


How to build stateful streaming applications with Apache Flink

By Fabian Hueske | Aug. 2nd, 2018

 Send to Kindle

Apache Flink is a framework for implementing stateful stream processing applications and running them at scale on a compute cluster. In a previous article we examined what stateful stream processing is, what use cases it addresses, and why you should implement and run your streaming applications with Apache Flink.

In this article, I will present examples for two common use cases of stateful stream processing and discuss how they can be implemented with Flink. The first use case is event-driven applications, i.e., applications that ingest continuous streams of events and apply some business logic to these events. The second is the streaming analytics use case, where I will present two analytical queries implemented with Flink's SQL API, which aggregate streaming data in real-time. We at Data Artisans provide the source code of all of our examples in a public GitHub repository.

Before we dive into the details of the examples, I will introduce the event stream that is ingested by the example applications and explain how you can run the code that we provide.

A stream of taxi ride events

Our example applications are based on a public data set about taxi rides that happened in New York City in 2013. The organizers of the 2015 DEBS (ACM International Conference on Distributed Event-Based Systems) Grand Challenge rearranged the original data set and converted it into a single CSV file from which we are reading the following nine fields.

- Medallion—an MD5 sum id of the taxi

- `Hack_license`—an MD5 sum id of the taxi license
- `Pickup_datetime`—the time when passengers were picked up
- `Dropoff_datetime`—the time when passengers were dropped off
- `Pickup_longitude`—the longitude of the pick-up location
- `Pickup_latitude`—the latitude of the pick-up location
- `Dropoff_longitude`—the longitude of the drop-off location
- `Dropoff_latitude`—the latitude of the drop-off location
- `Total_amount`—total paid in dollars

The CSV file stores the records in ascending order of their drop-off time attribute. Hence, the file can be treated as an ordered log of events that were published when a trip ended. In order to run the examples that we provide on GitHub, you need to download the data set of the DEBS challenge from Google Drive.

All example applications sequentially read the CSV file and ingest it as a stream of taxi ride events. From there on, the applications process the events just like any other stream, i.e., like a stream that is ingested from a log-based publish-subscribe system, such as Apache Kafka or Kinesis. In fact, reading a file (or any other type of persisted data) and treating it as a stream is a cornerstone of Flink's approach to unifying batch and stream processing.

Running the Flink examples

As mentioned earlier, we published the source code of our example applications in a GitHub repository. We encourage you to fork and clone the repository. The examples can be easily executed from within your IDE of choice; you don't need to set up and configure a Flink cluster to run them. First, import the source code of the examples as a Maven project. Then, execute the main class of an application and provide the storage location of the data file (see above for the link to download the data) as a program parameter.

Once you have launched an application, it will start a local, embedded Flink instance inside the application's JVM process and submit the application to execute

it. You will see a bunch of log statements while Flink is starting and the job's tasks are being scheduled. Once the application is running, its output will be written to the standard output.

Building an event-driven application in Flink

Now, let's discuss our first use case, which is an event-driven application. Event-driven applications ingest streams of events, perform computations as the events are received, and may emit new events or trigger external actions. Multiple event-driven applications can be composed by connecting them together via event log systems, similar to how large systems can be composed from microservices. Event-driven applications, event logs, and application state snapshots (known as savepoints in Flink) comprise a very powerful design pattern because you can reset their state and replay their input to recover from a failure, to fix a bug, or to migrate an application to a different cluster.

In this article we will examine an event-driven application that backs a service, which monitors the working hours of taxi drivers. In 2016, the NYC Taxi and Limousine Commission decided to restrict the working hours of taxi drivers to 12 hour shifts and require a break of at least eight hours before the next shift may be started. A shift starts with the beginning of the first ride. From then on, a driver may start new rides within a window of 12 hours. Our application tracks the rides of drivers, marks the end time of their 12-hour window (i.e., the time when they may start the last ride), and flags rides that violated the regulation. You can find the full source code of this example in our GitHub repository.

Our application is implemented with Flink's `DataStream` API and a `KeyedProcessFunction`. The `DataStream` API is a functional API and based on the concept of typed data streams. A `DataStream<T>` is the logical representation of a stream of events of type `T`. A stream is processed by applying a function to it that produces another data stream, possibly of a different type. Flink processes streams in parallel by distributing events to stream partitions and applying different instances of functions to each partition.

