# Windowing

ng: Dataflow SDK 1.x for Java is unsupported as of October 16, 2018. After August 12, 2020, Dataflow will no
sing Dataflow 1.x and below. See Migrating from Dataflow SDK 1.x for Java
aflow/docs/guides/migrate-java-1-to-2) for migration guidance.

**ocumentation on this page applies only to the Dataflow SDK 1.x for Java.**

ataflow SDK 2.x for Java and the Dataflow SDK for Python are based on Apache Beam. See the documentatio
aflow/model/programming-model-beam) for those SDKs.

The Dataflow SDKs use a concept called **Windowing** to subdivide a PCollection (/dataflow/model/pcollection) according to the timestamps of its individual elements. Dataflow transforms (/dataflow/model/transforms) that aggregate multiple elements, such as GroupByKey (/dataflow/model/group-by-key) and Combine (/dataflow/model/combine), work implicitly on a per-window basis—that is, they process each `PCollection` as a succession of multiple, finite windows, though the entire collection itself may be of unlimited or infinite size.

The Dataflow SDKs use a related concept called **Triggers** to determine when to "close" each finite window as unbounded data arrives. Using a trigger can help to refine the windowing strategy for your `PCollection` to deal with late-arriving data or to provide early results. See Triggers (/dataflow/model/triggers) for more information.

## Windowing Basics

Windowing is most useful with an **unbounded** `PCollection`, which represents a continuously updating data set of unknown/unlimited size (e.g. streaming data). Some Dataflow transforms, such as GroupByKey (/dataflow/model/group-by-key) and Combine (/dataflow/model/combine), group multiple elements by a common key. Ordinarily, that grouping operation groups all of the elements that have the same key *in the entire data set*. With an unbounded data set, it is impossible to collect all of the elements, since new elements are constantly being added.

In the Dataflow model, any `PCollection` can be subdivided into logical **windows**. Each element in a `PCollection` gets assigned to one or more windows according to the `PCollection`'s

**windowing function**, and each individual window contains a finite number of elements. Grouping transforms then consider each `PCollection`'s elements on a per-window basis. `GroupByKey`, for example, implicitly groups the elements of a `PCollection` by *key and window*. Dataflow *only* groups data within the same window, and doesn't group data in other windows.

**n:** Dataflow's default windowing behavior is to assign all elements of a **`PCollection`** to a single, global windo
or *unbounded* **`PCollections`**. Before you use a grouping transform such as **`GroupByKey`** on an unbounded
**`ection`**, **you must set a non-global windowing function.** See Setting Your PCollection's Windowing Function.
:ing)

don't set a non-global windowing function for your unbounded **`PCollection`** and subsequently use a groupin
orm such as **`GroupByKey`** or **`Combine`**, your pipeline will generate an error upon construction and your Dataflo
l.

n alternatively set a non-default Trigger (/dataflow/model/triggers) for a **`PCollection`** to allow the global wi
t "early" results under some other conditions.

## Windowing Constraints

Once you set the windowing function for a `PCollection`, the elements' windows are used *the next time you apply a grouping transform* to that `PCollection`. Dataflow performs the actual window grouping on an as-needed basis; if you set a windowing function using the `Window` transform, each element is assigned to a window, but the windows are not considered until you group the `PCollection` with `GroupByKey` or `Combine`. This can have different effects on your pipeline.

Consider the example pipeline in Figure 1 below:

