Building I/O connectors using Splittable DoFns

Israel Herraiz, Miren Esnaola

BEAM SUMMIT

A bit of history...

To connect to an unsupported data store in the Beam SDK, you need to create a custom I/O connector.

Until fairly recently there were 2 options for the implementation of a custom I/O.

1. Create a mini-pipeline made of the basic **`ParDo`** and **`GroupByKey`** transforms (Bounded sources only and certain scenarios).
2. Use the Source API.

Connectors as mini-pipelines

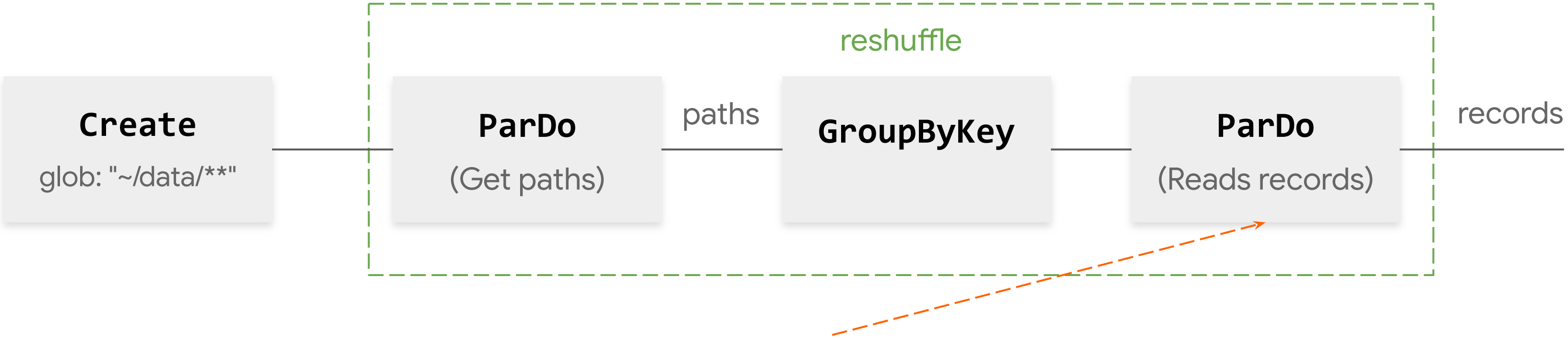
For bounded data sources where data can be read in parallel, a mini-pipeline can be created consisting of 2 steps:

1. Split incoming data into parts to be read in parallel
2. Read from each of those parts

Each of those steps is a `ParDo`, with a `GroupByKey` in between. For most runners the `GroupByKey` allows the runner to use different number of workers and dynamic work rebalancing if supported.

Connectors as mini-pipelines

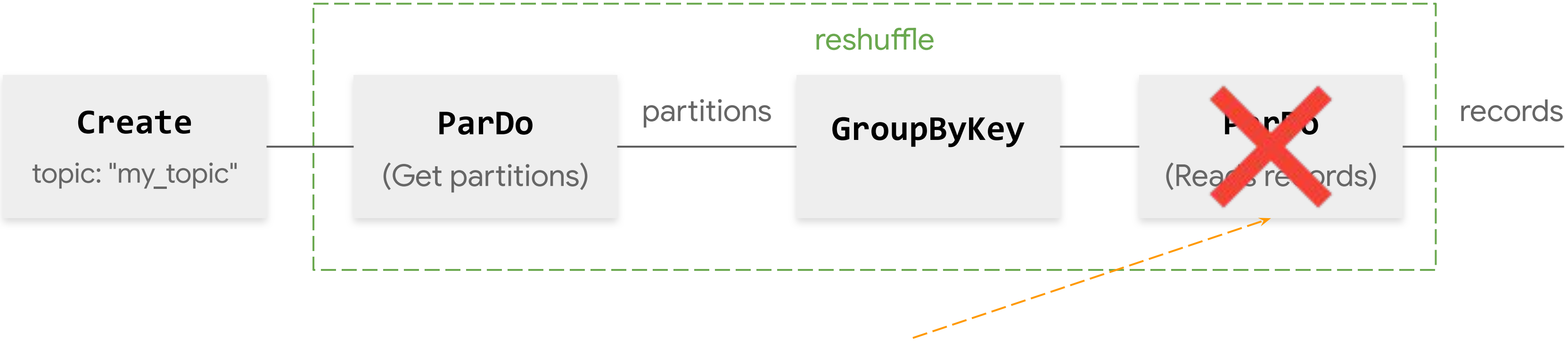
Problem statement — Given a file glob as input, read the records in the files matching the pattern



Limitation — Some files might be much larger than others. The second ParDo may have very long individual process calls and result in poor pipeline performance.

