Flink Event Time

Time and order are important concepts in stateful computations. To produce correct deterministic results, independent of processing time and processing speed, and in the presence of delayed and/or out-of-order records, Flink supports *event time* processing.

Event time processing can produce deterministic results even when the data has timestamps far in the past or future, or when records are received and processed out-of-order with respect to their timestamp, or when processing speed is very slow or very fast.

In event time processing, each record is associated with an application assigned timestamp, usually the time when the event recorded occurred, and computation is performed in reference to this timestamp. This contrasts with the more common, but less precise, case where processing references the current system time (processing time).

When time windowed operations, such as aggregates, are performed in event time, records are assigned to time windows based on their event time. In addition, records may be buffered and/or operators may delay emitting results, in order to handle out-of-order records.

For event time processing to function the following must happen: Event time processing must be enabled; records must be assigned timestamps; the progress of event time must be communicated via watermarks; and if computations produce output records, they must also be assigned timestamps, which is done by Flink internally.

Enabling event time processing

To enable event time processing the *time characteristic* must be set appropriately:
```scala
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
```

Timestamps and Watermarks, Extractors and Generators

Stream records can be assigned a timestamp, an event time. This timestamp can be assigned at the source or in a stream. Sources with a native timestamp can use it as the record timestamp, or the user can assign a timestamp to record at a source or in a stream via a *timestamp extractor*.

Watermarks are stream elements that signal the passage of event time. They also indicate completeness, or the expectation that no records with a timestamp earlier than the watermark will be received after the watermark is emitted. They are created by *watermark generators* and emitted by applying the generator to sources or data streams. Like records, watermarks flow downstream through operators, but unlike them they are processed internally by Flink. Flink ensures that watermarks and records never overtake each other between operators.

Since watermarks signal the passage of event time, operators that depend on the progress of time, such as time window operators, will not fire until they observe an appropriate watermark. Thus, if watermarks are not generated in a timely manner, a job may stall or delay emitting results.

Flink combines timestamp extractors and watermark generators into a single class, which must implement one of two interfaces: `AssignerWithPeriodicWatermarks` or `AssignerWithPunctuatedWatermarks`. The former supports periodic watermark generation, while the later enables punctuated watermarks. The two types cannot be combined.

Periodic Watermarks

`AssignerWithPeriodicWatermarks` generates watermarks periodically. The `getCurrentWatermark()` method will be called at an interval evaluated in processing-time (wall clock). By default the interval is 200 milliseconds and it can be configured via `ExecutionConfig.setAutoWatermarkInterval()`.

Punctuated Watermarks

`AssignerWithPunctuatedWatermarks` generates watermarks based on the records in the stream. For each record, `checkAndGetNextWatermark(lastElement: T, extractedTimestamp: Long)` is called. The method can return a `Watermark` if it wishes to generate one, otherwise it can return `null`.

Assigning Timestamps and Emitting Watermarks

Kafka / Kinesis Connector Sources

Since Kafka 0.10, Kafka records include a timestamp field that can represents either the time at which the Kafka broker received the record from a producer (`LogAppendTime`) or a time assigned by the producer to the record (`CreateTime`). The Flink Kafka connector consumers for Kafka 0.10 and later emit records with timestamps assigned from the Kafka record timestamps if the time characteristic is configured as event-time.

Similarly, Kinesis records are assigned a timestamp that approximately captures the time the record is inserted in the stream, and the Flink Kinesis connector consumer emits records with that timestamp assigned to them.

If the native Kafka or Kinesis record timestamps must be overridden, this can be accomplished in both connectors by configuring the source with a timestamp assigner via `assignTimestampsAndWatermarks()`.

Unlike timestamps, no connector consumers automatically generate watermarks. Watermarks must be explicitly generated by configuring the source with a watermark generator via `assignTimestampsAndWatermarks()`.

If you want to generate watermarks within the connector but desire to use the native Kafka or Kinesis record timestamp, you can define `extractTimestamp()` such that is simply returns the `previousElementTimestamp` argument. The `previousElementTimestamp` argument of the `extractTimestamp()` method contains the native record timestamp.