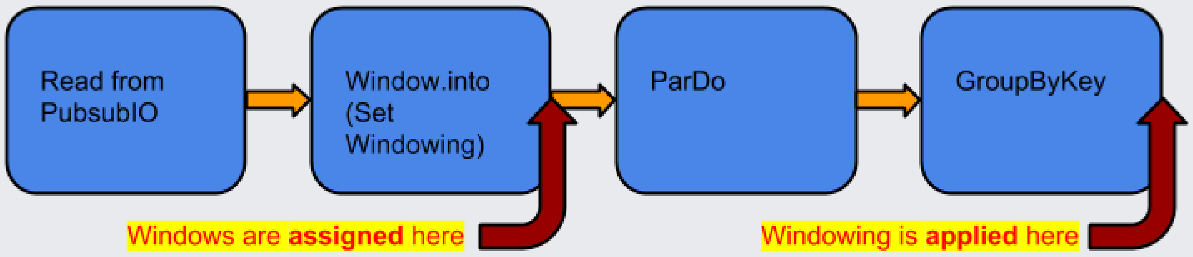
Read from PubsubIO
Window.into (Set Windowing)
ParDo
GroupByKey
Windows are assigned here
Windowing is applied here

Figure 1: Pipeline Applying Windowing

In the above pipeline, we create an unbounded `PCollection` by reading a set of key/value pairs using PubsubIO (/dataflow/model/pubsub-io), and then apply a windowing function to that collection using the `Window` transform. We then apply a `ParDo` to the collection, and then later group the result of that `ParDo` using `GroupByKey`. The windowing function *has no effect on the* `ParDo` *transform*, because the windows are not actually used until they're needed for the `GroupByKey`.

Subsequent transforms, however, are applied to the result of the `GroupByKey`–that is, data grouped by *both key and window.*

## Using Windowing With Bounded PCollections

You can use windowing with fixed-size data sets in **bounded** `PCollection`s. Note, however, that windowing considers only the implicit timestamps attached to each element of a `PCollection`, and data sources that create fixed data sets (such as `TextIO` and `BigQueryIO`) assign the same timestamp to every element. This means that all the elements are by default part of a single, global window. Having all elements assigned to the same window will cause a pipeline to execute in classic MapReduce batch style.

To use windowing with fixed data sets, you can assign your own timestamps (#TimeStamping) to each element. To assign timestamps to elements, you use a `ParDo` transform with a `DoFn` that outputs each element with a new timestamp.

Using windowing with a bounded `PCollection` can affect how your pipeline processes data. For example, consider the following pipeline:

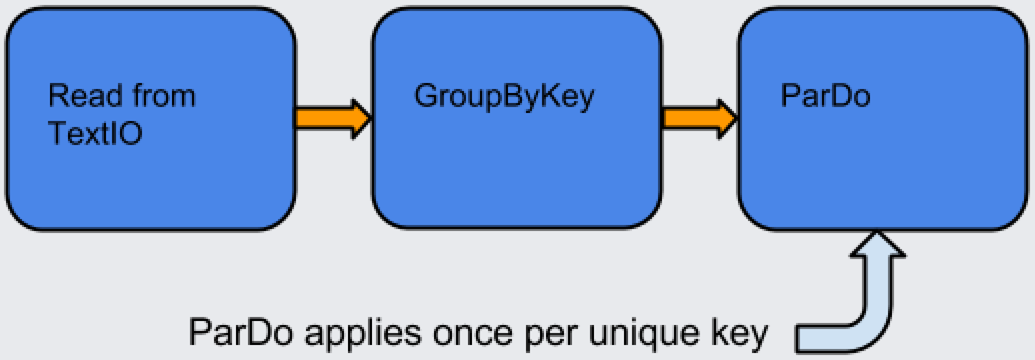

Figure 2: GroupByKey and ParDo without windowing, on a bounded collection.

In the above pipeline, we create a bounded `PCollection` by reading a set of key/value pairs using TextIO (/dataflow/model/text-io). We then group the collection using `GroupByKey`, and apply a `ParDo` transform to the grouped `PCollection`. In this example, the `GroupByKey` creates a collection of unique keys, and then `ParDo` gets applied *exactly once per key*.

