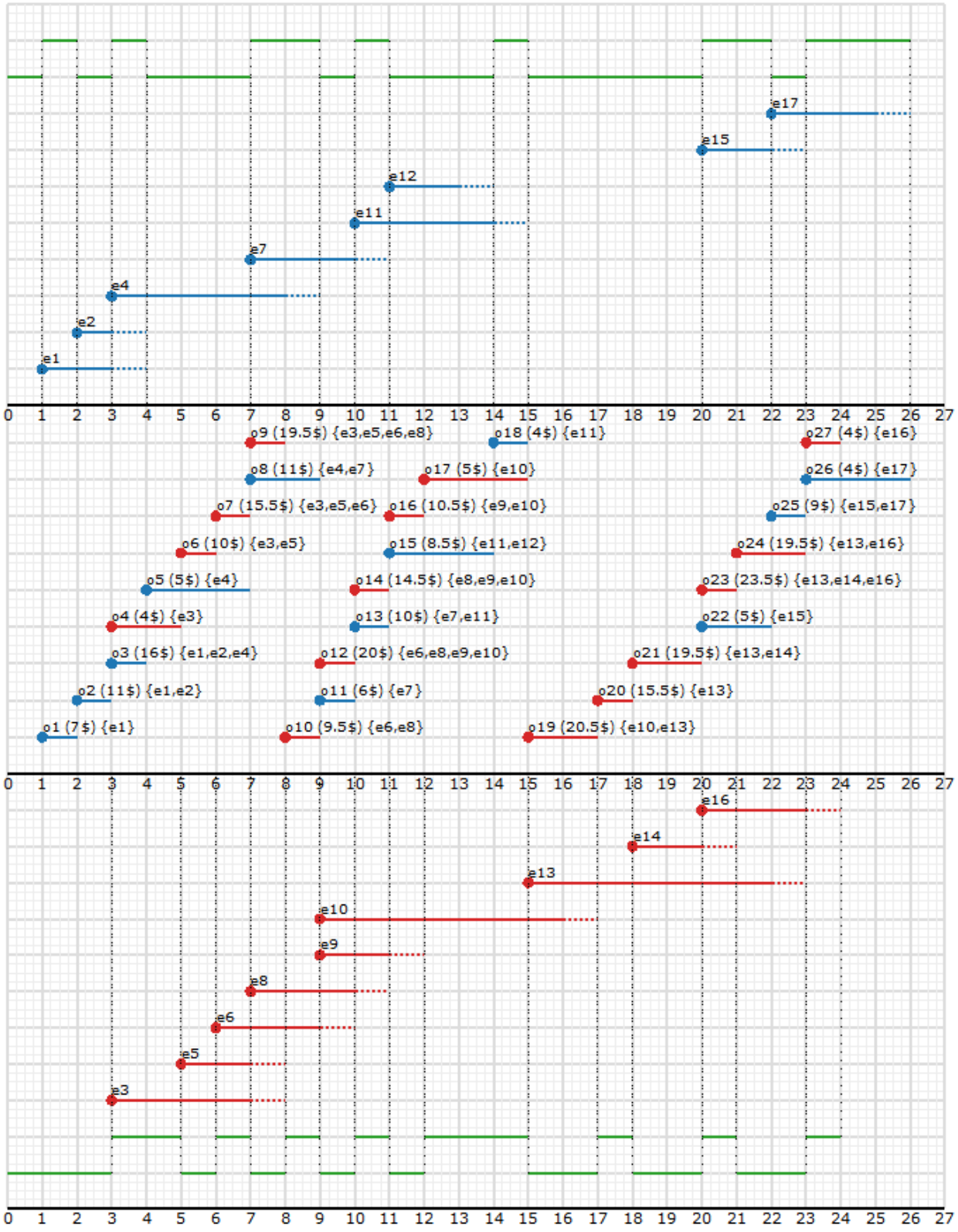


Figure 14. Output events from Group and Apply with snapshot windows

- Each window has at least one new event, rather than a repetition of the same set of events from the past window. This implies that each aggregate computation retains just enough and the most current state – which is significant when you consider thousands or millions of groups (we discussed this in the introduction to Group and Apply)

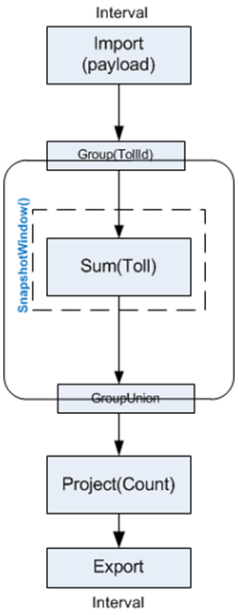


Figure 15. Query graph for [Partitioned Snapshot window] query

- The output reflects the most current event-driven state of the stream by looking back over a specified time period. From this perspective, **snapshot window combined with AlterEventDuration is the nearest equivalent to a sliding window in StreamInsight**, in that it represents the latest output of the stream at any point in time. We say "nearest equivalent" because a pure sliding window returns an output at every tick of advancing time.

Step 3 Gather elements of query logic to compute the output and develop an event flow graph.

- Group and Apply – we need a grouping operator to group the events in the stream based on Booth ID.
- Window – we choose a snapshot window – as per requirement.
- Sum() – to return the summed toll amount within a window
- Alter Event Duration – to extend the EndTime stamp of the event by one minute

The event flow graph showing the query with the above components is shown in Figure 15.

Step 4 Compose the query as a streaming transformation of input to output.

By now, the query should be very intuitive to understand. The input stream is grouped by TollId into per-TollBooth streams. On each of these streams, every event has its end time advanced by a minute, and then a snapshot window is applied on the stream. Within each snapshot window, we compute the aggregates Sum() and Count(). The second statement transforms the output events into Point events.

```
var query = from e in inputStream.AlterEventDuration(e => e.EndTime - e.StartTime + TimeSpan.FromMinutes(1))
            group e by e.TollId into perTollBooth
            from win in perTollBooth.SnapshotWindow()
            select new Toll
            {
                TollId = perTollBooth.Key,
                TollAmount = win.Sum(e => e.Toll),
                VehicleCount = win.Count() // computed as a bonus, not asked in the query
            };
```

One can now easily compute a moving average. First transform the output events into Point events and then, given the sum and the count, compute the moving average of toll collected through a given tollbooth, as follows:

```
var partitionedMovingAverage = from e in query.ToPointEventStream()
                               select new TollAverage
                               {
                                   TollId = e.TollId,
                                   AverageToll = e.TollAmount / e.VehicleCount
                               };
```

5. Multicast

A CEP stream is an unidirectional flow of events from input to the output with events participating in incremental operations as they flow through the query. There are many instances where we need to replicate a stream and use each replica to perform a different computation or analysis, and then correlate these streams to derive some insight. StreamInsight provides a **multicast** operator to replicate a stream. We will see examples of the multicast operation in the next section on Joins.

6. Join

StreamInsight is a temporal query processing engine, and one of the key capabilities of the engine is to correlate event occurrences with respect to time and their payload values. Many of our everyday queries involve a temporal correlation: "what was the S&P index when this stock went up by 15%?", "when did this incident happen (before/after another event)"?, "what caused this to happen?", "what should happen when events A, B and C happen in that order (pattern)"? The JOIN operation enables this correlation between event streams in StreamInsight.

The Join operation in StreamInsight is temporal, and is characterized by the following behavior. A Join between two events from the respective streams is valid:

(a) for the duration of overlap of lifetimes of the events,
(b) when the payload values of the events satisfy the join predicate.

Let's consider this simple application.

[Inner Join] "Report the output whenever Toll Booth 2 has processed the same number of vehicles as Toll Booth 1, computed over the last 1 minute, every time a change occurs in either stream."

Step 1 Model input and output events from the application's perspective.

We will use the input stream used for [Partitioned Sliding Window].

Step 2 Understand the desired query semantics by constructing sample input and output event CHTs.

Assume that we are able to separate the streams for Toll Booth 1 and 2. The objective is to correlate the first stream against the second, and generate an output whenever the count of vehicles between the two streams is the same.

The time plot for this input is shown in Figure 16. The phrase "every time a change occurs in either stream" motivates us to define a snapshot window on the streams. To get count computation over the last 1 minute we will need to extend event end times by 1 minute just like in previous examples (and also depicted as dotted-line extensions). And finally we would need to filter the results separately for toll booth 1 and toll booth 2. The resulting stream for toll booth 1 is depicted in blue in the top track, and for toll booth 2 in red in the bottom track. Each of these tracks shows the snapshot windows in green which are annotated for convenience with the resulting count corresponding to the window. The Join operation then compares the two streams. The final result is depicted in the middle and represents overlapping intervals where the count of vehicles for toll booth 1 matches the count for toll booth 2. Again, participating events are included in the event annotations for convenience.

Step 3 Gather elements of query logic to compute the output and develop an event flow graph.

In [Partitioned Sliding Window], the ability of the Group and Apply operator to partition events based on a key enabled us to group the events by TollId, and compute the counts for every snapshot window. In this example, we will replicate that stream, and filter out the events that correspond to TollId 1 and 2 respectively from each replica of the stream. Then we will correlate these two streams using the join based on the event count. StreamInsight provides a **multicast** operator to replicate streams.

Step 4 Compose the query as a streaming transformation of input to output.

We'll reuse the logic from [Partitioned Sliding Window] to get a stream of per toll booth aggregations. The two streams **stream1** and **stream2** are results of filtering just the events for toll booth 1 and 2 respectively. The logic in each of them is applied over the replicas of the output stream from **partitionedSlidingWindow** that are generated as a result of the Multicast operation. We then correlate these two streams using a Join operation based on the vehicle count from both streams.