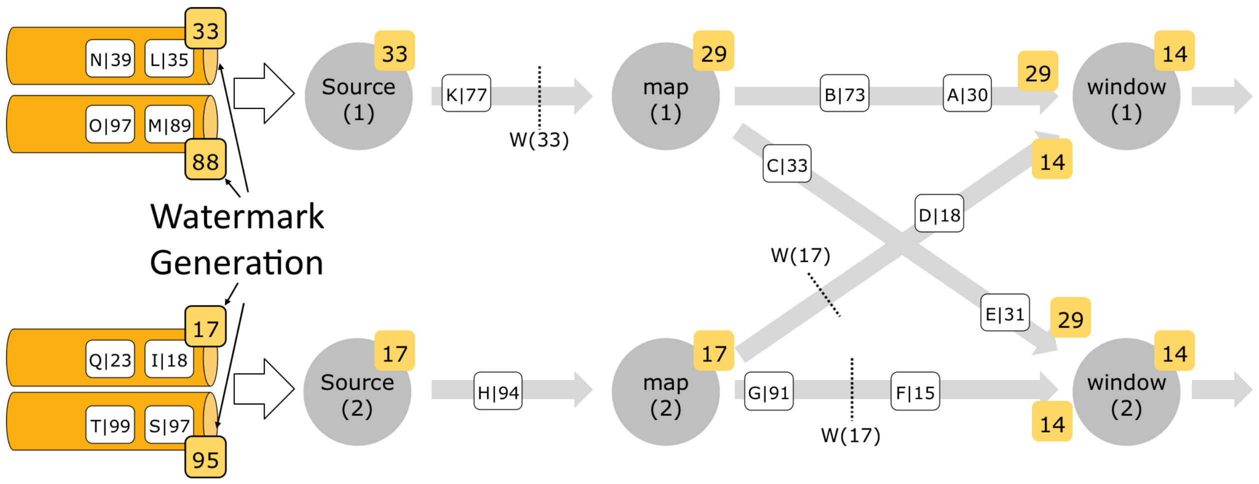
```scala
def extractTimestamp(element: T, prevElementTs: Long): Long = prevElementTs
```

Watermark Flow in the Kafka Connector Source

Kafka topics are split into partitions. Partitions are the unit of consumption. Each Kafka source subtask consumes zero or more partitions. Internally, timestamp extractors and watermark generators are applied to each partition separately so as to ensure that watermarks are not advanced too eagerly. If watermarks were assigned timestamps per subtask instead of per partition, then, depending on how the watermark is computed, subtasks consuming multiple partitions could advanced the watermark even if one of the partitions is behind the others. That could lead to some records being treated as late records.

As multiple partitions may be assigned to a source subtask, there is a need to merge the watermarks generated for each partition. The connector does this the same way that a multiple stream operator merges watermarks from it's input streams, by emitting the minimum latest watermark received across the partitions.

This diagram shows watermark flowing in a job topology, with the addition of watermarks per Kafka partition:



Source

Custom Sources

Custom sources can assign timestamps to records they emit by using `collectWithTimestamp()` in the `SourceContext` given as an argument to the `run` method in `SourceFunction` implementations.

Custom sources can emit watermarks by using `emitWatermark()` in the `SourceContext` given as an argument to the `run` method in `SourceFunction` implementations.

Streams

Records can be assigned timestamps, and watermarks can be emitted, in a stream by calling `DataStream.assignTimestampsAndWatermarks()` with a timestamp assigner. Any previous timestamp assigned to the records, if any, are overwritten from that point downstream.

Operators

Timestamp Handling

Operators can transform records from one or more DataStreams into new records. These new records must also have a timestamp associated with them.

There isn't an API within the operators for user code to emit new records with a specific timestamp. Flink automatically assigns timestamps to the new records. It does so such that the assigned timestamps are aligned with the watermark, meaning that non-late input records won't result in late output records.

If different timestamps are desired, a timestamp assigner can be applied on the data stream downstream from the transform to assign new timestamps. This must be done with care. At this stage of the processing graph, data may have been shuffled, and there may be no strong ordering guarantee to the shuffled records coming from different partitions. As in the case of the Kafka connector and Kafka partitions, applying a timestamp extractor and watermark generator downstream may result in watermarks advancing too eagerly as a result of the relative reordering of shuffled records.

