**Stream processing with streaming dataflows**

Graphical user interface, text, application

Description automatically generated

Let’s say we have a code snippet of a Flink program as shown in the image above.

Step 1: The jar file of this program is distributed amongst all nodes inside the Flink cluster.

Diagram

Description automatically generated

Step 2: Flink converts all the user-defined transformations into a **streaming dataflow**, which is a data structure that encloses the order of operations applied to the input data stream. This dataflow can be represented as a directed graph that always starts with source(s) and ends at sink(s).

Diagram, schematic

Description automatically generated

Step 3: Flink converts the dataflow into an execution graph. This graph not only depicts how the abstract streaming dataflow is mapped to the system’s available resources, it also shows the level of parallelism of the job’s execution process.

**Fault-tolerance with Snapshots**

Flink periodically inserts “markers” into the stream, flowing along with the data. These markers create barriers that split the data stream into 2 parts, containing either old records or new ones. Using this technique, Flink can differentiate the state of the applications at different points in time.

Timeline

Description automatically generated

In Flink’s terminology, markers are *checkpoint barriers* and parts are *checkpoints*. Checkpoint n encapsulates the state of the operators resulted from having consumed all events happening **only before** checkpoint barrier n. It works like loading a computer game from a saved version. This mechanism provides the abilities to, literally, *time travel*. Flink users can go back in time and recover an earlier but consistent version, or experiment with different implementations at a desired point in time.

**Event time and Watermark**

Flink supports different notions of time:

Event time: the time when an event actually occurred, recorded by the edge device producing the event.

Ingestion time: the time when an event is ingested by Flink. If Flink needs to reprocess the data from the message broker due to any failure, this type of timestamp cannot be reproduced.

Processing time: the system clock time when a specific operator processes the event.

As seen from these descriptions, event time is superior for performing most kind of computations and analytics. However, it does come with a cost of introducing complexity since the events can arrive at the operators **out of order**. The solution for this problem is **Watermark**.

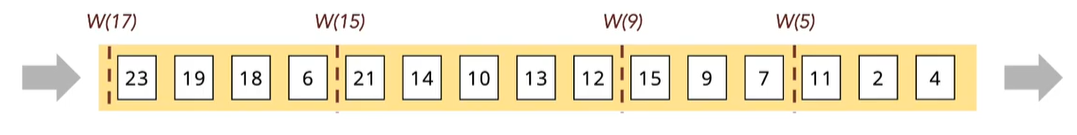
Let’s look at an example of a stream of events with timestamps.

A picture containing diagram

Description automatically generated

It is obvious from the image above that the stream is out of order. Imagine that we are trying to output the same stream of events, but sorted by timestamps. The first element is a 4, and it needs to be delayed at least until the 2 arrives. After these 2 elements are emitted, 11 comes in. From this view provided by the image, we can see *ahead in time* that Flink needs to wait for 7, 9, 10 and 6 to arrive first. But in hindsight, it is very diffcult to know when the results are ready after a while. If there is a chance that 3, 5 or 8 comes in later, Flink will have to decide whether to wait for those elements, or just skip them to maintain low latency, since they are not likely to exist.

Flink uses watermark as the solution for this complex problem. Watermarks define when to stop waiting for more events. Watermark generators inject watermarks into the stream that flow along with the data.



Flink views the out-of-order problem as if each event can arrive after a certain amount of delay, and these delays vary between each element. Developers can configure the maximum threshold on this delay. In the example above, the maximum delay is set to 6. The watermark coming right after element 15 will assert that all following events is probably higher than 15-6=9, thus the name W(9). Element 6 arrives long after W(9), and will be marked as *late*.

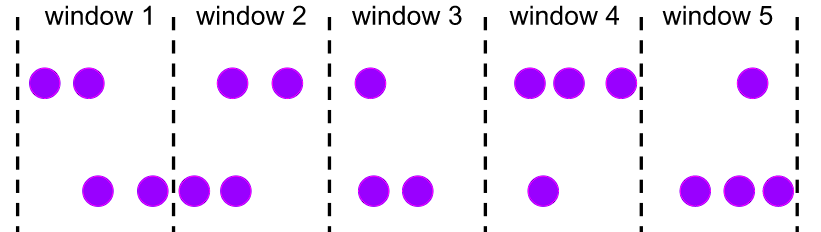
Flink refers to this approach as *bounded-out-of-orderness*. Developers use bounded-out-of-orderness to find the optimal compromise between latency and precision. Flink also provides methods to handle late events. More of this will be introduced in the next part.

**Windows**

When processing streams of data, we often face the tasks of doing aggregate analytics on different parts of the stream, such as: count total items viewed per hour, find the maximum temperature per minute, number of users per week,… In order to calculate these numbers, an analytics software needs to have the ability to slice the data stream into buckets with varying sizes based on the users’ needs. In Flink, this concept is referred to as *Windows.* Each windowed Flink program needs to include 2 main components: *Window Assigner* and *Window functions.*

A Window Assigner defines how its elements are assigned to windows. Flink introduces some pre-implemented window assigner types:

* *Global windows*: assign all elements with the same key to the same window. This kind of window does not have a predetermined end, thus no computation can be done on these global windows, unless developers have already specifed some kind of triggers (discussed later in the section).
* *Tumbling windows*: assign elements to non-overlapping windows of a fixed size. These windows do not overlap



* *Slide windows*: very similar to tumbling windows, but these windows overlap. An additional *window slide* parameter is responsible for deciding how much these windows overlap with each other. For example, say we have windows of size 10 minutes and slide of 5 minutes. This means we get a window containing events in the last 10 minutes every 5 minutes

A picture containing diagram

Description automatically generated

* *Session windows*: assign elements by sessions of activity. A session window closes when it stops receiving elements after a certain amount of time.

Window functions define the type of computations to be performed on each window. Flink gives 3 main types of window functions:

* *ProcessWindowFunction*: receives an iterable of all elements of the window, along with a Context object containing time and state information,
* *ReduceFunction* and *AggregateFunction*: incrementally aggregate the elements of a window.
* Mixture of both types

Flink also provides *Triggers* and *Evictors* in addition to these components. Triggers can react to certain events in a window and starts a window function. Flink comes with 4 basic types of Triggers: *onElement(), onEventTime(), onProcessingTime(), onMerge()*, but users can also design custom Triggers based on their tasks. These methods will be called when their respective invocation events happen, and trigger some computations specified by the users. Evictors can remove elements from a window after a trigger fires and before or after a window function has run.

When working with Event-time window, sometimes elements arrive after a big delay, and is marked as late by the Watermark’s policy, mentioned in the Watermark section. These late elements will be dropped by default, but users can configure a “timer” method called *AllowedLateness* to make Flink hold the state of the window until the timer expires. Late data that arrives during the allowed lateness can be collected by the *sideOutputLateData* and pushed to another output stream.

**Event-driven application**

**References:**

https://nightlies.apache.org/flink/flink-docs-release-1.17/