**CHAPTER 6**

**Time-Based and Window Operators**

In this chapter, we will cover DataStream API methods for time handling and time- based operators, like windows. As you learned in [“Time Semantics”](#_bookmark35), Flink’s time- based operators can be applied with different notions of time.

First, we will learn how to define time characteristics, timestamps, and watermarks. Then, we will cover the process functions, low-level transformations that provide access to timestamps and watermarks and can register timers. Next, we will get to use Flink’s window API, which provides built-in implementations of the most common window types. You will also get an introduction to custom, user-defined window operations and core windowing constructs, such as assigners, triggers, and evictors. Finally, we will discuss how to join streams on time and strategies to handle late events.

# Configuring Time Characteristics

To define time operations in a distributed stream processing application, it is impor‐ tant to understand the meaning of time. When you specify a window to collect events in one-minute buckets, which events exactly will each bucket contain? In the Data‐ Stream API, you can use the *time characteristic* to tell Flink how to define time when you are creating windows. The time characteristic is a property of the

StreamExecutionEnvironment and it takes the following values:

ProcessingTime

specifies that operators determine the current time of the data stream according

to the system clock of the machine where they are being executed. Processing- time windows trigger based on machine time and include whatever elements happen to have arrived at the operator until that point in time. In general, using processing time for window operations results in nondeterministic results

because the contents of the windows depend on the speed at which the elements arrive. This setting offers very low latency because processing tasks do not have to wait for watermarks to advance the event time.

EventTime

specifies that operators determine the current time by using information from

the data itself. Each event carries a timestamp and the logical time of the system is defined by watermarks. As you learned in [“Timestamps” on page 47](#_bookmark31), time‐ stamps either exist in the data before entering the data processing pipeline, or are assigned by the application at the sources. An event-time window triggers when the watermarks declare that all timestamps for a certain time interval have been received. Event-time windows compute deterministic results even when events arrive out of order. The window result will not depend on how fast the stream is read or processed.

IngestionTime

specifies the processing time of the source operator as an event time timestamp

to each ingested record and automatically generates watermarks. It is a hybrid of EventTime and ProcessingTime. The ingestion time of an event is the time it entered the stream processor. Ingestion time does not offer much practical value

compared to event time as it does not provide deterministic results and has simi‐ lar performance as event time.

[Example 6-1](#_bookmark17) shows how to set the time characteristic by revisiting the sensor stream‐ ing application code you wrote in [“Hello, Flink!” on page 79](#_bookmark18).

*Example 6-1. Setting the time characteristic to event time*

**object AverageSensorReadings** {

*// main() defines and executes the DataStream program*

**def** main(args**: Array**[**String**]) {

*// set up the streaming execution environment*

**val** env **= StreamExecutionEnvironment**.getExecutionEnvironment

*// use event time for the application*

env.setStreamTimeCharacteristic(**TimeCharacteristic**.**EventTime**)

*// ingest sensor stream*

**val** sensorData**: DataStream**[**SensorReading**] **=** env.addSource(...)

}

}

Setting the time characteristic to EventTime enables timestamp and watermark handling, and as a result, event-time operations. Of course, you can still use processing-time windows and timers if you choose the EventTime time characteristic.

In order to use processing time, replace TimeCharacteristic.EventTime with

TimeCharacteristic.ProcessingTime.

## Assigning Timestamps and Generating Watermarks

As discussed in [“Event-Time Processing” on page 47](#_bookmark7), your application needs to pro‐ vide two important pieces of information to Flink in order to operate in event time. Each event must be associated with a timestamp that typically indicates when the event actually happened. Event-time streams also need to carry watermarks from which operators infer the current event time.

Timestamps and watermarks are specified in milliseconds since the epoch of 1970-01-01T00:00:00Z. A watermark tells operators that no more events with a timestamp less than or equal to the watermark are expected. Timestamps and water‐ marks can be either assigned and generated by a SourceFunction or using an explicit user-defined timestamp assigner and watermark generator. Assigning timestamps and generating watermarks in a SourceFunction is discussed in [“Source Functions,](#_bookmark32) [Timestamps, and Watermarks” on page 204](#_bookmark32). Here we explain how to do this with a

user-defined function.

#### Overriding Source-Generated Timestamps and Watermarks

If a timestamp assigner is used, any existing timestamps and water‐ marks will be overwritten.

The DataStream API provides the TimestampAssigner interface to extract time‐ stamps from elements after they have been ingested into the streaming application.

Typically, the timestamp assigner is called right after the source function because most assigners make assumptions about the order of elements with respect to their timestamps when generating watermarks. Since elements are typically ingested in parallel, any operation that causes Flink to redistribute elements across parallel

stream partitions, such as parallelism changes, keyBy(), or other explicit redistribu‐

tions, mixes up the timestamp order of the elements.

It is best practice to assign timestamps and generate watermarks as close to the sour‐ ces as possible or even within the SourceFunction. Depending on the use case, it is possible to apply an initial filtering or transformation on the input stream before

assigning timestamps if such operations do not induce a redistribution of elements.

To ensure that event-time operations behave as expected, the assigner should be called before any event-time dependent transformation (e.g., before the first event- time window).

Timestamp assigners behave like other transformation operators. They are called on a stream of elements and produce a new stream of timestamped elements and water‐

marks. Timestamp assigners do not change the data type of a DataStream.

The code in [Example 6-2](#_bookmark8) shows how to use a timestamp assigner. In this example, after reading the stream, we first apply a filter transformation and then call the

assignTimestampsAndWatermarks() method where we define the timestamp assigner

MyAssigner().

*Example 6-2. Using a timestamp assigner*

**val** env **= StreamExecutionEnvironment**.getExecutionEnvironment

*// set the event time characteristic*

env.setStreamTimeCharacteristic(**TimeCharacteristic**.**EventTime**)

*// ingest sensor stream*

**val** readings**: DataStream**[**SensorReading**] **=** env

.addSource(**new SensorSource**)

*// assign timestamps and generate watermarks*

.assignTimestampsAndWatermarks(**new MyAssigner**())

In the example above, MyAssigner can either be of type AssignerWithPeriodicWater marks or AssignerWithPunctuatedWatermarks. These two interfaces extend the Time stampAssigner interface provided by the DataStream API. The first interface defines assigners that emit watermarks periodically while the second injects watermarks

based on a property of the input events. We describe both interfaces in detail next.

**Assigner with periodic watermarks**

Assigning watermarks periodically means that we instruct the system to emit water‐ marks and advance the event time in fixed intervals of machine time. The default

interval is set to two hundred milliseconds, but it can be configured using the Execu tionConfig.setAutoWatermarkInterval() method:

**val** env **= StreamExecutionEnvironment**.getExecutionEnvironment env.setStreamTimeCharacteristic(**TimeCharacteristic**.**EventTime**)

*// generate watermarks every 5 seconds*

env.getConfig.setAutoWatermarkInterval(5000)

In the preceding example, you instruct the program to emit watermarks every 5 sec‐ onds. Actually, every 5 seconds, Flink invokes the getCurrentWatermark() method of AssignerWithPeriodicWatermarks. If the method returns a nonnull value with a

timestamp larger than the timestamp of the previous watermark, the new watermark

is forwarded. This check is necessary to ensure event time continuously increases; otherwise no watermark is produced.

[Example 6-3](#_bookmark4) shows an assigner with periodic timestamps that produces watermarks by keeping track of the maximum element timestamp it has seen so far. When asked for a new watermark, the assigner returns a watermark with the maximum timestamp minus a 1-minute tolerance interval.

*Example 6-3. A periodic watermark assigner*

**class PeriodicAssigner**

**extends AssignerWithPeriodicWatermarks**[**SensorReading**] {

**val** bound**: Long** = 60 \* 1000 *// 1 min in ms*

**var** maxTs**: Long** = **Long**.**MinValue** *// the maximum observed timestamp*

**override def** getCurrentWatermark**: Watermark** = {

*// generated watermark with 1 min tolerance*

**new Watermark**(maxTs - bound)

}

**override def** extractTimestamp( r**: SensorReading**, previousTS**: Long**)**: Long** = {

*// update maximum timestamp*

maxTs **=** maxTs.max(r.timestamp)

*// return record timestamp*

r.timestamp

}

}

The DataStream API provides implementations for two common cases of timestamp assigners with periodic watermarks. If your input elements have timestamps that are

monotonically increasing, you can use the shortcut method assignAscending TimeStamps. This method uses the current timestamp to generate watermarks, since no earlier timestamps can appear. The following shows how to generate watermarks

for ascending timestamps:

**val** stream**: DataStream**[**SensorReading**] **=** ...

**val** withTimestampsAndWatermarks **=** stream

.assignAscendingTimestamps(e **=>** e.timestamp)

The other common case of periodic watermark generation is when you know the maximum lateness that you will encounter in the input stream—the maximum differ‐ ence between an element’s timestamp and the largest timestamp of all perviously

ingested elements. For such cases, Flink provides the BoundedOutOfOrder nessTimeStampExtractor, which takes the maximum expected lateness as an argu‐ ment:

**val** stream**: DataStream**[**SensorReading**] **=** ...

**val** output **=** stream.assignTimestampsAndWatermarks(

**new BoundedOutOfOrdernessTimestampExtractor**[**SensorReading**]( **Time**.seconds(10))(e =>.timestamp)

In the preceding code, elements are allowed to be late up to 10 seconds. This means if the difference between an element’s event time and the maximum timestamp of all previous elements is greater than 10 seconds, the element might arrive for processing after its corresponding computation has completed and the result has been emitted. Flink offers different strategies to handle such late events, and we discuss those later in [“Handling Late Data” on page 148](#_bookmark9).

### Assigner with punctuated watermarks

Sometimes the input stream contains special tuples or markers that indicate the stream’s progress. Flink provides the AssignerWithPunctuatedWatermarks interface for such cases, or when watermarks can be defined based on some other property of

the input elements. It defines the checkAndGetNextWatermark() method, which is called for each event right after extractTimestamp(). This method can decide to gen‐ erate a new watermark or not. A new watermark is emitted if the method returns a

nonnull watermark that is larger than the latest emitted watermark.

[Example 6-4](#_bookmark19) shows a punctuated watermark assigner that emits a watermark for every reading it receives from the sensor with the ID "sensor\_1".

*Example 6-4. A punctuated watermark assigner*

**class PunctuatedAssigner**

**extends AssignerWithPunctuatedWatermarks**[**SensorReading**] {

**val** bound**: Long** = 60 \* 1000 *// 1 min in ms*

**override def** checkAndGetNextWatermark( r**: SensorReading**,

extractedTS**: Long**)**: Watermark** = {

**if** (r.id == "sensor\_1") {

*// emit watermark if reading is from sensor\_1*

**new Watermark**(extractedTS - bound)

} **else** {

*// do not emit a watermark*

**null**

}

}

**override def** extractTimestamp( r**: SensorReading**, previousTS**: Long**)**: Long** = {

*// assign record timestamp*

r.timestamp

}

}

## Watermarks, Latency, and Completeness

So far we have discussed how to generate watermarks using a TimestampAssigner. What we have not discussed yet is the effect that watermarks have on your streaming

application.

Watermarks are used to balance latency and result completeness. They control how long to wait for data to arrive before performing a computation, such as finalizing a window computation and emitting the result. An operator based on event time uses watermarks to determine the completeness of its ingested records and the progress of its operation. Based on the received watermarks, the operator computes a point in time up to which it expects to have received relevant input records.

