High Availability in Trill

This document describes the features included in Trill that support High Availability (HA). HA is achieved by allowing a program to periodically checkpoint the state of the running query and restore a new instance of that query to the checkpointed state should the query be terminated in some way.

1. What is High Availability?

So, you've written a query using Trill, and you would like to ensure high availability of the query's execution. The query may be running on an unreliable node that may periodically reset, or the query may need to be moved between nodes to ensure uptime. How does one go about doing that?

Suppose that the query author has constructed the following query:

```
var subject = new Subject<StreamEvent<StructTuple<int, int>>>();

// Input data
var data = Enumerable.Range(0, 10000).ToList()
    .ToObservable()
    .Select(e => StreamEvent.CreateStart<StructTuple<int, int>>(
        e, new StructTuple<int, int> { Item1 = e % 10, Item2 = e }));

// Create the query
var input1 = subject.ToStreamable();
var query1 = input1.GroupApply(e => e.Item1,
                               str => str.Sum(e => (ulong)e.Item2),
                               (g, c) => new StructTuple<int, ulong>
                                         { Item1 = g.Key, Item2 = c });
var output1 = query1.ToStreamEventObservable()
                    .Where(e => e.IsData)
                    .Select(e => e.Payload);
```

What's notable about this query is that the operator GroupApply has state associated with it. So if someone wanted to shut down the node on which the query is running and start it up again on another node somewhere, it is not as simple as just starting up the query again. The key is to capture the state of the query at a known time so that, when the query is started up again, it can start out pre-primed with that state and can continue processing from there. The process of capturing that state is called *checkpointing*, and the process of priming a new instance of a query with the old state is called *restoration*.

To move a running query from one node to another, or to restart a query from a known point:

- Take a checkpoint of the running query, taking note of the state of the input and output streams at that point as well. Save that state to a known location.
- When restarting that same query, start it by restoring that state at the time of query creation, and start the inputs again from the corresponding point. If the query has generated output

between the time of checkpointing and the time of restoration, that output will be produced a second time and will need to be de-duplicated.

2. The Players

When HA is enabled in Trill, the query author should notice virtually no difference from query authoring without HA. The difference between the two modes of operation is in a particular object called a `QueryContainer`.

Ordinarily, when constructing queries in Trill, there are no concrete objects that represent the query definition or the running query. The query writer will ingress data into Streamable objects (using calls to ToStreamable), then perform query operations on those streamables, and finally egress data from the system (using calls to ToStreamEventObservable). In no part of that workflow is there a reference to the query itself. The `QueryContainer` object provides that reference.

A `QueryContainer` object is trivial to create:

```
var container = new QueryContainer();
```

The API for the `QueryContainer` object is designed to be a drop-in replacement for the calls to ToStreamable on data ingress and ToStreamEventObservable on data egress. The idea is to register the data ingress and egress sites with the `QueryContainer` object, defining the boundaries of the query. For instance, without using HA, one might ingress data from an observable `inputData` as follows:

```
var input1 = inputData.ToStreamable();
```

To use HA with a `QueryContainer` object, replace the above call with the following:

```
var input1 = container.RegisterInput(inputData);
```

For every overload of the ToStreamable method, there is an identical overload of the RegisterInput method with one additional parameter at the front, that being the input observable.

The same must be done on the data egress side by replacing the following call on a streamable object:

```
var output = query.ToStreamEventObservable();
```

With this:

```
var output = container.RegisterOutput(query);
```

Note: For a given query, there may be multiple sources of data, and multiple points of data output as well. Technically, at each point of data ingress, one has the independent choice of using either the RegisterInput or ToStreamable methods to get a Streamable object (and respectively the choice of either the RegisterOutput or ToStreamEventObservable for egress). In practice, for a given query, one should either use the Register methods on the points of data ingress and egress or on none of them. Mixing the two modes of operation may lead to an unknown state and operation will be unpredictable.

With one important exception, those method call replacements represent the full set of changes required to enable HA. The one remaining change is that the query container must be explicitly enabled before the query begins to execute. Without the `QueryContainer` object, as soon as the calls to ToStreamEventObservable evaluate, the query becomes live and data begins to flow through the stream. When using a `QueryContainer` object, it is important to have an explicit step for enabling the query. That way, the query author has an opportunity to dictate the initial query state – for instance, to restore the state from a previous, interrupted query evaluation – prior to data beginning to flow through the system and further affecting query state.