The following code snippet shows the high-level flow of our monitoring application.

```
// ingest stream of taxi rides.  
DataStream<TaxiRide> rides = TaxiRides.getRides(env, inputPath);  
  
DataStream<Tuple2<String, String>> notifications = rides  
  
    // partition stream by the driver's license id  
    .keyBy(r -> r.licenseId)  
    // monitor ride events and generate notifications  
    .process(new MonitorWorkTime());  
  
// print notifications  
  
notifications.print();
```

The application starts ingesting a stream of taxi ride events. In our example, the events are read from a text file, parsed, and stored in `TaxiRide` POJO objects. A real-world application would typically ingest the events from a message queue or event log, such as Apache Kafka or Pravega. The next step is to key the `TaxiRide` events by the `licenseId` of the driver. The `keyBy` operation partitions the stream on the declared field, such that all events with the same key are processed by the same parallel instance of the following function. In our case, we partition on the `licenseId` field because we want to monitor the working time of each individual driver.

Next, we apply the `MonitorWorkTime` function on the partitioned `TaxiRide` events. The function tracks the rides per driver and monitors their shifts and break times. It emits events of type `Tuple2<String, String>`, where each tuple represents a notification consisting of the license ID of the driver and a message. Finally, our application emits the messages by printing them to the standard output. A real-

world application would write the notifications to an external message or storage system, like Apache Kafka, HDFS, or a database system, or would trigger an external call to immediately push them out.

Now that we've discussed the overall flow of the application, let's have a look at the `MonitorWorkTime` function, which contains most of the application's actual business logic. The `MonitorWorkTime` function is a stateful `KeyedProcessFunction` that ingests `TaxiRide` events and emits `Tuple2<String, String>` records. The `KeyedProcessFunction` interface features two methods to process data: `processElement()` and `onTimer()`. The `processElement()` method is called for each arriving event. The `onTimer()` method is called when a previously registered timer fires. The following snippet shows the skeleton of the `MonitorWorkTime` function and everything that is declared outside of the processing methods.

```
public static class MonitorWorkTime
    extends KeyedProcessFunction<String, TaxiRide, Tuple2<String, St

// time constants in milliseconds

private static final long ALLOWED_WORK_TIME = 12 * 60 * 60 * 1000;
private static final long REQ_BREAK_TIME = 8 * 60 * 60 * 1000;
private static final long CLEAN_UP_INTERVAL = 28 * 60 * 60 * 1000;

private transient DateTimeFormatter formatter;

// state handle to store the starting time of a shift

ValueState<Long> shiftStart;

@Override

public void open(Configuration conf) {
```

```
// register state handle
shiftStart = getRuntimeContext().getState(
    new ValueStateDescriptor<>("shiftStart", Types.LONG));
// initialize time formatter
this.formatter = DateTimeFormat.forPattern("yyyy-MM-dd HH:mm:ss")
}

// processElement() and onTimer() are discussed in detail below.

}
```

The function declares a few constants for time intervals in milliseconds, a time formatter, and a state handle for keyed state that is managed by Flink. Managed state is periodically checkpointed and automatically restored in case of a failure. Keyed state is organized per key, which means that a function will maintain one value per handle and key. In our case, the `MonitorWorkTime` function maintains a `Long` value for each key, i.e., for each `licenseId`. The `shiftStart` state stores the starting time of a driver's shift. The state handle is initialized in the `open()` method, which is called once before the first event is processed.

Now, let's have a look at the `processElement()` method.

```
@Override
public void processElement(
    TaxiRide ride,
    Context ctx,
    Collector<Tuple2<String, String>> out) throws Exception {

    // look up start time of the last shift

    Long startTs = shiftStart.value();

    if (startTs == null ||
```

```
startTs < ride.pickUpTime - (ALLOWED_WORK_TIME + REQ_BREAK_TIME)

// this is the first ride of a new shift.

startTs = ride.pickUpTime;
shiftStart.update(startTs);
long endTs = startTs + ALLOWED_WORK_TIME;
out.collect(Tuple2.of(ride.licenseId,
    "You are allowed to accept new passengers until " + formatter.

// register timer to clean up the state in 24h

ctx.timerService().registerEventTimeTimer(startTs + CLEAN_UP_INT
} else if (startTs < ride.pickUpTime - ALLOWED_WORK_TIME) {
    // this ride started after the allowed work time ended.
    // it is a violation of the regulations!
    out.collect(Tuple2.of(ride.licenseId,
        "This ride violated the working time regulations."));
}
}
```

The `processElement()` method is called for each `TaxiRide` event. First, the method fetches the start time of the driver's shift from the state handle. If the state does not contain a start time (`startTs == null`) or if the last shift started more than 20 hours (`ALLOWED_WORK_TIME + REQ_BREAK_TIME`) earlier than the current ride, the current ride is the first ride of a new shift. In either case, the function starts a new shift by updating the start time of the shift to the start time of the current ride, emits a message to the driver with the end time of the new shift, and registers a timer to clean up the state in 24 hours.

If the current ride is not the first ride of a new shift, the function checks if it violates the working time regulation, i.e., whether it started more than 12 hours later than the start of the driver's current shift. If that is the case, the function emits a message to inform the driver about the violation.