However, the reality is that we can never have perfect watermarks because that would mean we are always certain there are no delayed records. In practice, you need to make an educated guess and use heuristics to generate watermarks in your applica‐ tions. You need to use whatever information you have about the sources, the network, and the partitions to estimate progress and an upper bound for the lateness of your input records. Estimates mean there is room for errors, in which case you might gen‐ erate watermarks that are inaccurate, resulting in late data or an unnecessary increase in the application’s latency. With this in mind, you can use watermarks to balance result latency and result completeness.

If you generate loose watermarks—where the watermarks are far behind the time‐ stamps of the processed records—you increase the latency of the produced results. You may have been able to generate a result earlier but you had to wait for the water‐ mark. Moreover the state size typically increases because the application needs to buffer more data until it can perform a computation. However, you can be quite cer‐ tain all relevant data is available when you perform a computation.

On the other hand, if you generate very tight watermarks—watermarks that might be larger than the timestamps of some later records—time-based computations might be performed before all relevant data has arrived. While this might yield incomplete or inaccurate results, the results are produced in a timely fashion with lower latency.

Unlike batch applications, which are built around the premise that all data is avail‐ able, the latency/completeness tradeoff is a fundamental characteristic of stream pro‐ cessing applications, which process unbounded data as it arrives. Watermarks are a powerful way to control the behavior of an application with respect to time. Besides watermarks, Flink has many features to tweak the exact behavior of time-based oper‐ ations, such as process functions and window triggers, and offers different ways to handle late data, which are discussed in [“Handling Late Data” on page 148](#_bookmark9).

# Process Functions

Even though time information and watermarks are crucial to many streaming appli‐ cations, you might have noticed that we cannot access them through the basic Data‐

Stream API transformations we have seen so far. For example, a MapFunction does

not have access to timestamps or the current event time.

The DataStream API provides a family of low-level transformations, the process functions, which can also access record timestamps and watermarks and register timers that trigger at a specific time in the future. Moreover, process functions feature side outputs to emit records to multiple output streams. Process functions are com‐ monly used to build event-driven applications and to implement custom logic for which predefined windows and transformations might not be suitable. For example, most operators for Flink’s SQL support are implemented using process functions.

Currently, Flink provides eight different process functions: ProcessFunction, Keyed ProcessFunction, CoProcessFunction, ProcessJoinFunction, BroadcastProcess Function, KeyedBroadcastProcessFunction, ProcessWindowFunction, and ProcessAllWindowFunction. As indicated by name, these functions are applicable in

different contexts. However, they have a very similar feature set. We will continue dis‐ cussing these common features by looking in detail at the KeyedProcessFunction.

The KeyedProcessFunction is a very versatile function and can be applied to a Keyed Stream. The function is called for each record of the stream and returns zero, one, or more records. All process functions implement the RichFunction interface and hence offer open(), close(), and getRuntimeContext() methods. Additionally, a KeyedPro cessFunction[KEY, IN, OUT] provides the following two methods:

1. processElement(v: IN, ctx: Context, out: Collector[OUT]) is called for each record of the stream. As usual, result records are emitted by passing them to the Collector. The Context object is what makes a process function special. It gives access to the timestamp and the key of the current record and to a TimerService. Moreover, the Context can emit records to side outputs.
2. onTimer(timestamp: Long, ctx: OnTimerContext, out: Collector[OUT]) is a callback function that is invoked when a previously registered timer triggers. The timestamp argument gives the timestamp of the firing timer and the Collec

tor allows records to be emitted. The OnTimerContext provides the same serv‐

ices as the Context object of the processElement() method and also returns the time domain (processing time or event time) of the firing trigger.

## TimerService and Timers

The TimerService of the Context and OnTimerContext objects offers the following methods:

* currentProcessingTime(): Long returns the current processing time.
* currentWatermark(): Long returns the timestamp of the current watermark.
* registerProcessingTimeTimer(timestamp: Long): Unit registers a processing time timer for the current key. The timer will fire when the processing time of the

executing machine reaches the provided timestamp.

* registerEventTimeTimer(timestamp: Long): Unit registers an event-time timer for the current key. The timer will fire when the watermark is updated to a

timestamp that is equal to or larger than the timer’s timestamp.

* deleteProcessingTimeTimer(timestamp: Long): Unit deletes a processing- time timer that was previously registered for the current key. If no such timer

exists, the method has no effect.

* deleteEventTimeTimer(timestamp: Long): Unit deletes an event-time timer that was previously registered for the current key. If no such timer exists, the

method has no effect.

When a timer fires, the onTimer() callback function is called. The processElement() and onTimer() methods are synchronized to prevent concurrent access and manipu‐ lation of state.

#### Timers on Nonkeyed Streams

Timers can only be registered on keyed streams. A common use case for timers is to clear keyed state after some period of inactivity for a key or to implement custom time-based windowing logic. To use timers on a nonkeyed stream, you can create a keyed stream by

using a KeySelector with a constant dummy key. Note that this

will move all data to a single task such that the operator would be

effectively executed with a parallelism of 1.

For each key and timestamp, exactly one timer can be registered, which means each key can have multiple timers but only one for each timestamp. By default, a

KeyedProcessFunction holds the timestamps of all timers in a priority queue on the

heap. However, you can configure the RocksDB state backend to also store the timers.

Timers are checkpointed along with any other state of the function. If an application needs to recover from a failure, all processing-time timers that expired while the application was restarting will fire immediately when the application resumes. This is also true for processing-time timers that are persisted in a savepoint. Timers are always asynchronously checkpointed, with one exception. If you are using the

RocksDB state backend with incremental checkpoints and storing the timers on the heap (default setting), they are checkpointed synchronously. In that case, it is recom‐ mended to not use timers excessively, to avoid long checkpointing times.

Timers that are registered for a timestamp in the past are not silently dropped but processed as well. Processing-time timers fire immediately after the registering method returns. Event-time timers fire when the next watermark is processed.

The following code shows how to apply a KeyedProcessFunction to a KeyedStream. The function monitors the temperatures of sensors and emits a warning if the tem‐

perature of a sensor monotonically increases for a period of 1 second in processing time:

**val** warnings **=** readings

*// key by sensor id*

.keyBy(**\_**.id)

*// apply ProcessFunction to monitor temperatures*

.process(**new TempIncreaseAlertFunction**)

The implementation of the TempIncreaseAlterFunction is shown in [Example 6-5](#_bookmark20).

*Example 6-5. A KeyedProcessFunction that emits a warning if the temperature of a sensor monotonically increases for 1 second in processing time*

*/\*\* Emits a warning if the temperature of a sensor*

*\* monotonically increases for 1 second (in processing time).*

*\*/*

**class TempIncreaseAlertFunction**

**extends KeyedProcessFunction**[**String**, **SensorReading**, **String**] {

*// stores temperature of last sensor reading*

**lazy val** lastTemp**: ValueState**[**Double**] **=** getRuntimeContext.getState(

**new ValueStateDescriptor**[**Double**]("lastTemp", **Types**.of[**Double**]))

*// stores timestamp of currently active timer*

**lazy val** currentTimer**: ValueState**[**Long**] **=** getRuntimeContext.getState(

**new ValueStateDescriptor**[**Long**]("timer", **Types**.of[**Long**]))

**override def** processElement( r**: SensorReading**,

ctx**: KeyedProcessFunction**[**String**, **SensorReading**, **String**]**#Context**, out**: Collector**[**String**])**: Unit** = {

*// get previous temperature*

**val** prevTemp **=** lastTemp.value()

*// update last temperature*

lastTemp.update(r.temperature)

**val** curTimerTimestamp **=** currentTimer.value();

**if** (prevTemp == 0.0 || r.temperature < prevTemp) {

*// temperature decreased; delete current timer* ctx.timerService().deleteProcessingTimeTimer(curTimerTimestamp) currentTimer.clear()

} **else if** (r.temperature > prevTemp && curTimerTimestamp == 0) {

*// temperature increased and we have not set a timer yet*

*// set processing time timer for now + 1 second*

**val** timerTs **=** ctx.timerService().currentProcessingTime() + 1000 ctx.timerService().registerProcessingTimeTimer(timerTs)

*// remember current timer*

currentTimer.update(timerTs)

}

}

**override def** onTimer( ts**: Long**,

ctx**: KeyedProcessFunction**[**String**, **SensorReading**, **String**]**#OnTimerContext**, out**: Collector**[**String**])**: Unit** = {

out.collect("Temperature of sensor '" + ctx.getCurrentKey + "' monotonically increased for 1 second.") currentTimer.clear()

}

}

## Emitting to Side Outputs

Most operators of the DataStream API have a single output—they produce one result stream with a specific data type. Only the split operator allows splitting a stream into multiple streams of the same type. Side outputs are a feature of process functions to emit multiple streams from a function with possibly different types. A side output is

identified by an OutputTag[X] object, where X is the type of the resulting side output

stream. Process functions can emit a record to one or more side outputs via the

Context object.

[Example 6-6](#_bookmark27) shows how to emit data from a ProcessFunction via DataStream of a side output.

*Example 6-6. Applying a ProcessFunction that emits to a side output*

**val** monitoredReadings**: DataStream**[**SensorReading**] **=** readings

*// monitor stream for readings with freezing temperatures*

.process(**new FreezingMonitor**)

*// retrieve and print the freezing alarms side output*

monitoredReadings

.getSideOutput(**new OutputTag**[**String**]("freezing-alarms"))

.print()

*// print the main output*

readings.print()

[Example 6-7](#_bookmark5) shows the FreezingMonitor function that monitors a stream of sensor readings and emits a warning to a side output for readings with a temperature below

32°F.

*Example 6-7. A ProcessFunction that emits records to a side output*

*/\*\* Emits freezing alarms to a side output for readings*

*\* with a temperature below 32F. \*/*

**class FreezingMonitor extends ProcessFunction**[**SensorReading**, **SensorReading**] {

*// define a side output tag*

**lazy val** freezingAlarmOutput**: OutputTag**[**String**] **= new OutputTag**[**String**]("freezing-alarms")

**override def** processElement( r**: SensorReading**,

ctx**: ProcessFunction**[**SensorReading**, **SensorReading**]**#Context**, out**: Collector**[**SensorReading**])**: Unit** = {

*// emit freezing alarm if temperature is below 32F*

**if** (r.temperature < 32.0) {

ctx.output(freezingAlarmOutput, s"Freezing Alarm for ${r.id}")

}

*// forward all readings to the regular output*

out.collect(r)

}

}

## CoProcessFunction

For low-level operations on two inputs, the DataStream API also provides the CoProcessFunction. Similar to a CoFlatMapFunction, a CoProcessFunction offers a transformation method for each input, processElement1() and processElement2(). Similar to the ProcessFunction, both methods are called with a Context object that

gives access to the element or timer timestamp, a TimerService, and side outputs. The CoProcessFunction also provides an onTimer() callback method. [Example 6-8](#_bookmark28) shows how to apply a CoProcessFunction to combine two streams.

*Example 6-8. Applying a CoProcessFunction*

*// ingest sensor stream*

**val** sensorData**: DataStream**[**SensorReading**] **=** ...

*// filter switches enable forwarding of readings*

**val** filterSwitches**: DataStream**[(**String**, **Long**)] **=** env

.fromCollection(**Seq**(

("sensor\_2", 10 \* 1000L), *// forward sensor\_2 for 10 seconds*

("sensor\_7", 60 \* 1000L)) *// forward sensor\_7 for 1 minute*

)

**val** forwardedReadings **=** readings

*// connect readings and switches*

.connect(filterSwitches)

*// key by sensor ids*

.keyBy(**\_**.id, **\_**.\_1)

*// apply filtering CoProcessFunction*

.process(**new ReadingFilter**)

The implementation of a ReadingFilter function that dynamically filters a stream of sensor readings based on a stream of filter switches is shown in [Example 6-9](#_bookmark10).