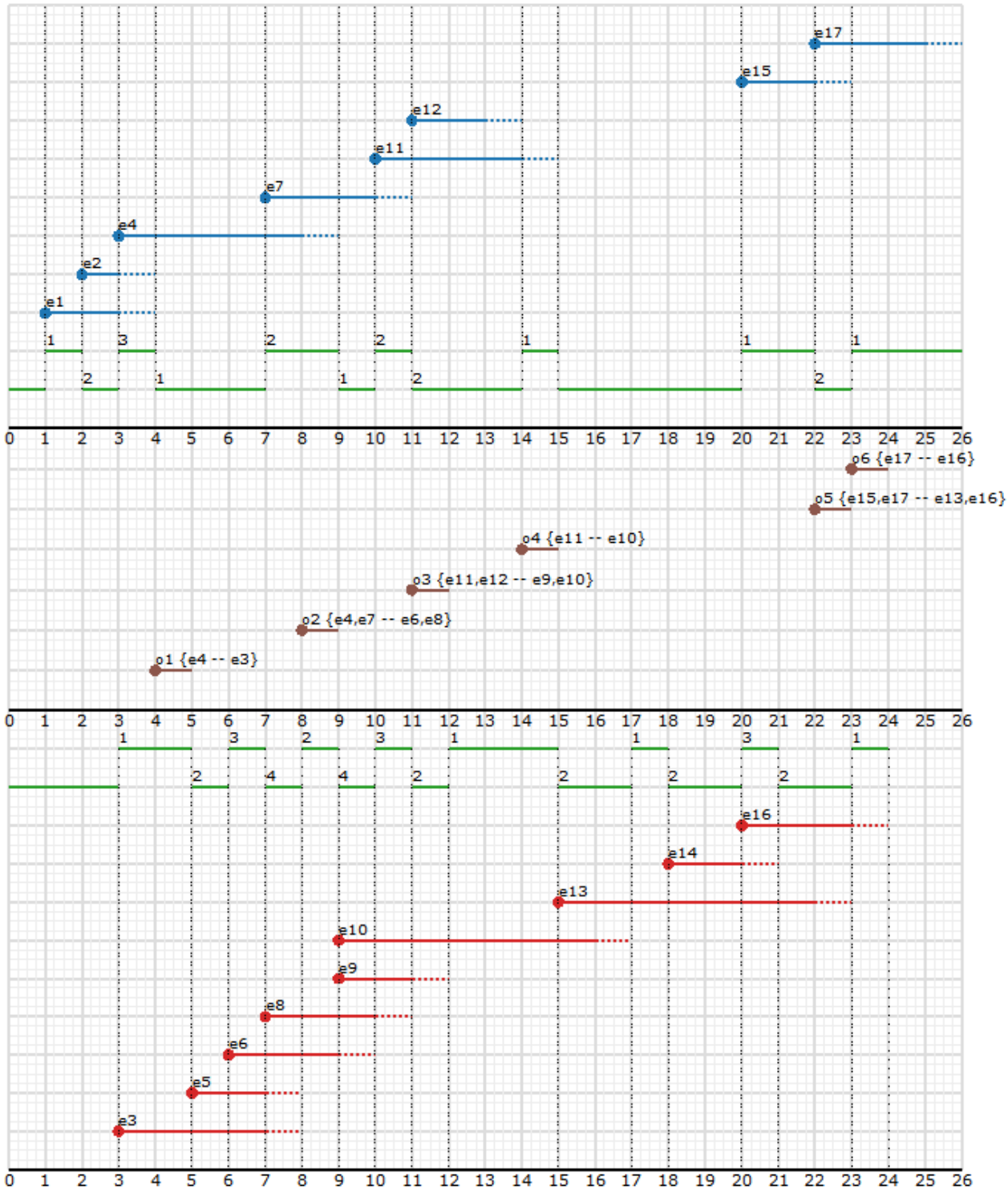


Figure 16. Inner Join between two streams (equal vehicle count is the JOIN predicate)

```
var stream1 = from e in partitionedSlidingWindow where e.TollId == "1" select e;
var stream2 = from e in partitionedSlidingWindow where e.TollId == "2" select e;
var query = from e1 in stream1
            join e2 in stream2
            on e1.VehicleCount equals e2.VehicleCount
            select new TollCompare
            {
                TollId_1 = e1.TollId,
                TollId_2 = e2.TollId,
```

```
            VehicleCount = e1.VehicleCount
        };
```

The Multicast operator is implicitly invoked as a consequence of the LINQ statements that define stream1 and stream2. Note that any number of such replicas can be generated. The JOIN that you see above is a case of **inner join**.

A **cross join (*aka* Cartesian join)** involves combining payload fields of every event in the first stream with payload fields of every event from the other, joined stream. It has no join condition.

```
var crossJoin = from e1 in stream1
                from e2 in stream2
                select new TollCompare
                {
                    TollId_1 = e1.TollId,
                    TollId_2 = e2.TollId,
                    VehicleCount = e1.VehicleCount
                };
```

Cartesian joins bring up the specter of an explosion of rows in relational systems. In this case, that risk exists only if the events that are joined also have exactly overlapping lifetimes on an event-by-event basis.

StreamInsight natively supports Cross Join, Inner Join, and Anti Join. We will take some time to discuss how we can realize other join types from these three primitives.

A **theta join** derives from inner join, where the join condition is something other than equality of values in the specific fields. Consider this query as an example.

[Theta Join] "Report the output whenever Toll Booth 2 has processed lesser number of vehicles than Toll Booth 1, computed over the last 1 minute, every time a change occurs in either stream".

```
var query = from e1 in stream1
            from e2 in stream2
            where e1.VehicleCount > e2.VehicleCount
            select new TollCompare
            {
                TollId_1 = e1.TollId,
                TollId_2 = e2.TollId,
                VehicleCount = e1.VehicleCount
            };
```

An **equi join** is a special case of the theta join – where the join condition is an equality of values in specific fields from the payload of both streams. This is the example we saw above in **[Inner Join]**.

A **self-join** derives from inner-join – it is a join of the stream to itself. Combined with multicast, it can be used to correlate different fields in the payload (which by itself, one might argue, does not require a join) with *different temporal constraints* being imposed on each stream. A commonly recurring pattern in streaming queries is to replicate a stream using a multicast, having one of the replicated streams compute an aggregate or perform another complex operation, and then use the other replica, typically with temporal transformations, and to combine the two results to achieve the desired application semantics.

6.1 Alerting and Anomalies – Left Anti Join (LAJ)

A common need in streaming applications is the ability to detect *non-occurrences* of events. Relevant to our toll station scenario, note that several toll stations offer automated tollbooths. In the US, there is a network of highways that provide this service called EZPass. Commuters who subscribe to this service can drive through the automated tollbooths with lesser delay. A scanner in the automated tollbooth reads the tag whenever the vehicle passes through one of the toll booths. Given this background, a major pain point for EZPass is lost revenue due to toll violations on two accounts: (1) toll evasion – where the vehicle that passes through the toll does not have a tag (2) speed – where the vehicle passes too fast through the toll. We will look at the toll evasion case here.

In general terms, we detect non-occurrence in the following manner:

1. Define a reference stream of events, each characterizing the occurrence of something of interest.
2. Compare the stream that is being observed against this reference stream on a continuous basis.
3. O*nly when* the observed stream has an absence (*i.e.*, non-occurrence) of the reference event, you report this non-occurrence from the reference stream.

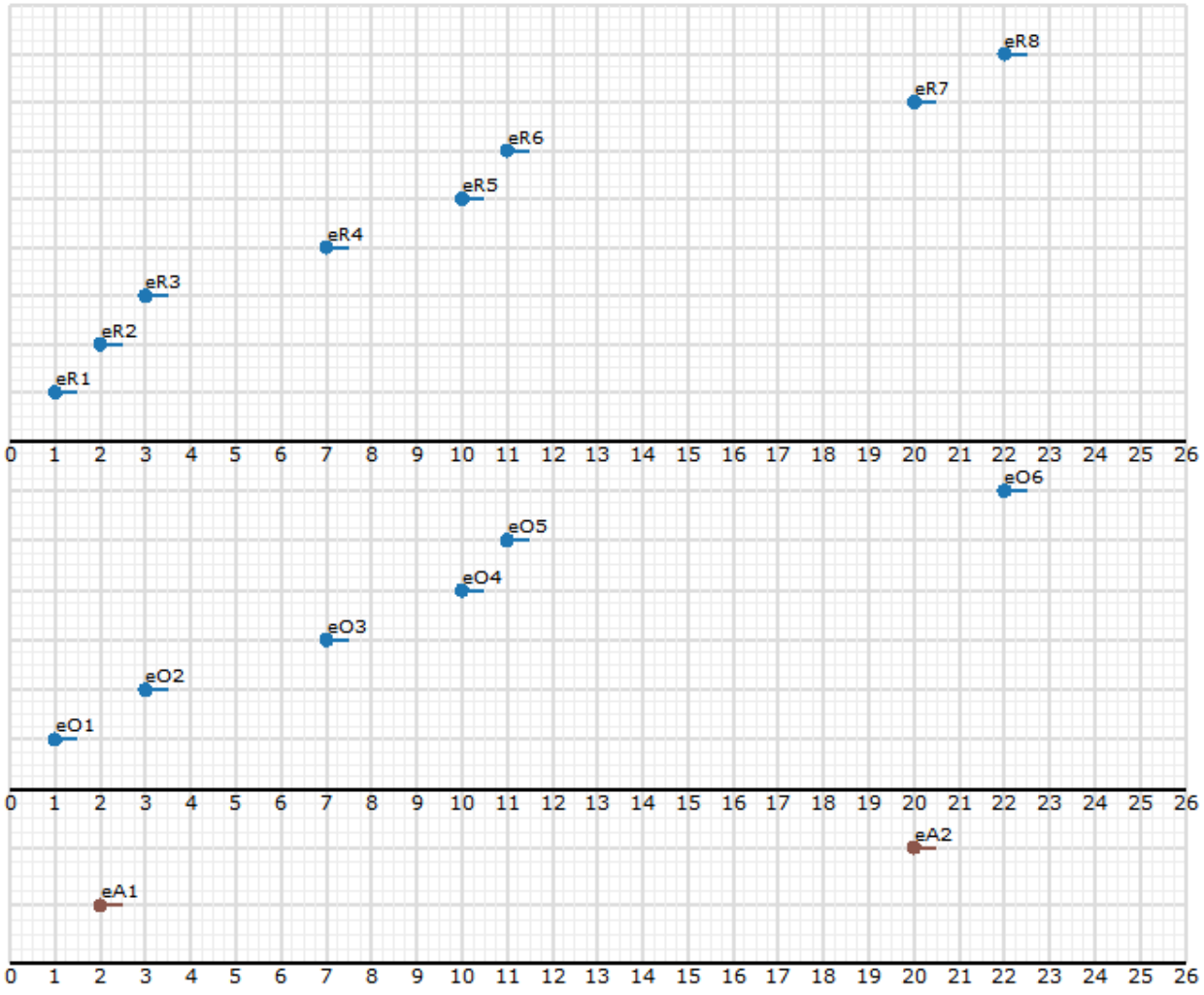


Figure 17. Left Anti Join in action

Consider the query:

