## Getting Data Into (and out of) Trill

Before you can use Trill, you must first get some data into the proper format for Trill to operate on. This means you must create a *stream*, i.e., a value of type `IStreamable`. This note assumes that you have read Section 2 of the Trill Query Writing Guide which explains the basics of streams (a stream is a sequence of *stream events*: each stream event is either a data value (tagged with timestamps) or else a *punctuation*).

An overview of how all the data types discussed in this note are related is seen in Figure 1.

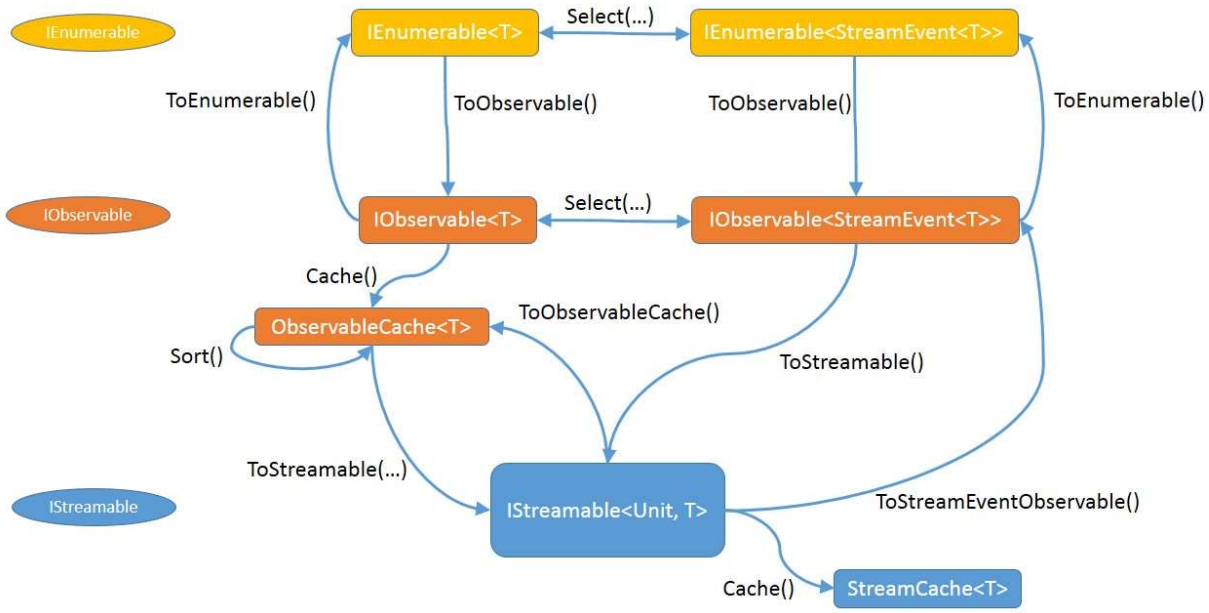


*Figure 1: Data Ingress and Egress*

Since a stream is a sequence, it is most likely that you will start with either an enumerable or an observable containing your data.

### From IEnumerable to IObservable

The first step is to convert the sequence of data contained in an `IEnumerable` to being a sequence represented by an `IObservable`. Since Trill does not deal with data directly, but instead values of type `StreamEvent`, you also need to convert the sequence of data (of type `T`) to `IObservable<StreamEvent<T>>`. You can do this by first creating an enumerable of stream events and then an observable of that:

```
var enumOfStreamEvent = e.Select(t =>
    StreamEvent.CreateStart<T>(StreamEvent.MinSyncTime, t));
var observableOfStreamEvent = enumOfStreamEvent.ToObservable();
```

or else first create the observable of `T` and then an observable of stream events:

```
var observableOfT = e.ToObservable();
var observableOfStreamEvent =
   observableOfT.Select(t =>
      StreamEvent.CreateStart<T>(StreamEvent.MinSyncTime, t));
```

Both examples show the creation of a sequence of *start events*. Of course you can create other kinds of stream events by changing the body of the anonymous delegate that is creating instances of stream events.

## From IObservable to IStreamable

To create a stream from an observable of stream events, you first need to decide on the *ingress policies*. These indicate how the system should handle the notion of time that is associated with every stream. There are five policies, all of which are optional.

### DisorderPolicy

A sequence of stream events is considered to be a valid Trill stream only if it is *well-formed*. A well-formed sequence of stream events must have start times that are in non-descending order. The first important option on the `ToStreamable` method allows you to specify how out-of-order stream events should be handled. This is called the *disorder policy*. There are three disorder policies that specify what to do if an out-of-order stream event is encountered during ingress.

- **Throw**: an `InvalidOperationException` exception is thrown. This is the default.
- **Adjust**: ignore the specified start time and use the most recent valid start time seen in the sequence.
- **Drop**: do not include the stream event in the stream.

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

#### *Reordering Streams*

Trill now supports the ability to buffer and reorder streams that arrive in almost sorted order of sync timestamp. This facility is invoked by the user providing an optional ***reorderLatency*** argument to the existing disorder policies of *Throw*, *Adjust*, and *Drop*. *reorderLatency* is a long number (time span) that indicates how much data Trill should buffer and before reordering and processing them. Data that is out-of-order by more than this amount (in application time) from "now" is handled by the specified disorder policy (of throw, adjust, or drop). The user can also set a *reorderLatency* of infinity (*StreamEvent.InfinitySyncTime*), which forces Trill to reorder and output events only when punctuations or low watermarks are ingressed by the user or automatically generated by Trill via PeriodicPunctuationPolicy or PeriodicLowWatermarkPolicy.