*Example 6-9. Implementation of a CoProcessFunction that dynamically filters a stream of sensor readings*

**class ReadingFilter**

**extends CoProcessFunction**[**SensorReading**, (**String**, **Long**), **SensorReading**] {

*// switch to enable forwarding*

**lazy val** forwardingEnabled**: ValueState**[**Boolean**] **=** getRuntimeContext.getState(

**new ValueStateDescriptor**[**Boolean**]("filterSwitch", **Types**.of[**Boolean**]))

*// hold timestamp of currently active disable timer*

**lazy val** disableTimer**: ValueState**[**Long**] **=** getRuntimeContext.getState(

**new ValueStateDescriptor**[**Long**]("timer", **Types**.of[**Long**]))

**override def** processElement1( reading**: SensorReading**,

ctx**: CoProcessFunction**[**SensorReading**, (**String**, **Long**), **SensorReading**]**#Context**, out**: Collector**[**SensorReading**])**: Unit** = {

*// check if we may forward the reading* **if** (forwardingEnabled.value()) { out.collect(reading)

}

}

**override def** processElement2( switch**:** (**String**, **Long**),

ctx**: CoProcessFunction**[**SensorReading**, (**String**, **Long**), **SensorReading**]**#Context**, out**: Collector**[**SensorReading**])**: Unit** = {

*// enable reading forwarding*

forwardingEnabled.update(**true**)

*// set disable forward timer*

**val** timerTimestamp **=** ctx.timerService().currentProcessingTime() + switch.\_2

**val** curTimerTimestamp **=** disableTimer.value()

**if** (timerTimestamp > curTimerTimestamp) {

*// remove current timer and register new timer* ctx.timerService().deleteEventTimeTimer(curTimerTimestamp) ctx.timerService().registerProcessingTimeTimer(timerTimestamp) disableTimer.update(timerTimestamp)

}

}

**override def** onTimer( ts**: Long**,

ctx**: CoProcessFunction**[**SensorReading**, (**String**, **Long**), **SensorReading**]

**#OnTimerContext**, out**: Collector**[**SensorReading**])**: Unit** = {

*// remove all state; forward switch will be false by default*

forwardingEnabled.clear() disableTimer.clear()

}

}

# Window Operators

Windows are common operations in streaming applications. They enable transfor‐ mations such as aggregations on *bounded intervals* of an unbounded stream. Typi‐ cally, these intervals are defined using time-based logic. Window operators provide a way to group events in buckets of finite size and apply computations on the bounded contents of these buckets. For example, a window operator can group the events of a stream into windows of 5 minutes and count for each window how many events have been received.

The DataStream API provides built-in methods for the most common window oper‐ ations as well as a very flexible windowing mechanism to define custom windowing logic. In this section, we show you how to define window operators, present the built- in window types of the DataStream API, discuss the functions that can be applied on a window, and finally explain how to define custom windowing logic.

## Defining Window Operators

Window operators can be applied on a keyed or a nonkeyed stream. Window opera‐ tors on keyed windows are evaluated in parallel, and nonkeyed windows are processed in a single thread.

To create a window operator, you need to specify two window components:

1. A *window assigner* that determines how the elements of the input stream are grouped into windows. A window assigner produces a WindowedStream (or All WindowedStream if applied on a nonkeyed DataStream).
2. A *window function* that is applied on a WindowedStream (or AllWindowedStream)

and processes the elements that are assigned to a window.

The following code shows how to specify a window assigner and a window function on a keyed or nonkeyed stream:

*// define a keyed window operator*

stream

.keyBy(...)

.window(...) *// specify the window assigner*

.reduce/aggregate/process(...) *// specify the window function*

*// define a nonkeyed window-all operator*

stream

.windowAll(...) *// specify the window assigner*

.reduce/aggregate/process(...) *// specify the window function*

In the remainder of the chapter we focus on keyed windows only. Nonkeyed windows (also called all-windows in the DataStream API) behave exactly the same, except that they collect all data and are not evaluated in parallel.

Note that you can customize a window operator by providing a custom trigger or evictor and declaring strategies to deal with late elements. Custom window operators are discussed in detail later in this section.

## Built-in Window Assigners

Flink provides built-in window assigners for the most common windowing use cases. All assigners we discuss here are time-based and were introduced in [“Operations on](#_bookmark34) [Data Streams” on page 22](#_bookmark34). Time-based window assigners assign an element based on its event-time timestamp or the current processing time to windows. Time windows have a start and an end timestamp.

All built-in window assigners provide a default trigger that triggers the evaluation of a window once the (processing or event) time passes the end of the window. It is important to note that a window is created when the first element is assigned to it. Flink will never evaluate empty windows.

#### Count-Based Windows

In addition to time-based windows, Flink also supports count- based windows—windows that group a fixed number of elements in the order in which they arrive at the window operator. Since they depend on the ingestion order, count-based windows are not deter‐ ministic. Moreover, they can cause issues if they are used without a custom trigger that discards incomplete and stale windows at some point.

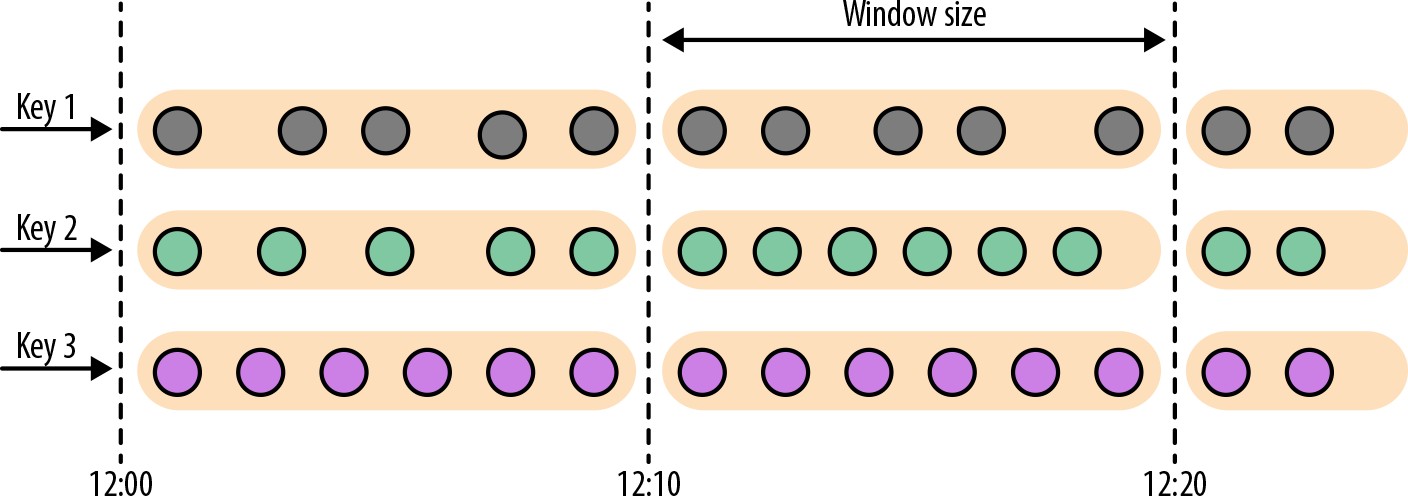
Flink’s built-in window assigners create windows of type TimeWindow. This window type essentially represents a time interval between the two timestamps, where start is

inclusive and end is exclusive. It exposes methods to retrieve the window boundaries, to check whether windows intersect, and to merge overlapping windows.

In the following, we show the different built-in window assigners of the DataStream API and how to use them to define window operators.

**Tumbling windows**

A tumbling window assigner places elements into nonoverlapping, fixed-size win‐ dows, as shown in [Figure 6-1](#_bookmark11).



*Figure 6-1. A tumbling window assigner places elements into fixed-size, nonoverlapping windows*

The Datastream API provides two assigners—TumblingEventTimeWindows and Tum blingProcessingTimeWindows—for tumbling event-time and processing-time win‐ dows, respectively. A tumbling window assigner receives one parameter, the window

size in time units; this can be specified using the of(Time size) method of the assigner. The time interval can be set in milliseconds, seconds, minutes, hours, or

days.

The following code shows how to define event-time and processing-time tumbling windows on a stream of sensor data measurements:

**val** sensorData**: DataStream**[**SensorReading**] **=** ...

**val** avgTemp **=** sensorData

.keyBy(**\_**.id)

*// group readings in 1s event-time windows*

.window(**TumblingEventTimeWindows**.of(**Time**.seconds(1)))

.process(**new TemperatureAverager**)

**val** avgTemp **=** sensorData

.keyBy(**\_**.id)

*// group readings in 1s processing-time windows*

.window(**TumblingProcessingTimeWindows**.of(**Time**.seconds(1)))

.process(**new TemperatureAverager**)

The window definition looked a bit different in our first DataStream API example, [“Operations on Data Streams” on page 22](#_bookmark34). There, we defined an event-time tumbling

window using the timeWindow(size) method, which is a shortcut for either window. (TumblingEventTimeWindows.of(size)) or for window.(TumblingProcessing TimeWindows.of(size)) depending on the configured time characteristic. The fol‐ lowing code shows how to use this shortcut:

**val** avgTemp **=** sensorData

.keyBy(**\_**.id)

*// shortcut for window.(TumblingEventTimeWindows.of(size))*

.timeWindow(**Time**.seconds(1))

.process(**new TemperatureAverager**)

By default, tumbling windows are aligned to the epoch time, 1970-01-01-00:00:00.000. For example, an assigner with a size of one hour will define windows at 00:00:00, 01:00:00, 02:00:00, and so on. Alternatively, you can spec‐

ify an offset as a second parameter in the assigner. The following code shows win‐ dows with an offset of 15 minutes that start at 00:15:00, 01:15:00, 02:15:00, and so on:

**val** avgTemp **=** sensorData

.keyBy(**\_**.id)

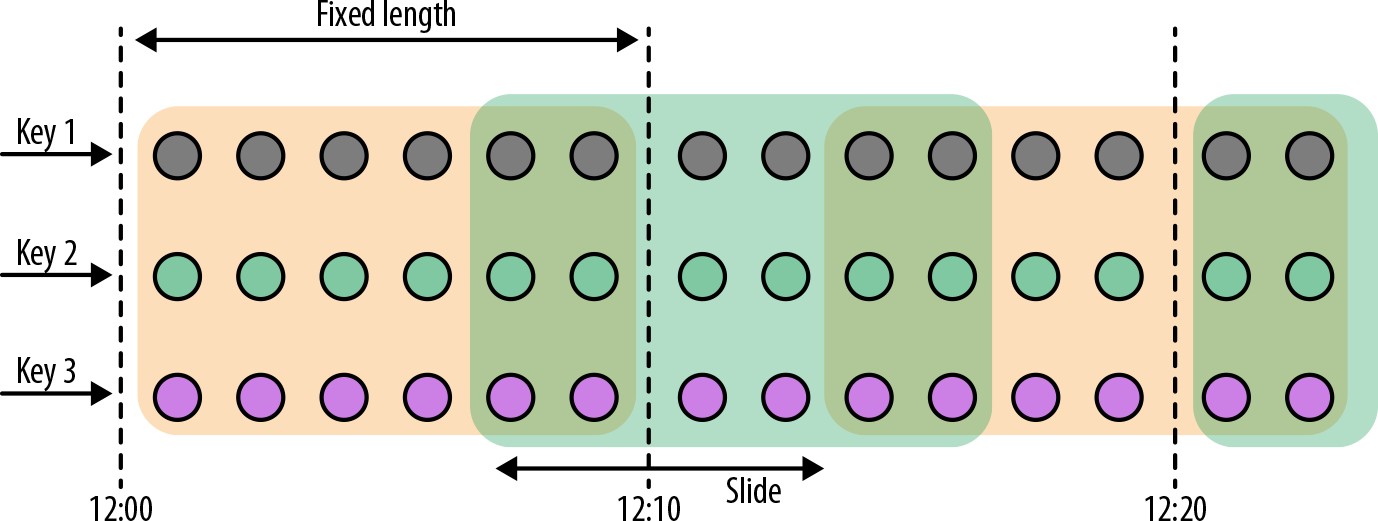
*// group readings in 1 hour windows with 15 min offset*

.window(**TumblingEventTimeWindows**.of(**Time**.hours(1), **Time**.minutes(15)))

.process(**new TemperatureAverager**)

### Sliding windows

The sliding window assigner assigns elements to fixed-sized windows that are shifted by a specified slide interval, as shown in [Figure 6-2](#_bookmark21).