Now, consider the same pipeline, but using a windowing function:

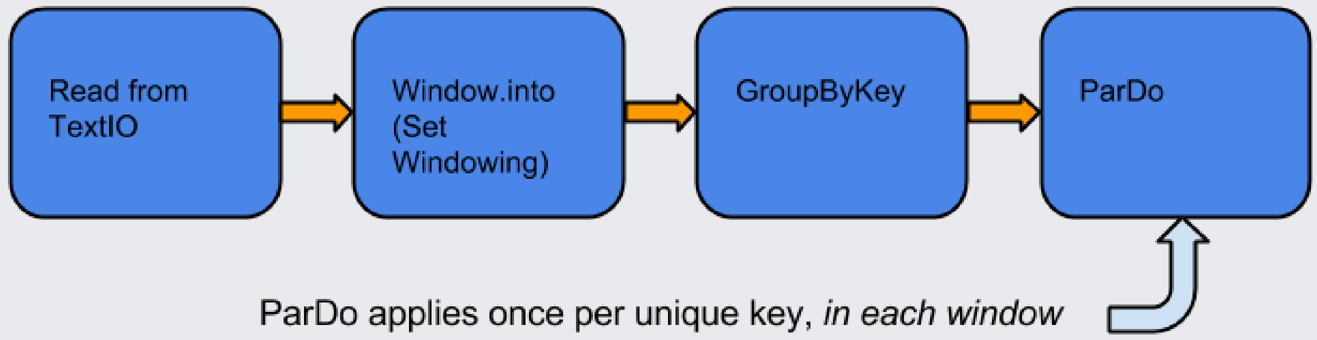
Read from TextIO
Window.into (Set Windowing)
GroupByKey
ParDo
ParDo applies once per unique key, *in each window*

Figure 3: GroupByKey and ParDo with windowing, on a bounded collection.

As before, the pipeline creates a bounded `PCollection` of key/value pairs. We then set a windowing function (/dataflow/model/windowing#Setting) for that `PCollection`. The `GroupByKey` transform now groups the elements of the `PCollection` *by both key and window*. The subsequent `ParDo` transform gets applied *multiple times per key*, once for each window.

## Windowing Functions

The Dataflow SDKs let you define different kinds of windows to divide the elements of your `PCollection`. The SDK provides several windowing functions, including:

- Fixed Time Windows
- Sliding Time Windows
- Per-Session Windows
- Single Global Window

:hat each element can logically belong to *more than one window*, depending on the windowing on you use. Sliding time windowing, for example, creates overlapping windows wherein a single :nt can be assigned to multiple windows.

### Fixed Time Windows

The most simple form of windowing is a **fixed time window**: given a timestamped `PCollection`, which might be continuously updating, each window might capture (for example) five minutes worth of elements.

A fixed time window represents the time interval in the data stream that defines a bundle of data for processing. Consider a window that operates at five-minute intervals: all of the elements in your unbounded `PCollection` with timestamp values between 0:00:00 and 0:04:59 belong to the first window, elements with timestamp values between 0:05:00 and 0:09:59 belong to the second window, and so on.

