Trill Query Writing Guide

1 Introduction

This guide introduces the reader to the basic concepts and programming constructs needed to author queries in Trill, a highly performant query processing engine for real time and offline temporal-relational data. Trill stands for a trillion events per day, which in many cases, is a very conservative estimate of Trill's query processing speeds.

Trill's query model is similar to that of Microsoft StreamInsight, which is Microsoft's commercial product offering in event processing. There is a separate guide available for users who are familiar with Microsoft StreamInsight.

This guide describes and demonstrates these concepts through the development of a running machine telemetry example. Note that this guide assumes that the reader is fluent in C# and is familiar with some of the basics of LINQ. It is also helpful if the reader has an introductory level of familiarity with Reactive LINQ (i.e. Rx). Working source code for all examples shown in this document in available as part of our downloads at https://github.com/Microsoft/TrillSamples.

1.1 Machine Telemetry

One common use case for systems like Trill, is the online analysis and visualization of machine and application telemetry. In today's datacenter-oriented applications, one of the first types of questions application writers need to answer is:

> What is happening in my application for individual users, and across all users? Also what is happening to the infrastructure that these applications are running on?

Figure 1: Common Types of Questions Answerable with Trill for Datacenter Apps

Note that the questions in Figure 1 make sense in both real-time and historical contexts. Proof of the power of Trill's query language is how little the actual query changes when going from one context to the other, making Trill particularly suitable for posing and answering queries over offline data for the purpose of developing real-time alerts and dashboards, which are also executed with Trill.

Throughout this document, we will build upon an example scenario based on infrastructure monitoring. Initially, we will answer the question:

> For a particular multi-core machine, for each 3600 tick period of the log, which process (by name) consumed the most CPU time on cores 1 and 2?

Figure 2: Initial Question to Focus On

It is worth pointing out that, while certain aspects of the program model presented here may seem complicated at first, they add the temporal query writing power to more traditional data programming paradigms (e.g. SQL) needed to address both the offline and online query processing scenarios in a consistent manner. After a bit of query-writing practice, these constructs will become natural. The learning experience is similar to discovering how to write non-trivial SQL queries.

2 DATA MODEL, INGRESS AND EGRESS

To answer the question in Figure 2, we assume that we receive data from an operating system service like ETW. ETW is a windows event logging/delivery framework that the OS and some applications use to communicate events to whoever subscribes. ETW delivers a sequence of different types of events, depending on what is happening in the system. For instance, an event is generated whenever there is a context switch. Similarly, events are generated whenever processes are created or destroyed. While these events are all delivered through one information conduit, we will, for now, assume that events have been partitioned by event type into separate event streams. We will later describe how we can use Trill itself to do this partitioning. Given this partitioning, there are two types of events we need to answer the question in Figure 2. For the purpose of this example, we simplify these two event streams. The first, called `ContextSwitch`, consists of context switch events. The fields (i.e. members) for each `ContextSwitch` event are shown in Figure 3.

Field Name	Type	Description
`Tick`	`long`	The time of the context switch, in ticks
`ProcessId`	`long`	Id of the process being context switched in
`CpuId`	`long`	Id of the core on which the context switch is occurring
`CpuTemp`	`long`	Temperature of the CPU at the time of the context switch

Figure 3: ContextSwitch Fields

In this example, we assume the `ContextSwitch` stream has the input shown in Figure 4. Each row in the table represents one input event. This event signifies that at time `Tick`, the process `ProcessId` running on core `CpuId` was context switched in. At the time it was context switched in, the CPU temperature was `CpuTemp`.

Tick	ProcessId	CpuId	CpuTemp
0	1	1	120
0	3	2	121
0	5	3	124
120	2	1	123
300	1	1	122
1800	4	2	125
3540	2	1	119
3600	1	1	120

Figure 4: ContextSwitch Data

In Trill, data is ingressed into the system through Observables (see Rx for a complete description of Observables in .NET). Observables are a .NET programming abstraction for in-process piping of push based, potentially unending data sources. In particular, programmers can write Observables which acquire data of a specified type, which is then pushed to a subscriber. Data acquisition is triggered in response to a subscriber registering with the Observable. Note that since the acquisition of data is part of the Observable code which programmers write, this could even involve communicating with other machines.

Since there is a conversion to Observables from Enumerables (see Enumerables for an in-depth discussion of LINQ and Enumerables), and there are very convenient ways of specifying Enumerable constants, most of our examples here will begin with an Enumerable constant specification, followed by a conversion to an Observable. Note that this conversion does not in any way change the result of the queries shown throughout this

document. If the Observable was delivering identical data from a real time feed, the results would be unchanged.

Figure 5 shows the code for creating an Observable of `ContextSwitch` objects, which when subscribed to, will deliver the data shown in Figure 4.

```
private struct ContextSwitch
{
    public ContextSwitch(long tick, long pid, long cid, long cpuTemp)
    {
        this.Tick = tick;
        this.ProcessId = pid;
        this.CpuId = cid;
        this.CpuTemp = cpuTemp;
    }

    public long Tick;
    public long ProcessId;
    public long CpuId;
    public long CpuTemp;
    ...
};
...
IObservable<ContextSwitch> contextSwitchObservable = new[]
{
    new ContextSwitch(0, 1, 1, 120),
    new ContextSwitch(0, 3, 2, 121),
    new ContextSwitch(0, 5, 3, 124),
    new ContextSwitch(120, 2, 1, 123),
    new ContextSwitch(300, 1, 1, 122),
    new ContextSwitch(1800, 4, 2, 125),
    new ContextSwitch(3540, 2, 1, 119),
    new ContextSwitch(3600, 1, 1, 120),
}.ToObservable();
```

Figure 5: Ingressing Enumerable Constants into Observables

Once the data is in an Observable, we may ingress it into Trill. Similar to Observable, Trill has a type for describing a data stream. This type is called Streamable. While Observables and Streamables are similar in that they are both push based and support a wide range of LINQ operations, they also have very significant differences which make them useful in very different situations.

For instance, Observables have no explicit notion of time, and simply “run specified code” whenever data arrives. Alternatively, Trill has a very explicit notion of time, which is programmed directly against (e.g. compute a 1 minute moving average). This means that it is the job of Trill, not the application programmer, to ensure that operations in the system result in producing the correct answer according to this notion of time, regardless of data representation or internal race conditions. In some sense, while operations on both Streamables and Observables are functional and produce identical results, Trill is declarative while Rx is imperative. As with other declarative languages, Trill’s declarative nature creates a query writing environment in Trill which is indifferent to the system it runs on, enables automatic parallelization, and allows transparent changes to internal data representation. In addition, since time is explicitly part of Streamable, operations over streams are

fundamentally temporal, greatly simplifying temporal query authoring, since in Rx, even the most basic temporal operations must be implemented by the query writer.

On the other hand, because Streamables have an explicit notion of time, the operations on them are constrained in ways that ensure that the resulting streams are temporally valid. For instance, time can only move forward in a Streamable (data from the past cannot suddenly appear). Operations are restricted to ensure that the stream data remains well-formed. There are two situations where Rx's ability to process ill formed data streams can be leveraged before interfacing with Trill:

- Data Ingress – When data initially arrives, it may come from many sources, some of which may experience longer transmission delays than others. One may even need to "give up" on some sources once enough time has passed. Dealing with these problems and presenting the resulting well-formed stream to Trill is a great application of Rx.
- Data dependent passage of time in mid-query – In this possibly rare situation, Trill doesn't have enough information to conclude that time is passing and needs "help". One can use Rx to assert that time has passed in the middle of the Streaming query. Trill interoperates nicely with Rx, making query authoring in these situations straightforward.

Since Streamables carry inherently timestamped data, Trill introduces a type – StreamEvent<T> – that corresponds to individual elements in a Streamable. The template parameter is an event's payload, which may or may not contain time information. T is comparable to payloads in Observables or Enumerables. In addition to the payload, each data event also contains two timestamps, called StartTime and EndTime. These timestamps, and Trill's treatment of them, is one of the most fundamental differences between Trill and Rx.