*Figure 6-2. A sliding window assigner places elements into fixed-size, possibly overlap‐ ping windows*

For a sliding window, you have to specify a window size and a slide interval that defines how frequently a new window is started. When the slide interval is smaller than the window size, the windows overlap and elements can be assigned to more than one window. If the slide is larger than the window size, some elements might not be assigned to any window and hence may be dropped.

The following code shows how to group the sensor readings in sliding windows of 1 hour size with a 15-minute slide interval. Each reading will be added to four win‐ dows. The DataStream API provides event-time and processing-time assigners, as well as shortcut methods, and a time interval offset can be set as the third parameter to the window assigner:

*// event-time sliding windows assigner*

**val** slidingAvgTemp **=** sensorData

.keyBy(**\_**.id)

*// create 1h event-time windows every 15 minutes*

.window(**SlidingEventTimeWindows**.of(**Time**.hours(1), **Time**.minutes(15)))

.process(**new TemperatureAverager**)

*// processing-time sliding windows assigner*

**val** slidingAvgTemp **=** sensorData

.keyBy(**\_**.id)

*// create 1h processing-time windows every 15 minutes*

.window(**SlidingProcessingTimeWindows**.of(**Time**.hours(1), **Time**.minutes(15)))

.process(**new TemperatureAverager**)

*// sliding windows assigner using a shortcut method*

**val** slidingAvgTemp **=** sensorData

.keyBy(**\_**.id)

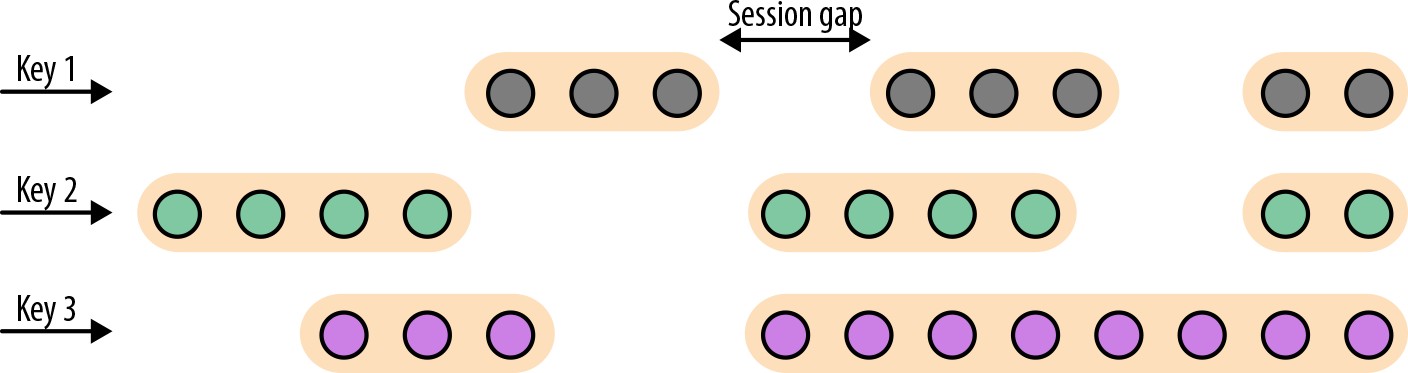
*// shortcut for window.(SlidingEventTimeWindow.of(size, slide))*

.timeWindow(**Time**.hours(1), **Time**(minutes(15)))

.process(**new TemperatureAverager**)

### Session windows

A session window assigner places elements into nonoverlapping windows of activity of varying size. The boundaries of session windows are defined by gaps of inactivity, time intervals in which no record is received. [Figure 6-3](#_bookmark25) illustrates how elements are assigned to session windows.



*Figure 6-3. A session window assigner places elements into varying size windows defined*

*by a session gap*

The following examples show how to group the sensor readings into session windows where each session is defined by a 15-minute period of inactivity:

*// event-time session windows assigner*

**val** sessionWindows **=** sensorData

.keyBy(**\_**.id)

*// create event-time session windows with a 15 min gap*

.window(**EventTimeSessionWindows**.withGap(**Time**.minutes(15)))

.process(...)

*// processing-time session windows assigner*

**val** sessionWindows **=** sensorData

.keyBy(**\_**.id)

*// create processing-time session windows with a 15 min gap*

.window(**ProcessingTimeSessionWindows**.withGap(**Time**.minutes(15)))

.process(...)

Since the start and the end of a session window depend on the received elements, a window assigner cannot immediately assign all the elements to the correct window.

Instead, the SessionWindows assigner initially maps each incoming element into its

own window with the element’s timestamp as the start time and the session gap as the

window size. Subsequently, it merges all windows with overlapping ranges.

## Applying Functions on Windows

Window functions define the computation that is performed on the elements of a window. There are two types of functions that can be applied on a window:

1. *Incremental aggregation functions* are directly applied when an element is added to a window and hold and update a single value as window state. These functions are typically very space-efficient and eventually emit the aggregated value as a

result. ReduceFunction and AggregateFunction are incremental aggregation

functions.

1. *Full window functions* collect all elements of a window and iterate over the list of all collected elements when they are evaluated. Full window functions usually require more space but allow for more complex logic than incremental aggrega‐

tion functions. The ProcessWindowFunction is a full window function.

In this section, we discuss the different types of functions that can be applied on a window to perform aggregations or arbitrary computations on the window’s con‐ tents. We also show how to jointly apply incremental aggregation and full window functions in a window operator.

### ReduceFunction

The ReduceFunction was introduced in [“KeyedStream Transformations” on page 87](#_bookmark0) when discussing running aggregations on keyed streams. A ReduceFunction accepts two values of the same type and combines them into a single value of the same type.

When applied on a windowed stream, ReduceFunction incrementally aggregates the elements that are assigned to a window. A window only stores the current result of

the aggregation—a single value of the ReduceFunction’s input (and output) type. When a new element is received, the ReduceFunction is called with the new element and the current value that is read from the window’s state. The window’s state is

replaced by the ReduceFunction’s result.

The advantages of applying ReduceFunction on a window are the constant and small state size per window and the simple function interface. However, the applications for ReduceFunction are limited and usually restricted to simple aggregations since the input and output type must be the same.

[Example 6-10](#_bookmark12) shows a reduce lambda function that computes the mininum tempera‐ ture per sensor every 15 seconds.

*Example 6-10. Applying a reduce lambda function on a WindowedStream*

**val** minTempPerWindow**: DataStream**[(**String**, **Double**)] **=** sensorData

.map(r **=>** (r.id, r.temperature))

.keyBy(**\_**.\_1)

.timeWindow(**Time**.seconds(15))

.reduce((r1, r2) **=>** (r1.\_1, r1.\_2.min(r2.\_2)))

### AggregateFunction

Similar to ReduceFunction, AggregateFunction is also incrementally applied to the elements that are applied to a window. Moreover, the state of a window operator with an AggregateFunction also consists of a single value.

While the interface of the AggregateFunction is much more flexible, it is also more complex to implement compared to the interface of the ReduceFunction. The follow‐ ing code shows the interface of the AggregateFunction:

**public interface AggregateFunction**<IN, ACC, OUT> **extends** Function, Serializable {

*// create a new accumulator to start a new aggregate.*

ACC createAccumulator();

*// add an input element to the accumulator and return the accumulator.*

ACC add(IN value, ACC accumulator);

*// compute the result from the accumulator and return it.*

OUT getResult(ACC accumulator);

*// merge two accumulators and return the result.*

ACC merge(ACC a, ACC b);

}

The interface defines an input type, IN, an accumulator of type ACC, and a result type OUT. In contrast to the ReduceFunction, the intermediate data type and the output type do not depend on the input type.

[Example 6-11](#_bookmark6) shows how to use an AggregateFunction to compute the average tem‐ perature of sensor readings per window. The accumulator maintains a running sum and count and the getResult() method computes the average value.

*Example 6-11. Applying an AggregateFunction on a WindowedStream*

**val** avgTempPerWindow**: DataStream**[(**String**, **Double**)] **=** sensorData

.map(r **=>** (r.id, r.temperature))

.keyBy(**\_**.\_1)

.timeWindow(**Time**.seconds(15))

.aggregate(**new AvgTempFunction**)

*// An AggregateFunction to compute the average tempeature per sensor.*

*// The accumulator holds the sum of temperatures and an event count.*

**class AvgTempFunction**

**extends AggregateFunction**

[(**String**, **Double**), (**String**, **Double**, **Int**), (**String**, **Double**)] {

**override def** createAccumulator() **=** { ("", 0.0, 0)

}

**override def** add(in**:** (**String**, **Double**), acc**:** (**String**, **Double**, **Int**)) **=** { (in.\_1, in.\_2 + acc.\_2, 1 + acc.\_3)

}

**override def** getResult(acc**:** (**String**, **Double**, **Int**)) **=** { (acc.\_1, acc.\_2 / acc.\_3)

}

**override def** merge(acc1**:** (**String**, **Double**, **Int**), acc2**:** (**String**, **Double**, **Int**)) **=** { (acc1.\_1, acc1.\_2 + acc2.\_2, acc1.\_3 + acc2.\_3)

}

}

### ProcessWindowFunction

ReduceFunction and AggregateFunction are incrementally applied on events that are assigned to a window. However, sometimes we need access to all elements of a

window to perform more complex computations, such as computing the median of values in a window or the most frequently occurring value. For such applications,

neither the ReduceFunction nor the AggregateFunction are suitable. Flink’s Data‐ Stream API offers the ProcessWindowFunction to perform arbitrary computations on the contents of a window.

The DataStream API of Flink 1.7 features the WindowFunction interface. WindowFunction has been superseded by ProcessWindow Function and will not be discussed here.

The following code shows the interface of the ProcessWindowFunction:

**public abstract class ProcessWindowFunction**<IN, OUT, KEY, W **extends** Window>

**extends** AbstractRichFunction {

*// Evaluates the window*

**void** process(

KEY key, Context ctx, Iterable<IN> vals, Collector<OUT> out) **throws** Exception;

*// Deletes any custom per-window state when the window is purged*

**public void** clear(Context ctx) **throws** Exception {}

*// The context holding window metadata*

**public abstract class Context implements** Serializable {

*// Returns the metadata of the window*

**public abstract** W window();

*// Returns the current processing time*

**public abstract long** currentProcessingTime();

*// Returns the current event-time watermark*

**public abstract long** currentWatermark();

*// State accessor for per-window state*

**public abstract** KeyedStateStore windowState();

*// State accessor for per-key global state*

**public abstract** KeyedStateStore globalState();

*// Emits a record to the side output identified by the OutputTag.*

**public abstract** <X> **void** output(OutputTag<X> outputTag, X value);

}

}

The process() method is called with the key of the window, an Iterable to access the elements of the window, and a Collector to emit results. Moreover, the method has a Context parameter similar to other process methods. The Context object of the ProcessWindowFunction gives access to the metadata of the window, the current pro‐

cessing time and watermark, state stores to manage per-window and per-key global

states, and side outputs to emit records.