The example below shows us using a disorder policy of 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 upto T – 100. Any late-arriving data with a timestamp of lower than T – 100 would be dropped (since we chose a disorder policy of drop; one could instead choose adjust or throw if they wanted). 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.

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

For example, assume with the above query that we get an input data stream with sync timestamps of 2, 3, 21, 15, 63, 42, 84, and 105 (in that order). At the moment we receive 105, the high watermark of the stream (max. sync time) 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.

### FlushPolicy

This policy specifies when to flush batched output events through the entire query from the ingress site.

- **None**: Does not automatically flush. Output events will be batched and egressed normally.
- **FlushOnPunctuation** (non-partitioned streams only): When a punctuation is ingressed or generated, a flush will also be propagated through the query. This is the default policy for non-partitioned streams.
- **FlushOnLowWatermark** (partitioned streams only): When a low watermark is ingressed or generated, a flush will also be propagated through the query. This is the default policy for partitioned streams.
- **FlushOnBatchBoundary**: When a batch is filled on ingress, a flush will be propagated through the query.

```
var s = observableOfStreamEvent.ToStreamable(
                DisorderPolicy.Adjust(),
                FlushPolicy.FlushOnPunctuation);
```

### PeriodicPunctuationPolicy

It may be that you would like to have punctuation events automatically included in the stream. A *punctuation event* is a way for you to move time forward for the stream, or for a partitioned stream, a specific stream partition. The punctuation policy allows you to request certain periodic punctuations to be injected into the stream:

- **None**: Don't add any punctuations. This is the default.
- **Time**: If the *current* time of the stream (i.e., the most recent valid start time) is greater than *n* ticks later than the previous current time, then a punctuation should be added to the stream. The punctuation's timestamp will be rounded down to the previous multiple of *generationPeriod*.

```
var s = observableOfStreamEvent.ToStreamable(
                DisorderPolicy.Adjust(),
                FlushPolicy.FlushOnPunctuation,
                PeriodicPunctuationPolicy.Time(10));
```

### PeriodicLowWatermarkPolicy

For partitioned streams, this policy specifies how to low watermarks should be generated into the stream. Low watermarks set a lower bound on all partitions in the stream, and any event ingressed before that lower bound will be considered out of order.

- **None**: Don't inject any low watermarks. This is the default policy.
- **Time**: Inject low watermarks every *generationPeriod* time ticks, rounded down to the previous generationPeriod multiple, based on the highest event time ingressed into the stream. Low watermark timestamp is calculated as (highest event time - *lowWatermarkTimestampLag*). Highest ingressed event time is calculated before any buffering such as specified in other policies, DisorderPolicy.

```
var observableOfPartitionedStreamEvent = observableOfT.Select(t =>
        PartitionedStreamEvent.CreateStart<T>(key, StreamEvent.MinSyncTime, t));

var s = observableOfStreamEvent.ToStreamable(
                DisorderPolicy.Adjust(),
                PartitionedFlushPolicy.FlushOnLowWatermark,
                PeriodicPunctuationPolicy.None(),
                PeriodicLowWatermarkPolicy.Time(100, 10));
```

### OnCompletedPolicy

This policy specifies how the system should handle stream completion, i.e., what to do when an OnCompleted call is made on a subscriber to a stream. There are three possibilities:

1. **None**: Immediately stop and produce no more tuples.
2. **Flush**: Any partial output that is being held within the system is pushed through the query plan before completing.
3. **EndOfStream**: Move the "current time" to infinity before completing. This has the effect of causing all of the tuples that may be within the system to be pushed through the query plan. This is the default policy.

```
var s = observableOfStreamEvent.ToStreamable(
                DisorderPolicy.Adjust(),
                FlushPolicy.FlushOnPunctuation,
                PeriodicPunctuationPolicy.Time(10),
                OnCompletedPolicy.EndOfStream);
```

## Caches

There are situations where you want to make sure that Trill can execute at its maximum potential. In order to achieve this, it is best to have all of your data loaded into memory. We call this *caching*.

There is a simple extension method `Cache()` on IStreamable which returns a stream which has been fully loaded into memory. A stream cache is still a value of type `IStreamable` and can be used in all contexts that expect a stream.

```
var cache = stream.Cache();
```

There are a couple of optional arguments you might wish to use:

- **limit**: Specifies a limit on the number events that can be stored in the cache.
- **inferProperties**: Specifies whether each stream event in the incoming stream should be checked to infer the properties of the stream, such as whether all stream events have constant duration, or does not contain any intervals.
- **coalesceEndEdges**: Specifies whether end edges should be coalesced with their corresponding start edges to form interval events.

## And back again...

```
var obs = stream.ToStreamEventObservable();
```

There are inverse methods that can reverse all of the steps that have been shown so far. For instance, to convert a stream into an observable of stream events:

From there you can use `Select` and/or `ToEnumerable` to convert back to pure data or an enumerable, respectively.

## Threading

For a given input stream, you need to ingress events into Trill on a single thread. To be more precise, two threads should not be calling OnNext to Trill at the same instant on the same input. Different inputs can however be ingressing data in parallel without any restriction. One (slow but quick) way of enforcing single threaded behavior for your input observable is to wrap it in a Synchronize() method. See http://msdn.microsoft.com/en-us/library/system.reactive.linq.observable.synchronize(v=vs.103).aspx for more details on this.