Going back to our example, we will now convert each data element in contextSwitchObservable into a StreamEvent. For now, we assume that StartTime should be equal to ContextSwitch.Tick, and EndTime should be StartTime + 1. It will become clear later why we have made these choices. While there is only one logical notion of an event, with both a start and end time, it is sometimes necessary to describe the logical event using two physical events, called edge events, which are presented to the system at different wall clock times. For the purpose of this example, we assume that both the start and end times are known simultaneously and specified in a single event, called an interval event. The conversion of the data in our example into an Observable of StreamEvents is shown in Figure 6.

```
IObservable<StreamEvent<ContextSwitch>> contextSwitchStreamEventObservable =
    contextSwitchObservable.Select(e => StreamEvent.CreateInterval(e.Tick, e.Tick + 1, e));
```

Figure 6: Creating an Observable of StreamEvents

Any Observable of StreamEvents may be converted into a Streamable. But since any Observable may contain any sequence of StreamEvents, even if the timestamps are out-of-order, we must, in the conversion to Streamable, specify how to handle out-of-order events using DisorderPolicy, which ensures that the result is a well-formed stream. The conversion used in our sample is shown in Figure 7.

```
IObservableIngressStreamable<ContextSwitch> contextSwitchIngressStreamable =
    contextSwitchStreamEventObservable.ToStreamable(DisorderPolicy.Drop());
var contextSwitchStreamable =
    (IStreamable<Empty, ContextSwitch>)contextSwitchIngressStreamable;
```

Figure 7: Creating a Streamable of Context Switches

Note that there is no reference to StreamEvents in the Streamable type. This is due to the fact that Streamables contain only StreamEvents. Therefore, explicit reference to StreamEvents would be redundant. Also, note that in addition to the payload type (e.g. ContextSwitch), there is another type parameter (e.g. Unit). This type parameter is used, in some situations, to catch query errors at compile time (i.e. in Visual Studio), and is otherwise uninteresting to users. We therefore defer further discussion of this type parameter until the relevant point in this guide. The DisorderPolicy specified is Drop, which means that any event which arrives out-of-order according to its relevant timestamp (e.g. start time for interval events), is dropped. In this example, there are no out-of-order events, so nothing is dropped. Figure 8 shows the contents of contextSwitchStreamable. Lastly, note that contextSwitchIngressStreamable is an instance of IObservableIngressStremable, an interface derived from IStreamable. The explicit cast to IStreamable is not necessary in practice, and only included here to outline the types.

StartTime	EndTime	Tick	ProcessId	CpuId	CpuTemp
0	1	0	1	1	120
0	1	0	3	2	121
0	1	0	5	3	124
120	121	120	2	1	123
300	301	300	1	1	122
1800	1801	1800	4	2	125
3540	3541	3540	2	1	119
3600	3601	3600	1	1	120

Figure 8: Contents of contextSwitchStreamable

Note that the first two fields, StartTime and EndTime, are timestamp fields present for every interval event in a stream, no matter the payload type. The three remaining fields are the actual payload data.

A conversion may also be performed from Streamables to Observables of StreamEvents. For instance, Figure 9 shows how we can convert our stream back to an Observable.

```
IObservable<StreamEvent<ContextSwitch>> passthroughContextSwitchStreamEventObservable =
    contextSwitchStreamable.ToStreamEventObservable();
```

Figure 9: Converting from Streamable back to Observable

Unlike the conversion to Streamable, there is no option of specifying a DisorderPolicy, since Observables don't have any notion of temporality or temporal well-formedness.

We have now authored our first Trill query, which is a passthrough query. Note that we haven't run our query yet. In order to do this, we need to initiate the process by creating a subscribe call on the output Observable. This can be done using the LINQ operator ForEach. The result is shown in Figure 10.

```
passthroughContextSwitchStreamEventObservable
    .Where(e => e.IsData)
    .ForEachAsync(e => Console.WriteLine(e.ToString())).Wait();
```

Figure 10: Executing a Streaming Query

Note the addition of the Where operator, which retains all events such that the "IsData" property is true. For the purpose of this example, only data events are relevant. Refer to Information Flow and the Passage of Time section for a detailed description of non-data events.

To explicitly show the result types of operations, we have, throughout our example, explicitly typed all variables. In practice, it is much easier using the “var” statement, which implicitly assigns the correct type to the result of an assignment statement. We therefore show our complete pass-through Trill program below, using “var” where appropriate.

Note the “using” statement at the top of the program. This statement refers to the assemblies that need to be included in every Trill program. There is no installation or configuration for these assemblies, which facilitates deployment in datacenter apps.

Through the rest of this guide, rather than presenting complete programs, we present code fragments, most of which are operations on Streamables. Refer to the QueryWritersGuide sample for a complete example.

```
using System;
using System.Reactive;
using System.Reactive.Linq;
using Microsoft.StreamProcessing;

internal class Program
{
    private struct ContextSwitch
    {
        public ContextSwitch(long tick, long pid, long cid, long cpuTemp)
        {
            this.Tick = tick;
            this.ProcessId = pid;
            this.CpuId = cid;
            this.CpuTemp = cpuTemp;
        }

        public long Tick;
        public long ProcessId;
        public long CpuId;
        public long CpuTemp;

        public override string ToString() => $"Tick={this.Tick}\tProcessId={this.ProcessId}\t" +
            $"CpuId={this.CpuId}\tCpuTemp={this.CpuTemp}";
    };

    private static void WriteEvent<T>(StreamEvent<T> e) => Console.WriteLine(
        $"Event Kind = Interval\tStart Time = {e.StartTime}\t" +
        $"End Time = {e.EndTime}\tPayload = ({e.Payload.ToString()})");

    public static void Main(string[] args)
    {
        IObservable<ContextSwitch> contextSwitchObservable = new[]
        {
            new ContextSwitch(0, 1, 1, 120),
            new ContextSwitch(0, 3, 2, 121),
            new ContextSwitch(0, 5, 3, 124),
            new ContextSwitch(120, 2, 1, 123),
            new ContextSwitch(300, 1, 1, 122),
            new ContextSwitch(1800, 4, 2, 125),
            new ContextSwitch(3540, 2, 1, 119),
            new ContextSwitch(3600, 1, 1, 120),
        }.ToObservable();

        IObservable<StreamEvent<ContextSwitch>> contextSwitchStreamEventObservable =
            contextSwitchObservable.Select(e => StreamEvent.CreateInterval(e.Tick, e.Tick + 1, e));
        IObservableIngressStreamable<ContextSwitch> contextSwitchIngressStreamable =
            contextSwitchStreamEventObservable.ToStreamable(DisorderPolicy.Drop());

        IObservable<StreamEvent<ContextSwitch>> passthroughContextSwitchStreamEventObservable =
            contextSwitchIngressStreamable.ToStreamEventObservable();

        passthroughContextSwitchStreamEventObservable
            .Where(e => e.IsData)
            .ForEachAsync(e => WriteEvent(e)).Wait();
    }
}
```

Figure 11: Complete Pass-Through Trill Query Program

Note that the above example, and the remaining queries in this guide, explicitly ingress and egress StreamEvent objects. However, some queries may more naturally be written without explicitly referencing StreamEvent. Figure 12: Non-StreamEvent Passthrough provides a passthrough example that extracts the start time as the ContextSwitch.Tick, and selects the ContextSwitch as the result.

```
var payloadStreamable = contextSwitchObservable.ToTemporalStreamable(cs => cs.Tick);
var passthroughObservable = payloadStreamable.ToTemporalObservable((start, cs) => cs);
```

Figure 12: Non-StreamEvent Passthrough

3 WHERE AND SELECT

Like other query languages, Trill can both filter and transform data. To do this, Trill adopts the same convention as LINQ to objects and Rx, except that Trill is much faster than both. Filtering in Trill is done using the Where operator. The query writer provides an expression that, given a payload, evaluates to true or false. If the result of applying the expression is true, the event is passed along, otherwise it is dropped. For instance, going back to our example, we now write the query described in Figure 13.

```
What are the context switch events on cores 1 and 2?
```

Figure 13: Where Query Text

The resulting LINQ query is shown in Figure 14: Where Query Code. Note that the expression passed to the where operator is simply shorthand for writing a function which takes a payload and returns a bool. Such inline functions are called lambdas. The p on the left side of the "=>" is the name for the input parameter used in the body of the function, which is to the right of the "=>". Also, note that the input is a payload and not an event. As a result, this version of Where may not use any temporal event information in the filter expression.

```
var contextSwitchTwoCores = contextSwitchStreamable.Where(p => p.CpuId == 1 || p.CpuId == 2);
```

Figure 14: Where Query Code

The result of the query in is shown in Figure 15.