The `processElement()` method of the `MonitorWorkTime` function registers a timer to clean up the state 24 hours after the start of a shift. Removing state that is no longer needed is important to prevent growing state sizes due to leaking state. A timer fires when the time of the application passes the timer's timestamp. At that point, the `onTimer()` method is called. Similar to state, timers are maintained per key, and the function is put into the context of the associated key before the `onTimer()` method is called. Hence, all state access is directed to the key that was active when the timer was registered.

Let's have a look at the `onTimer()` method of `MonitorWorkTime`.

```
@Override
public void onTimer(
    long timerTs,
    OnTimerContext ctx,
    Collector<Tuple2<String, String>> out) throws Exception {

    // remove the shift state if no new shift was started already.

    Long startTs = shiftStart.value();
    if (startTs == timerTs - CLEAN_UP_INTERVAL) {
        shiftStart.clear();
    }
}
```

The `processElement()` method registers timers for 24 hours after a shift started to clean up state that is no longer needed. Cleaning up the state is the only logic that the `onTimer()` method implements. When a timer fires, we check if the driver

started a new shift in the meantime, i.e., whether the shift starting time changed. If that is not the case, we clear the shift state for the driver.

Page 2

The implementation of the working hour monitoring example demonstrates how Flink applications operate with state and time, the core ingredients of any slightly advanced stream processing application. Flink provides many features for working with state and time, such as support for processing and event time. In addition, Flink provides several strategies for dealing with late records, different types of state, an efficient checkpointing mechanism to guarantee exactly-once state consistency, and many unique features centered around the concept of “savepoints,” just to name a few. The combination of all of these features results in a stream processor that provides the flexibility that is required to build very sophisticated event-driven applications and run them at scale and with operational ease.

Analyzing data streams with SQL in Flink

The previous example showed the expressiveness and low-level control that you have at hand when implementing applications with Flink’s `DataStream API` and `ProcessFunctions`. However, a large fraction of applications (and subtasks of more advanced applications) have very similar requirements and do not need this level of expressiveness. They can be defined more concisely and conveniently using SQL, the standard language for data processing and analytics.

Flink features unified SQL support for batch and streaming data. For a given query Flink computes the same result regardless of whether it is executed on a stream of continuously ingested records or on a bounded set of records, given that the stream and data set provide the same data. Flink supports the syntax and semantics of ANSI SQL; it does *not* define a language that looks similar to SQL and comes with proprietary syntax and semantics.

Supporting standard SQL as an API for unified batch and stream processing provides many advantages and benefits. Users who are familiar with standard SQL immediately know how to write queries and what results to expect. Queries can be applied to bounded and unbounded data, as well as recorded and real-time streams. Therefore, batch-stream unification enables use cases such as bootstrapping state from historical data and backfilling results to incorporate late data or to compensate for outages—all without query modifications. It's also easy to perform data exploration on a small static data set, and later run the same query on a continuous stream.

So, how does Flink evaluate SQL queries on streams? In principle, running a SQL query on a stream of data is the same as maintaining a materialized view. Materialized views are a well-known feature of relational database systems that are commonly used to speed up analytical workloads. They can be understood as a hybrid of regular virtual views and indexes. A materialized view is defined as a query (similar to a regular view), but the query result is immediately computed, stored as a table on disk or in memory, and automatically updated when the view's base tables change (similar to an index). When planning a query, the query optimizer aims to rewrite queries to read pre-computed results from a materialized view instead of computing them again.

The similarity between materialized view maintenance and the evaluation of SQL queries on streams should become clear when we treat the continuous SQL query as the query defining a materialized view, the input stream of a SQL query as a stream of changes that are applied on the view's base table, and the result of the continuous query as the stream of changes to update the materialized view. Hence, evaluating a SQL query on a stream is the same as maintaining a materialized view. Basically, Flink maintains materialized views by running incremental SQL queries on a distributed, stateful stream processor.

We can appreciate the elegance and power of Flink's SQL support by taking a look at two queries. The first query computes the average total ride fare per hour and location. To group by location, we discretize the drop-off locations' coordinates

into areas of approximately 250 by 250 meters. The query to compute this result is shown below. You can find the source code of the application that runs the query in our repository.

```
SELECT  toCellId(dropOffLon, dropOffLat) AS area,  TUMBLE_START(drop  
TUMBLE(dropOffTime, INTERVAL '1' HOUR)
```

The query looks and behaves just like a regular `SELECT-FROM-GROUP-BY` query. However, there are three functions that we need to explain:

1. `TUMBLE` is a function to assign records to windows of a fixed length, here one hour.
2. `TUMBLE_START` is a function that returns the start timestamp of a window, e.g., `TUMBLE_START` will return “2018-06-01 13:00:00.000” for the window that starts at 1 p.m. and ends at 2 p.m. on June 1, 2018.
3. `toCellId` is a user-defined function that computes a cell ID, representing an area of 250 by 250 meters, from a pair of longitude-latitude values.