We already discussed some of the features of the Context object when introducing the process functions, such as access to the current processing and event-time and

side outputs. However, ProcessWindowFunction’s Context object also offers unique features. The metadata of the window typically contains information that can be used

as an identifier for a window, such as the start and end timestamps in the case of a time window.

Another feature are per-window and per-key global states. Global state refers to the keyed state that is not scoped to any window, while per-window state refers to the window instance that is currently being evaluated. Per-window state is useful to

maintain information that should be shared between multiple invocations of the pro cess() method on the same window, which can happen due to configuring allowed lateness or using a custom trigger. A ProcessWindowFunction that utilizes per- window state needs to implement its clear() method to clean up any window-

specific state before the window is purged. Global state can be used to share

information between multiple windows on the same key.

[Example 6-12](#_bookmark22) groups the sensor reading stream into tumbling windows of 5 seconds and uses a ProcessWindowFunction to compute the lowest and highest temperature that occur within the window. It emits one record for each window consisting of the

window’s start and end timestamp and the minimum and maximum temperature.

*Example 6-12. Computing the minimum and maximum temperature per sensor and window using a ProcessWindowFunction*

*// output the lowest and highest temperature reading every 5 seconds*

**val** minMaxTempPerWindow**: DataStream**[**MinMaxTemp**] **=** sensorData

.keyBy(**\_**.id)

.timeWindow(**Time**.seconds(5))

.process(**new HighAndLowTempProcessFunction**)

**case class MinMaxTemp**(id**: String**, min**: Double**, max**:Double**, endTs**: Long**)

*/\*\**

* *A ProcessWindowFunction that computes the lowest and highest temperature*
* *reading per window and emits them together with the*
* *end timestamp of the window.*

*\*/*

**class HighAndLowTempProcessFunction**

**extends ProcessWindowFunction**[**SensorReading**, **MinMaxTemp**, **String**, **TimeWindow**] {

**override def** process( key**: String**,

ctx**: Context**,

vals**: Iterable**[**SensorReading**],

out**: Collector**[**MinMaxTemp**])**: Unit** = {

**val** temps **=** vals.map(**\_**.temperature)

**val** windowEnd **=** ctx.window.getEnd

out.collect(**MinMaxTemp**(key, temps.min, temps.max, windowEnd))

}

}

Internally, a window that is evaluated by ProcessWindowFunction stores all assigned events in a ListState.[1](#_bookmark33) By collecting all events and providing access to window meta‐ data and other features, ProcessWindowFunction can address many more use cases than ReduceFunction or AggregateFunction. However, the state of a window that

collects all events can become significantly larger than the state of a window whose

elements are incrementally aggregated.

### Incremental aggregation and ProcessWindowFunction

ProcessWindowFunction is a very powerful window function, but you need to use it with caution since it typically holds more data in state than incrementally aggregating

functions. Quite often the logic that needs to be applied on a window can be expressed as an incremental aggregation, but it also needs access to window metadata or state.

If you have incremental aggregation logic but also need access to window metadata, you can combine a ReduceFunction or AggregateFunction, which perform incre‐ mental aggregation, with a ProcessWindowFunction, which provides access to more

functionality. Elements that are assigned to a window will be immediately aggregated

and when the trigger of the window fires, the aggregated result will be handed to ProcessWindowFunction. The Iterable parameter of the ProcessWindowFunc tion.process() method will only provide a single value, the incrementally aggrega‐

ted result.

In the DataStream API this is done by providing a ProcessWindowFunction as a sec‐ ond parameter to the reduce() or aggregate() methods as shown in the following code:

input

.keyBy(...)

.timeWindow(...)

.reduce(

incrAggregator**: ReduceFunction**[**IN**],

function**: ProcessWindowFunction**[**IN**, **OUT**, **K**, **W**])

input

.keyBy(...)

.timeWindow(...)

.aggregate(

1 ListState and its performance characteristics are discussed in detail in [Chapter 7](#_bookmark1).

incrAggregator**: AggregateFunction**[**IN**, **ACC**, **V**], windowFunction**: ProcessWindowFunction**[**V**, **OUT**, **K**, **W**])

The code in Examples [6-13](#_bookmark13) and [6-14](#_bookmark26) shows how to solve the same use case as the code in [Example 6-12](#_bookmark22) with a combination of a ReduceFunction and a Pro cessWindowFunction, emitting every 5 seconds the minimun and maximum temper‐

ature per sensor and the end timestamp of each window.

*Example 6-13. Applying a ReduceFunction for incremental aggregation and a ProcessWindowFunction for finalizing the window result*

**case class MinMaxTemp**(id**: String**, min**: Double**, max**:Double**, endTs**: Long**) **val** minMaxTempPerWindow2**: DataStream**[**MinMaxTemp**] **=** sensorData

.map(r **=>** (r.id, r.temperature, r.temperature))

.keyBy(**\_**.\_1)

.timeWindow(**Time**.seconds(5))

.reduce(

*// incrementally compute min and max temperature*

(r1**:** (**String**, **Double**, **Double**), r2**:** (**String**, **Double**, **Double**)) **=>** { (r1.\_1, r1.\_2.min(r2.\_2), r1.\_3.max(r2.\_3))

},

*// finalize result in ProcessWindowFunction*

**new AssignWindowEndProcessFunction**()

)

As you can see in [Example 6-13](#_bookmark13), the ReduceFunction and ProcessWindowFunction are both defined in the reduce() method call. Since the aggregation is performed by the ReduceFunction, the ProcessWindowFunction only needs to append the window end timestamp to the incrementally computed result as shown in [Example 6-14](#_bookmark26).

*Example 6-14. Implementation of a ProcessWindowFunction that assigns the window end timestamp to an incrementally computed result*

**class AssignWindowEndProcessFunction extends**

**ProcessWindowFunction**[(**String**, **Double**, **Double**), **MinMaxTemp**, **String**, **TimeWindow**] {

**override def** process( key**: String**,

ctx**: Context**,

minMaxIt**: Iterable**[(**String**, **Double**, **Double**)], out**: Collector**[**MinMaxTemp**])**: Unit** = {

**val** minMax **=** minMaxIt.head

**val** windowEnd **=** ctx.window.getEnd

out.collect(**MinMaxTemp**(key, minMax.\_2, minMax.\_3, windowEnd))

}

}

## Customizing Window Operators

Window operators defined using Flink’s built-in window assigners can address many common use cases. However, as you start writing more advanced streaming applica‐ tions, you might find yourself needing to implement more complex windowing logic, such as windows that emit early results and update their results if late elements are encountered, or windows that start and end when specific records are received.

The DataStream API exposes interfaces and methods to define custom window oper‐ ators by allowing you to implement your own assigners, triggers, and evictors. Along with the previously discussed window functions, these components work together in a window operator to group and process elements in windows.

When an element arrives at a window operator, it is handed to the WindowAssigner. The assigner determines to which windows the element needs to be routed. If a win‐

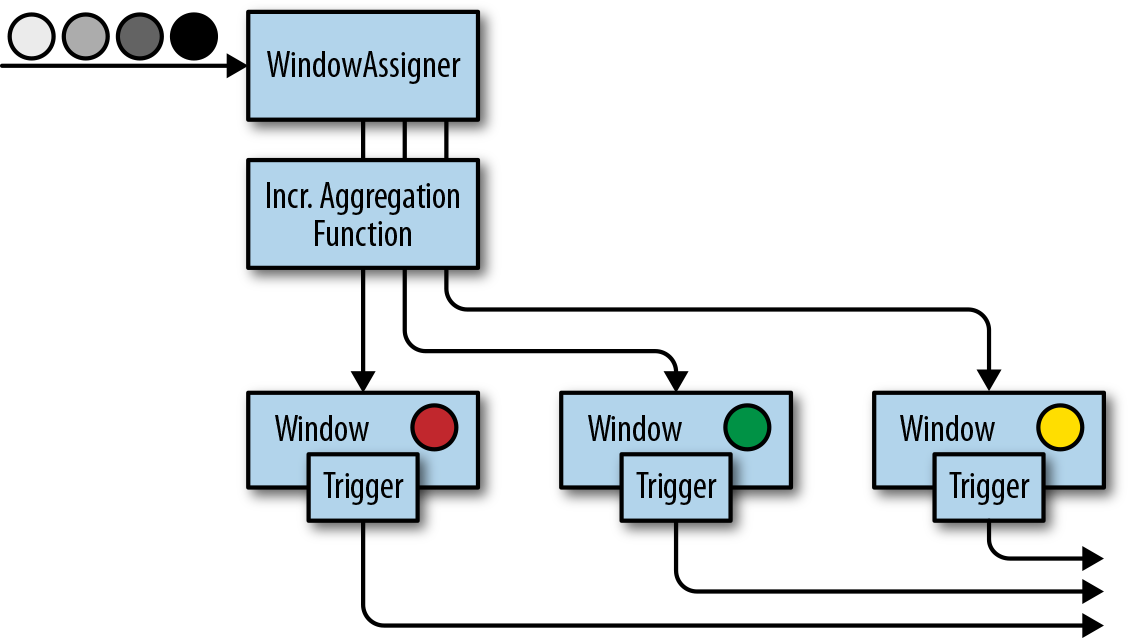
dow does not exist yet, it is created.

If the window operator is configured with an incremental aggregation function, such as a ReduceFunction or AggregateFunction, the newly added element is immediately aggregated and the result is stored as the contents of the window. If the window oper‐

ator does not have an incremental aggregation function, the new element is appended to a ListState that holds all assigned elements.

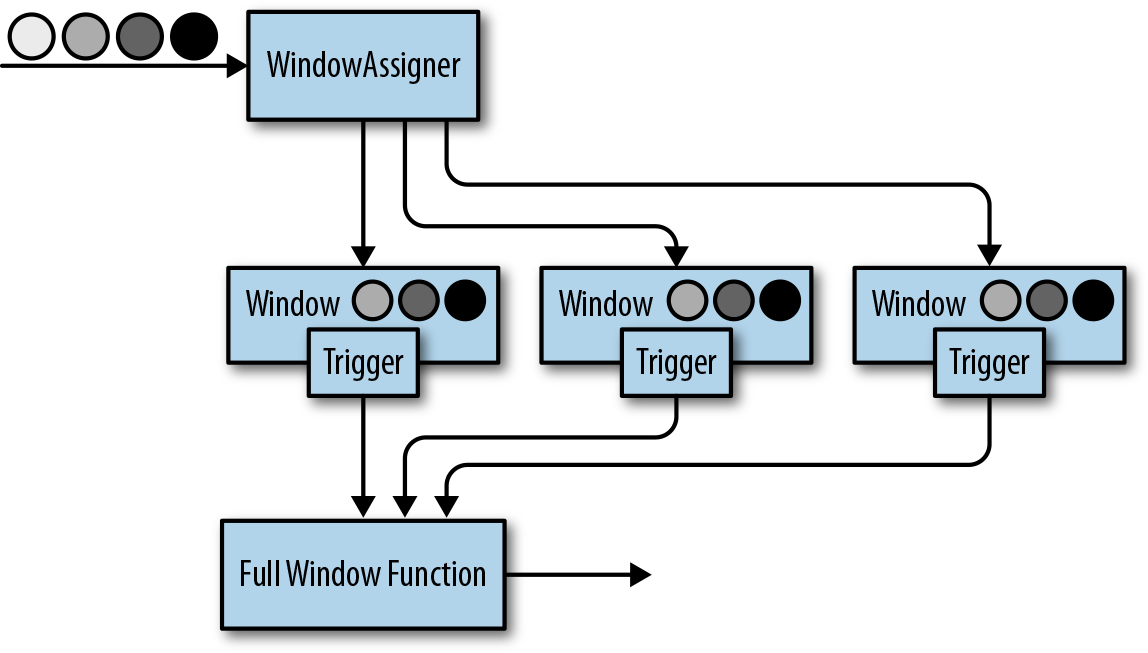
Every time an element is added to a window, it is also passed to the trigger of the win‐ dow. The trigger defines (fires) when a window is considered ready for evaluation and when a window is purged and its contents are cleared. A trigger can decide based on assigned elements or registered timers (similar to a process function) to evaluate or purge the contents of its window at specific points in time.

