Streaming analytics with Java and Apache Flink

How to use Flink’s built-in complex event processing engine for real-time streaming analytics

by Eric J. Bruno

July 6, 2020

With IoT and edge applications, some analytics approaches and frameworks use minibatch processing to approximate real-time analytics. This can prove limiting due to the latency that’s injected. This article discusses the benefits of the minibatch approach and suggests using the Apache Flink framework for stateful computations on data streams using minibatches. The goal here is to use Flink’s built-in complex event processing (CEP) engine for such real-time streaming analytics.

Stream analytics processing

Upon receiving an event from a continuous data stream, applications should react to the event immediately. However, with a batch processing approach, you collect data and store it, stop data collection at some point, and then run analytics on what was collected. Then you handle the next batch and aggregate, maintaining context with each iteration (see Figure 1). If the collection times are relatively small, this is usually referred to as minibatch processing.

Figure 1. Minibatching streaming data creates latency and can cause limitations.
Batching live data introduces some challenges, such as determining the batch interval (that is, when to stop batching), controlling latency, and managing storage. Latency is introduced as the data is repeatedly collected and stored while processing to read the data is delayed. Additionally, storing volumes of data can add to cost and hurt performance. Let’s explore an alternative to avoid these challenges.

The benefits of Flink for real-time analytics

Data is a perishable commodity: It holds the most value at the time it’s produced or captured. Alas, the latency of minibatch processing can negatively affect data’s value. By contrast, true stream analytics handles multiple, continuous data streams, allowing you to inspect results, filter and aggregate data, detect trends and patterns, predict outcomes, and change the level of focus in real time (see Figure 2).

![Figure 2. True real-time processing of streaming data limits latency and preserves data value](image)

Stream analytics lets you automate insights beyond historical data analytics. In particular, stream analytics is useful in use cases such as predictive maintenance, algorithmic trading, smart patient care in healthcare, and monitoring factory production lines, as well as in many other use cases. For example, see an article I wrote on complex event processing in healthcare IoT solutions.

The Flink framework provides real-time processing of streaming data without batching. It can also combine streaming data with historical data sources (such as databases) and perform analytics on the aggregate. In my opinion, one of Flink’s most powerful features is its support for CEP, which is perfect for building event-driven analytics applications.

CEP for streaming data

Relational databases and file systems are mostly used to store static data, not process real-time streaming data. CEP addresses this problem by matching incoming streams of events against one or more patterns. It does so in real time, enabling an application to filter out noisy data, uncover trends, or track thresholds without delay. CEP can handle more than one data stream, and the streams may be of varying types. CEP also
supports transformation, aggregation, filtering, and correlation. For instance, a CEP system may combine the processing of data streams from one or more data sources or devices with data stored in a database (see Figure 3).

Figure 3. Overview of stream-based query processing, end to end

Once the system has seen all events for a matching sequence, as defined by the query, results are propagated in real time.

Inside the Flink API

Flink features several libraries for common data processing use cases. The libraries are typically embedded in an API and can be integrated with other libraries:

- **DataSet API**: This is the core API for batch processing applications and data transformations, with state processing.
- **DataStream API**: This API supports stateful streaming applications, using both data streams and time as inputs.
- **FlinkCEP**: Flink’s CEP library provides an API to define patterns of events. It’s integrated with Flink’s DataStream API so patterns are evaluated on DataStreams.
- **Flink Graph API**: Also known as Gelly, this is a library for scalable graph processing and analysis. Gelly is implemented on top of and integrated with the DataSet API and features built-in algorithms.

This article focuses mainly on the DataStream and FlinkCEP APIs.

The Flink CEP engine

According to the online documentation, Apache Flink is designed to run streaming analytics at any scale. Applications are parallelized into tasks that are distributed and executed in a cluster. Its asynchronous and incremental algorithm ensures minimal latency while guaranteeing “exactly once” state consistency. It includes a built-in open source CEP library that embeds WSO2’s CEP engine, which means Flink is versatile enough to handle different kinds of streams.

Let’s look at a sample CEP workflow. Using the FlinkCEP API, you start by defining conditions to monitor, and then apply one or
more of these conditions to a stream of data such as temperature data, as started by the code in Listing 1.

