Flink, Queryable State, and High Frequency Time Series Data

Joe Olson
Data Architect
PhysIQ
11Apr2017

About Us / Our Data….

- What? Tech company that collects, stores, enriches, and presents vitals data for a given patient (heart rate, O2 levels, respiration rate, etc)
- Why? To build a predictive model of patient's state of health.
- Who? End users are patients and health care staff at care facilities (or at home!)
- How?
  - Data originates from wearable patches
  - Collected as a waveform – must be converted into friendly numeric types (think PLCs in a IoT type application)
  - Stream in in 1 second chunks (2KB – 6KB)
  - May represent data sampled anywhere from 1Hz to 200Hz
  - Treat data as a stream throughout all data flows

State in a Flink Stream…

- **Keyed State** – state associated with a partition determined by a keyed stream. One partition space per key.
- **Operator State** – state associated with an operator instance. Example: Kafka connector. Each parallel instance of the connector maintains it own state.

Both of these states exist in two forms:

- **Managed –** data structures controlled by Flink
- **Raw** – user defined byte array

- Our use case leverages **managed keyed state**

Managed Keyed State

- `ValueState<T>:` a value that can be updated and retrieved. Two main methods:
  - `.update()` Set the value
  - `.value()` Get the value
- `ListState<T>:` a list of elements that can be added to, or iterated over
  - `.add(T)` Add an item to the list
  - `Iterable<T> get()` Use to iterate
- `ReducingState<T>:` a single value that represents an aggregation of all values added to the state
  - `.add(T)` add to the state using a provided `ReduceFunction`

How Is State Persisted?

- 3 back end options for preserving state:
- `MemoryStateBackend`
  - Stored on the Java heap
  - Aggregate state must fit into Job Manager RAM.
  - Good for small state situations / testing
- `FsStateBackend`
  - Data held in task manager RAM
  - Upon checkpointing, writes state to file system (must be shared for HA)
  - Good for larger states, and HA situations
- `RocksDBStateBackend`
  - All data stored on disk in a RocksDB – an optimized KV store
  - Upon checkpointing, the entire RocksDB is copied
  - Good for very large states!
- Persistence options can be defined on the job level

Putting it all together…

Here is our stream:

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStateBackend(new RocksDBStateBackend(
                              "hdfs://namenode:40010/flink/checkpoints"))
val stream = env.addSource(someConsumer)
      .keyBy(anObject => (anObject.id1, anObject.id2))
      .flatMap(new doSomethingClass(param1, param2))
```

Here is our managed state:

```
class doSomethingClass(p1:Long, p2:Long) extends RichFlatMapFunction(…)

private val listStateDescriptor = new ListStateDescriptor("list-example",
        TypeInformation.of(new TypeHint[java.lang.Long]() {}))
private val listState = getRuntimeContext.getListState(listStateDescriptor)

override def flatMap(value:anObject, out: Collector[T]): Unit = {

    listState.add(value.somethingID)
    if (value.time > someTimeThreshold) {
          listState.clear()

    }
```

Queryable State

- The “Queryable State” feature refers to managed state that is accessible outside of the Flink runtime environment via a Flink Streaming API call.
- How?

```
aStateDescriptor.setQueryable("queryable-name")
```

- To access a managed state descriptor outside of Flink:

```
val config:Configuration = new Configuration();
    config.setString(ConfigConst.JOB_MANAGER_IPC_ADDRESS_KEY, serverIP)
    config.setInteger(ConfigConst.JOB_MANAGER_IPC_PORT_KEY, port)

val client:QueryableStateClient = new QueryableStateClient(config)

val key = (id1, id2)

// jobID: make a REST call to http://<serverIP>:8081/joboverview/running
val results = queryClient.executeQuery(key, jobID, "queryable-name")

// Not shown: deserializing results into a scala class
```

More thoughts

- Managed state is created within the Flink runtime context
- To access state, you'll need access to the runtime context (the `Rich..` classes in the Flink Streaming API)
- The windowing functionality in the API is not exposed to the runtime context

```
// No QS visibility into this until *after* apply finishes

val stream = env.addSource(aConsumer)
      .assignTimestampsAndWatermarks(new timeStampWatermark())
      .keyBy(x => (x._1, x_.2)
      .window(GlobalWindows.create())
      .trigger(new customTrigger())
      .apply(new applyRule())
      .addSink(new dataSink())
```

- More complex state management – (e.g. maps)
- Partitioning managed state into more manageable chunks
  - State variables addressable by nam

Our Use Cases

- Trying to move from batch mentality to stream mentality
- UC 1: ETL -> Kafka -> Flink -> (external) KV Store
  - Scalable, fault tolerant, etc
  - Replace traditional ETL stack
  - Much more scalable.
- UC 2: “Given T_1 and T_2 , find all places where we have data”
  - (i.e. “Show me the gaps in a big list of integers”)
  - Historically, calculated after the data is at rest
  - Now calculated in real time and served with Flink – fast, accurate
- UC 3: “Given data at 1s resolution, buffer it into Xs blobs”
  - Minimize reads (yeah, batch…)
  - Not currently in production
  - Need QS visibility into windows to service time series requests for data being blobbed
- Future Use Case? Replace (external) KV Store with Flink / Kafka?