Best Practices for Using Trill in Real-Time Deployments

Row-oriented vs. Columnar Trill

Trill can operate in two modes: *row-oriented*, where payloads types stay as originally defined, and *columnar*, where we transform a batch of payloads to instead be stored as a set of arrays (one per field in the payload). The row-oriented version of Trill is the most production-tested, so it may be a good idea to start with that mode in your real-time pipelines. The columnar mode is usually used for offline query processing scenarios where very high throughput is a primary requirement. Even in the row-oriented mode, Trill is usually fast enough to handle typical real-time scenarios, and the bottleneck is often elsewhere in the system, such as input acquisition, serialization, deserialization, memory, etc.

You instruct Trill to operate in row-oriented mode using the following configuration variable:

```
Config.ForceRowBasedExecution = true
```

All configuration variables in Trill are static global – they must be set immediately at program entry point, and should not be modified in the rest of the application.

Batch Size

Trill batches input data before feeding it into the query processing logic. Batching is controlled by the **`Config.DataBatchSize`** variable (with the extreme value of 1 disabling batching entirely – but this is not recommended). This setting is global for all queries - you cannot selectively enable it for particular operators, but there is little reason in practice to worry about this – a uniform small batch size usually does just fine. Larger batch sizes increase the memory footprint of Trill, so it is a good idea to tune this variable to the smallest possible value that makes sense for your scenario.

The ideal setting for this parameter is a little more than the number of data events you expect between two punctuations. Any larger value implies that the remaining space in the batch is wasted. A batch size of at most 1000 is perfectly reasonable for most real-time scenarios I have encountered. Most of our memory-critical real-time customers (e.g., Exchange) use a value in this ballpark, while Halo uses an even smaller value (~10).

The default value of batch size in Trill is **80,000** – this is a large value, but was chosen because Trill is configured by default to work optimally for offline query performance. In offline scenarios, large batch sizes are needed to match the performance of columnar databases, since latency is not a primary consideration in those scenarios.

More on batching

The idea of batching in Trill is that we allow you to specify what latency you can accept, using the punctuation policy. We then pack events into batches and send these batches through the query execution plan. This gives much higher performance due to the benefit of batching, but there are diminishing returns with larger batch sizes (depending on the complexity of the query). For example, with

grouped aggregation, I've found that a batch size of 1000 gives you nearly all the benefits possible with batching.

Note that batching in Trill is purely physical. Every operator batches as much as it can (up to the specified maximum batch size), before sending it to the next operator. GroupApply is no different. A **flush** flows through the query plan and causes batches to close-out and be forwarded immediately. It is used when you want to force output "right now" instead of waiting for batches to fill up and output to eventually trickle out. Flushes by default will be initiated in response to Punctuations or Low Watermarks, but this is configurable via the FlushPolicy.

As an aside, a punctuation is associated with a timestamp T and plays double duty of telling users that time has passed forward until T – this is useful when there are lulls in the input data stream, and you want time to move forward in the query and be reported at the output as well.

Memory Management

Trill does memory pooling (ref-counting + reuse) of these batches for improved performance. If your payload types are classes or anonymous types, you will likely want Trill to Zero-Out the batch contents before putting them back in the pool, so the payload entries can be GCed immediately. You do this using the following Config setting (this is not done by default): **`Config.ClearColumnsOnReturn = true`**

If you want to disable memory pooling entirely, for optimizing towards low memory, you can try to disable memory pooling entirely using: **`Config.DisableMemoryPooling = true`** (play with this to see if it helps).

App Config

If you have access to app.config for your application, you should add the following lines. See http://msdn.microsoft.com/en-us/library/ms184658.aspx for details on how to add it. The GC lines in particular allow the GC to run on all threads, which is appropriate in a server environment and allow Trill to work with larger amounts of data, for example, in hash dictionaries.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <runtime>
    <gcServer enabled="true"/>
    <gcAllowVeryLargeObjects enabled="true" />
  </runtime>
</configuration>
```

The suggestion to turn on server GC doesn't work for Azure worker roles. One has to take a different approach, as noted in several articles, including http://blogs.msdn.com/b/cclayton/archive/2014/06/05/server-garbage-collection-mode-in-microsoft-azure.aspx.

Ingress

With respect to ingress, it is usually not advisable to ingress start-edge events with a disorder policy. This combination has an unexpected, but correct, consequence that every adjusted event has to be remembered (potentially forever). Consider a late start event under the Adjust DisorderPolicy. Trill will adjust the late start event forward but will also have to remember it until a matching end edge arrives in the future (or forever, if such a matching end edge never arrives). This is because, for correctness, we would have to adjust the original start time of the end-edge event to the correct original adjusted start-edge's timestamp, before pushing it into the query plan. A similar reason holds for a drop policy as well.

However, in most cases, the user knows that their input consists only of start-edges (no end edges), so it is safe to discard the late-arriving data. This is achieved using the following pattern:

```
[Create observable stream of StreamEvents with lifetime from T to StreamEvent.MaxSyncTime]

// create intervals to tell Trill that there is no need to store late-arriving events

.ToStreamable(DisorderPolicy.Adjust(…), …) // apply ingress policies

.AlterEventDuration(StreamEvent.InfinitySyncTime) // convert the events back to start edges (if you want)
```

If you wanted to adjust events only within a tolerance factor (say one hour) and drop events that are later than an hour, you would set the interval endpoint to that value (T + 1 hour) instead of StreamEvent.MaxSyncTime (thus, any later-arriving events would simply get dropped).

Reordering Input by Time

The same pattern described above (ingesting intervals instead of start edges) works when you have a reorder latency as well. You can reorder up to a latency using the reorder-latency argument to `DisorderPolicy.Adjust(…)`. If events are later than this, you can adjust events forward up to a particular threshold, as defined by the end time of the input interval event. Trill drops the event entirely if it is late even beyond this threshold.

When a reorder latency is specified, reordering at ingress in Trill uses an algorithm called Impatience Sort, which is efficient for almost sorted streams of data, but may have a higher memory footprint. You can play with replacing impatience by a standard priority-queue-based sorting technique, using the following config parameter:

```
Config.IngressSortingTechnique = SortingTechnique.PriorityQueue;
```