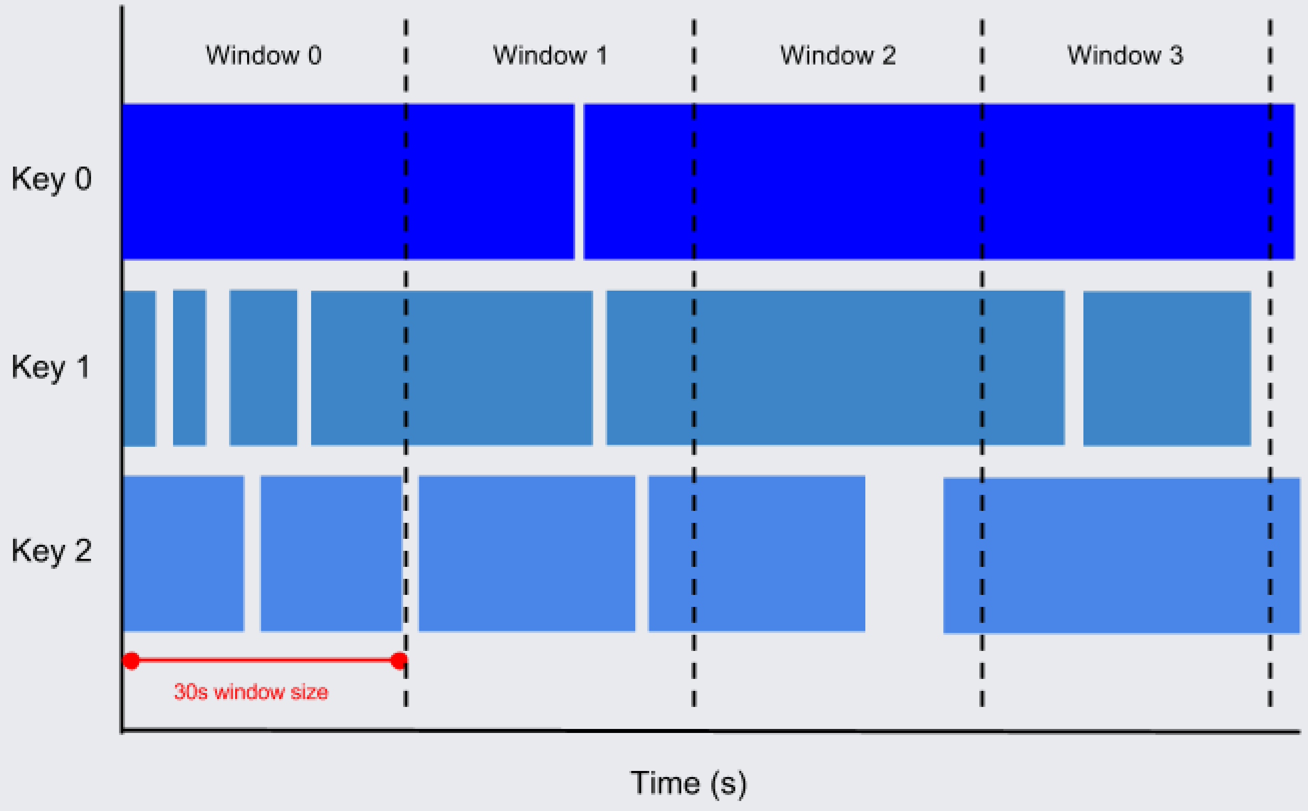

Figure 4: Fixed time windows, 30s in size.

## Sliding Time Windows

A **sliding time window** also uses time intervals in the data stream to define bundles of data; however, with sliding time windowing, the windows overlap. Each window might capture five minutes worth of data, but a new window starts every ten seconds. The frequency with which sliding windows begin is called the *period*. Therefore, our example would have a window *size* of five minutes and a *period* of ten seconds.

Because multiple windows overlap, most elements in a data set will belong to more than one window. This kind of Windowing is useful for taking running averages of data; using sliding time windows, you can compute a running average of the past five minutes' worth of data, updated every ten seconds, in our example.