StartTime	EndTime	Tick	ProcessId	CpuId	CpuTemp
0	1	0	1	1	120
0	1	0	3	2	121
120	121	120	2	1	123
300	301	300	1	1	122
1800	1801	1800	4	2	125
3540	3541	3540	2	1	119
3600	3601	3600	1	1	120

Figure 15: Where query result

Similarly, Trill also allows per-event transformation through the Select operator. This operator takes an expression which, given a payload, returns another payload. Note that this transformation should, like Where, be stateless. Stateful transformations are performed using other operations.

In our example, we use Select to answer the query shown in Figure 16.

What are the context switch events on cores 1 and 2, without the unneeded temperature information?

Figure 16: Select Query Text

In other words, the Select must drop the temperature information while retaining all the other information. Note that the output payload type, therefore, has one less field than the input payload type. Since defining a new explicit type in such situations can be very burdensome when writing queries in LINQ, C# has the ability to define new types, called anonymous types, as part of the expression passed into the Select statement. Figure 17

```
var contextSwitchTwoCoresNoTemp = contextSwitchTwoCores.Select(
        e => new { e.Tick, e.ProcessId, e.CpuId });
```

Figure 17: Select Query Code

shows how this is used in our example. Note that C# is careful to ensure that all anonymous types which have the same members defined in the same order are of the same type.

Note that LINQ doesn't require a redefinition of the field name when output fields are the same as input fields, including the field name. For instance, Figure 17 can be rewritten as shown in Figure 18.

```
var contextSwitchTwoCoresNoTemp = contextSwitchTwoCores.Select(
        e => new { e.Tick, e.ProcessId, e.CpuId });
```

Figure 18: Alternate Select Query Code

The corresponding query result is shown in Figure 19.

StartTime	EndTime	Tick	ProcessId	CpuId
0	1	0	1	1
0	1	0	3	2
120	121	120	2	1
300	301	300	1	1
1800	1801	1800	4	2
3540	3541	3540	2	1
3600	3601	3600	1	1

Figure 19: Select Query Result

In addition to accessing operator functionality through methods on IStreamable, Trill also supports some of the LINQ comprehension syntax, which makes these operations look like SQL. Figure 20 shows the comprehension syntax version of our where and select queries. While this syntax may be a bit easier to understand for SQL users, it adds no expressiveness to the query language, and cannot be used to write all aspects of LINQ (and therefore Trill) queries.

```
var contextSwitchTwoCoresNoTemp = from e in contextSwitchStreamable
                                  where e.CpuId == 1 || e.CpuId == 2
                                  select new { e.Tick, e.ProcessId, e.CpuId };
```

Figure 20: Where and Select with the LINQ Comprehension Syntax

4 JOIN (INNER JOIN)

Consider our original example in Figure 2: Initial Question to Focus On. In the end, we are supposed to output process names, not process IDs. Since there are no process names in our context switch information, we combine `contextSwitchTwoCoresNoTemp` with another stream called `namesStream`, which associates process IDs with their corresponding process names for the duration of time during which that association was valid. The contents of the `namesStream` is shown in Figure 21. In this example all five process ids to name associations are valid for the entire period of time covered by the context switch data. Note that this data model is expressive enough to describe process ids reused by the operating system. In such a case, the time periods covered by the two associations with the same ProcessId would be mutually exclusive, each covering the period of time during which its association was valid.

StartTime	EndTime	ProcessId	Name
0	10000	1	Word
0	10000	2	Internet Explorer
0	10000	3	Excel
0	10000	4	Visual Studio
0	10000	5	Outlook

Figure 21: Contents of namesStream

Going back to our example, we will now answer the query shown in Figure 22.

> What are the context switch events on cores 1 and 2, annotated with the correct process names?

Figure 22: Join Query Text

This brings us to a very important point, the meaning of the start and end timestamps. Trill interprets these timestamps as the period of time during which the associated data is used in calculations. For instance, the timestamps in the `namesStream` indicate that for each row, its information will be combined only with other information that shares some or all of its interval of validity (i.e. they temporally overlap).

With this information in mind, we now use an operator familiar to SQL and LINQ users called Join. Join is used in Trill to combine rows from two streams. Each possible pair, one from each stream, is considered for output, and if the provided predicate over the pair evaluates to true, and the two events temporally overlap, an output is produced. Otherwise, the pair is dropped.

In this case, if our predicate always evaluated to true, join would produce 35 rows: since all events in `contextSwitchStream` temporally overlap all events in `namesStream`, we would get all combinations of the seven context switch events with the five process name events. In reality, we only want the combinations where the ProcessId fields match. This is by far the most common type of join, called an equality join, which has an equality predicate between a field from one stream, and a key field (i.e. a field which uniquely identifies events) from the other stream. Figure 23 shows how `contextSwitchTwoCoresNoTemp and namesStream` are joined to produce a context switch stream annotated with process names.

```
var contextSwitchWithNames = contextSwitchTwoCoresNoTemp.Join(namesStream,
    e => e.ProcessId, e => e.ProcessId,
    (leftPayload, rightPayload) => new
    {
        leftPayload.Tick,
        leftPayload.ProcessId,
        leftPayload.CpuId,
        rightPayload.Name
    });
```

Figure 23: Join Query Code

First, note that the stream on which the method is called (i.e. `contextSwitchTwoCoresNoTemp`) is considered the "left" stream. The stream with which `contextSwitchTwoCoresNoTemp` is joined is passed in as the first argument. The second and third arguments are selectors for the left and right fields used in the equality predicate. In this case, we are only producing output for pairs where the ProcessIds from the two streams match. Finally, the last argument is a selector, which when output is produced for a pair of events, computes the output payload, given the two input payloads for the pair. In this case we keep all fields from `contextSwitchTwoCoresNoTemp`, but only keep the `Name` field from `namesStream`. The contents of `contextSwitchWithNames` are shown in Figure 24.

StartTime	EndTime	Tick	ProcessId	CpuId	Name
0	1	0	1	1	Word
0	1	0	3	2	Excel
120	121	120	2	1	Internet Explorer
300	301	300	1	1	Word
1800	1801	1800	4	2	Visual Studio
3540	3541	3540	2	1	Internet Explorer
3600	3601	3600	1	1	Word

Figure 24: Join Query Result, contextSwitchWithNames

One important aspect of the output in Figure 24 are the lifetimes. They are identical to the lifetimes of `contextSwitchTwoCoresNoTemp`. This makes sense when one considers that for all output producing pairs of events, the contextSwitchTwoCoresNoTemp event's lifetime is a subset of the `namesStream` event's lifetime. The contextSwitchTwoCoresNoTemp event's lifetime is exactly the period of time during which both payloads were used in the calculation of output. It is therefore this period of time in which the output was produced.

Like where and select, join can also be written using the comprehension syntax. Figure 25 shows how our join query can be written using this alternate syntax.

```
var contextSwitchWithNames = from leftPayload in contextSwitchTwoCoresNoTemp
                             join rightPayload in namesStream on
                                 leftPayload.ProcessId equals rightPayload.ProcessId
                             select new
                             {
                                 leftPayload.Tick,
                                 leftPayload.ProcessId,
                                 leftPayload.CpuId,
                                 rightPayload.Name
                             };
```

Figure 25: Join Query Comprehension Syntax

Similarly, it is possible to write the entire query so far using the comprehension syntax. Figure 26: Entire Join Query Comprehension Syntax shows how this is done.

```
var contextSwitchWithNames = from leftPayload in contextSwitchStreamable
                             join rightPayload in namesStream on
                                 leftPayload.ProcessId equals rightPayload.ProcessId
                             where leftPayload.CpuId == 1 || leftPayload.CpuId == 2
                             select new
                             {
                                 leftPayload.Tick,
                                 leftPayload.ProcessId,
                                 leftPayload.CpuId,
                                 rightPayload.Name
                             };
```

Figure 26: Entire Join Query Comprehension Syntax

5 AlterEventDuration

Going back to our example, in order to compute how much CPU time a process is consuming, we must compute, in the payload, the duration of time for each context switch. To do this, we must combine consecutive context switches on the same core, so that we may determine the duration of each time slice. Combining consecutive events on the same core should be done with join. Unfortunately, none of the event lifetimes in the `contextSwitchWithNames` stream overlap, yet the events – and their associated payloads – can only be brought together during periods when the lifetimes overlap.