[Left Anti Join] "Report toll violators – owners of vehicles that pass through an automated toll booth without a valid EZ-Pass tag read".

Let's walk through the query building process.

Step 1 Model input and output events from the applications perspective.

One of these will be the *reference stream* – it indicates the physical passage of a vehicle through the tollbooth – where an event will be generated whenever a vehicle crosses the toll booth. The other will be the *observed* stream – it represents a valid tag read – where an event will be generated whenever a valid tag read occurs when the vehicle crosses the tollbooth.

The Start time of the point event in the reference stream will be the point in time when the pressure sensor underneath the asphalt at the entry to the toll booth senses a vehicle. For the sake of simplicity, let's assume that the hardware in the toll station is fast enough to read the tag on the windshield at the exact point in time when the vehicle itself was sensed. So the start time of the point event in the observed stream will be the same, provided it was a valid read – if the tag was missing, expired, or is invalid, there would be no event generated.

Step 2 Understand the desired query semantics by constructing sample input and output CHTs.

The time plot in Figure 17 shows a reference stream with events named eR_i and indicates the passage of a vehicle through the toll booth 1. The middle track shows an observed stream with events named eO_i and indicates valid tag reads in toll booth 1. The bottom-most track on the time plot shows the output events, each of these events indicates a tag violation.

Step 3 Gather elements of query logic to compute the output and develop an event flow graph.

- The reference stream is the left branch of any join that we propose to use here, the observed stream is the right stream.
- We need a join type that outputs an event from the reference stream whenever an event in the left (reference) stream of the join does NOT find a matching event in the right (observed) stream. The match is defined as the points in time:
    1. when an event does not exist on the observed stream;
    2. when the event's payload does not satisfy the join predicate.

  This example demonstrates case 1 above. The lifetime of this output event will be the duration over which the two events do NOT overlap – which implies, again, a point event. [Left Anti Join] demonstrates this behavior.

Step 4 Compose the query as a streaming transformation of input to output.

```
// Simulate the reference stream from inputStream itself - convert it to a point event stream
var referenceStream = from e in inputStream.ToPointEventStream() select e;

// Simulate the tag violations in the observed stream by filtering out specific
// vehicles. Let us filter out all the events in the dataset with a Tag length of 0.
// In a real scenario, these events will not exist at all – this simulation is only
// because we are reusing the same input stream for this example.
// The events that were filtered out should be the ones that show up in the output of Q7
var observedStream = from e in inputStream.ToPointEventStream()
                     where 0 != e.Tag.Length
                     select e;
```

```
// Report tag violations
var query = referenceStream.LeftAntiJoin(observedStream, (left, right) => true);
```

Note that there is no requirement that the reference stream and the observed stream consist of events with the same duration for Left Anti Join to work. For cases where these events have different lifetimes, the output from Left Anti Join will have a lifetime that corresponds to the non-overlapping areas of the two lifetimes. It is, however, important that you make sure that the output behavior correctly reflects your application semantics – here is where the above process will be immensely helpful.

Note that there was no payload considered in the determination of the join – this is another axis on which you could pivot the join behavior. Also note that we did not specify any window of time over which to make this correlation. You can easily compose another query based on **[Left Anti Join]** to "find out the number of violations over the past 1 hour" using the count and windowing constructs we discussed earlier.

Left Anti Join is the third join type natively supported in StreamInsight.

7. Let's Play Ball – Heavy Hitters

Recall the query [Partitioned Sliding Window] from Section 4.1.

[Partitioned Sliding Window] Find the most recent toll generated from vehicles being processed at each station over a one minute window reporting the result every time a change occurs in the input.

Now, assume that a toll administrator wants to obtain some aggregate statistics on the output from this query. Assume that she is interested in knowing the top 2 toll amounts every 3 minutes from the output of [Partitioned Sliding Window], as described in the following query:
[TopK] "Report the top 2 toll amounts over a 3 minute tumbling window over the results obtained from [Partitioned Sliding Window]."

Step 1 Model input and output events from the application's perspective.

We will retain the same event models as [Partitioned Sliding Window].

Step 2 Understand the desired query semantics by constructing sample input and output event tables.

The input to this example is the output from [Partitioned Sliding Window], shown in Figure 18.

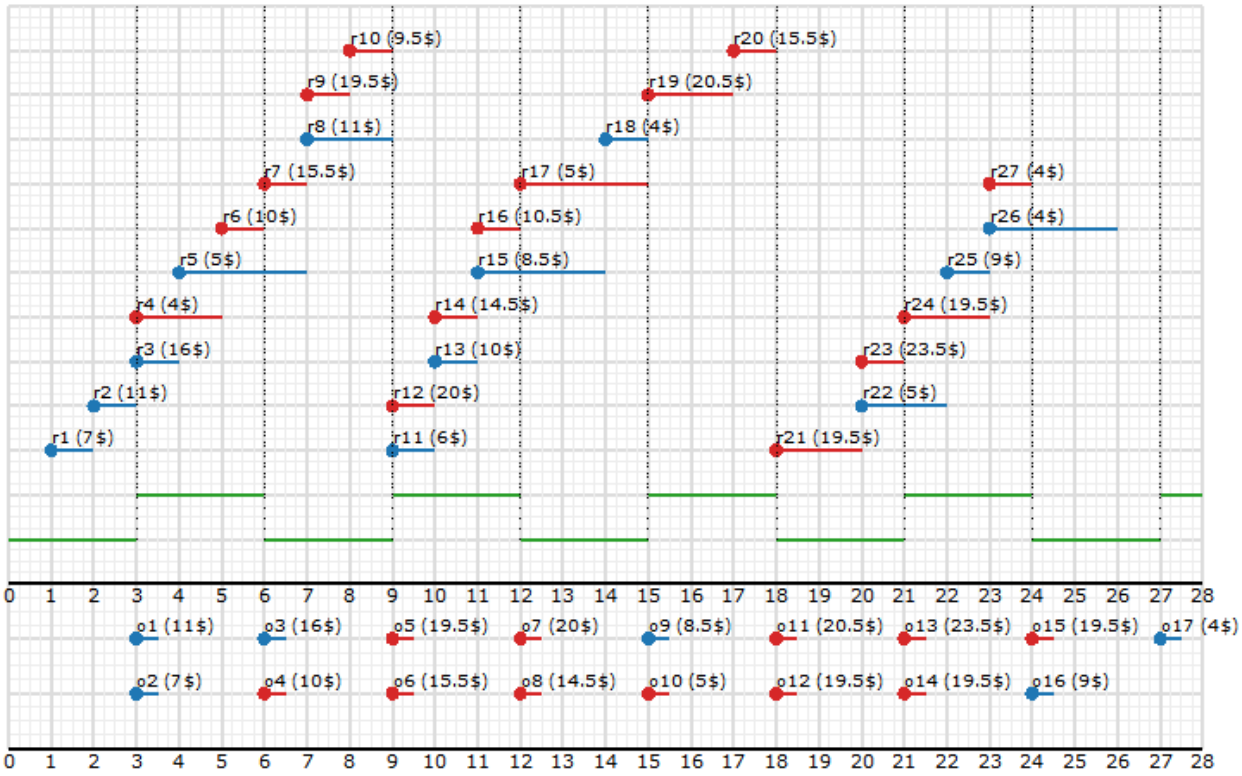


Figure 18. Top-K query with tumbling windows built using output from [Partitioned Sliding Window]

Event presentation follows established convention – events related to toll booth 1 in blue, toll booth 2 in red, and green line segments show tumbling windows of 3 minutes. We named events with an "r" prefix as they are the results from the [Partitioned Sliding Window] query. Provided total tolls are in parenthesis.

The expected output is shown in the bottom track of the time plot. We also used convention to put events ranked 1 in the top row and those ranked 2 in the second row below.

Step 3 Consider the different elements of query logic required to compute the output above.

We need a mechanism to rank in descending order the TollAmount in each of the buckets defined by the tumbling window. The orderby clause with the descending qualifier in the StreamInsight query achieves this.

We need to take the top 2 of these rank-ordered entries – the Take() operator in LINQ allows us to do this. LINQ also provides us the rank in the output.

Step 4 Compose the query as a streaming transformation of input to output.