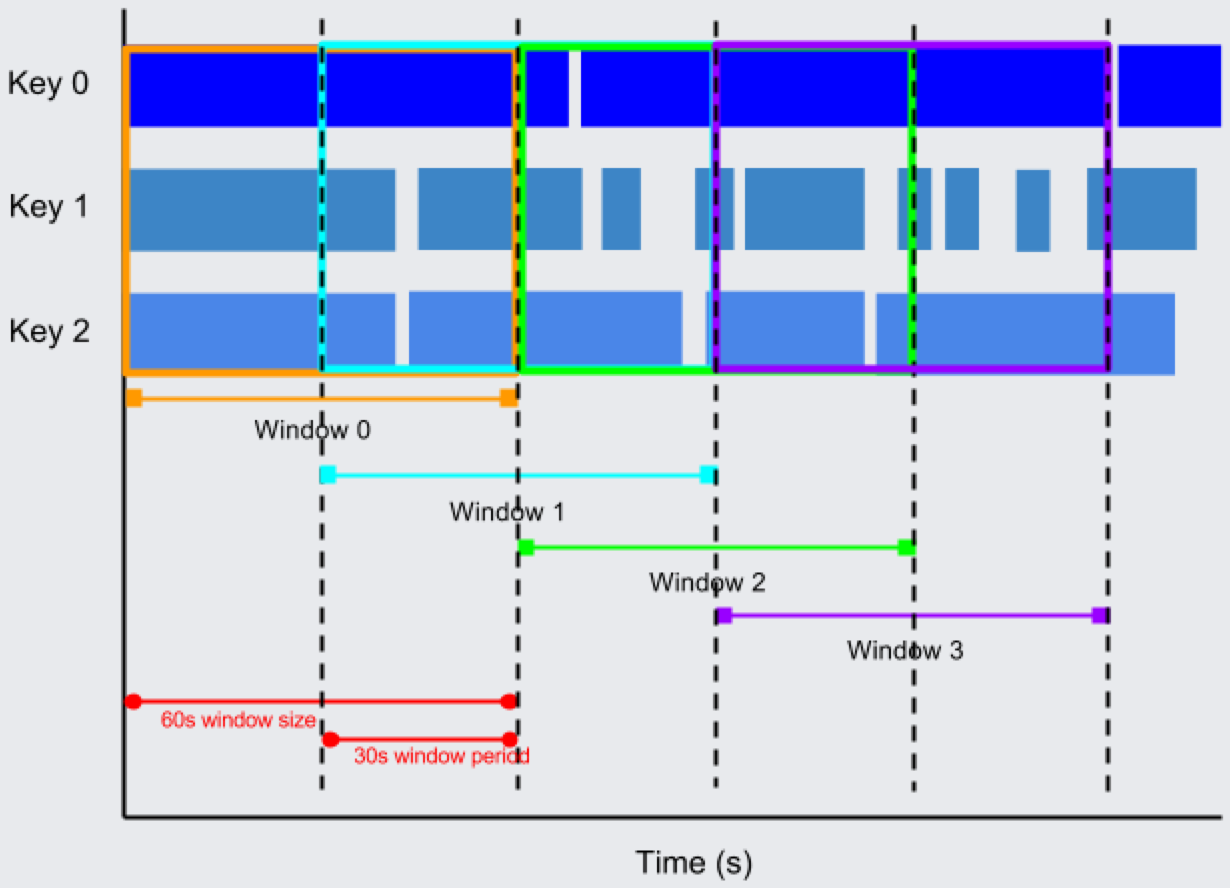

Figure 5: Sliding time windows, with 1 minute window size and 30s window period.

## Session Windows

A **session window** function defines windows around areas of concentration in the data. Session windowing is useful for data that is irregularly distributed with respect to time; for example, a data stream representing user mouse activity may have long periods of idle time interspersed with high concentrations of clicks. Session windowing groups the high concentrations of data into separate windows and filters out the idle sections of the data stream.

Note that **session windowing applies on a per-key basis**; that is, grouping into sessions **only** takes into account data that has the same key. Each key in your data collection will therefore be grouped into disjoint windows of differing sizes.

The simplest kind of session windowing specifies a *minimum gap duration*. All data arriving below a minimum threshold of time delay is grouped into the same window. If data arrives after the minimum specified gap duration time, this initiates the start of a new window.