**Listing 1.** A Flink DataStream capturing data from a network socket connection

```java
DataStream<String> deviceRawStream = env.sock
DataStream<TemperatureEvent> inputEventStream = deviceRawStream.map(
    new MapFunction<String, TemperatureEvent>() {
        @Override
        public TemperatureEvent map(String evt) {
            TemperatureEvent tempEv = new TemperatureEvent(evt);
            return tempEv;
        }
    });
```

The code in Listing 1 defines a socket-based `DataStream` and maps the incoming data values to `TemperatureEvent` objects (with a `MapFunction`) to form a secondary `DataStream`. Next, building on the event stream defined above, use CEP to filter out all temperature data, except those values exceeding a threshold (see Listing 2).

**Listing 2.** CEP is useful for filtering out noisy or unneeded data.

```java
IterativeCondition<TemperatureEvent> tempThresholdCondition1 = new IterativeCondition<TemperatureEvent>() {
    @Override
    public boolean filter(TemperatureEvent event, IterativeCondition.Context<TemperatureEvent> context) {
        return event.getTemperature() >= TEMP;
    }
};
...
```

Next, define a CEP pattern to warn when the temperature threshold is exceeded at least twice within a time window (see Listing 3).

**Listing 3.** With FlinkCEP, define patterns to apply to any `DataStream`.

```java
Pattern<TemperatureEvent, ?> warningPattern = Pattern.begin("first")
    .subtype(TemperatureEvent.class)
    .where(tempThresholdCondition1)
    .next("second")
    .subtype(TemperatureEvent.class)
    .where(tempThresholdCondition2)
    .within(Time.seconds(20));
```
Finally, as shown in **Listing 4**, define a **PatternStream** (derived from **DataStream**) and another **DataStream** for actual warnings generated when the filtered temperature data matches the conditions of the pattern. In this case, that condition is defined as two temperature readings that exceed a threshold within a given time frame.

**Listing 4.** Define a DataStream that matches a pattern based on defined conditions.

```java
PatternStream<TemperatureEvent> temperaturePa = CEP.pattern( inputEventStream.keyBy("warningPattern"));

DataStream<TemperatureWarning> warnings = temp
  (Map<String, List<TemperatureEvent>> patt
    TemperatureEvent first = (Temperature
    TemperatureEvent second = (Temperature

    return new TemperatureWarning(
      first.getDeviceId(),
      (first.getTemperature() + second.
    }?
  warnings.print();
```

The **PatternStream** simply selects data from the **DataStream** based on device ID and applies the warning pattern. Next, the code selects from the **PatternStream**, similar to a database **SELECT** statement, to pull the two **TemperatureEvent** objects that make up each warning event. The resulting temperature data is averaged and a **TemperatureWarning** object is emitted as part of the new warning **DataStream**. In this example, the application writes each warning event to the command line:

```
Temperature Warning! Device 1 average tempera
Temperature Warning! Device 1 average tempera
```

Let’s build on this example to work on multiple devices, where a federated average temperature is calculated across all devices.

**Federated analytics example**

The example discussed next builds a solution that averages the temperature readings across each device (say, one per office in a building) as in the example above, and then it creates another stream that averages the temperature across all offices to determine the overall building temperature. **Figure 4** is an overview of the pipelines created by joining these streams. Each “Temp Zone” in this diagram represents an office in a building.
Next, let’s look at an example of aggregating data over time to generate an average using Flink (see Figure 5).

This example shows how easy it is to:

- Add queries to a workflow
- Transform data
- Create time-based windows or trigger-based windows (for example, the availability of data items from the streams) for operations to occur
- Support the aggregation and grouping of data
- Generate results as streams to optionally be used in the next query as part of a workflow

Data transformation

Doing something useful with streams of data often involves transforming that data. Flink supports a wide range of transformation operators with user-defined functions to map data to objects, filter data, or perform operations on that data. Transformation can be as simple as parsing a String to an integer or adding data to a collection, or it can be something more complex such as aggregating or averaging arriving values, which is precisely what needs to be done here.

To aggregate data, define a Flink SingleOutputStreamOperator, which does what its name implies: operates on a single stream of data. In contrast, JoinOperator works on multiple streams. In this case, take the inputEventStream object from Listing 1 and perform an aggregation (see Listing 5).

Listing 5. Perform a transformation on an incoming Flink data stream.
First, define a stream selector to grab values for a specific device. Next, trigger the transformation on a window of events, which in this case is for every new event that arrives. Alternatively, you can process a set of data by defining a window of more than one event. Finally, the `AggregateFunction` interface requires an input object, an accumulator, and a `Tuple`.