The resulting query is shown below.

```
var query = from window in partitionedSlidingWindow.TumblingWindow(TimeSpan.FromMinutes(3))
            from top in
                (from e in window
                 orderby e.TollAmount descending
                 select e
                ).Take(2,
                    e => new TopEvents
                    {
```

```
                TollRank = e.Rank,
                TollAmount = e.Payload.TollAmount,
                TollId = e.Payload.TollId,
                VehicleCount = e.Payload.VehicleCount
            })
select top;
```

8. Filter and Project

We have quietly used filter and project in most of our examples, without specifically highlighting them. Let's take a minute to do so. Simply put, the **Filter** operation is the *where* clause specification in a StreamInsight LINQ query, along with the select clause that outputs the filtered events. Consider the query [TopK]. Suppose we wanted to see the output only for TollStation 1. The time plot in Figure 19 shows the effect of this operation on the output of the [TopK] query.

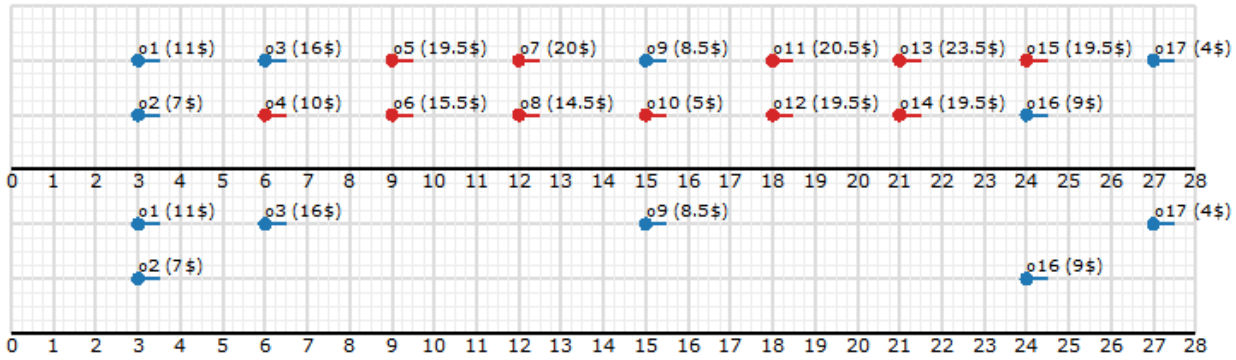


Figure 19. Filter for toll booth 1 applied to the result of [TopK] query

The query becomes:

```
var query = from e in topK
            where e.TollId == "1"
            select e;
```

The filter clause here is `e.TollId == "1"`. This can be any logical expression allowed by LINQ. UDFs (User Defined Functions, which we will see shortly) find their common use here.

A **Project** operation "transforms and projects out" the payload fields for the stream output, packaged into a known type (as we have seen in examples so far) or into an anonymous type (when used for query composition) as part of the select clause. Projection allows you to transform the payload – in terms of in-place changes to a field, or in adding or removing fields. Similar to the case for filters, there are no limitations on the kinds of LINQ and C# expressions that can be supported for projection transforms.

9. Operators as Legos – building up Outer Join

So far, we have followed the top-down, four step query writing methodology to arrive at the LINQ query. If you are able to complement this top-down approach with a bottom-up introspection of the query in **Step 3**, this will greatly improve your ability to write efficient queries. In this section, we will use this bottom-up approach to constructing a query to build up an Outer Join operation. First, let's understand how a StreamInsight query works.

StreamInsight enriches LINQ with temporal query capability through LINQ extension methods. On the client, the C# compiler runs HelloToll.cs through a StreamInsight LINQ Provider, which translates a LINQ query into *query template, query binder, and query definition* XML documents. The StreamInsight engine understands only this XML-based representations of the query– it generates the execution plan, loads and links the required assemblies that constitute a running query, and executes the query.

In its compiled form, a StreamInsight query is an event flow graph made up of *operators* and *streams*. Each **Operator** is a stateful or stateless machine that performs a specific operation on each event arriving via one or more input streams, generates new events as a result of this operation, and outputs the event via one or more of its output streams. **Streams** are efficient in-memory couriers of events from one operator to another.

For any given query, you can view this query plan using the StreamInsight Debugger. In the simplest use case, start a query trace, run the query, stop the trace, and then view the trace file in the debugger (we will not digress into Debugger setup here – please consult the documentation). Let's consider the query [TopK] for this exercise.

Figure 20 Query Graph of [TopK] as shown in the Event Flow Debugger

```
c:\Program Files\Microsoft StreamInsight 2.1\Tools>trace start c:\q9-trace.etl
The command completed successfully.
```

Now execute the query from Visual Studio. Once the query completes…

```
c:\Program Files\Microsoft StreamInsight 2.1\Tools>trace stop
The command completed successfully.
```

Next, start the debugger from the Start menu, and open the file c:\q9-trace.etl. The resulting query graph is shown in Figure 11. The plan shows events flowing from top to bottom through an operator chain `IMPORT -> CLEANSEINPUT -> ALTERLIFETIME -> GROUPING` (which contains an `AGGREGATE` and `PROJECT`) -> `PROJECT -> TOP-K -> PROJECT -> CLEANSE`. We have looked at all of the LINQ level constructs that correspond to these operators, except for `CLEANSEINPUT` and `CLEANSE`. Note that there are other powerful uses for the debugger – we will not digress into those capabilities now.

Now, imagine generating such a plan manually, using just the higher level constructs we have learned so far, to build either other operators or complete queries. This is the bottom-up approach to query building that we referred to earlier. The repertoire of primitive operators in StreamInsight is not very large, and you do not have to know all of them. What is important is the process of trying to compose a query by working bottom up to construct a query using a flow diagram substituting for the actual query plan.

In the earlier section on Joins, we left out one important join type for later: **Outer Join**. In this section, we will build this operator bottom up by combining other primitive operators such as inner join, multicast, and project - we can put them together like so many Lego pieces to compose the outer join query.

An Outer join derives from inner join, and has the following characteristics.

- The left outer join results in an output event that contains all the fields of the payload from the left stream, with matching values from the fields in the right stream, and NULL values for the fields where no matches are found.
- The right outer join contains all fields of the payload from the right stream, and matching values from the left stream, and NULL if no matches are found.
- The full outer join results in events that contain all the fields from payloads of both the streams, with NULL values for non-matching fields.

Let's consider a **left outer join**.

Step 1 Model input and output events from the application's perspective.

We will use interval events for this example.

Step 2 Understand the desired query semantics by constructing sample input and output event tables.

We will first consider the payload values to illustrate the outer join. From a temporal perspective, the semantics are that of an inner join. Assume we have two streams with payloads shown side by side below.

Start Time	End Time	License Plate	Make	Model
12:01	12:03	XYZ 1003	Toyota	Camry
12:02	12:04	XYZ 1001	Toyota	Camry
12:03	12:07	ZYX 1002	Honda	Accord
12:03	12:08	BAC 1005	Toyota	Camry

Start Time	End Time	License Plate	Toll	TagId
12:03	12:08	BAC 1005	5.50	567891234
12:05	12:07	ABC 1004	5.00	456789123

12:05	12:07	ABC 1004	Ford	Taurus						
12:07	12:09	NJB 1006	Ford	Taurus						

The result from a left outer join of the above two streams on equality of License Plates, and based on an intrinsic comparison of the timestamps done by the inner join semantics, is shown below.

Start Time	End Time	License Plate	Make	Model	Toll	TagId
12:01	12:03	XYZ 1003	Toyota	Camry	NULL	NULL
12:02	12:04	YXZ 1001	Toyota	Camry	NULL	NULL
12:03	12:07	ZYX 1002	Honda	Accord	NULL	NULL
12:03	12:08	BAC 1005	Toyota	Camry	5.00	456789123
12:05	12:07	ABC 1004	Ford	Taurus	5.50	567891234
12:07	12:09	NJB 1006	Ford	Taurus	NULL	NULL

Step 3 Consider the different elements of query logic required to compute the output above.

Given that this operator is not available out of the box, we will emulate left outer join behavior with the primitives at hand.

- An inner join between the two streams will give us the matching rows from the two streams
- A Left Anti Join between the two streams will give all the rows in left stream that do NOT have matching rows in the right stream. Since the left stream is the reference stream, this will give us the payload fields (License Plate, Make, Model). Now we need to add the NULL columns to this payload – this is done very easily using a Project operation on this output.
- Combine the above two streams with a Union operator to get the output we want.

This is shown graphically in Figure 20 below.

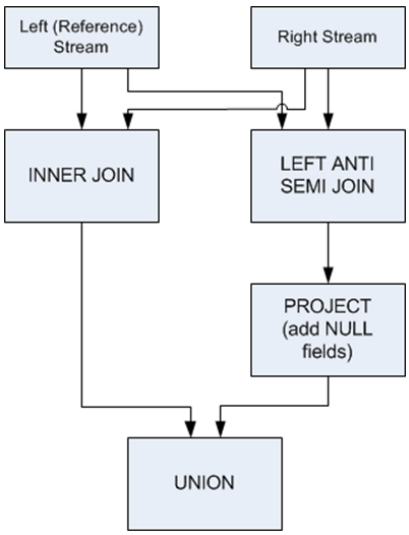


Figure 20. Outer Join composed from primitives

Step 4 Compose the query as a streaming transformation of input to output.