The query will return a result similar to this:

```
8> 29126,2013-01-04 11:00:00.0,44.04> 46551,2013-01-04 11:00:00.0,14  
7> 56781,2013-01-04 12:00:00.0,7.5
```

An important detail of our example query is that `dropOffTime` is an event-time timestamp attribute. The timestamps of an event-time attribute are (roughly) increasing and are guarded by watermarks. Watermarks are special records that provide the system with a lower bound for the event-time timestamps of all future rows. Among other things, Flink’s SQL engine uses watermarks to determine when the result of a window can be computed and emitted.

For example, from a watermark with a timestamp of “2018-06-14 15:00:00.0” Flink infers that all windows that end before “2018-06-14 15:00:00.0” can be finalized because no more records with a timestamp equal to or less than the watermark should arrive. Event-time attributes and their watermarks are declared in the table definition.

The second query that I’d like to discuss is very similar to the first one. It computes the average total ride fare per location and hour of day. The difference from the first query is that we do not compute the average for each hour but for each hour of the day; i.e., we aggregate the fares of rides that happened every day at the same time. You can check out the application’s source code in our repository.

```
SELECT  toCellId(dropOffLon, dropOffLat) AS area,  EXTRACT(HOUR FROM  
EXTRACT(HOUR FROM dropOffTime)
```

The query looks very similar to the first one. Syntactically the only difference is that we replaced the `TUMBLE` function with the `EXTRACT` function that extracts the hour of day of a timestamp. However, this modification affects how the query is processed. For the first query, Flink tracks watermarks to decide when to compute the result of a windowed aggregation. This is not possible for the second query, because the query aggregates per hour of the day, such that Flink cannot wait until all data are present before emitting a result. Therefore, Flink emits results and continuously updates them as new rows are ingested.

The query will return a result similar to the one below.

```
5> (false,45563,14,7.125)5> (true,45563,14,7.25)7> (true,49792,14,10  
3> (true,48295,15,6.76)
```

Comparing this result with the result of the first query, you will notice the boolean flags (`false`, `true`) at the start of each row. These flags indicate whether a row is

added to the result (true) or removed from the result (false). Whenever a result needs to be updated, Flink emits two rows, one to remove the previous result and one to add a new result. Also note that in the second set of results the time of day appears as an integer representing the hour, such as 14, rather than something like 2013-01-04 14:00:00.0, as seen in the first set of results.

Using insert and delete messages, Flink is able to encode arbitrary changes and continuously maintain the result of a streaming SQL query in an external storage system, such as PostgreSQL or Elasticsearch. Dashboards or other external applications can connect to such storage systems and retrieve and visualize the results, which are updated with low latency as the input streams are ingested.

The example queries that we have examined demonstrate the versatility of Flink's SQL support. Besides traditional batch analytics, SQL queries can perform common stream analytics operations such as windowed joins and aggregations. At the same time, Flink can maintain and update materialized views with low latency. All use cases benefit from the unified standard SQL syntax and semantics for batch and streaming input data. Flink SQL supports the most common relational operations including projection, selection, aggregation, and joins. Moreover, Flink offers many built-in functions and provides excellent support for user-defined functions, such that custom logic can be easily embedded in SQL queries.

Time to start exploring

The applications and queries discussed in this article should have given you a good intuition of the capabilities and style of Flink's APIs. While Flink's `DataStream` API and `ProcessFunctions` provide low-level control of time and state, Flink's SQL support offers a well-known and concise interface to process streaming and batch data in a unified fashion.

The GitHub repository that we published for this article is a good starting point to learn more about Flink and its APIs. We encourage you to fork and clone it. You can run the examples that I have presented, modify them, or create completely new

applications or queries. Please check out the documentation or reach out to the friendly Flink community via the mailing lists if you have any questions. Happy hacking!

Fabian Hueske is a committer and Project Management Committee member of the Apache Flink project. He is one of the three original creators of Apache Flink and a co-founder of data Artisans, a Berlin-based startup devoted to fostering Flink. Fabian is currently writing a book, “Stream Processing with Apache Flink,” to be published by O’Reilly later this year. Follow him on Twitter at @fhueske.

New Tech Forum provides a venue to explore and discuss emerging enterprise technology in unprecedented depth and breadth. The selection is subjective, based on our pick of the technologies we believe to be important and of greatest interest to InfoWorld readers. InfoWorld does not accept marketing collateral for publication and reserves the right to edit all contributed content. Send all inquiries to newtechforum@infoworld.com.