We therefore introduce a new operator called `AlterEventDuration` that query writers use to manipulate lifetimes, and therefore change how data is combined in subsequent computations. For instance, Figure 27: AlterEventDuration Query Code shows how we can extend the lifetime of every event so that every lifetime is infinitely long (i.e. the data can be combined with all other data in the fullness of time).

```
var infiniteContextSwitch =
        contextSwitchWithNames.AlterEventDuration(StreamEvent.InfinitySyncTime);
```

Figure 27: AlterEventDuration Query Code

The parameter passed to the operator is the new lifetime of the event. Note that this version of AlterEventDuration only allows constant lifetimes. As we will see later, there is another overload that allows the computation of the lifetime to make use of the initial start and end times of the transformed event. It is worth noting, though, that the overload used in this example should be used whenever possible, as performance will be significantly better. The AlterEventDuration query result is shown in Figure 28: AlterEventDuration Query Result, infiniteContextSwitch.

StartTime	EndTime	Tick	ProcessId	CpuId	Name
0	InfinitySyncTime	0	1	1	Word
0	InfinitySyncTime	0	3	2	Excel
120	InfinitySyncTime	120	2	1	Internet Explorer
300	InfinitySyncTime	300	1	1	Word
1800	InfinitySyncTime	1800	4	2	Visual Studio

3540	InfinitySyncTime	3540	2	1	Internet Explorer
3600	InfinitySyncTime	3600	1	1	Word

Figure 28: AlterEventDuration Query Result, infiniteContextSwitch

6 ClipEventDuration

Now we have brought all of our data together, but it is actually too much, since we only want to join consecutive context switches on the same core. We therefore make use of an operator called `ClipEventDuration` that clips an event's lifetime using the first qualifying event from a (potentially) different stream with a later timestamp. In other words, for each event in `infiniteContextSwitch`, we clip its lifetime with the first event from the same stream which is on the same core with a later timestamp.

The signature for `ClipEventDuration` is similar to Join in that, given two streams, we find pairs of matching events. Unlike Join, we don't consider the cross product. Rather, we match each event on the left with at most one event on the right. The output is all the events on the left, possibly with shorter lifetimes. Figure 29 shows the code to clip the lifetimes of the events in with the lifetimes of consecutive events on the same core.

Like join, there are two selectors, one for the left stream and one for the right. These selectors, like Join, are used in the equality predicate that matches events on the left and events on the right. This particular clip, given an event on the left, matches the first subsequent event on the right whose lifetime overlaps, and who's CPU Id is equal to the CPU Id of the event on the left. The `clippedContextSwitch` query result is shown in Figure 30.

```
var clippedContextSwitch = infiniteContextSwitch
        .ClipEventDuration(infiniteContextSwitch, e => e.CpuId, e => e.CpuId);
```

Figure 29: ClipEventDuration Query Code

StartTime	EndTime	Tick	ProcessId	CpuId	Name
0	120	0	1	1	Word
0	1800	0	3	2	Excel
120	300	120	2	1	Internet Explorer
300	3540	300	1	1	Word
1800	InfinitySyncTime	1800	4	2	Visual Studio
3540	3600	3540	2	1	Internet Explorer
3600	InfinitySyncTime	3600	1	1	Word

Figure 30: ClipEventDuration Query Result, clippedContextSwitch

Note that last event for each of the two CPU Ids remain unclipped, because there were no subsequent matching events for the ClipEventDuration operator to use for the clip.

7 Multicast

While the code shown in Figure 29 will produce the correct result using the techniques for ingressing static data shown in these examples, when executed, it has some interesting behavior: rather than computing `infiniteContextSwitch` once and using the results for both inputs to ClipEventDuration, it will instead compute

infiniteContextSwitch twice, once for each ClipEventDuration input. This behavior is standard across all LINQ dialects. While inefficient for offline processing, this can be even more problematic for real time queries, where multiple subscriptions to a data source may not be possible, and where two subscriptions that start at different times may produce different actual input!

We therefore introduce the operator Multicast, which allows a single stream to be shared with multiple parts of a query plan, with the requirement that they must, ultimately, be combined back into a single output stream. Figure 31 shows the clip query code, rewritten using multicast.

```
var clippedContextSwitch = infiniteContextSwitch.Multicast(
        s => s.ClipEventDuration(s, e => e.CpuId, e => e.CpuId));
```

Figure 31: Multicast Version of ClipEventDuration Query Code

This Multicast shares the stream infiniteContextSwitch by exposing it as the input (in this example, "s") to its argument lambda. The stream valued function computed by the argument lambda is the output after all the different copies of s have been brought back together. In this case, the two copies of s are brought back together using ClipEventDuration, whose output is returned by the passed function.

8 SHIFTEVENTLIFETIME

Continuing with our example, we will now use an operator called ShiftEventLifetime that, like AlterEventDuration, modifies the time interval associated with each event. This operator, combined with join and another multicast, allows us to solve the query shown in Figure 32.

```
What are the the process names and time slice durations for all processes that ran on cores 1 and 2?
```

Figure 32: Timeslice Query Text

In order to answer the query, we must bring together each event in clippedContextSwitch with the subsequent context switch on the same core. We are quite close to being able to do this. Since we clipped each infinite context switch with the subsequent context switch on the same core, each context switch lifetime is just one time unit (called a chronon) before the subsequent context switch. We could, therefore, bring the two context switches together with join if we could shift the lifetimes of the events in clippedContextSwitch forward by one chronon. Since both the start and end times would be pushed forward one chronon, each event would join only to the following event on the same core, and not to itself. Figure 33 shows how ShiftEventLifetime shifts event lifetimes forward by one chronon.

```
var shiftedClippedContextSwitch = clippedContextSwitch.ShiftEventLifetime(1);
```

Figure 33: ShiftEventLifetime Query Code

The resulting stream contents of shiftedClippedContextSwitch, are shown in Figure 34: ShiftEventLifetime Query Result, shiftedClippedContextSwitch, and contextSwitchWithNames, along with contextSwitchWithNames again from the Join section for comparison. Notice that consecutive context switches share one chronon of time across the two tables. E.g., the first event in shiftedClippedContextSwitch for Cpu 1, Process 1, "Word", has a valid time interval from [1, 121); the event for the subsequent context switch on the same core in

`contextSwitchWithNames` for Cpu 1, Process 2 “Internet Explorer”, has a valid time interval from [120, 121). These share exactly one chronon of time – [120, 121), while no other event for Cpu 1 overlaps this range.

shiftedClippedContextSwitch					
StartTime	EndTime	Tick	ProcessId	CpuId	Name
1	121	0	1	1	Word
1	1801	0	3	2	Excel
121	301	120	2	1	Internet Explorer
301	3541	300	1	1	Word
1801	InfinitySyncTime	1800	4	2	Visual Studio
3541	3601	3540	2	1	Internet Explorer
3601	InfinitySyncTime	3600	1	1	Word
contextSwitchWithNames					
StartTime	EndTime	Tick	ProcessId	CpuId	Name
0	1	0	1	1	Word
0	1	0	3	2	Excel
120	121	120	2	1	Internet Explorer
300	301	300	1	1	Word
1800	1801	1800	4	2	Visual Studio
3540	3541	3540	2	1	Internet Explorer
3600	3601	3600	1	1	Word

Figure 34: ShiftEventLifetime Query Result, shiftedClippedContextSwitch, and contextSwitchWithNames

With the previous observation in mind, we can join the two streams together and compute the correct timeslices. The code for computing the query in Figure 32 is shown in Figure 35, and the results in Figure 36.

```
var timeslices = shiftedClippedContextSwitch.Join(contextSwitchWithNames,
    e => e.CpuId, e => e.CpuId,
    (left, right) => new
    {
        left.ProcessId,
        left.CpuId,
        left.Name,
        Timeslice = right.Tick - left.Tick
    });
```

Figure 35: Timeslices Query Code

StartTime	EndTime	ProcessId	CpuId	Name	Timeslice
120	121	1	1	Word	120
300	301	2	1	Internet Explorer	180
1800	1801	3	2	Excel	1800
3540	3541	1	1	Word	3240
3600	3601	2	1	Internet Explorer	60

Figure 36: Timeslices Query Result

Note that the query in Figure 35, similar to our example in Section 7, refers to the `contextSwitchWithNames` stream twice, once explicitly, and once in the computation of `shiftedClippedContextSwitch`. We therefore need to use Multicast to eliminate this problem. The result is shown in Figure 37.

```
var timeslices = contextSwitchWithNames.Multicast(t => t
    .AlterEventDuration(StreamEvent.InfinitySyncTime)
    .Multicast(s => s
        .ClipEventDuration(s, e => e.CpuId, e => e.CpuId))
        .ShiftEventLifetime(1)
        .Join(t,
            e => e.CpuId, e => e.CpuId,
            (left, right) => new
            {
                left.ProcessId,
                left.CpuId,
                left.Name,
                Timeslice = right.Tick - left.Tick
            }));
```

Figure 37: Timeslices Query Code Using Multicast

9 AlterEventLifetime and Sum

Going back to our initial query, shown in Figure 2, we need to know, for each process, for each core, how much CPU time the process consumed. In order to compute this information, we first consider the query in Figure 38.