We will use the same input stream to generate the left and right streams using suitable projections. See how the query matches the graph provided above.

```
// Simulate the left stream input from inputStream
var outerJoin_L = from e in inputStream
                  select new
                  {
                      LicensePlate = e.LicensePlate,
                      Make = e.Make,
                      Model = e.Model,
                  };
// Simulate the right stream input from inputStream – eliminate all events with Toyota as the vehicle
// These should be the rows in the outer joined result with NULL values for Toll and LicensePlate
var outerJoin_R = from e in inputStream
                  where e.Make != "Toyota"
                  select new
                  {
                      LicensePlate = e.LicensePlate,
                      Toll = e.Toll,
                      TollId = e.TollId,
                  };
// Inner join the two simulated input streams
var innerJoin = from left in outerJoin_L
                from right in outerJoin_R
                where left.LicensePlate == right.LicensePlate
                select new TollOuterJoin
                {
                    LicensePlate = left.LicensePlate,
                    Make = left.Make,
                    Model = left.Model,
                    Toll = right.Toll,
                    TollId = right.TollId,
                };
// Left anti join the two input simulated streams, and add the Project
var leftAntiJoin = outerJoin_L
    .LeftAntiJoin(outerJoin_R, left => left.LicensePlate, right => right.LicensePlate)
    .Select(left => new TollOuterJoin
        {
            LicensePlate = left.LicensePlate,
            Make = left.Make,
            Model = left.Model,
            Toll = null,
            TollId = null
        };
// Union the two streams to complete a Left Outer Join operation
var query = innerJoin.Union(leftAntiJoin);
```

Such graphing of event flows, followed by query composition in LINQ, is the bread and butter of StreamInsight query development.

10. Union

The stream `outerJoin` is the result of a Union operation. This operator joins two streams which have the same event payload type (*aka* schema). The Group and Apply operator has an implicit Group Union operator to group the results from various apply branches. The Union operation is very useful in such scenarios where the requirement is just to merge the two streams, without performing any correlation checks like Join.

11. Extending the Query with Smarts

Simple aggregates such as Sum, Count, and Avg are built into StreamInsight extensions for LINQ. If you have the need the compute more custom aggregates and functions, you have four choices:

- **User Defined Functions** – useful for per-event computations. The function is executed for each incoming event and an expression is returned. It can be used wherever expressions can be used.
- **User Defined Aggregate** – useful for computing aggregates over a set of events. It returns a scalar value from a computation based on a set of events. Typically, the events that fall within a window of time constitute the set of events over which you are aggregating.
- **User Defined Operator** – useful for defining complete event transformers. A UDO processes a set of events and returns another set of events. In most general terms, this feature can be used to build custom operators that perform unique operations not covered by the existing set of operators such as join, union, group and apply, project, and so on. The more common use case is to build time sensitive and time insensitive operations against a set of events.

Let's explore these capabilities with toll station based examples.

11.1 User Defined Function

There are roughly 19 million EZ-Pass tags in circulation in the US (source). This surprisingly low number indicates that this technology has not yet captured the user's imagination as something that could be a more useful commuting tool than a mere payment device. Depending on how the backend data store is designed, you can obtain different tag information (expired, reported-stolen, other) from the account database in anywhere from a few seconds to a minute.

The opportunities for *immediate* action – for customer service (email or mail notification, traffic updates, auto service reminders and promos "time for oil change, get 10% off at Firestone"), or prevention of misuse or crime (flagging authorities) – are limitless. What takes several days using traditional batch processing can be shortened down to seconds or minutes after the occurrence of the event of a vehicle crossing a toll station. Consider another variant of the toll violation query.

[UDF] For each vehicle that is being processed at an EZ-Pass booth, report the toll reading if the tag does not exist, has expired, or is reported stolen.

The query for the above problem statement will look like this:

```
var query = from e in inputStream
            where 0 == e.Tag.Length || TagInfo.IsLostOrStolen(e.Tag) || TagInfo.IsExpired(e.Tag)
            select new TollViolation
            {
```

```
                LicensePlate = e.LicensePlate,
                Make = e.Make,
                Model = e.Model,
                State = e.State,
                Tag = e.Tag,
                TollId = e.TollId
            };
```

The two user defined functions `IsExpired` and `IsLostOrStolen` are defined below.

```
// Assume this is the reference database; the user defined function will search against this.
public static TagInfo[] tags = new TagInfo[]
{
    new TagInfo { TagId = "123456789", RenewalDate = ParseDate("2/20/2009"),
                  IsReportedLostOrStolen = false, AccountId = "NJ100001JET1109" },
    new TagInfo { TagId = "234567891", RenewalDate = ParseDate("12/6/2008"),
                  IsReportedLostOrStolen=true, AccountId="NY100002GNT0109" },
    new TagInfo { TagId = "345678912", RenewalDate = ParseDate("9/1/2008"),
                  IsReportedLostOrStolen = true, AccountId = "CT100003YNK0210" }
};
public static bool IsLostOrStolen(string tagId)
{
    return Tags.Any(tag => tag.TagId == tagId && tag.IsReportedLostOrStolen);
}
public static bool IsExpired(string TagId)
{
    return Tags.Any(tag => tag.TagId == tagId && tag.RenewalDate.AddYears(1) > DateTime.Now);
}
```

Note that these routines can be of arbitrary complexity. The motivation behind the simple array of TagInfo objects in this example is to demonstrate something self-contained within this program. In a production system, this will most likely be replaced by LINQ to SQL query against a database, or against an in-memory cache like Windows AppFabric.

User defined functions are generally used in the filtering clause of the LINQ query, but they can be applied wherever an expression is returned, so they can be used in Project operations. For example, you can imagine a function that is used to retrieve the VIN (vehicle identification number) of a vehicle given its license plates – a sample query might be as shown below:

```
var query = from e in inputStream
            where 0 == e.Tag.Length || TagInfo.IsLostOrStolen(e.Tag) || TagInfo.IsExpired(e.Tag)
            select new TollViolation
            {
                LicensePlate = e.LicensePlate,
                Make = e.Make,
                Model = e.Model,
                State = e.State,
                Tag = e.Tag,
                TollId = e.TollId
                Vin = GetVinFromLicensePlate(e.LicensePlate) // UDF in a Project operation
            };
```

`GetVinFromLicensePlate` is a UDF that does referential access against some external store. In this place, any other complex operation that returns an expression will help.

11.2 User Defined Aggregates

Out of the box, StreamInsight supports Min, Max, Avg, Sum, and Count. But there are several other custom aggregates that you may be interested in computing against an event stream. StreamInsight provides the capability to develop such aggregates via the extensibility mechanism of user defined aggregates.

In the eastern part of the US, Eastern New York, New Jersey, and Connecticut form what is called the Tri-State area, and southern and central NJ has a similar affinity with the state of PA. Given the population and geographic density, it is common for people to live in one state and work in another. Assume that for a given toll station, we want to compute the ratio of out-of-state vehicles to in-state vehicles.

Note that this example can be solved using a single query with simple counts of in-state, out-of-state, and total number of vehicles, and simply dividing them to obtain the result – it does not require the heavy machinery of a UDA. But the goal in this section is to show the different pieces of crafting a UDA, so we will consider this simple application logic.

[UDA] Over a 3 minute tumbling window, find the ratio of out-of-state vehicles to in-state vehicles being processed at a toll station.

The query for this problem statement will look like this:

```
var query = from win in inputStream.TumblingWindow(TimeSpan.FromMinutes(3))
            select win.UserDefinedAggregate<TollReading, OutOfStateVehicleRatio, float>(null);
```

The `OutOfStateVehicleRatio` is a User Defined Aggregate that is externalized to the LINQ surface through the `UserDefinedAggregate` operator extension. The actual class that contains the UDA implementation looks as follows:

```
public class OutOfStateVehicleRatio : CepAggregate<TollReading, float>
{
    public override float GenerateOutput(IEnumerable<TollReading> tollReadings)
    {
        float tempCount = 0;
        float totalCount = 0;
        foreach (var tollReading in tollReadings)
        {
            totalCount++;
            if (tollReading.State != "NY")
            {
                tempCount++;
            }
        }
        return tempCount / totalCount;
    }
}
```

Some notes about the implementation of `OutOfStateVehicleRatio` class:

- The input into this aggregate is a set of events with payload of type TollReading. The output of this aggregate is a value of type float (by virtue of this class inheriting from `CepAggregate<TollReading, float>` and the `IEnumerable<TollReading>`).
- Alternatively, since we know that the out-of-state is determined based on the payload field State of type String, we can define a more restrictive class as `CepAggregate<string, float>` as shown below (changes from the previous class is highlighted in bold)

```
public class OutOfStateVehicleRatio2 : CepAggregate<string, float>
{
    public override float GenerateOutput(IEnumerable<string> stateReadings)
```

```
    {
        float tempCount = 0;
        float totalCount = 0;
        foreach (var state in stateReadings)
        {
            totalCount++;
            if (state != "NY")
            {
                tempCount++;
            }
        }
        return tempCount / totalCount;
    }
}
```

Another operator extension, `UserDefinedAggregateWithMapping`, enables an additional formal parameter of an Expression that maps to the type of a single field of the payload (this mapping could simply be a reference to a specific field in the payload itself). This will enable us to invoke the UDA with a lambda expression that maps the event payload to a string – in this case, by simply referencing the State field in the payload. This causes an IEnumerable collection of just the State values to be sent into the UDA instead of the entire event payload.

```
var query = from win in inputStream.TumblingWindow(TimeSpan.FromMinutes(3))
            select win.UserDefinedAggregateWithMapping<TollReading, OutOfStateVehicleRatio2, string, float>
                       (e => e.State, null)
```

UDAs are not incremental. For every aggregate computation for a given event input, the full set of events that "belong" to the window over which the aggregate is computed is considered. However, this provides an acceptable performance for most applications.

11.3 User Defined Operator

User Defined Operators are useful for processing a set of events and returning another set of events. This can be used to build custom operators that perform unique operations not covered by the existing set of operators such as join, union, multicast, and so on. UDOs are also useful when you need to deal with the complete event structure, *i.e.*, inclusive of the Start and End Time fields, rather than dealing just with payload fields.