This application will use the three-field `Tuple3`, which includes the device ID, the count of temperature readings received, and the sum of all temperature readings. As data arrives, Flink calls the `add` method on the `Aggregation` class (see Listing 6).

**Listing 6.** The `AggregateFunction` to aggregate temperature data

```java
public class Aggregation implements AggregateFunction<TemperatureEvent, AverageAccumulator>
    Tuple3<Integer, Long, Double> {

    public AverageAccumulator add(TemperatureEvent evt) {
        acc.key = evt.getDeviceId();
        acc.sum += evt.getTemperature();
        acc.count++;
        return acc;
    }
}
```

You need to define an accumulator class (described above) and a `serializer`, which defines how the data is written to and read back from a Java `OutputStream` and `InputStream`, respectively. Listing 7 shows how the aggregated data is published (serialized) over NATS.io by defining a Flink `Sink` on the `aggregateProcess` object from Listing 5.

**Listing 7.** Publishing the result of the Flink custom aggregation process

```java
aggregateProcess.addSink(
    new AggregateStreamPublisher(
        pubSubServerAddr, pubSubServerPort, "AggregateTemp"+deviceId) );
```

It's time to explore in detail how this data is used in the federated workflow.
Receiving distributed temperature data

To support distribution, the average and aggregated temperature data from each office device is sent via a Flink pub/sub plugin (implemented in the NATSioPubSubConnector project (the code is available from me here) using the NATS.io message broker, and it is received via a custom Flink source. The workflow is implemented in the FederatedAverageTemperature class and starts by setting up NATS.io listeners for each temperature zone (see Listing 8).

Listing 8. Setting up a NATS.io pub/sub listener for a temperature zone

```java
private void monitorAverageAndAggregateTemperature
    StreamExecutionEnvironment env) throws Exception {
    NATSMessageSchema<String> schema = new
        new PubSubDeserializationSchemaImpl<String>.
    // Connect to device 1 "aggregate tempera
    SourceFunction<String> aggTemp1PubSubSource
        .newBuilder()
            .withDeserializationSchema(deseri
        .withProjectName("project")
    .build();
```

First, a NATSMessageSchema custom object is created and used to let Flink know how to serialize and deserialize NATS.io messages. Next, a PubSubSource custom object is created to listen for published messages from a NATS.io server. The server address and port are provided, along with the subscription name to listen to, such as AggregateTemp1. With that set up, the next step is to create a Flink DataStream that accepts the published aggregate temperature data, as shown in Listing 9.

Listing 9. A Flink DataStream of a zone’s aggregated temperature data

```java
DataStream<String> device1AggregateRawStream
    env.addSource(aggTemp1PubSubSource).r
    device1AggregateRawStream.map(new Map
        AggregateTemperatureEvent>() {
            @Override
            public AggregateTemperatureEv
                new AggregateTemp
                return evt;
        }
});
```
The raw `DataStream` object, `device1AggregateRawStream`, is a stream of temperature values of type `String`. The `DataStream` object, `device1AggregateStream`, is created using these values as input to emit objects of type `AggregateTemperatureEvent`. One of these data flows is created per zone.

Finally, to create the federated average temperature (that is, the building temperature) from the data received from each zone, you combine the `DataStream` for each temperature zone, use a moving “window” based on the number of zones, and compute the global average. For example, if there are 20 temperature zones, you set the window count to 20 and effectively wait until the last 20 zone readings arrive before calculating the global average and waiting for the next 20 readings. In this case, to make the example easier to illustrate and to run, the code uses only two temperature zones (see Listing 10).

**Listing 10.** A union `DataStream` of all data to calculate the global average temperature

```java
// Union of all temp zone (device) data stream
DataStream<AggregateTemperatureEvent> unionStream = device1AggregateStream.union(device2AggregateStream) .keyBy( value -> value.getDeviceId() )

// Apply averaging on temp data across all zones
DataStream<GlobalAverageTemperatureEvent> globalAverageStream = unionStream.countWindowAll(2, 2) // based on number of zones .aggregate(new GlobalAverage());
globalAverageStream.print();
```

The first union `DataStream` arranges the temperature readings by device ID. The second union waits until data is received from each zone, based on a sliding window. In Listing 10, the window indicates that the application should aggregate data after every two zone readings arrive, using the last two zone readings received. (In the real world, you might add time out handlers for individual streams and simply use their last known reading to ensure a proper global average of all temperature zones.) The output is shown in Figure 6.