What happens when a trigger fires depends on the configured functions of the win‐ dow operator. If the operator is configured just with an incremental aggregation func‐ tion, the current aggregation result is emitted. This case is shown in [Figure 6-4](#_bookmark0).



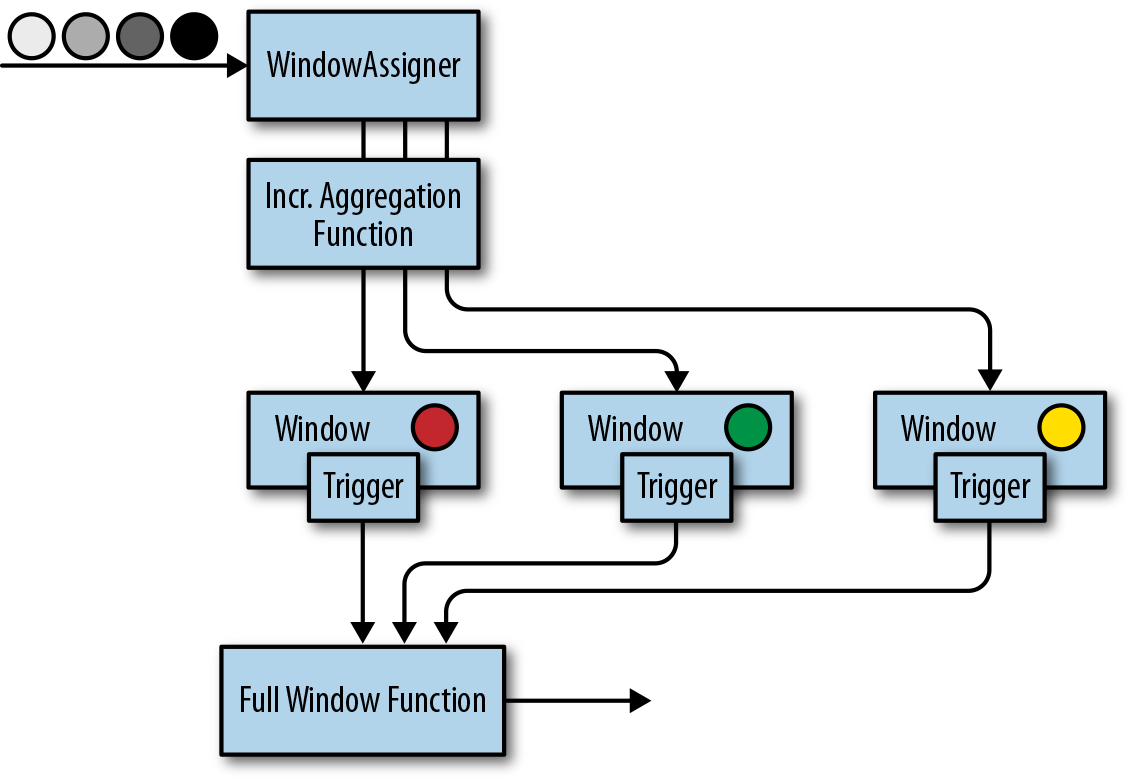
*Figure 6-4. A window operator with an incremental aggregation function (the single cir‐ cle in each window represents its aggregated window state)*

If the operator only has a full window function, the function is applied on all ele‐ ments of the window and the result is emitted as shown in [Figure 6-5](#_bookmark23).



*Figure 6-5. A window operator with a full window function (the circles in each window represent its collected raw input records)*

Finally, if the operator has an incremental aggregation function and a full window function, the full window function is applied on the aggregated value and the result is emitted. [Figure 6-6](#_bookmark0) depicts this case.



*Figure 6-6. A window operator with an incremental aggregation and full window func‐ tion (the single circle in each window represents its aggregated window state)*

The evictor is an optional component that can be injected before or after a Process WindowFunction is called. An evictor can remove collected elements from the con‐ tents of a window. Since it has to iterate over all elements, it can only be used if no

incremental aggregation function is specified.

The following code shows how to define a window operator with a custom trigger and evictor:

stream

.keyBy(...)

.window(...) *// specify the window assigner*

[**.trigger**(**...**)] *// optional: specify the trigger*

[**.evictor**(**...**)] *// optional: specify the evictor*

.reduce/aggregate/process(...) *// specify the window function*

While evictors are optional components, each window operator needs a trigger to decide when to evaluate its windows. In order to provide a concise window operator

API, each WindowAssigner has a default trigger that is used unless an explicit trigger

is defined.

Note that an explicitly specified trigger overrides the existing trig‐ ger and does not complement it—the window will only be evalu‐ ated based on the trigger that was last defined.

In the following sections, we discuss the lifecycle of windows and introduce the inter‐ faces to define custom window assigners, triggers, and evictors.

### Window lifecycle

A window operator creates and typically also deletes windows while it processes incoming stream elements. As discussed before, elements are assigned to windows by

a WindowAssigner, a trigger decides when to evalute a window, and a window func‐

tion performs the actual window evaluation. In this section, we discuss the lifecycle of

a window—when it is created, what information it consists of, and when it is deleted.

A window is created when the WindowAssigner assigns the first element to it. Conse‐ quently, there is no window without at least one element. A window consists of

different pieces of state as follows:

*Window content*

The window content holds the elements that have been assigned to the window or the result of the incremental aggregation in case the window operator has a

ReduceFunction or AggregateFunction.

*Window object*

The WindowAssigner returns zero, one, or multiple window objects. The window operator groups elements based on the returned objects. Hence, a window object

holds the information used to distinguish windows from each other. Each win‐ dow object has an end timestamp that defines the point in time after which the window and its state can be deleted.

*Timers of a trigger*

A trigger can register timers to be called back at certain points in time—for example, to evaluate a window or purge its contents. These timers are maintained by the window operator.

*Custom-defined state in a trigger*

A trigger can define and use custom, per-window and per-key state. This state is completely controlled by the trigger and not maintained by the window operator.

The window operator deletes a window when the end time of the window, defined by the end timestamp of the window object, is reached. Whether this happens with

processing-time or event-time semantics depends on the value returned by the Win dowAssigner.isEventTime() method.

When a window is deleted, the window operator automatically clears the window content and discards the window object. Custom-defined trigger state and registered trigger timers are not cleared because this state is opaque to the window operator.

Hence, a trigger must clear all of its state in the Trigger.clear() method to prevent

leaking state.

### Window assigners

The WindowAssigner determines for each arriving element to which windows it is assigned. An element can be added to zero, one, or multiple windows. The following shows the WindowAssigner interface:

**public abstract class WindowAssigner**<T, W **extends** Window>

**implements** Serializable {

*// Returns a collection of windows to which the element is assigned*

**public abstract** Collection<W> assignWindows( T element,

**long** timestamp, WindowAssignerContext context);

*// Returns the default Trigger of the WindowAssigner* **public abstract** Trigger<T, W> getDefaultTrigger( StreamExecutionEnvironment env);

*// Returns the TypeSerializer for the windows of this WindowAssigner* **public abstract** TypeSerializer<W> getWindowSerializer( ExecutionConfig executionConfig);

*// Indicates whether this assigner creates event-time windows*

**public abstract boolean** isEventTime();

*// A context that gives access to the current processing time*

**public abstract static class WindowAssignerContext** {

*// Returns the current processing time*

**public abstract long** getCurrentProcessingTime();

}

}

A WindowAssigner is typed to the type of the incoming elements and the type of the windows to which the elements are assigned. It also needs to provide a default trigger

that is used if no explicit trigger is specified. The code in [Example 6-15](#_bookmark29) creates a cus‐ tom assigner for 30-second tumbling event-time windows.

*Example 6-15. A window assigner for tumbling event-time windows*

*/\*\* A custom window that groups events into 30-second tumbling windows. \*/*

**class ThirtySecondsWindows**

**extends WindowAssigner**[**Object**, **TimeWindow**] {

**val** windowSize**: Long** = 30 \* 1000L

**override def** assignWindows( o**: Object**,

ts**: Long**,

ctx**: WindowAssigner.WindowAssignerContext**)**: java.util.List**[**TimeWindow**] **=** {

*// rounding down by 30 seconds*

**val** startTime **=** ts - (ts % windowSize)

**val** endTime **=** startTime + windowSize

*// emitting the corresponding time window*

**Collections**.singletonList(**new TimeWindow**(startTime, endTime))

}

**override def** getDefaultTrigger(

env**: environment.StreamExecutionEnvironment**)**: Trigger**[**Object**, **TimeWindow**] **=** {

**EventTimeTrigger**.create()

}

**override def** getWindowSerializer(

executionConfig**: ExecutionConfig**)**: TypeSerializer**[**TimeWindow**] **=** {

**new TimeWindow**.**Serializer**

}

**override def** isEventTime **= true**

}

#### The GlobalWindows Assigner

The GlobalWindows assigner maps all elements to the same global window. Its default trigger is the NeverTrigger that, as the name suggests, never fires. Consequently, the GlobalWindows assigner requires a custom trigger and possibly an evictor to selectively

remove elements from the window state.

The end timestamp of GlobalWindows is Long.MAX\_VALUE. Conse‐ quently, GlobalWindows will never be completely cleaned up. When applied on a KeyedStream with an evolving key space, Global Windows will maintain some state for each key. It should only be

used with care.

In addition to the WindowAssigner interface there is also the MergingWindowAssigner

interface that extends WindowAssigner. The MergingWindowAssigner is used for win‐ dow operators that need to merge existing windows. One example for such an assigner is the EventTimeSessionWindows assigner we discussed before, which works by creating a new window for each arriving element and merging overlapping win‐

dows afterward.

When merging windows, you need to ensure that the state of all merging windows and their triggers is also appropriately merged. The Trigger interface features a call‐ back method that is invoked when windows are merged to merge state that is associ‐

ated with the windows. Merging of windows is discussed in more detail in the next section.

### Triggers

Triggers define when a window is evaluated and its results are emitted. A trigger can decide to fire based on progress in time- or data-specific conditions, such as element count or certain observed element values. For example, the default triggers of the pre‐ viously discussed time windows fire when the processing time or the watermark exceed the timestamp of the window’s end boundary.

Triggers have access to time properties and timers, and can work with state. Hence, they are as powerful as process functions. For example, you can implement triggering logic to fire when the window receives a certain number of elements, when an ele‐ ment with a specific value is added to the window, or after detecting a pattern on added elements like “two events of the same type within 5 seconds.” A custom trigger can also be used to compute and emit early results from an event-time window, before the watermark reaches the window’s end timestamp. This is a common strat‐ egy to produce (incomplete) low-latency results despite using a conservative water‐ marking strategy.

Every time a trigger is called it produces a TriggerResult that determines what should happen to the window. TriggerResult can take one of the following values:

CONTINUE

No action is taken.

FIRE

If the window operator has a ProcessWindowFunction, the function is called and the result is emitted. If the window only has an incremetal aggregation function (ReduceFunction or AggregateFunction) the current aggregation result is emit‐ ted. The state of the window is not changed.