Consider a hypothetical example where the toll station also acts as a weigh station for trucks and other commercial vehicles. If the vehicle is hauling a load that is not in line with its "class" – in terms of number of axles and such parameters - weigh stations charge an extra toll for the additional tonnage. Assume that for each truck or commercial vehicle that passes through the EZ-Pass toll booth over a particular time period, the toll station has to compute this score and output an event that has a different structure – with the vehicle's tonnage, license plate, weight compliance score, and other information.

[UDO] Over a one hour tumbling window, report all commercial vehicles with tonnage greater than one ton (2K lbs), along with their arrival times at the toll, and any charges due to weight violation. Overweight charges during the rush hour (7am to 7pm) are double that of non-rush hours.

Step 1 Model input and output events from the application's perspective.

We will use the same input events as discussed in previous examples, augmented with weight-related information for this example. The output event will be structurally different than the input event.

Step 2 Understand the desired query semantics by constructing sample input and output event CHTs.

This is similar to the tumbling window example we noticed earlier. The emphasis in this example is on the output of an event with different payload type *with different start and end timestamps*.

Step 3 Consider the different elements of query logic required to compute the output above.

While the windowing constructs are simple, the output event structure is dependent on the timestamp of the event. This requires a construct called **time-sensitive** User Defined Operator.

Step 4 Compose the query as a streaming transformation of input to output.

```
var query = from win in inputStream.TumblingWindow(TimeSpan.FromHours(1))
            from e in win.UserDefinedOperator(() => new VehicleWeights());
            select e;
```

The UDO VehicleWeights() is defined as follows. In this example, we have made the weight charge a constant value – it is possible to pass this into the UDO implementation as a configuration parameter (in case the weight charge varies every month or every week depending on some other external factors). This is only to keep the example simpler to understand.

```
public class VehicleWeights : CepTimeSensitiveOperator<TollReading, VehicleWeightInfo>
{
    double weightcharge = 0.5;

    public override IEnumerable<IntervalEvent<VehicleWeightInfo>>
        GenerateOutput(IEnumerable<IntervalEvent<TollReading>> events,
        WindowDescriptor windowDescriptor)
    {
        List<IntervalEvent<VehicleWeightInfo>> output = new List<IntervalEvent<VehicleWeightInfo>>();

        // Identify any commercial vehicles in this window for the given window duration
        foreach (var e in events.Where(e => e.StartTime.Hour >= 0 && e.Payload.VehicleType == 2))
        {
            // create an output interval event
            IntervalEvent<VehicleWeightInfo> vehicleWeightEvent = CreateIntervalEvent();

            // populate the output interval event
            vehicleWeightEvent.StartTime = e.StartTime;
            vehicleWeightEvent.EndTime = e.EndTime;
            vehicleWeightEvent.Payload = new VehicleWeightInfo
            {
                LicensePlate = e.Payload.LicensePlate,
                Weight = e.Payload.VehicleWeight,

                // here is the interesting part; note how the output is dependent on
                // the start and end timestamps of the input event. The weight charge
                // is a function of the rush hour definition, the weigh charge factor
                // and the vehicle tonnage itself
                WeightCharge =
                    ((e.StartTime.Hour >= 7 && e.StartTime.Hour <= 14) ? 2 : 1)
                    * weightcharge
                    * e.Payload.VehicleWeight
            };

            // output the event via the IEnumerable interface
```

```
            output.Add(vehicleWeightEvent);
        }

        return output;
    }
}
```

Some important points to consider from both the User Defined Aggregate and UDO discussions above:

- UDAs and UDOs can be defined to be either time sensitive, or time insensitive. The UDA example is time insensitive, and the UDO is time sensitive.
- The main difference between the two is that time sensitive UDAs or UDOs expose the complete event to the UDO or UDA implementation – meaning that you can access the StartTime and EndTime of the event that "belongs" or "falls into" a window of time, and you can define your output event to be dependent on these timestamps. In other words, the event timestamps and payload fields of the output event can be based on these input event timestamps. In addition, the time sensitive UDA or UDO can be designed to provide the window's start and end time values.

  In contrast, a time insensitive UDO or UDA exposes only the payload to the UDO or UDA implementation. The output event from these extensibility mechanisms can be dependent only on the payload field values of the incoming event.
- For a time sensitive UDO or UDA, an important caveat is that ALL values in the implementation of the UDO or UDA MUST be deterministic. In other words, you cannot use constructs like `Random()`, or `DateTime.Now()` whose values can change from one invocation of the UDA or UDO to the next.
- UDO or UDA is invoked once per window, IF the events arrive into the UDO/UDA in an ordered fashion. We will defer the discussion on UDO behavior with out of order events until after the next section.

Part II Understanding Streaming Behavior of a Query

The discussion in Part 1 centered around query development – the process of designing a query, the various operators that constitute a query, and the composition of the query itself. We deliberately ignored any discussion on stream behavior – *i.e.*, how an event stream flowed from its source to the sink in a query. We did not delve into how and when query output is generated, and the factors that impact the correctness and responsiveness of the query.

This is the final step in our query development process:

Step 5 – Consider the timeliness of query output, balanced against correctness of query output

12. Notes on Events and Event Modeling

12.1 Event Shapes

A StreamInsight query can support three event models (*aka* shape) – Point, Interval, and Edge – based on which you can model your application events. It is worth mentioning that a **Project** operator can change the shape of the payload only, not the event shape itself.

StreamInsight provides you with the flexibility to model the shape of your events:

- **In the input adapter**: Using the Adapter API in the StreamInsight SDK, you can implement your input application event as a PointEvent, IntervalEvent, or an EdgeEvent and enqueue it into the query. The input adapter instance is represented in a logical query graph as an IMPORT operator.
- **In the query itself**: Inside the query, it is still a good practice to think of all events as interval events, irrespective of its shape. But from the standpoint of programmability, once an event crosses this threshold into the query, constructs like AlterEventDuration(), AlterEventLifetime(), and ToPointStream() enable explicit transformation of the event shape, while operations like JOIN can also implicitly change the shape of the event being output from them. A shape change from a point to an interval event is reversible, but changes to other event types may be irreversible. For example, if you transform an interval event to a point event, it is impossible within the context of the query execution to "go back" and discover the validity interval of the event you just transformed. Of course, you can record the event flow and inspect what happened using the Event Flow Debugger. We have seen examples of these constructs earlier.
- **In the output adapter**: Similar to the input adapter, the output adapter can also model the event along one of the three shapes for output. An output adapter instance is represented in the query graph as an EXPORT operator. Once the event crosses this threshold, the output adapter instance influences the shape of the output event via the binding of the query with the instance.

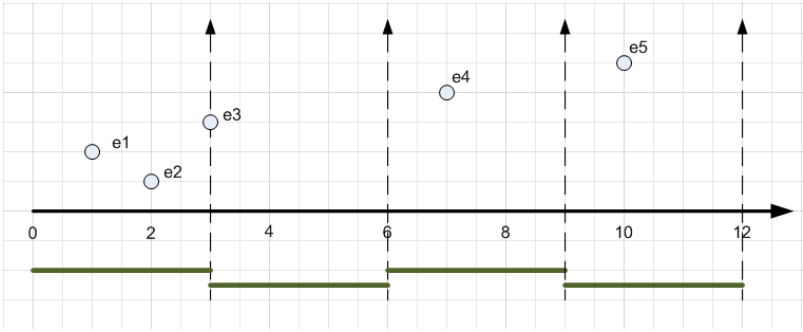


Figure 21. Point Event Example

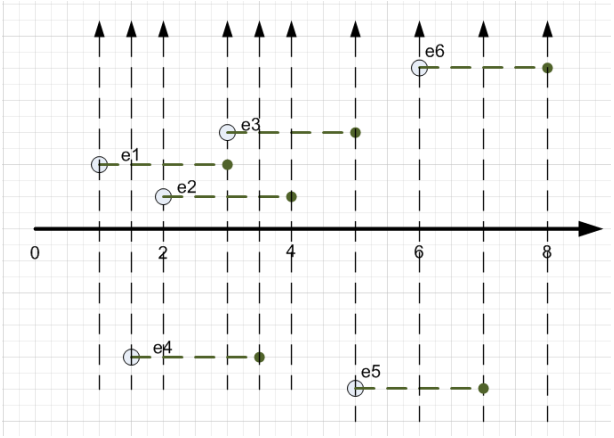
We will briefly look at the various event models in the context of query development, and to set the stage for discussions around stream behavior in upcoming sections.

PointEvent is used to model application events that typically occur at a point in time. A simple example is a web click.

From a query development standpoint:

- Point events used in conjunction with Hopping/Tumbling windows convey the idea of events "falling into" or "belonging to" a particular, fixed time slice, as shown in the Figure 21.
- Point events have the lifetime of a single tick. When you want to report the impact of a point event over a particular time period (as in "web clicks over the past one week, one hour" *etc.*) *every time a change occurs in the stream* (*i.e.*, in an event-driven manner, using snapshot windows) the programming technique is to "stretch" or alter the duration of the point events uniformly by the time period, and then define the snapshots. You can use the same trick when you have to correlate point events from two streams.

The figure below shows an example of correlating point events from two streams in a data-driven manner *looking back* over a 2 minute window.



- Conversely, you can use point events to vertically slice an interval or an edge event through a JOIN to generate a point-in-time result.

IntervalEvent is used to model application events that span a duration of time. We have built most of our examples in the previous chapter with interval events. Some key features from a query development standpoint:

- Events modeled as intervals are a quick means to correlate and detect co-occurrence of events across two different streams. The overlap of lifetimes is an indicator of this co-occurrence.
- Interval events influence CTI events to exhibit specific temporal behavior (discussed next).