Non-windowed single input transforms such as `map`, `reduce`, `fold`, and `sum` assign the timestamp of the input record to output records, even when the transform, such as `flatMap`, produces multiple output records.

Similarly, non-windowed two input transforms such as CoMap and CoFlatMap, assign the timestamp of the input record being processed to any records output, regardless of which stream the record came from or state the operator may have maintained.

On the other hand, windowed operators assigned emitted records the maximum timestamp that can belong to the window, which for non-global windows is 1 millisecond less than the end timestamp of the window (Flink measures time at millisecond resolution), whereas for global windows it is `Long.MAX_VALUE`. They emit the same timestamp regardless of how many times the window if trigger is fired.

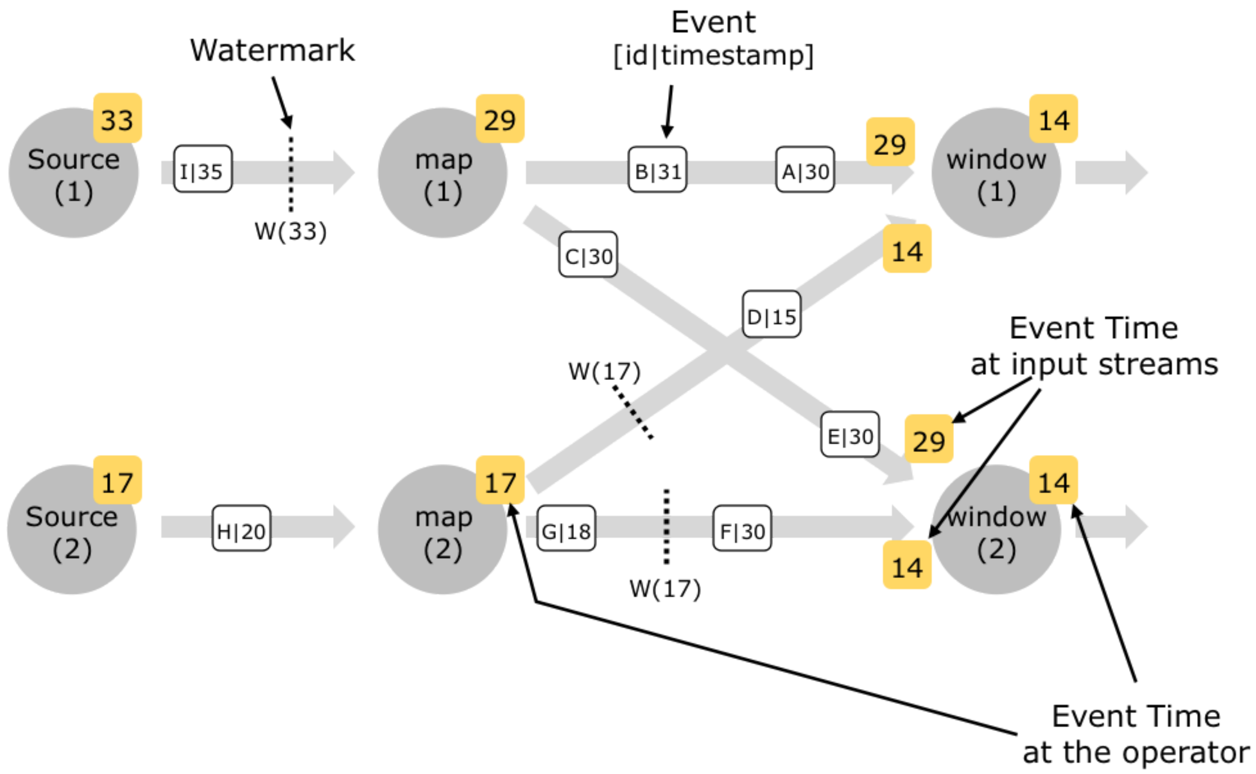
Additionally, operators on keyed streams that implement the the low-level `ProcessFunction` interface have access to timers that can trigger both in processing-time and event-time. If the operator emits a record in an event time `onTimer` callback, the record will be assigned the timer's timestamp, whereas if the record is emitted on a processing time `onTimer` callback, the record won't be assigned any timestamp. Before Flink 1.4.0, in an event time job, records emitted on a precessing time `onTimer` callback were assigned the current processing time in error.