> Given all the timeslice information events for a particular process for a particular core, how do I compute the CPU consumption for that process in 3600 tick periods?

Figure 38: Hourly CPU Consumption for One Process on One Core

Figure 39 shows example input to the query posed in Figure 38, and for the simplicity of this section, only considers process 1 on core 1.

StartTime	EndTime	ProcessId	CpuId	Name	Timeslice
120	121	1	1	Word	120
3540	3541	1	1	Word	3240

Figure 39: Aggregate Query Input, timeslicesForProcess1Cpu1

At this point in our example query, we will always assume that processes are context switched at every 3600 tick boundary, even if it is immediately context switched back in. We will eliminate this assumption later in section 11. Under this assumption, we can assume that if, on each 3600 tick boundary, we sum all of the timeslices which occurred in the 3600 tick period prior to that boundary, we have computed the CPU consumption, in ticks, for that period. In other words, for each 3600 tick period, we wish to bring together and sum all timeslices which occurred during that period. Assuming the output is a signal, which at each point in time, measures the last reported sum, each event in Figure 39 contributes to the result reported during the 3600 tick period which begins on or after the context switch, since the start time of the event is when the process was context switched out. This is shown pictorially in Figure 40, where the lifetimes of our two context switches at 120 and 3540 are mapped into the subsequent 3600 tick period during which they contribute to the current answer.

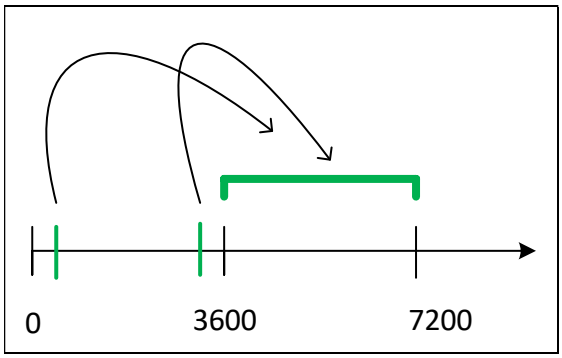

Figure 40: Lifetime conversion before aggregation

With the two lifetime changing operators introduced do far, ShiftEventLifetime and AlterEventDuration, one can perform the lifetime transformation depicted in Figure 40, but for convenience, we introduce a single operator called AlterEventLifetime that can change both the start time and event duration. In this case, we want to push forward the start time to the correct reporting time, and set the duration to 3600 ticks, which is the time between reported values. Figure 41 shows how we reassign timeslice lifetimes in this manner.

```
var windowedTimeslices = timeslicesForProcess1Cpu1.
        AlterEventLifetime(origStartTime => (1 + ((origStartTime - 1) / 3600)) * 3600, 3600);
```

Figure 41: Rassigning Timeslice Lifetimes with AlterEventLifetime

In fact, this technique is frequently used for lifetime reassignment, called hopping window lifetimes, where something is computed using all the data from the last hop size amount of time, and where results are produced every hop period units of time. In our example, the hop size and hop period are the same, but not necessarily so. For instance, we might want to know, every second (hop period), the resources consumed over the last day (hop size). For convenience, we include in Trill an operator for assigning hopping window lifetimes. This is shown, for our example, in Figure 42. In this example, the two parameters are the hop size and period respectively.

```
var windowedTimeslicesForProcess1Cpu1 =
    timeslicesForProcess1Cpu1.HoppingWindowLifetime(3600, 3600);
```

Figure 42: Reassigning Lifetimes with HoppingWindowLifetime

Now that we have brought the correct data together by manipulating their lifetimes, we simply apply the operation we want to perform for each unique collection of data. In this case, for each collection of data (3600 tick period), we wish to sum the Timeslice field. Figure 43 shows how this is done.

```
var totalConsumptionPerPeriodForProcess1Cpu1 =
    windowedTimeslicesForProcess1Cpu1.Sum(e => e.Timeslice);
```

Figure 43: Sum Query Code

The result is a stream of events, each of which covers a 3600 tick reporting period which reports the aggregate timeslice information for the prior 3600 period. Note that the output payload is a long, not a struct or class, which is why no field name is needed. The results are shown in Figure 44.

StartTime	EndTime	Payload
3600	7200	3360

Figure 44: Sum Query Results

In this case, there is only one output row, since there was only one reporting period that contained data.

10 Group and Apply

In section 9, we limited our sum query to a particular process for a particular core. In reality, we want to perform the same combination of HoppingWindowLifetime and Sum for each process-core combination. This is precisely what the GroupAndApply operation is for. GroupAndApply partitions an input stream according to a partitioning expression, and for each partition, evaluates the same streaming query. The results are then combined with a union operation, which ensures that time order is maintained in the final result.

In our case, we partition the timeslices stream by process and core, and for each partition, run our query from section 9. The code for this is shown in Figure 45.

```
var totalConsumptionPerPeriod = timeslices.GroupApply(
        e => new { e.CpuId, e.ProcessId, e.Name },
        s => s.HoppingWindowLifetime(3600, 3600).Sum(e => e.Timeslice),
        (g, p) => new { g.Key.CpuId, g.Key.Name, TotalTime = p });
```

Figure 45: Group and Apply Query Code

More specifically, the group and apply operator partitions each individual event in the timeslices stream with all events that have the same CpuId/ProcessId combination. The expression in the first parameter evaluates, for each event, which partition that event goes to. For each partition, the streaming subquery described in the second parameter is run. The third parameter is a selector that takes each output event from the streaming subquery and combines it with the return value of the partitioning function for that stream, which is important for this query, since the payload after the Sum operator has dropped the ProcessId and CpuId for the current stream. We therefore use the final selector to put these fields back. The output for this group and apply query is shown in Figure 46.

StartTime	EndTime	CpuId	Name	TotalTime
3600	7200	1	Word	3360
3600	7200	1	Internet Explorer	240
3600	7200	2	Excel	1800

Figure 46: Group and Apply Query Result

11 Chop

In our example so far, no timeslices spanned our reporting boundaries. Without this assumption, our query would have returned an incorrect result. Consider the alternate input shown in Figure 47.

Tick	ProcessId	CpuId	CpuTemp
0	1	1	120
0	3	2	121
0	5	3	124
120	2	1	123

300	1	1	122
1800	4	2	125
3540	2	1	119
3600	1	1	120
5400	3	2	122
7200	4	2	121

Figure 47: Alternate ContextSwitch Data (alternativeContextSwitchStream)

The alternate timeslices output is shown in Figure 48.

StartTime	EndTime	ProcessId	CpuId	Name	Timeslice
120	121	1	1	Word	120
300	301	2	1	Internet Explorer	180
1800	1801	3	2	Excel	1800
3540	3541	1	1	Word	3240
3600	3601	2	1	Internet Explorer	60
5400	5401	4	2	Visual Studio	3600
7200	7201	3	2	Excel	1800

Figure 48: Alternate Timeslices Output

Observe the additional two rows in the alternate timeslices table. While this table correctly computes when processes were context-switched out, and how long they ran, consider what happens when we run this through the rest of our query. The group and apply output is shown in Figure 49.

StartTime	EndTime	CpuId	Name	TotalTime
3600	7200	1	Word	3360
3600	7200	1	Internet Explorer	240
3600	7200	2	Excel	1800
7200	10800	2	Visual Studio	3600
7200	10800	2	Excel	1800

Figure 49: Alternate Group and Apply Query Result

There are two problems with the output shown in Figure 49:

1. Incorrect output: Note that according to the output, Visual Studio and Excel collectively consumed 5400 cycles of the 3600 available cycles. This impossibility happened because 1800 of the Visual Studio cycles were incorrectly attributed to the last reported period. They should, instead, have been correctly billed to the previous reporting period. This occurred because all cycles for a context switch are attributed to the time period when the process is switched out.
2. Missing output: What happened to the output for core 3? We're issuing output for the second time period, but are completely missing the output for core 3 for the first time period. It's actually highly related to the incorrect output problem. Since there has only been one context switch on core 3, we are waiting for the next context switch before attributing the time for the long running process.