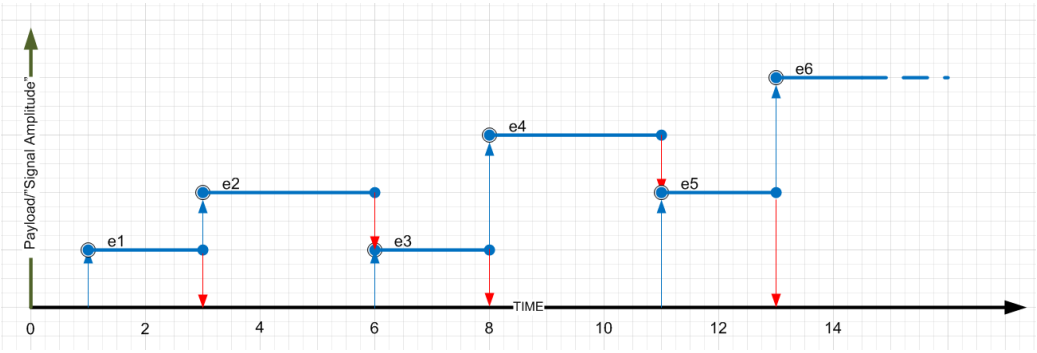


Figure 22. Edge Event

EdgeEvent is used to model application events for which the end time is unknown at the start time of the event, but is provided at a later point in time. The input source must have clear markers against each event indicating if a given event that has to be enqueued is a Start edge or an End edge. The input adapter can then use this marker to set the EdgeType property in the CreateInsertEvent() method. The two main caveats in enqueueing edge events are:

- The End edge must have the same start time and the payload field values as the Start edge.
- If an End edge event arrives without a corresponding Start edge event it will be ignored.

These are some of the most common use cases and motivations for using Edge Events.

- Edge events are ideal for modeling "long-running" events, such as the uptime of a process or a machine, given the flexibility of deciding the termination of the event on demand. Against this "anchoring" event, several small events such as CPU cycles, I/O, memory usage, and such short term metrics can be correlated and summarized.
- A popular application of edge events is in modeling signals – to implement step functions, discretize analog signals, or model event patterns with arbitrary event lifetimes.
- Following the same theme of "anchoring" event, edge events are used for modeling anomaly detection applications in conjunction with Left Anti Join. The edge event can be the anchoring event in the left hand side (or reference) stream, while the event stream with the "anomaly" such as non-occurrence of the event, gaps/mismatch in event payload values can be the right stream. An output from the left stream is sent out whenever an event in the left (reference) stream of the join does NOT find a matching event in the right (observed) stream. The match is defined as the points in time (1) when an event does not exist on the observed stream (2) when the event's payload does not satisfy the join predicate.

12.2 Event Kinds

You can enqueue just two kinds of events into a StreamInsight query from the input adapter. You do not explicitly tag these events as such – they are generated by virtue of your using different methods in the Adapter API.

- **INSERT event** – This is the event kind that gets enqueued when you choose one of the above event shapes, populate the payload structure, and use the Enqueue() method in your input adapter program.

- **CTI event** – **At the outset, know this much – if CTI events are not enqueued into a query, the query will NOT generate any output**. This is a special punctuation event that gets enqueued when you use the method EnqueueCtiEvent() in your input adapter program, or define their enqueueing process declaratively. We will understand the significance and purpose of this event in the next section.

12.3 Event "CRUD"

Of the familiar CRUD (Create, Read, Update, Delete) operations, only Create will resonate with users familiar with persisted storage or messaging systems – the other operations are distinctly different in a streaming system.

Create: You use CreateInsertEvent() to create an INSERT event of a given shape, populate its application timestamp(s) and payload fields, and enqueue the event.

Read: Recall from our introduction that the streaming model is very different from the query–response model seen in persisted systems. There is no equivalent of a "row/tuple store" from which you can selectively read an event based on a tuple identifier. The only way you'd "read" the exact event that you enqueued into a query is at the *output of the query* – assuming the query was a straight pass through, as in:

```
var query = from e in inputStream select e;
```

Update: StreamInsight events are immutable. Once an event is enqueued, there is no means to reach into the system and *delete* or modify that specific enqueued event in-place in the stream.

The only event model that supports a semblance of an update is the **Edge Event**. This is demonstrated in Figure 23 Edge Event, and shown in this table below. You can enqueue a start edge event with EndTime ∞ and then submit an end edge event with a modified end time. Then you can submit another start edge in a gapless manner with the new payload value. A sequence of such start and end edge events mimics the behavior of a step function, or an AD convertor.

EventKind	Event	Edge	StartTime	EndTime	Payload	Comment
INSERT	e1	Start	12:01	∞	1	
INSERT	e1	End	12:01	12:03	1	Same start time and payload value as e1
INSERT	e2	Start	12:03	∞	2	
INSERT	e2	End	12:03	12:06	2	-- ditto for e2 --
INSERT	e3	Start	12:06	∞	1	
INSERT	e3	End	12:06	12:08	1	
INSERT	e4	Start	12:08	∞	3	
INSERT	e4	End	12:08	12:11	3	
INSERT	e5	Start	12:11	∞	2	
INSERT	e5	End	12:11	12:13	2	
INSERT	e6	Start	12:13	∞	4	
…						and so on…

Inside the query processing engine, StreamInsight supports RETRACT and EXPAND events. These events are not exposed to the external user. The only time you will confront these events is while examining a query plan from the Event Flow Debugger. Event retraction and expansion, and the related operators responsible for cleaning up these events – CLEANSEINPUT and CLEANSE – is outside the scope of this document.

Delete: Similar to update, there is no concept of deleting an enqueued event. The closest equivalent to a Delete operation is the ReleaseEvent() method. You will invoke this in the output adapter after you Dequeue an event and copy the results to the event structure in your local/managed memory.

This short digression on event models and structures is a prelude to the main topic in this chapter – which is to understand stream behavior and query advancement. This is also the final step to consider in completing the query development process.

13. Step 5 – Time and Commitment

Until now, our discussion of query semantics has been devoid of any notion of timeliness of output delivery. We have proceeded with the notion that any input event that enters the system is used in the computation and the correct results are streamed out. However, for any output to be streamed out of a StreamInsight query, the input stream must contain special events called CTI events. This section explains the need for these special events in your input stream.

StreamInsight is a temporal query processing engine over complex event streams. It is built for applications where *time* – as in real clock time ("now" versus "over the past few seconds" versus "past two hours") – plays a central role in the computation and comprehension of results. Each event that is processed by a query represents a payload (*i.e.*, data elements) along with start and end timestamps that demarcate the lifetime of that event.

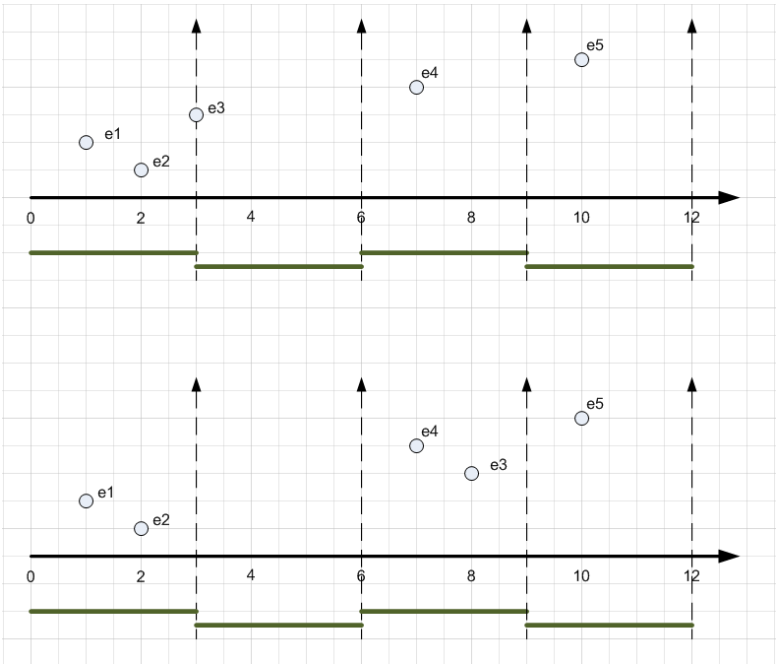
StreamInsight expects these start and end timestamps to be assigned by the application – in other words, these are **application timestamps**. StreamInsight does NOT implicitly assign any values to these timestamps – the program that enqueues events into the server sets these timestamp from a source. We will see why this is an important distinction shortly.

So far, all of our discussion of query processing has been at the "logical" level. To define query semantics, we used an input event table, built the time plot to understand how events interact with each other, and derived the output event table. In all of our examples, the StartTime of the most recent event input into the system was equal to, or at least one tick greater than, the event that arrived in the system just before it. The End Time of a particular interval event may have overlapped or exceeded that of its preceding event, which is not a problem. In short, we assumed that all incoming events arriving into a StreamInsight server have their application StartTimes ordered in relation to the system clock of the host server – *i.e.*, they were **Ordered Events**.

Ordered Events are achievable when the StreamInsight server is on the same machine as the event source. But in many real-life deployments, the server and the event sources/sinks are in disparate machines connected by networks for various reasons. Such multiple systems contribute to latency in the delivery of events from one system to another. This gives rise to the possibility of events arriving into a StreamInsight server with their timestamps out of order in relation to the system clock of the host server. StreamInsight queries are designed from ground-up to enable the user to accommodate such **Out Of Order Events**.

13.1 Why does order matter?

Note that a StreamInsight query may have both stateful (Joins, built-in and user-defined Aggregates, user-defined Operators) and stateless operators (Project, Group and Apply, Union, Filter) in its queries. The ordering of events can directly impact the results of stateful operators, as shown in the example below of a Count with tumbling windows.