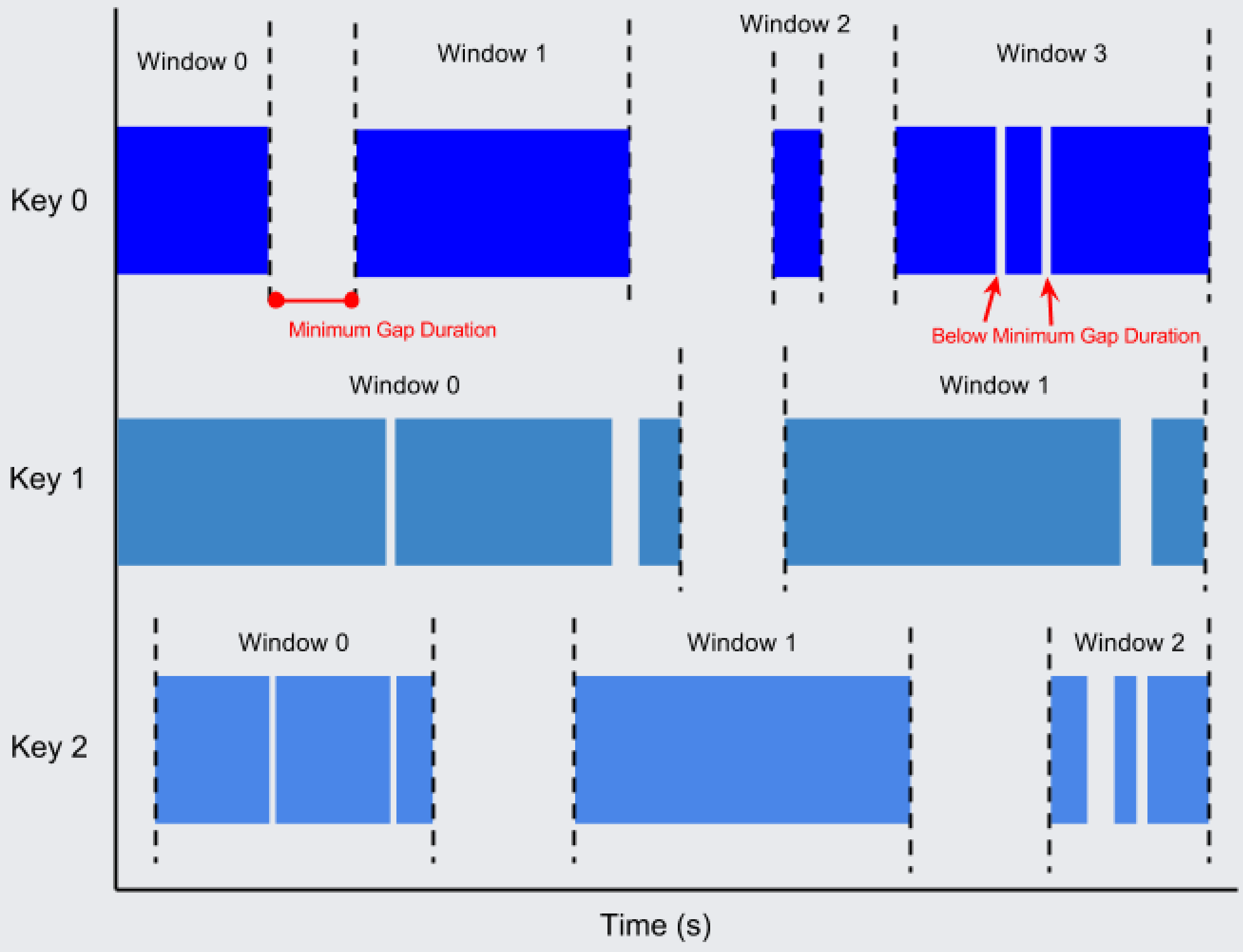

Figure 5: Session windows, with a minimum gap duration. Note how each data key has different windows, according to its data distribution.

## Single Global Window

By default, all data in a `PCollection` is assigned to a single global window. If your data set is of a fixed size, you can leave the global window default for your `PCollection`. If the elements of your `PCollection` all belong to a single global window, your pipeline will execute much like a batch processing job (as in MapReduce-based processing).