PURGE

The content of the window is completely discarded and the window including all metadata is removed. Also, the ProcessWindowFunction.clear() method is invoked to clean up all custom per-window state.

FIRE\_AND\_PURGE

FIRE\_AND\_PURGE: Evaluates the window first (FIRE) and subsequently removes all state and metadata (PURGE).

The possible TriggerResult values enable you to implement sophisticated window‐ ing logic. A custom trigger may fire several times, computing new or updated results

or purging a window without emitting a result if a certain condition is fulfilled. The next block of code shows the Trigger API:

public **abstract class Trigger**<T, W **extends Window**> implements **Serializable** {

*// Called for every element that gets added to a window*

**TriggerResult** onElement(

T element, long timestamp, W window, **TriggerContext** ctx);

*// Called when a processing-time timer fires* public **abstract TriggerResult** onProcessingTime( long timestamp, W window, **TriggerContext** ctx);

*// Called when an event-time timer fires*

public **abstract TriggerResult** onEventTime(

long timestamp, W window, **TriggerContext** ctx);

*// Returns true if this trigger supports merging of trigger state*

public boolean canMerge();

*// Called when several windows have been merged into one window*

*// and the state of the triggers needs to be merged*

public void onMerge(W window, **OnMergeContext** ctx);

*// Clears any state that the trigger might hold for the given window*

*// This method is called when a window is purged*

public **abstract** void clear(W window, **TriggerContext** ctx);

}

*// A context object that is given to Trigger methods to allow them*

*// to register timer callbacks and deal with state*

public interface **TriggerContext** {

*// Returns the current processing time*

long getCurrentProcessingTime();

*// Returns the current watermark time*

long getCurrentWatermark();

*// Registers a processing-time timer*

void registerProcessingTimeTimer(long time);

*// Registers an event-time timer*

void registerEventTimeTimer(long time);

*// Deletes a processing-time timer*

void deleteProcessingTimeTimer(long time);

*// Deletes an event-time timer*

void deleteEventTimeTimer(long time);

*// Retrieves a state object that is scoped to the window and the key of the trigger*

<S **extends State**> S getPartitionedState(**StateDescriptor**<S, ?> stateDescriptor);

}

*// Extension of TriggerContext that is given to the Trigger.onMerge() method*

public interface **OnMergeContext extends TriggerContext** {

*// Merges per-window state of the trigger*

*// The state to be merged must support merging*

void mergePartitionedState(**StateDescriptor**<S, ?> stateDescriptor);

}

As you can see, the Trigger API can be used to implement sophisticated logic by pro‐ viding access to time and state. There are two aspects of triggers that require special care: cleaning up state and merging triggers.

When using per-window state in a trigger, you need to ensure that this state is prop‐ erly deleted when the window is deleted. Otherwise, the window operator will accu‐ mulate more and more state over time and your application will probably fail at some

point. In order to clean up all state when a window is deleted, the clear() method of

a trigger needs to remove all custom per-window state and delete all processing-time and event-time timers using the TriggerContext object. It is not possible to clean up state in a timer callback method, since these methods are not called after a window is

deleted.

If a trigger is applied together with a MergingWindowAssigner, it needs to be able to handle the case when two windows are merged. In this case, any custom states of the triggers also need to be merged. canMerge() declares that a trigger supports merging

and the onMerge() method needs to implement the logic to perform the merge. If a

trigger does not support merging it cannot be used in combination with a Merging WindowAssigner.

When triggers are merged, all descriptors of custom states must be provided to the

mergePartitionedState() method of the OnMergeContext object.

Note that mergable triggers may only use state primitives that can be automatically merged—ListState, ReduceState, or Aggrega tingState.

[Example 6-16](#_bookmark30) shows a trigger that fires early, before the end time of the window is reached. The trigger registers a timer when the first event is assigned to a window, 1 second ahead of the current watermark. When the timer fires, a new timer is regis‐ tered. Therefore, the trigger fires, at most, every second.

*Example 6-16. An early firing trigger*

*/\*\* A trigger that fires early. The trigger fires at most every second. \*/*

**class OneSecondIntervalTrigger**

**extends Trigger**[**SensorReading**, **TimeWindow**] {

**override def** onElement( r**: SensorReading**, timestamp**: Long**,

window**: TimeWindow**,

ctx**: Trigger.TriggerContext**)**: TriggerResult** = {

*// firstSeen will be false if not set yet*

**val** firstSeen**: ValueState**[**Boolean**] **=** ctx.getPartitionedState(

**new ValueStateDescriptor**[**Boolean**]("firstSeen", classOf[**Boolean**]))

*// register initial timer only for first element*

**if** (!firstSeen.value()) {

*// compute time for next early firing by rounding watermark to second*

**val** t **=** ctx.getCurrentWatermark + (1000 - (ctx.getCurrentWatermark % 1000)) ctx.registerEventTimeTimer(t)

*// register timer for the window end* ctx.registerEventTimeTimer(window.getEnd) firstSeen.update(**true**)

}

*// Continue. Do not evaluate per element*

**TriggerResult**.**CONTINUE**

}

**override def** onEventTime( timestamp**: Long**, window**: TimeWindow**,

ctx**: Trigger.TriggerContext**)**: TriggerResult** = {

**if** (timestamp == window.getEnd) {

*// final evaluation and purge window state*

**TriggerResult**.**FIRE\_AND\_PURGE**

} **else** {

*// register next early firing timer*

**val** t **=** ctx.getCurrentWatermark + (1000 - (ctx.getCurrentWatermark % 1000))

**if** (t < window.getEnd) { ctx.registerEventTimeTimer(t)

}

*// fire trigger to evaluate window*

**TriggerResult**.**FIRE**

}

}

**override def** onProcessingTime( timestamp**: Long**,

window**: TimeWindow**,

ctx**: Trigger.TriggerContext**)**: TriggerResult** = {

*// Continue. We don't use processing time timers*

**TriggerResult**.**CONTINUE**

}

**override def** clear( window**: TimeWindow**,

ctx**: Trigger.TriggerContext**)**: Unit** = {

*// clear trigger state*

**val** firstSeen**: ValueState**[**Boolean**] **=** ctx.getPartitionedState(

**new ValueStateDescriptor**[**Boolean**]("firstSeen", classOf[**Boolean**])) firstSeen.clear()

}

}

Note that the trigger uses custom state, which is cleaned up using the clear() method. Since we are using a simple nonmergable ValueState, the trigger is not mer‐ gable.

### Evictors

The Evictor is an optional component in Flink’s windowing mechanism. It can remove elements from a window before or after the window function is evaluated.

[Example 6-17](#_bookmark14) shows the Evictor interface.

*Example 6-17. The Evictor interface*

public interface **Evictor**<T, W **extends Window**> **extends Serializable** {

*// Optionally evicts elements. Called before windowing function.*

void evictBefore( **Iterable**<**TimestampedValue**<T>> elements, int size,

W window,

**EvictorContext** evictorContext);

*// Optionally evicts elements. Called after windowing function.*

void evictAfter( **Iterable**<**TimestampedValue**<T>> elements, int size,

W window,

**EvictorContext** evictorContext);

*// A context object that is given to Evictor methods.*

interface **EvictorContext** {

*// Returns the current processing time.*

long getCurrentProcessingTime();

*// Returns the current event time watermark.*

long getCurrentWatermark();

}

The evictBefore() and evictAfter() methods are called before and after a window function is applied on the content of a window, respectively. Both methods are called with an Iterable that serves all elements that were added to the window, the number

of elements in the window (size), the window object, and an EvictorContext that

provides access to the current processing time and watermark. Elements are removed

from a window by calling the remove() method on the Iterator that can be obtained from the Iterable.

#### Preaggregation and Evictors

Evictors iterate over a list of elements in a window. They can only be applied if the window collects all added events and does not

apply a ReduceFunction or AggregateFunction to incrementally

aggregate the window content.

Evictors are often applied on a GlobalWindow for partial cleaning of the window— without purging the complete window state.

# Joining Streams on Time

A common requirement when working with streams is to connect or join the events of two streams. Flink’s DataStream API features two built-in operators to join streams with a temporal condition: the interval join and the window join. In this section, we describe both operators.

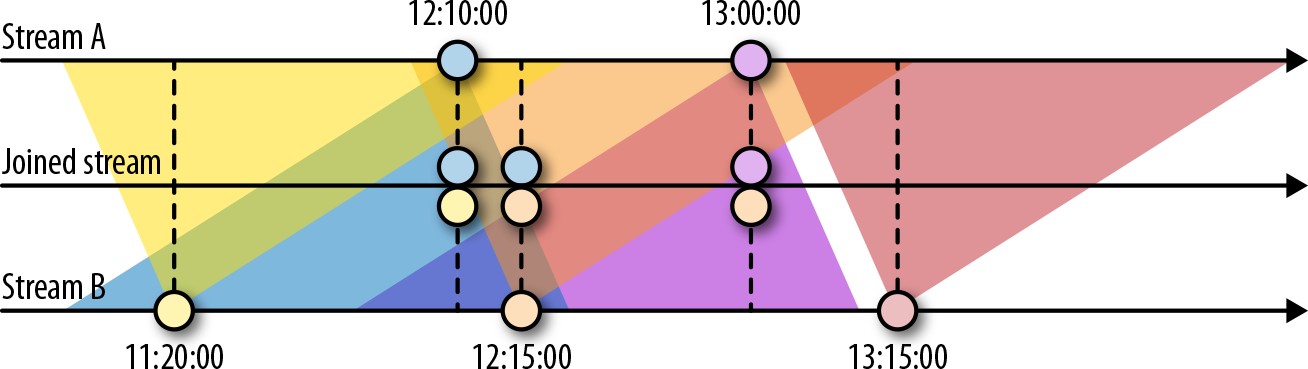
If you cannot express your required join semantics using Flink’s built-in join opera‐ tors, you can implement custom join logic as a CoProcessFunction, BroadcastPro cessFunction, or KeyedBroadcastProcessFunction.

Note that you should design such an operator with efficient state access patterns and effective state cleanup strategies.

## Interval Join

The interval join joins events from two streams that have a common key and that have timestamps not more than specified intervals apart from each other.

[Figure 6-7](#_bookmark0) shows an interval join of two streams, A and B, that joins an event from A with an event from B if the timestamp of the B event is not less than one hour earlier and not more than 15 minutes later than the timestamp of the A event. The join interval is symmetric, i.e., an event from B joins with all events from A that are no more than 15 minutes earlier and at most one hour later than the B event.



*Figure 6-7. An interval join joining two streams A and B*

The interval join currently only supports event time and operates with INNER JOIN

semantics (events that have no matching event will not be forwarded). An interval

join is defined as shown in [Example 6-18](#_bookmark15).

*Example 6-18. Using the interval join*

input1

.keyBy(…)

.between(<lower-bound>, <upper-bound>) *// bounds with respect to input1*

.process(**ProcessJoinFunction**) *// process pairs of matched events*

Pairs of joined events are passed into a ProcessJoinFunction. The lower and upper bounds are defined as negative and positive time intervals, for example, as between(Time.hour(-1), Time.minute(15)). The lower and upper bounds can be arbitrarily chosen as long as the lower bound is smaller than the upper bound; you

can join all A events with all B events that have timestamps between one and two hours more than the A event.

An interval join needs to buffer records from one or both inputs. For the first input, all records with timestamps larger than the current watermark—the upper bound— are buffered. For the second input, all records with timestamps larger than the current watermark + the lower bound are buffered. Note that both bounds may be negative. The join in [Figure 6-7](#_bookmark0) stores all records with timestamps larger than the current watermark—15 minutes from stream A—and all records with timestamps larger than the current watermark—one hour from stream B. You should be aware that the storage requirements of the interval join may significantly increase if the event time of both input streams is not synchronized because the watermark is deter‐ mined by the “slower” stream.