First, let's understand the time plot better. The X axis shows the progression of *application time*. The event labels, however, are directly representative of their *relative ordering with respect to the clock on the host server*. For simplicity, we have considered point in time events. The values of these local timestamps do not matter – but their relative ordering matters. The following table explicitly calls out what "out of order" means – in the second table on the right, e4 arrives with an application timestamp that is out of order relative to the system/wall clock time. Since e3 is the first incoming event, no order is yet established. But once e3 has happened, then the next event could be in order or out of order. Note that tumbling windows report results for the next 3 minutes by default.

Event	Application System/WallClock Time StartTime
e3	12:03:00 10:00:00
e4	12:07:00 10:02:00

Event	Application System/WallClock Time StartTime
e3	12:08:00 10:00:00
e4	12:07:00 10:02:00

Given this understanding, the count values for the five tumbling windows from the two time plots are:

Event	StartTime	Count
o1	12:00	2
o2	12:03	1
o3	12:06	1
o4	12:09	2
o5	12:12	1

Event	StartTime	Count
o1	12:00	2
	12:03	--
o2	12:06	2
o3	12:09	1
o4	12:12	1

If you have agreed with these results, it is because of two implicit assumptions in your understanding of the example:

1. That the count is computed for a window defined by specific start and end times, spanning 3 time units – this is correct, and a constant.
2. That we are asking the query to *report the* output *once every three minutes* aligned with the window boundaries (*i.e.*, we are asking for the output at each horizontal line in the table).

Assume that our application acts on the output of this query with this logic:

```
if (count < 2)
{
        BuyOrder();
}
else
{
        SellOrder();
}
```

Then, with no out of order events (table on the left), o2 would have triggered a BuyOrder by virtue of e3's arrival. With e3 arriving out of order (table on the right), the action would be SellOrder – even if at a different time interval (12:06). If this application were a trading station, we will discover this mistake in a post-audit of the transaction, and undertake some compensatory action.

Now, assume that we know that the network transmission is unreliable, and there is a possibility of events arriving out of order into the StreamInsight server. Instead of every 3 minutes, assume that we request that the output be reported *once every six minutes*. Then the second table becomes:

Event	StartTime	Count
o1	12:00 12:03	2 <empty>
o2 o3	12:06 12:09	2 3
o4	12:12	1 <tbd>

Again, each horizontal line in the table signifies the reporting of the output. By doing this, we have accommodated for the late arrival of event e3, and thereby presented a correct (SellOrder) result even in the face of unordered input. But this correctness of output comes at a price – we have doubled the latency (or put another way, reduced the "liveliness") of the query output by a factor of two.

The key takeaways from this discussion are:

1. StreamInsight confronts out of order processing of events intrinsically in its query algebra
2. StreamInsight provides a mechanism for the user to make a conscious compromise between liveliness of the query – at the expense of possible incorrect output, or absolute correctness – at the expense of slowness in reporting of output.

The mechanism for conveying this compromise via the input event stream to the query is a special punctuation event called **Current Time Increment**. This is the topic of our next section.

13.2 Current Time Increment (CTI) Events

A CTI event provides a watermarking functionality in the stream to affect two fundamental attributes of the query:

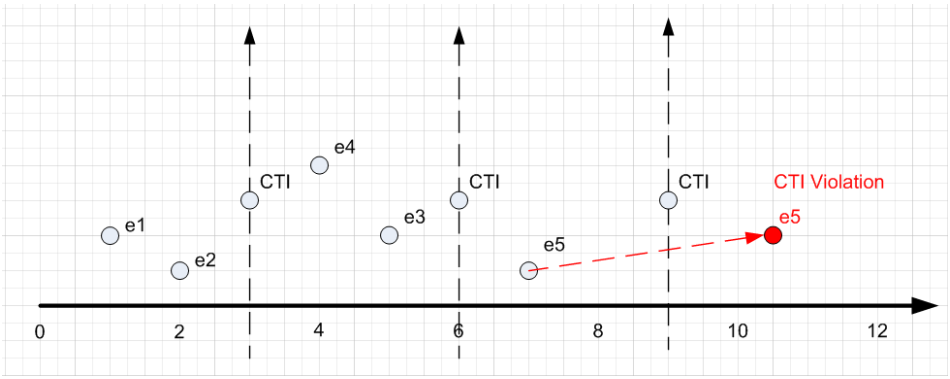
1. The liveliness of the query – the pace at which the query is able to emit output.
2. The statefulness of the query – the pace at which the query is able to release events that contribute towards state (for joins, aggregates, *etc.*).

You can explicitly issue the CTI event using the EnqueueCtiEvent() method – with the timestamp as a parameter.

When the streaming application enqueues a CTI into the input stream, you accomplish two things:

1. It provides a guarantee to the query that, from the point of issuance of the CTI, it will not enqueue any event whose Start Time is less than the timestamp of the CTI Event (*i.e.*, the query will not see an event that is earlier in application time than the CTI event). If a situation does arise where such an event is enqueued into the query, the query will return a "CTI Violation" exception.

   The figure below depicts a scenario where event e5 arrives with a timestamp of 12:07 *after* a CTI event that has been issued with timestamp 12:09, which is a cause for a CTI violation.



   IMPORTANT – In your program, note that the exception due to CTI violation will happen only in the scenario where you explicitly enqueue the CTI Events without using AdvanceTimeSettings() to define your CTI progression. This interface is explained below.

2. It informs the query that any stateful operator that has processed events that arrived preceding the CTI can release the events – under the following specific conditions:
   a. an interval event that preceded this CTI event can be released IFF the CTI's timestamp is greater than the **End Time** of the interval event.
   b. a point event which preceded this CTI event can release the point event IFF the CTI's timestamp is greater than the Start Time of the point event.
   c. an edge event (Start or End Edge does not matter) which preceded this CTI event can release the edge event IFF the CTI's timestamp is greater than the Start Time of the edge event.

13.2.1 Time and Commitment revisited

Given the above background, it is easy to see how CTI events help in achieving the compromise between blocking (*i.e.*, waiting for unordered events) for correctness of output versus achieving a lively stream (*i.e.*, not waiting for events, but accepting that some of the output results may be incorrect). In the above figure, point events e3 and e4 are unordered – if the CTI was issued once every event,

then there would have been an error in the output. But if this error was acceptable, then a CTI after every event (as opposed to once every 3 minutes) would have resulted in a maximally responsive query.

Important: If you know that the events arrive into a query ordered on time, then it is highly recommended that you issue a CTI coinciding with the Start Time of every event that is being enqueued.

13.3 AdvanceTime Settings

A practical problem with CTIs is that most of us cannot exactly predict *when* we want to issue a CTI explicitly. There are a couple of ways a developer can implement the streaming application (or to be more precise, the input adapter) to issue CTIs:

1. Populate the input stream itself with CTI events after one or more INSERT events. This is feasible if the event source is persisted, but even then, it is a cumbersome process or an overhead to seed the input stream with CTI markers – which the input adapter can then use as an indicator to call EnqueueCtiEvent().
2. Programmatically enqueue CTIs from the input adapter. You can specify a CTI frequency in terms of one CTI every N events through the adapter configuration, and the adapter can use this to invoke EnqueueCtiEvent() once every N events. This however makes the adapter implementation complex – if the EnqueueCti fails because of pushback from the engine (see Adapter documentation), then you have to keep track of enqueueing the CTI event along with INSERT events at the next opportunity provided by the engine.

So the solution is to automate the enqueue of CTIs in a declarative fashion. StreamInsight provides the mechanism to specify this as an interface implementation in the adapter factory (see Adapter documentation) or as part of the query binding specification itself. See the product documentation on AdvanceTimeSettings.

14. Conclusion and Future Work

In this paper, we provided a developer's introduction to the Microsoft StreamInsight Queries. We discussed a systematic 5 step process by which a developer can think through the formulation of a problem, and working through the steps to arrive at a StreamInsight query. We introduced the components of a StreamInsight application through a simple tutorial, and used this tutorial to discuss typical needs in a streaming application, showing how StreamInsight queries addressed these needs. We covered the windowing constructs and other language elements that enabled filtering, computations, and correlation over events. We showed examples of both top-down and bottom-up analysis and understanding of composing queries. We then expanded on the various forms of stream behavior and how StreamInsight enables the user to arrive at a compromise between latency requirements and the need for correctness of results.

While this document provides a good introductory coverage, it is not complete by any means. We have not covered some useful constructs in Count Window and Dynamic Query Composition capability. The Join construct is immensely powerful for various forms of event correlation – the examples provided in this document are simplistic. The query behavior section could be expanded to explain the temporal algebra in greater detail. Particularly, we have not touched upon the Edge event and signal processing using StreamInsight.

15. About the Sample Code

This document ships with sample code packaged in a file called `HitchHiker.zip`. This file contains a Visual Studio 2010 solution called HitchHiker.sln, which contains the main project, including a routine that lets you iterate through the various queries discussed in the document and watch their execution.

Feedback and Further Information

We hope that this paper, along with the sample code, will give you a quick start on programming your event-driven applications using StreamInsight. We'd greatly appreciate your feedback on this document. If there are errors or bugs in the document or code, please excuse us, and let us know – we'll arrange to address the error immediately.

Please send your questions, suggestions, comments and errata on this document to Alex.Raizman@microsoft.com.

We'd appreciate your feedback on the problems that StreamInsight is currently able to address, and more importantly, on any pain points that it does NOT address. For general product feedback, we'd encourage you to post your questions and comments at the StreamInsight Forum: http://social.msdn.microsoft.com/Forums/en-US/streaminsight

For the latest news and updates on the product, please subscribe to our blog at: http://blogs.msdn.com/b/streaminsight/

Microsoft StreamInsight ships as part of Microsoft SQL Server 2012, and it is available as a separate download here: http://go.microsoft.com/fwlink/?LinkID=253700

Acknowledgments

We thank Torsten Grabs, Galex Yen, Georgi Chkodrov, Balan Sethu Raman, and Anton Kirilov for their diligent review of this document and for their time and guidance.

Microsoft