Both problems above can be addressed by introducing an artificial context switch at every reporting boundary for every core. More specifically, if there isn't already a context switch at the reporting period boundary, we logically introduce one by context switching the current process to itself. This approach addresses both problems above.

We create these extra context switches using an operation called Chop. Chop creates, from a single event, multiple events with identical payloads and consecutive lifetimes that, collectively, cover the entire lifetime of the original event. Therefore, we are creating the new lifetimes by "chopping" the original lifetime. The chop locations are periodic and are specified using a period and an offset. Therefore, if we take the original context switches, create lifetimes to the next context switch on that core, and then chop these lifetimes with a period equal to our reporting period, we will generate the needed extra context switches.

```
var contextSwitchChoppedUnbounded = alternativeContextSwitchStream
    .AlterEventDuration(StreamEvent.InfinitySyncTime)
    .Multicast(s => s.ClipEventDuration(s, e => e.CpuId, e => e.CpuId))
    .Chop(0, 3600)
    .Select((origStartTime, e) =>
        new { Tick = origStartTime, e.ProcessId, e.CpuId, e.CpuTemp })
    .AlterEventDuration(1);
```

Figure 50: Chop Query Code

Figure 50 shows how these extra context switches are created. Note that we first need to generate the lifetimes using AlterEventDuration, Multicast, and ClipEventDuration. We then Chop these lifetimes every 3600 ticks aligned to a chop on the 0^{th} tick. Finally, we replace the original ContextSwitch.Tick value with the start times of the events. This ensures that the extra events created by the chop no longer have the original ContextSwitch.Tick value, but rather have the beginning of the new lifetime after chopping. Finally, we change the lifetimes back to 1 chronon, resulting in context switch data just like the original `contextSwitchStreamable`, but with the extra context switches. The contents of `contextSwitchChoppedUnbounded` is shown in Figure 51.

Tick	ProcessId	CpuId	CpuTemp
0	1	1	120
0	3	2	121
0	5	3	124
120	2	1	123
300	1	1	122
1800	4	2	125
3540	2	1	119
3600	1	1	120
3600	4	2	125
3600	5	3	124
5400	3	2	122
7200	1	1	120
7200	4	2	121
7200	5	3	124
10800	1	1	120
10800	4	2	121
10800	5	3	124
…	…	…	…

Figure 51: Chop Query Result

Note the …s at the end of Figure 51. This is to indicate that the three final rows of the table, which represent the processes 1, 4, and 5 running on cores 1, 2, and 3 are repeated for all time, leading to a massive amount of output. This happens because these three lifetimes are not clipped, since they correspond to the final context switch events for each core. To avoid producing infinite output in this case, we simply join the unchopped lifetimes with a stream that has a single element with a fixed time interval, limiting the reporting of output to the period described by that interval. The resulting code is shown in Figure 52.

```
var fixedInterval = new[] { StreamEvent.CreateInterval(0, 10800, Unit.Default) }
        .ToObservable().ToStreamable();
var contextSwitchChopped = alternativeContextSwitchStream
        .AlterEventDuration(StreamEvent.InfinitySyncTime)
        .Multicast(s => s.ClipEventDuration(s, e => e.CpuId, e => e.CpuId))
        .Join(fixedInterval, (left, right) => left)
        .Chop(0, 3600)
        .Select((origStartTime, e) =>
            new { e.CpuId, e.ProcessId, e.CpuTemp, Tick = origStartTime })
        .AlterEventDuration(1);
```

Figure 52:Improved Chop Query Code

The contents of `contextSwitchChopped` is shown in Figure 53.

StartTime	EndTime	Tick	ProcessId	CpuId	CpuTemp
0	1	0	1	1	120
0	1	0	3	2	121
0	1	0	5	3	124
120	121	120	2	1	123
300	301	300	1	1	122
1800	1801	1800	4	2	125
3540	3541	3540	2	1	119
3600	3601	3600	1	1	120
3600	3601	3600	4	2	125
3600	3601	3600	5	3	124
5400	5401	5400	3	2	122
7200	7201	7200	1	1	120
7200	7201	7200	4	2	121
7200	7201	7200	5	3	124

Figure 53: Improved Chop Query Result

As a result of adding the join, the output in Figure 53 is truncated. Note the use of Unit and Unit.Default in the Join. This is to signify that the payload is empty (of type Unit), and the value of an empty payload is Unit.Default.

12 Final Query

Now we have all the pieces, we can use the query for “chopped” context switches as the input to the timeslices query, then compute the maximum total consumed CPU for each 3600 tick period per process. For completeness, the entire query is provided in Figure 54: Final Query, and its results in Figure 55: Final Query Result.

```
var choppedContextSwitch = alternativeContextSwitchStream
        .Where(cs => cs.CpuId == 1 || cs.CpuId == 2)
        .AlterEventDuration(StreamEvent.InfinitySyncTime)
        .Multicast(s => s.ClipEventDuration(s, e => e.CpuId, e => e.CpuId))
        .Join(fixedInterval, (left, right) => left)
        .Chop(0, 3600)
        .Select((start, e) => new { Tick = start, e.ProcessId, e.CpuId, e.CpuTemp })
        .AlterEventDuration(1);

var choppedContextSwitchWithNames = choppedContextSwitch
    .Join(namesStream, e => e.ProcessId, e => e.ProcessId, (left, right) => new
        {
            left.Tick,
            left.ProcessId,
            left.CpuId,
            right.Name
        });

var timeslicesPerCpu = choppedContextSwitchWithNames
    .Multicast(t => t
        .AlterEventDuration(StreamEvent.InfinitySyncTime)
        .Multicast(s => s.ClipEventDuration(s, e => e.CpuId, e => e.CpuId))
        .ShiftEventLifetime(1)
        .Join(t,
            e => e.CpuId, e => e.CpuId,
            (left, right) => new
            {
                left.ProcessId,
                left.CpuId,
                left.Name,
                Timeslice = right.Tick - left.Tick
            }));

var mostCpuConsumedPerPeriod = timeslicesPerCpu
    .GroupApply(
        e => new { e.ProcessId, e.Name },
        s => s.HoppingWindowLifetime(3600, 3600).Sum(e => e.Timeslice),
        (g, p) => new { g.Key.Name, TotalTime = p })
    .Max((l, r) => l.TotalTime.CompareTo(r.TotalTime))
    .Select((startTime, payload) => new
    {
        PeriodStart = startTime - 3600,
        PeriodEnd = startTime,
        ProcessName = payload.Name,
        TotalCpuConsumed = payload.TotalTime
    });
```

Figure 54: Final Query

PeriodStart	PeriodEnd	ProcessName	TotalCpuConsumed
0	3600	Word	3360
3600	7200	Word	3600

Figure 55: Final Query Result

13 Information Flow and the Passage of Time

So far, in our example, we have approached query writing with the mindset that we have all the (temporal) input data, and are performing (temporal) operations over this dataset to produce the correct output. This approach is very similar to writing SQL queries, and is the right mindset to have when authoring query logic. In real time processing, though, the data arrives and is presented to Trill at particular points in time. When the data is presented to Trill, output may, at that time, also be produced. In order for Trill to correctly produce output before all data has arrived, Trill must know that all data, up to some application time, has been received, and therefore any result computed based on input up to that application time is correct. In other words, Trill must be aware of the passage of application time in the input, and propagate that progress into the output.