ın use a single global window if you are working with an unbounded data set, e.g. from a streaming data sour
ʼer, use caution when applying aggregating transforms such as GroupByKey (/dataflow/model/group-by-key)

ne (/dataflow/model/combine). A single global window with a default trigger generally requires the entire data vailable before processing, which is not possible with continuously updating data.

form aggregations on an unbounded **PCollection** that uses global windowing, you should specify a non-def (/dataflow/model/triggers) for that **PCollection**. If you attempt to perform an aggregation such as **Group** unbounded, globally windowed **PCollection** with default triggering, the Cloud Dataflow service will generate ion when your pipeline is constructed.

### Other Windowing Functions

The Dataflow SDKs provide more windowing functions beyond fixed, sliding, session, and global windows, such as Calendar-based windows.

Java

See the package com.google.cloud.dataflow.sdk.transforms.windowing (/dataflow/java-sdk/JavaDoc/com/google/cloud/dataflow/sdk/transforms/windowing/package-summary)
for a complete list of the available windowing functions in the Dataflow SDK for Java.

## Setting Your PCollection's Windowing Function

You can set the windowing function for a `PCollection` by applying the `Window` transform. When you apply the `Window` transform, you must provide a `WindowFn`. The `WindowFn` determines the windowing function your `PCollection` will use for subsequent grouping transforms, such as a fixed or sliding time window.

The Dataflow SDKs provide pre-defined `WindownFn`s for the basic windowing functions (#Functions), or you can define your own `WindowFn` in advanced cases.

ically, like all transforms, **Window** takes an input **PCollection** and outputs a new **PCollection** wi element assigned to one or more logical, finite windows.

setting a windowing function, you may also want to set a trigger for your **PCollection**. The trigger determine each individual window is aggregated and emitted, and helps refine how the windowing function performs wit t to late data and computing early results. See Triggers (/dataflow/model/triggers) for more information.

## Setting Fixed-Time Windows

The following example code shows how to apply `Window` to divide a `PCollection` into fixed windows, each one minute in length:

Java

```
PCollection<String> items = ...;
PCollection<String> fixed_windowed_items = items.apply(
  Window.<String>into(FixedWindows.of(Duration.standardMinutes(1))));
```

## Setting Sliding Time Windows

The following example code shows how to apply `Window` to divide a `PCollection` into sliding time windows. Each window is 30 minutes in length, and a new window begins every five seconds:

Java

```
PCollection<String> items = ...;
PCollection<String> sliding_windowed_items = items.apply(
  Window.<String>into(SlidingWindows.of(Duration.standardMinutes(30)).every(Durati
```

## Setting Session Windows

The following example code shows how to apply `Window` to divide a `PCollection` into session windows, where each session must be separated by a time gap of at least 10 minutes:

```
PCollection<String> items = ...;
PCollection<String> session_windowed_items = items.apply(
  Window.<String>into(Sessions.withGapDuration(Duration.standardMinutes(10))));
```

Note that the sessions are **per-key**—each key in the collection will have its own session groupings depending on the data distribution.

### Setting a Single Global Window

If your `PCollection` is bounded (the size is fixed), you can assign all the elements to a single global window. The following example code shows how to set a single global window for a `PCollection`:

To set a single global window for your `PCollection`, pass `new GlobalWindows()` as the `WindowFn` when you apply the `Window` transform. The following example code shows how to apply `Window` to assign a `PCollection` into a single global window:

Java

```
PCollection<String> items = ...;
PCollection<String> batch_items = items.apply(
  Window.<String>into(new GlobalWindows()));
```

## Time Skew, Data Lag, and Late Data

In any data processing system, there is a certain amount of lag between the time a data event occurs (the "event time", determined by the timestamp on the data element itself) and the time the actual data element gets processed at any stage in your pipeline (the "processing time", determined by the clock on the system processing the element).

erfect system, the event time for each data element and the processing time would be equal, or a
have a consistent delta. However, in any real-world computing system, data generation and deliv
ıbject to any number of temporal limitations. In large or distributed systems, such as a distribute

tion of web front-ends generating customer orders or log files, there are no guarantees that data
s will appear in your pipeline in the same order that they were generated in various places on the

For example, let's say we have a `PCollection` that's using fixed-time windowing, with windows that are five minutes long. For each window, Dataflow must collect all the data with an *event time* timestamp in the given window range (between 0:00 and 4:59 in the first window, for instance). Data with timestamps outside that range (data from 5:00 or later) belong to a different window.