**Figure 6.** The output from the aggregation application
**Note:** To get a true global average across all zone temperature readings, the calculation uses the aggregate temperature data per zone (the count of readings and the sum of all temperature readings) to generate an average. Due to this, the global average will be slightly different (and more accurate) than simply averaging the average temperatures per zone.

**Running the sample federated temperature application**

The first step in running this sample Flink application is to download and install Apache Flink, which runs on Windows, macOS, and Linux equally well. Next, start Flink by executing the provided shell script in the bin directory where you installed Flink (~/.flink-1.10.0/bin/start-cluster.sh). To stop Flink on UNIX, execute the stop-cluster.sh script, and on Windows, simply close the two command windows that open when Flink starts. Read this tutorial for more information on running Flink.

There are four projects/applications (download them here) that make up this example:

- **TempSource:** Generates a continuous stream of simulated device temperature data
- **NATSioPubSubConnector:** An Apache Flink connector that follows a pattern to allow Flink-based analytics to subscribe to NATS.io pub/sub topics
- **FlinkAverageTemperature:** An Apache Flink application that receives the stream of temperature data from one device and calculates a running average, tracks the aggregate of all temperatures, and publishes the results on a pub/sub topic via NATS.io
- **FederatedAverageTemp:** An Apache Flink application that subscribes to the data from each temperature zone and calculates the global average across all zones

You can run the temperature zones in two individual containers, in two separate VMs, or on physical computers. Note that each container, VM, or computer must run its own instance of Flink. Also, the NATS.io broker needs to run somewhere, and the sample applications (FlinkAverageTemperature and FederatedAverageTemp) each need to be updated with the IP address of the node they are running on. Here are the detailed steps for Temp Zone 1:

1. Start the NATS.io Docker container (available for download at nats.io):
   ```
   sudo docker run -p 4222:4222 -ti nats:latest -m 8222
   ```
2. Start Flink:
   ```
   /home/<user>/flink-1.10.0/bin/start-cluster.sh
   ```
3. Monitor the Flink task executor log:
   tail -f \ flink-1.10.0/log/flink-<username>-taskexecutor-0--<computername>.out

4. Start the simulated temperature device:
   java -jar TempSource/target/TempSource-1.0-SNAPSHOT.jar port 9091

5. Start the Flink average temperature pipeline application:
   flink run
   FlinkAverageTemp/target/FlinkAverageTemp-1.0.jar host <datazone1-IP> port 4222
   deviceid 1

Execute the same steps for Temp Zone 2, except for Step 1 (only one instance of NATS.io needs to run on your network).

The output from this Flink workflow should look like Figure 7:

![Figure 7. Output from the Flink workflow for Temp Zone 2](image)

Next, back on Temp Zone 1, run the Flink application to generate the federated, or global, average temperature. You can run it within an IDE or by executing the following command (all on one line):

```
> java -classpath NATSioPubSubConnector-1.0-SFederatedAverageTemp-1.0.jar host <NATS.io-Ho
```

If all is correct, the output will be similar to what was shown in Figure 6.

**Conclusion**

Flink offers a lot of capability for batch and real-time stream data processing, far beyond what is covered in this article. It offers built-in fault tolerance through savepoints, high availability, distribution and scaling, built-in integrations, and performance tuning. Unlike other frameworks, Flink builds batch processing on top of real-time streaming, not the other way around.

Additionally, there’s support for multiple languages, such as Java, Scala, and Python, and also AI workflows with FlinkML. Going forward, the Flink roadmap includes building out the machine learning pipeline support; unifying the APIs for stream, event-based, and batch processing applications; continually increasing the performance of stream processing; making it easier to deploy Flink applications; expanding the ecosystem; and much more. As the list of companies that use and support Flink grows, you should consider becoming part of that community as well.
Eric J. Bruno

Eric J. Bruno is in the advanced research group at Dell focused on Edge and 5G. He has almost 30 years experience in the information technology community as an enterprise architect, developer, and analyst with expertise in large-scale distributed software design, real-time systems, and edge/IoT. Follow him on Twitter at @ericjbruno.