Watermark Handling

Watermarks are generated independently by parallel instances of the watermark generators in subtasks, and flow downstream through operators, advancing event time as they go.

When an operator advances its event time due to a received watermark, it processes all triggering timers, which may result in the emission of new records, before it forwards the watermark. For instance, window operators will first evaluate any windows that fire as a result of the watermark, and after emit the watermark downstream.

This diagram shows watermarks flowing in a job topology:



[Source]

Operators that consume multiple streams, such as union operator or an operator following a `keyBy()`, will receive watermarks from multiple subtasks. Such operators advance event time by using the minimum latest watermark received from its input streams.

If there is a need to redefine watermarks somewhere within a stream, a new watermark generate can be applied to the stream. Upstream watermarks will not be forwarded by the watermark generator. See the warning about about applying a timestamp extractor and watermark generator downstream from the sources of records.

The watermark propagation logic is located in the `StatusWatermarkValve` class.

Stalled Watermarks

As mentioned, operators with multiple input streams and that follow a shuffle (e.g. keyBy), advance event time by using the minimum latest watermark received across the input streams. This means that event time will not advance readily in a multiple input stream operator if one of the input streams is low volume and punctuated watermarks or periodic watermarks based solely on record data are being generated. This can lead to job stalls.

Special cases of this issue are stalled Kafka partitions and stalled Kinesis shards. In these cases, a single partition or shard consumed by a source subtask may be idle, leading to watermark generation by the source stalling.

Punctuated watermarks are problematic in these cases as `checkAndGetNextWatermark(lastElement: T, extractedTimestamp: Long)` is only called when there are new records. And periodic watermarks can be problematic if they generate watermarks based solely on observed records.

For instance, this periodic assigner will not advanced event time if the stream stalls, even though `getCurrentWatermark()` is called periodically:
```scala
class Assigner extends AssignerWithPeriodicWatermarks[T] {
  private var maxTime: Long = _

  def getCurrentWatermark(): Watermark =
    new Watermark(maxTime - 1)

  def extractTimestamp(element: T, prevTS: Long): Long = {
    maxTime = max(maxTime , element.ts)
    element.ts
  }
}
```

To solve this issue, low volume streams can be assigned a periodic watermark extractor that bounds how long the job is willing to wait for records from the stream and to issue watermarks with some bounded delay with reference to wall clock time (processing time) when the stream appears idle, so as to allow operators downstream to proceeded. Beware that this should only be done if the event time is somewhat correlated with processing time (e.g. their difference is bounded) and that it may result in non-deterministic results otherwise.

A more drastic solution is to define multiple input operators that ignore watermarks from some streams with forwarding those of others, such as this `CoStreamFlatMap` operator:
```scala
class SingleWatermarkCoFlatMap[IN1,IN2,OUT](flatMapper:
CoFlatMapFunction[IN1,IN2,OUT]) extends CoStreamFlatMap(flatMapper)  {

  // Pass through the watermarks from the first stream
  override def processWatermark1(mark: Watermark): Unit =
processWatermark(mark)

  // Ignore watermarks from the second stream
  override def processWatermark2(mark: Watermark): Unit = {}
}
```
This should only be used if one of the streams is expected to be low volume, while event time is controlled by the high volume stream. For instance, an operator processing a high volume streams that can be configured via a low volume control stream. Note that this may also result in non-deterministic behavior if the job is restarted.

Stream Status

To deal with idling sources and stalled watermarks, Flink 1.3.0 introduced the concept of stream status, implemented via the `StreamStatus` class, to inform operators whether they can expect records from the upstream operator or source. Stream statuses are generated by sources, while streams can be active or idle.

When an operator's input stream is idle, watermarks from that input stream are no longer taken into consideration by the operator when determining its watermark by computing the minimum latest watermark received from its input streams. Only the watermarks of active input streams are considered.