13.1 Coping with Data Disorder

In Trill, all data disorder is removed when data is ingressed into Trill, with the options of either throwing, dropping, or adjusting the time of out-of-order data (reordering the data within a specified latency constraint is also supported, and is discussed separately in the next subsection). We define the notion of sync time, which is the application time, associated with an event, which is used to determine if that event is out-of-order. All sync times of all events, after dropping or adjusting, must be non-decreasing. The sync time of interval events is their start time. Therefore, the start times of all the interval data must be ingressed into Trill in order. For instance, consider the out-of-order input shown in Figure 56.

```
var outOfOrderStreamableThrow = new[]
{
    StreamEvent<int>.CreateInterval(10, 100, 1),
    StreamEvent<int>.CreateInterval(0, 50, 2),
    StreamEvent<int>.CreateInterval(0, 10, 3),
    StreamEvent<int>.CreateInterval(11, 90, 4)
}.ToObservable().ToStreamable(DisorderPolicy.Throw());
```

Figure 56: Out-of-Order Input (Throw)

Note that the second event (i.e. with payload value “2”) has an earlier start time than the preceding event. Since the disorder policy is set to throw, the query will throw on the thread which tries to ingress the data. Alternatively, look at the modified example in Figure 57.

```
var outOfOrderStreamableDrop = new[]
{
    StreamEvent<int>.CreateInterval(10, 100, 1),
    StreamEvent<int>.CreateInterval(0, 50, 2),
    StreamEvent<int>.CreateInterval(0, 10, 3),
    StreamEvent<int>.CreateInterval(11, 90, 4)
}.ToObservable().ToStreamable(DisorderPolicy.Drop());
```

Figure 57: Out-of-Order Input (Drop)

In this version, we drop out-of-order events instead of throwing. We therefore drop the second and third events, because their sync times (i.e. start times) are less than the sync time of the last event which was not dropped (i.e., the first event). We keep the fourth event, since the fourth event’s sync time (i.e. start time) is greater than or equal to the sync time of the first event. Finally, consider the modified example in Figure 58.

```
var outOfOrderStreamableAdjust = new[]
{
    StreamEvent<int>.CreateInterval(10, 100, 1),
    StreamEvent<int>.CreateInterval(0, 50, 2),
    StreamEvent<int>.CreateInterval(0, 10, 3),
    StreamEvent<int>.CreateInterval(11, 90, 4)
}.ToObservable().ToStreamable(DisorderPolicy.Adjust());
```

Figure 58: Out-of-Order Input (Adjust)

This version of our example uses the Adjust policy. In this version, if an event is out-of-order, its sync time is moved to the sync time of the last ingressed event. For instance, the start time of the second event is moved to 10. In cases, like the third event, where the resulting lifetime would be invalid, the event is dropped. The fourth event in this example is unchanged, since it's not out-of-order.

13.2 Reordering Data within Trill

All the disorder policies now accept an optional *reorderLatency* argument. This argument indicates the duration of application time that Trill waits before reordering events and processing them. The default value is 0, which produces the exact behavior described above.

The example below shows use of DisorderPolicy.Drop with a reorder latency of 100. This means that as data arrives and moves the maximum sync time to T, we reorder and release events up to (T - 100). Any late-arriving data with a timestamp of lower than (T - 100) would be dropped. One could instead choose a DisorderPolicy of Adjust or Throw if the out-of-order events should not be dropped. In addition, if the user manually pushes a punctuation into the system with a timestamp T' that is greater than (T - 100), Trill will reorder and process events until T'. The usual punctuation policy and drop policies are performed on the events released by the reordering operation. The user can also set a reorderLatency of infinity (StreamEvent.InfinitySyncTime), which forces Trill to reorder and output events only when punctuations are manually ingressed by the user.

```
var s = observableOfStreamEvent.ToStreamable(DisorderPolicy.Drop(100));
```

For example, assume with the above query that we get an input data stream that receives events with sync timestamps of 2, 3, 21, 15, 63, 42, 84, and 105 (in that order). At the moment we receive 105, the *high-water-mark* of the stream (max. sync time across all events) reaches 105. With a reorder latency of 100, this means that Trill needs to reorder and process data until timestamp 105-100=5. This will cause Trill to release the data events with timestamps 2 and 3 for further processing (i.e., applying the punctuation policy).

Next, suppose we get an event with a sync-time of 1. Since this event has a timestamp <= 5 (the current high watermark – reorder latency), it is released immediately for further processing and will be dropped because of our specified disorder policy of Drop().

Next, suppose we get an event with sync-time 132, which causes us move Trill's output to timestamp 132-100=32. This results in a production of the events 15 and 21 in that order. Note that the disorder between 15 and 21 has been eliminated in this process.

Next, if we get an input punctuation with sync-time 100, we force the reorder and processing of all events up to and including timestamp 100, even though our high watermark is still 132. This causes us to release events 42, 63, 84 in that order, followed by the punctuation at 100.

Finally, we get a punctuation at StreamEvent.InfinitySyncTime, which causes us to process and output data events 105 and 132, followed by the punctuation at infinity.

13.3 EDGE EVENTS

In some cases, ingressing intervals into Trill in start time order presents an output timeliness problem: consider the namesStream shown in Figure 21. The ProcessId/Name associations are given, along with the lifetime of the association, which describes the period of time during which the association was valid. At the time a ProcessId/ association becomes valid, it is not known when that association will become invalid. If we insist that all streaming data is entered into the system as fully specified interval events, we must wait until the ProcessId/Name association becomes invalid before entering the association into the system. Furthermore, since, for composability, interval data must be produced in start time order, we must delay the output of any other interval whose start time is greater than the unended event. This could result in arbitrary delays in the production of output, while we wait for a ProcessId/Name association to end.

We therefore introduce edge events. Edge events enable the expression of an interval event as two edge events. The first edge, called a start edge, establishes the start time, the provisional end time of StreamEvent. InfinitySyncTime, and payload of the event. The end edge, which arrives later, establishes the (finite) end time associated with the earlier matching start edge. Start edges are not required to have matching end edges. In these cases, the event is semantically treated as having an unbounded lifetime. Figure 59 shows the edge version of the namesStream stream. Note that start edges only have start times and payloads, while end edges have start times, end times, and payloads. The start times and payloads for end edges are used to identify which start edge is associated with the end edge.

Edge Type	StartTime	EndTime	ProcessId	Name
Start	0	Infinity	1	Word
Start	0	Infinity	2	Internet Explorer
Start	0	Infinity	3	Excel
Start	0	Infinity	4	Visual Studio
Start	0	Infinity	5	Outlook
End	0	10000	1	Word
End	0	10000	2	Internet Explorer
End	0	10000	3	Excel
End	0	10000	4	Visual Studio
End	0	10000	5	Outlook

Figure 59: Edge based namesStream

In Trill, edge events are a kind of StreamEvent. The code to ingress the edge version of the namesStream stream is shown in Figure 60. Note the two selects in the LINQ query – the first Select creates all the start edges, all of which happen at time 0, while the second Select creates all the end edges, all of which happen at time 10,000. The Concat operator is used to append the results of the second Select to the results of the first Select.

```
private struct ProcessName
{
    public ProcessName(long processId, string processName)
    {
        this.ProcessId = processId;
        this.Name = processName;
    }

    public long ProcessId;
    public string Name;
};
...
var processNamesObservable = new[]
    {
        new ProcessName(1, "Word"),
        new ProcessName(2, "Internet Explorer"),
        new ProcessName(3, "Excel"),
        new ProcessName(4, "Visual Studio"),
        new ProcessName(5, "Outlook"),
    }.ToObservable();
var namesStream = processNamesObservable
        .Select(=> StreamEvent<ProcessName>.CreateStart(0, e))
        .Concat(processNamesObservable.Select(
            e => StreamEvent<ProcessName>.CreateEnd(10000, 0, e)))
        .ToStreamable();
```

Figure 60: Creating namesStream with Edge Events

Note that interval events and edge events can be mixed together in Trill streams. For instance Figure 61 shows an alternate representation of the namesStream which is semantically equivalent to previous examples. Note that the first three events are interval events, while the last four events are edge events. Together, these events describe the exact same interval events as all previous examples of namesStream. From Trill's point of view, specifying the input as edge events instead of interval events has no real impact on the query result. Edge events simply enable the timely presentation of the same information to Trill. Similarly, Trill itself will output edge events to convey information about the output in a timely fashion. Consumers of Trill output must, therefore, be prepared to handle a combination of interval and edge events.

```
var processNamesObservable1 = new[]
    {
        new ProcessName(1, "Word"),
        new ProcessName(2, "Internet Explorer"),
        new ProcessName(3, "Excel"),
    }.ToObservable();
var processNamesObservable2 = new[]
    {
        new ProcessName(4, "Visual Studio"),
        new ProcessName(5, "Outlook"),
    }.ToObservable();
var namesStream = namesObservable1.Select(e => StreamEvent.CreateInterval(0, 10000, e))
        .Concat(namesObservable2.Select(e => StreamEvent.CreateStart(0, e)))
        .Concat(namesObservable2.Select(e => StreamEvent.CreateEnd(10000, 0, e)))
        .ToStreamable();
```

Figure 61: Creating namesStream using Interval and Edge Events

When performing queries over offline data, it is frequently convenient, when viewing results, to coalesce matching start and end edges into the single events they truly describe. We therefore provide an optional reshaping policy parameter to ToStreamEventObservable that coalesces these edges. WARNING: Coalescing edges may cause very high latency, which may be acceptable for offline queries, but is highly discouraged for real time queries. Figure 62 shows the ToStreamEventObservable call which coalesces edges for the query in Figure 61.

```
namesStream.ToStreamEventObservable(ReshapingPolicy.CoalesceEndEdges())
```

Figure 62: Coalescing Matching Edges

Observe that the sync time of a start edge is the start time, while the sync time of an end edge is the end time. Therefore, in the example above, we first ingress the interval events, all of which have sync times of 0, followed by the start edges, which also have sync times of 0, and finish with the end edges, which all have sync times of 10000. This input is, therefore, ordered.