## Window Join

As the name suggests, the window join is based on Flink’s windowing mechanism. Elements of both input streams are assigned to common windows and joined (or cogrouped) when a window is complete.

[Example 6-19](#_bookmark2) shows how to define a window join.

*Example 6-19. Joining two windowed streams*

input1.join(input2)

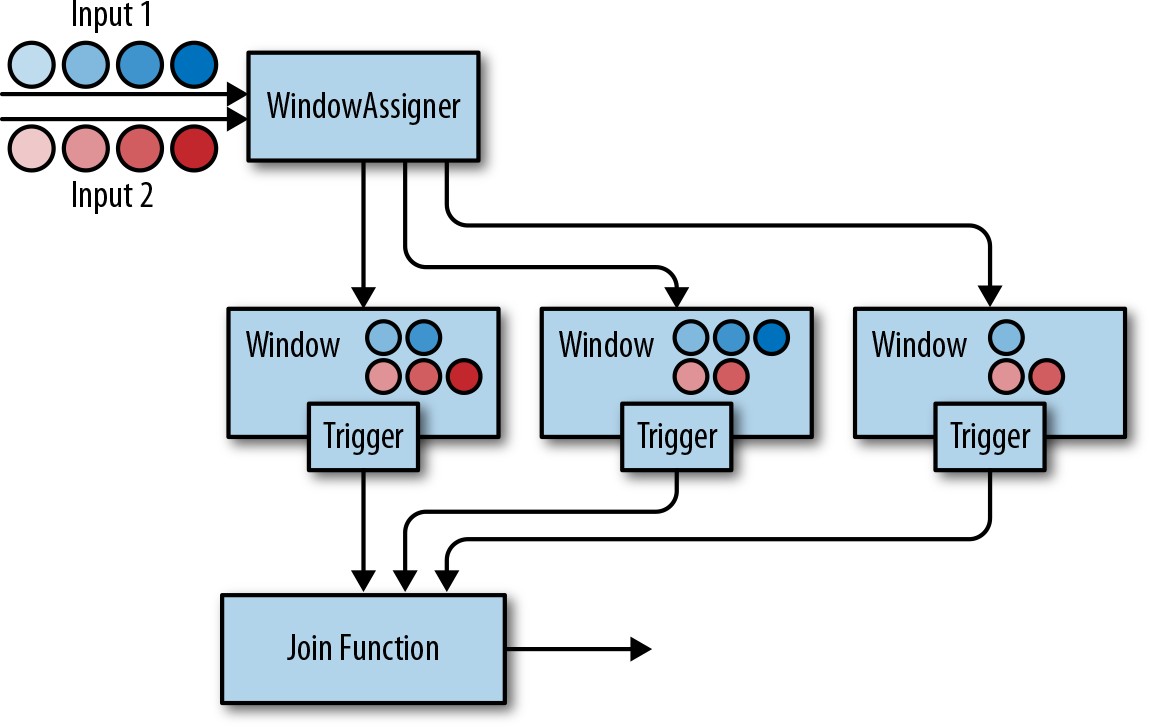
.where(...) *// specify key attributes for input1*

.equalTo(...) *// specify key attributes for input2*

.window(...) *// specify the WindowAssigner* [**.trigger**(**...**)] *// optional: specify a Trigger* [**.evictor**(**...**)] *// optional: specify an Evictor*

.apply(...) *// specify the JoinFunction*

[Figure 6-8](#_bookmark16) shows how the window join of the DataStream API works.



*Figure 6-8. Operation of a window join*

Both input streams are keyed on their key attributes and the common window assigner maps events of both streams to common windows, meaning a window stores

the events of both inputs. When the timer of a window fires, the JoinFunction is

called for each combination of elements from the first and the second input—the

cross-product. It is also possible to specify a custom trigger and evictor. Since the events of both streams are mapped into the same windows, triggers and evictors behave exactly as in regular window operators.

In addition to joining two streams, it is also possible to cogroup two streams on a window by starting the operator definition with coGroup() instead of join(). The overall logic is the same, but instead of calling a JoinFunction for every pair of events from both inputs, a CoGroupFunction is called once per window with iterators over the elements from both inputs.

It should be noted that joining windowed streams can have unex‐ pected semantics. For instance, assume you join two streams with a join operator that is configured with 1-hour tumbling window. An element of the first input will not be joined with an element of the second input even if they are just 1 second apart from each other but assigned to two different windows.

# Handling Late Data

As discussed, watermarks can be used to balance result completeness and result latency. Unless you opt for a very conservative watermark strategy that guarantees that all relevant records will be included at the cost of high latency, your application will most likely have to handle late elements.

A late element is an element that arrives at an operator when a computation to which it would need to contribute has already been performed. In the context of an event- time window operator, an event is late if it arrives at the operator and the window assigner maps it to a window that has already been computed because the operator’s watermark passed the end timestamp of the window.

The DataStream API provides different options for how to handle late events:

* Late events can be simply dropped.
* Late events can be redirected into a separate stream.
* Computation results can be updated based on late events and updates have to be emitted.

In the following, we discuss these options in detail and show how they are applied for process functions and window operators.

## Dropping Late Events

The easiest way to handle late events is to simply discard them. Dropping late events is the default behavior for event-time window operators. Hence, a late arriving ele‐ ment will not create a new window.

A process function can easily filter out late events by comparing their timestamps with the current watermark.

## Redirecting Late Events

Late events can also be redirected into another DataStream using the side-output fea‐ ture. From there, the late events can be processed or emitted using a regular sink

function. Depending on the business requirements, late data can later be integrated into the results of the streaming application with a periodic backfill process.

[Example 6-20](#_bookmark3) shows how to specify a window operator with a side output for late events.

*Example 6-20. Defining a window operator with a side output for late events*

**val** readings**: DataStream**[**SensorReading**] **=** ???

**val** countPer10Secs**: DataStream**[(**String**, **Long**, **Int**)] **=** readings

.keyBy(**\_**.id)

.timeWindow(**Time**.seconds(10))

*// emit late readings to a side output*

.sideOutputLateData(**new OutputTag**[**SensorReading**]("late-readings"))

*// count readings per window*

.process(**new CountFunction**())

*// retrieve the late events from the side output as a stream*

**val** lateStream**: DataStream**[**SensorReading**] **=** countPer10Secs

.getSideOutput(**new OutputTag**[**SensorReading**]("late-readings"))

A process function can identify late events by comparing event timestamps with the current watermark and emitting them using the regular side-output API.

[Example 6-21](#_bookmark24) shows a ProcessFunction that filters out late sensor readings from its

input and redirects them to a side-output stream.

*Example 6-21. A ProcessFunction that filters out late sensor readings and redirects them to a side output*

**val** readings**: DataStream**[**SensorReading**] **=** ???

**val** filteredReadings**: DataStream**[**SensorReading**] **=** readings

.process(**new LateReadingsFilter**)

*// retrieve late readings*

**val** lateReadings**: DataStream**[**SensorReading**] **=** filteredReadings

.getSideOutput(**new OutputTag**[**SensorReading**]("late-readings"))

*/\*\* A ProcessFunction that filters out late sensor readings and*

*\* re-directs them to a side output \*/*

**class LateReadingsFilter**

**extends ProcessFunction**[**SensorReading**, **SensorReading**] {

**val** lateReadingsOut **= new OutputTag**[**SensorReading**]("late-readings") **override def** processElement(

r**: SensorReading**,

ctx**: ProcessFunction**[**SensorReading**, **SensorReading**]**#Context**, out**: Collector**[**SensorReading**])**: Unit** = {

*// compare record timestamp with current watermark*

**if** (r.timestamp < ctx.timerService().currentWatermark()) {

*// this is a late reading => redirect it to the side output*

ctx.output(lateReadingsOut, r)

} **else** { out.collect(r)

}

}

}

## Updating Results by Including Late Events

Late events arrive at an operator after a computation to which they should have con‐ tributed was completed. Therefore, the operator emits a result that is incomplete or inaccurate. Instead of dropping or redirecting late events, another strategy is to recompute an incomplete result and emit an update. However, there are a few issues that need to be taken into account in order to be able to recompute and update results.

An operator that supports recomputing and updating of emitted results needs to pre‐ serve all state required for the computation after the first result was emitted. How‐ ever, since it is typically not possible for an operator to retain all state forever, it needs to purge state at some point. Once the state for a certain result has been purged, the result cannot be updated anymore and late events can only be dropped or redirected.

In addition to keeping state around, the downstream operators or external systems that follow an operator, which updates previously emitted results, need to be able to handle these updates. For example, a sink operator that writes the results and updates of a keyed window operator to a key-value store could do this by overriding inaccu‐ rate results with the latest update using upsert writes. For some use cases it might also be necessary to distinguish between the first result and an update due to a late event.

The window operator API provides a method to explicitly declare that you expect late elements. When using event-time windows, you can specify an additional time period called *allowed lateness*. A window operator with allowed lateness will not delete a window and its state after the watermark passes the window’s end timestamp. Instead, the operator continues to maintain the complete window for the allowed lateness period. When a late element arrives within the allowed lateness period it is handled like an on-time element and handed to the trigger. When the watermark passes the window’s end timestamp plus the lateness interval, the window is finally deleted and all subsequent late elements are discarded.

Allowed lateness can be specified using the allowedLateness() method as [Example 6-22](#_bookmark0) demonstrates.

*Example 6-22. Defining a window operator with an allowed lateness of 5 seconds*

**val** readings**: DataStream**[**SensorReading**] **=** ???

**val** countPer10Secs**: DataStream**[(**String**, **Long**, **Int**, **String**)] **=** readings

.keyBy(**\_**.id)

.timeWindow(**Time**.seconds(10))

*// process late readings for 5 additional seconds*

.allowedLateness(**Time**.seconds(5))

*// count readings and update results if late readings arrive*

.process(**new UpdatingWindowCountFunction**)

*/\*\* A counting WindowProcessFunction that distinguishes between*

*\* first results and updates. \*/*

**class UpdatingWindowCountFunction extends ProcessWindowFunction**[

**SensorReading**, (**String**, **Long**, **Int**, **String**), **String**, **TimeWindow**] {

**override def** process( id**: String**,

ctx**: Context**,

elements**: Iterable**[**SensorReading**],

out**: Collector**[(**String**, **Long**, **Int**, **String**)])**: Unit** = {

*// count the number of readings*

**val** cnt **=** elements.count(**\_ => true**)

*// state to check if this is the first evaluation of the window or not*

**val** isUpdate **=** ctx.windowState.getState(

**new ValueStateDescriptor**[**Boolean**]("isUpdate", **Types**.of[**Boolean**]))

**if** (!isUpdate.value()) {

*// first evaluation, emit first result* out.collect((id, ctx.window.getEnd, cnt, "first")) isUpdate.update(**true**)

} **else** {

*// not the first evaluation, emit an update*

out.collect((id, ctx.window.getEnd, cnt, "update"))

}

}

}

Process functions can also be implemented to support late data. Since state manage‐ ment is always custom and manually done in process functions, Flink does not pro‐ vide a built-in API to support late data. Instead, you can implement the necessary logic using the building blocks of record timestamps, watermarks, and timers.

# Summary

In this chapter you learned how to implement streaming applications that operate on time. We explained how to configure the time characteristics of a streaming applica‐ tion and how to assign timestamps and watermarks. You learned about time-based operators, including Flink’s process functions, built-in windows, and custom win‐ dows. We also discussed the semantics of watermarks, how to trade off result com‐ pleteness and result latency, and strategies for handling late events.