To begin query evaluation, simply include the following call:

```
var process = container.Restore();
```

Two things are of note with that method call:

- The method is named "Restore" because it is the same method that is used to dictate initial query state (to be revisited in Section 3: Restoration). Calling the method with no parameters is functionally identical to passing in null as the argument.
- The call to Restore returns an object of type `Process`. The returned object represents the running query, and is the object that can be checkpointed.

3. Checkpointing and Restoration

The state involved in checkpointing and restoration comes from a Stream object. Any stream object will do; Trill is agnostic when it comes to the storage medium of query state.

The `Process` object returned from the call to Restore has a single method called Checkpoint:

```
process.Checkpoint(stateStream);
```

Given a stream object `stateStream`, the code above will iteratively move through each of the working components of the active query and serialize its state to `stateStream`.

Note: The call to Checkpoint is expected to be synchronous. Any data received at the data ingress sites during a checkpoint may be lost.

Restoring a query to a checkpointed state is done using the same Restore method that begins active query evaluation:

```
var process = container.Restore(stateStream);
```

Note: The call to Restore does not do any structural validation yet. That means that the state given to the Restore call may correspond to a query that is not the same as the query represented by the container. When using restoration, take care to make sure that the checkpointed state came from the correct query or unexpected behavior may occur.

4. A Complete Example

What follows is a complete end-to-end example of using HA in Trill. It is a toy example, using only start edge stream events.

```
// Create a list to contain the query's output and a buffer to store the query state
var outputListWithCheckpoint = new List<StructTuple<int, ulong>>();
Stream state = new MemoryStream();

////////////////////////////////////////////////////////////////////
// Step 1: Checkpoint
////////////////////////////////////////////////////////////////////

var preCheckpointSubject = new Subject<StreamEvent<StructTuple<int, int>>>();
var container1 = new QueryContainer();

// Input data
var preCheckpointData = Enumerable.Range(0, 10000).ToList()
    .ToObservable()
    .Select(e => StreamEvent.CreateStart<StructTuple<int, int>>(
        e, new StructTuple<int, int> { Item1 = e % 10, Item2 = e }));

// Create the query
var input1 = container1.RegisterInput(preCheckpointSubject,
                                      OnCompletedPolicy.EndOfStream());
var query1 = input1.GroupApply(e => e.Item1,
                               str => str.Sum(e => (ulong)e.Item2),
                               (g, c) => new StructTuple<int, ulong>
                                         { Item1 = g.Key, Item2 = c });
var output1 = container1.RegisterOutput(query1);

// Dump output data to the output list
var outputAsync1 = output1.Where(e => e.IsData)
                          .Select(e => e.Payload)
                          .ForEachAsync(o => outputListWithCheckpoint.Add(o));

// Start the query
var pipe1 = container1.Restore();

// Feed the query's input data
preCheckpointData.ForEachAsync(e => preCheckpointSubject.OnNext(e)).Wait();

// Checkpoint the state
pipe1.Checkpoint(state);

// Restore the memory buffer to the beginning
state.Seek(0, SeekOrigin.Begin);

////////////////////////////////////////////////////////////////////
// Step 2: Restore
////////////////////////////////////////////////////////////////////

var postCheckpointSubject = new Subject<StreamEvent<StructTuple<int, int>>>();
var container2 = new QueryContainer();

// Input data
var postCheckpointData = Enumerable.Range(10000, 10000).ToList()
```

```
    .ToObservable()
    .Select(e => StreamEvent.CreateStart<StructTuple<int, int>>(
        e, new StructTuple<int, int> { Item1 = e % 10, Item2 = e }));

// Create the query
var input2 = container2.RegisterInput(postCheckpointSubject);
var query2 = input2.GroupApply(e => e.Item1,
                               str => str.Sum(e => (ulong)e.Item2),
                               (g, c) => new StructTuple<int, ulong>
                                         { Item1 = g.Key, Item2 = c });
var output2 = container2.RegisterOutput(query2);

// Dump output data to the output list
var outputAsync2 = output2.Where(e => e.IsData)
                          .Select(e => e.Payload)
                          .ForEachAsync(o => outputListWithCheckpoint.Add(o));

// Start the query, using the previously checkpointed state
var pipe2 = container2.Restore(state);

// Feed the query's input data, and mark the input data as being finished
postCheckpointData.ForEachAsync(e => postCheckpointSubject.OnNext(e)).Wait();
postCheckpointSubject.OnCompleted();
outputAsync2.Wait();
```