When all of an operator's input streams are idle, the operator emits an idle stream status to downstream operators. Once at least one of the operator’s input streams becomes active, the operator emits an active stream status downstream.

As of Flink 1.5.0, only the Kafka and Kinesis connector sources make use of this functionality.

The Kafka source marks a stream as idle if a subtask has not yet been assigned any partition start offsets to consume from.

The Kinesis source will mark a stream as idle if a subtask is not assigned any active shards on startup or all shards assigned to it have reached their end. Shards have an end when they have been split or merged in Kinesis.

Custom sources can mark themselves as idle by calling SourceContext.markAsTemporarilyIdle.

As of 1.5.0, no source yet is marked as idle if it has not emitted records for some period of time. There is an open issue to enable this functionality.

Handling Out-of-Order Records

Out-of-order records are a fact of life in distributed systems. Therefore, your job must be prepared to handle them. Flink provides a couple of mechanisms to take care of them.

Delaying Watermarks

One approach to handling out-of-order records is to delay watermarks by some fixed amount of time, either from the maximum observed record timestamp or from the current processing time, with the assumption that most or all out-of-order events will be received within that time span.

Flink has a predefined periodic watermark generator, `BoundedOutOfOrdernessTimestampExtractor` that delays watermarks a fixed amount from the maximum observed record timestamp.

This is equivalent to setting your event time clock back some fixed amount of time. Thus, it can result in the job output being delayed by the same amount of time, although this can be mitigated in some cases by the early firing of windows.

Delaying watermarks works well for short-term out-of-order records, such as those induced by the normal latencies involved in well functioning distributed system. These are usually in the order of seconds or at most minutes.

Late Records in Window operators

Flink's time windows operators support processing of late records. Late records are records within a time window that are received by the operator after it has processed a watermark with a timestamp greater than the window end timestamp, or the point in event time when the window computation is normally triggered.

Normally, after event time passes the window's end timestamp and the window is evaluated, the window's state is discarded and late records are dropped.

But if `WindowedStream.allowedLateness()` is used to specify an allowed lateness duration, the window state will be kept for that amount of time after the watermark has passed the end of the window, and window records that arrive during that period will be added to the window. Depending on the trigger used, this may cause the window computation to trigger again.

This can be used to emit the result of a window computation when a watermark passes the end of the window, and again, with an updated result, if there are late records. For this to succeed, downstream operators or sinks must be able to handle result updates.

Session windows represent a special case. They also support processing of late records, but in their case they may result in the merging of windows, as the new records may bridge previous windows.

In addition, late records that do not fall within the allowed lateness can be sent to side output by using `WindowedStream.sideOutputLateData()`. This side output can then be process as needed.

Allowed lateness is a better mechanism for handling records that are more highly out-of-order, in the order of minutes or hours, as it will allow the job to emit timely results while handling late records. But it will do so at the expense of more resources, needed to keep the window state longer, and downstream operators and sinks that can handle updated window records.

Late Records in ProcessFunction

The low-level `ProcessFunction` has access to the timestamp of the record via the context `timestamp()` method and to the current watermark via the TimerServer `getCurrentWatermark()` method, which is available via the context `timerService()` method. It can also output records to a side output via the context `output()`. This permits the `ProcessFunction` to filter out or redirect late records and mimic the allowed lateness behavior of windows.

Debugging Watermarks

The low watermark, or the minimum latest watermark received by a task, can be viewed in the Flink UI by selected the job, then /Overview/, and /Watermarks/. You can click on the operator to see the per subtask watermarks. Note these are the operator's current watermark. Sources won't display any, which is not very helpful. And if the operator has multiple input sources, you'll only be able to see the minimum of the latest watermarks from the sources (ergo “low” watermark), not the per source watermarks, which would be more useful for debugging. Ideally, the system would show watermarks emitted by generators.

The low watermarks are also available via the REST API, using the path **/jobs/:jobid/vertices/:vertexid/metrics**, and a query parameter such as **get=0.currentLowWatermark,...,N.currentLowWatermark**, where N is the vertex parallelism minus 1.