13.4 PUNCTUATIONS

In Trill, data itself marks the passage of time. Specifically, as the sync time of input data advances, time in the output overall also advances. However, there are two ways in which output time fails to advance as desired:

1) There may be a long period during which no data is sent, even though time is passing. Since there is no data, no output is produced and output time doesn't advance. This is particularly problematic for multi-input queries, where one input may be sparse, and the other dense. Output will not be produced unless data is received on both inputs.
2) In order to improve throughput, Trill "batches" many inputs into a single location in memory, and then performs operations en masse on the whole batch. Since data must be batched at the input to exploit this advantage, output may be delayed while batches are forming.

Both cases are addressed by the introduction of a new type of non-data event, called a punctuation. Punctuations have an application timestamp indicating that the time of the stream should advance to that time. This ensures that any data which follows the punctuation but has a sync time earlier than the punctuation's timestamp is considered out-of-order and handled appropriately. Punctuations, therefore, can be used to indicate that time is passing for an input stream, and can be used to unblock operators which are waiting for time to advance.

In addition, with the specification of the ingress policy FlushPolicy.FlushOnPunctuation, punctuations can be used to force the production of output associated with prior input, even if it means failing to fill a batch. In combination, the two roles of punctuations combine to guarantee that the system will immediately produce all output associated with input whose sync time precedes the punctuation's timestamp.

Punctuations can be introduced in two ways. For handling lulls in input data, punctuations typically come from the input provider itself, and simply show up in the sequence of input. Figure 63 shows an example of input with punctuations. Note the lull in the input data between sync time 3 and sync time 40. If this input stream were combined, in a query, with another very dense stream, output would be produced over the query at times 10, 20, 30, and 40, even though there is no data in this input stream. Also, note that punctuations produce output for times prior to their sync time, and that in order data may arrive after the punctuation with the same sync

time. In Figure 63, this is the case at time 40, where a punctuation is received, followed by an interval event with the same sync time.

```
var streamablePunctuations = new[]
{
    StreamEvent.CreateInterval(0, 1, 1),
    StreamEvent.CreateInterval(3, 4, 2),
    StreamEvent.CreatePunctuation<int>(10),
    StreamEvent.CreatePunctuation<int>(20),
    StreamEvent.CreatePunctuation<int>(30),
    StreamEvent.CreatePunctuation<int>(40),
    StreamEvent.CreateInterval(40, 41, 3)
}.ToObservable().ToStreamable(DisorderPolicy.Drop(), FlushPolicy.FlushOnPunctuation,
                              null, OnCompletedPolicy.None);
```

Figure 63: Input with Punctuations

Punctuations can be automatically generated after certain periods of time in the data have elapsed without a punctuation. For example, Figure 64 shows an example where punctuations are generated whenever a period of 10 or more ticks occur after the prior punctuation (or from the beginning of the stream). In this example, the first punctuation is generated at time 10, since the first event starts at time 0 and there are no punctuations in the input observable. A punctuation is not generated with the 3rd event since only 9 ticks have elapsed after the prior punctuation. Another punctuation is generated at time 40, since the event at time 40 is the first event whose sync time is 10 or more ticks after the prior punctuation. Note that punctuations are not generated for the period of time between 19 and 40, since there is no data.

```
var streamableTimePeriodEventPunctuations = new[]
{
    StreamEvent.CreateInterval(0, 1, 1),
    StreamEvent.CreateInterval(10, 4, 2),
    StreamEvent.CreateInterval(19, 4, 3),
    StreamEvent.CreateInterval(40, 41, 4)
}.ToObservable().ToStreamable(DisorderPolicy.Drop(), FlushPolicy.FlushOnPunctuation,
                              PeriodicPunctuationPolicy.Time(10), OnCompletedPolicy.None);
```

Figure 64: Input with Generated Punctuations After 10 Tick Periods

Finally, it is important to note that due to both batching and the fact that Trill only produces output prior to the latest sync time, not all output for the given input will be generated when the input IObservable terminates and sends an OnCompleted message to Trill. For instance, consider Figure 65. Because Trill batches both inputs to process them more efficiently, and since no additional action is taken when the query completes, no output will be generated. This is desirable for actual running real time queries, where they are periodically taken down for maintenance or load balancing, and we simply want to shut down the query, whatever its output state.

```
var incompleteOutputQuery = new[]
{
    StreamEvent.CreateInterval(0, 10, 1),
    StreamEvent.CreateInterval(1, 11, 2)
}.ToObservable()
    .ToStreamable(null, FlushPolicy.FlushOnPunctuation, null, OnCompletedPolicy.None)
    .Count()
    .ToStreamEventObservable();
```

Figure 65: Query with No Output

One could eliminate the batching issue by injecting a punctuation at the end, as is shown in Figure 66. Note that all output is produced up to time 1, which is the last sync time with data. This final punctuation strategy makes sense when back testing real time cases using offline data, where we want to see all correct answers up to the latest time, but we are uninterested in later results (e.g. the incomplete decrementing future count values).

```
var incompleteOutputQuery = new[]
{
    StreamEvent.CreateInterval(0, 10, 1),
    StreamEvent.CreateInterval(1, 11, 2),
    StreamEvent.CreatePunctuation<int>(1)
}.ToObservable()
    .ToStreamable(null, FlushPolicy.FlushOnPunctuation, null, OnCompletedPolicy.None)
    .Count()
    .ToStreamEventObservable();
```

Figure 66: Query with Output up to Time 1

For convenience, we have added an OnCompletedPolicy which injects a punctuation with the latest sync time in the input data, which has the effect of flushing all complete answers. This is shown in Figure 67. Note that we no longer need the explicit punctuation since the OnCompletedPolicy ensures that all complete answers are flushed.

```
var incompleteOutputQuery = new[]
{
    StreamEvent.CreateInterval(0, 10, 1),
    StreamEvent.CreateInterval(1, 11, 2)
}.ToObservable()
    .ToStreamable(null, FlushPolicy.FlushOnPunctuation, null, OnCompletedPolicy.Flush)
    .Count()
    .ToStreamEventObservable();
```

Figure 67: Alternate Query with Output up to Time 1

One the other hand, if one is using Trill for temporal analytics, one may want all answers across the entire time domain. Figure 68 shows how to produce all output across the entire time domain, by adding a punctuation at InfinitySyncTime. Now, all output is produced, including the count of 2 between 1 and 10, and the count of 1 between 10 and 11. Trill actually provides a straightforward way, through OnCompletedPolicy to request the generation of an infinite final punctuation upon query completion, shown in Figure 69. Note that OnCompletedPolicy.EndOfStream, and thus this behavior, is the default if no policy is specified.

```
var completeOutputQuery = new[]
{
    StreamEvent.CreateInterval(0, 10, 1),
    StreamEvent.CreateInterval(1, 11, 2),
    StreamEvent.CreatePunctuation<int>(StreamEvent.InfinitySyncTime)
}.ToObservable()
    .ToStreamable(null, FlushPolicy.FlushOnPunctuation, null, OnCompletedPolicy.None)
    .Count()
    .ToStreamEventObservable();
```

Figure 68: Query with all Output

```
var completeOutputQuery = new[]
{
    StreamEvent.CreateInterval(0, 10, 1),
    StreamEvent.CreateInterval(1, 11, 2)
}.ToObservable()
    .ToStreamable(null, FlushPolicy.FlushOnPunctuation, null, OnCompletedPolicy.EndOfStream)
    .Count()
    .ToStreamEventObservable();
```

Figure 69: Alternative Query with all Output