However, data isn't always guaranteed to arrive in a pipeline in correct time order, or to always arrive at predictable intervals. Dataflow tracks a *watermark*, which is the system's notion of when all data in a certain window can be expected to have arrived in the pipeline. Data that arrives with a timestamp after the watermark is considered **late data**.

From our example, suppose we have a simple watermark that assumes approximately 30s of lag time between the data timestamps (the event time) and the time the data appears in the pipeline (the processing time), then Dataflow would close the first window at 5:30. If a data record arrives at 5:34, but with a timestamp that would put it in the 0:00-4:59 window (say, 3:38), then that record is late data.

For simplicity, we've assumed that we're using a very straightforward watermark that estimates
ne/time skew. In practice, your **`PCollection`**'s data source determines the watermark, and watern
e more precise or complex.

## Managing Time Skew and Late Data

You can allow late data by invoking the `.withAllowedLateness` operation when you set your `PCollection`'s windowing strategy. The following code example demonstrates a windowing strategy that will allow late data up to two days after the end of a window.

Java

```
PCollection<String> items = ...;
PCollection<String> fixed_windowed_items = items.apply(
  Window.<String>into(FixedWindows.of(Duration.standardMinutes(1)))
        .withAllowedLateness(Duration.standardDays(2)));
```

When you set `.withAllowedLateness` on a `PCollection`, that allowed lateness propagates forward to any subsequent `PCollection` derived from the first `PCollection` you applied allowed lateness to. If you want to change the allowed lateness later in your pipeline, you must do so explicitly by applying `Window.withAllowedLateness()` again.

You can also use Dataflow's Triggers (/dataflow/model/triggers) API to help you refine the windowing strategy for a `PCollection`. You can use triggers to determine exactly when each individual window aggregates and reports its results, including how the window emits late elements.

Dataflow's default windowing and trigger strategies *discard* late data. If you want to ensure that ne handles instances of late data, you'll need to explicitly set **`.withAllowedLateness`** when you s **`Collection`**'s windowing strategy and set triggers for your **`PCollection`**s accordingly.

## Adding Timestamps To a PCollection's Elements

You can assign new timestamps to the elements of a `PCollection` by applying a ParDo (/dataflow/model/par-do) transform that outputs new elements with timestamps that you set. Assigning timestamps can be useful if you want to use Dataflow's windowing features, but your data set comes from a source without implicit timestamps (such as a file from TextIO (/dataflow/model/text-io)).

This is a good pattern to follow when your data set includes timestamp data, but the timestamps are *not* generated by the Dataflow data source. An example might be if your pipeline reads log records from an input file, and each log record includes a timestamp field; since your pipeline reads the records in from a file, the file source doesn't assign timestamps automatically. You can parse the timestamp field from each record and use a `ParDo` transform to attach the timestamps to each element in your `PCollection`.

Java

To assign timestamps, your `ParDo` transform needs to use a `DoFn` that outputs elements using `ProcessContext.outputWithTimestamp` (rather than the usual `ProcessContext.output` used to emit elements to the main output collection). The following example code shows a `ParDo` with a `DoFn` that outputs elements with new timestamps:

```
PCollection<LogEntry> unstampedLogs = ...;
PCollection<LogEntry> stampedLogs =
    unstampedLogs.apply(ParDo.of(new DoFn<LogEntry, LogEntry>() {
      public void processElement(ProcessContext c) {
        // Extract the timestamp from log entry we're currently processing.
        Instant logTimeStamp = extractTimeStampFromLogEntry(c.element());
        // Use outputWithTimestamp to emit the log entry with timestamp attached.
        c.outputWithTimestamp(c.element(), logTimeStamp);
      }
    }));
```