Connectors as mini-pipelines

Problem statement — Given a Kafka topic as input, read the records in the partitions



Impossible!! It would need to output an infinite number of records per partition

Umm, that’s why I created the Source API, to overcome these limitations...

Source API

Pros

- It allows the reading both bounded and unbounded data sources, in parallel using multiple workers
- It allows checkpointing and resuming reads from unbounded data sources.
- It provides advanced features such as progress reporting and dynamic rebalancing (which together enable autoscaling) for bounded sources,
- It supports reporting the source's watermark and backlog for unbounded sources.

Source API

Cons

- Coding involves a lot of boilerplate and is error-prone.,
- It does not compose well because a `Source` can appear only at the root of a pipeline.
- It is not possible to reuse code between seemingly very similar bounded and unbounded sources.
- It is not clear how to classify the ingestion of a very large and continuously growing dataset. Ingesting its “already available”

How could I address the Source API limitations? What about using `DoFn`*s as a starting point? They can be applied to bounded or unbounded data sources, plus they are composable...*

… but `DoFns` have a few limitations

Splittability

Applying a `DoFn` to a single element is monolithic.

Runner interaction

Runners apply a `DoFn` to an element as a “black box”.

I got it!!! The solution is a **Splittable DoFn**

What is a SDF?

A **Splittable DoFn** (SDF) is a generalization of a DoFn enabling Apache Beam developers to create modular and composable I/O components. Although that’s their main use, they can also be applied in other advanced non-I/O scenarios.

How does a SDF work?

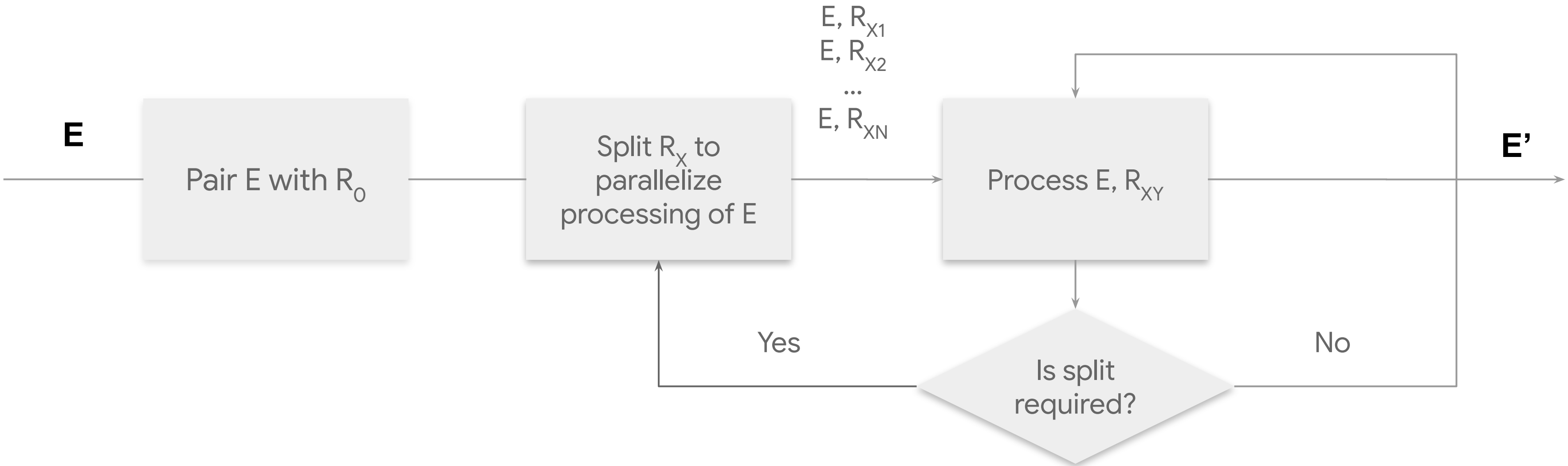
The processing of an element by a SDF is decomposed into a number of restrictions (potentially infinite).

A **restriction** describes some part of the work to be done for the whole element.

Executing an SDF follows the following steps:

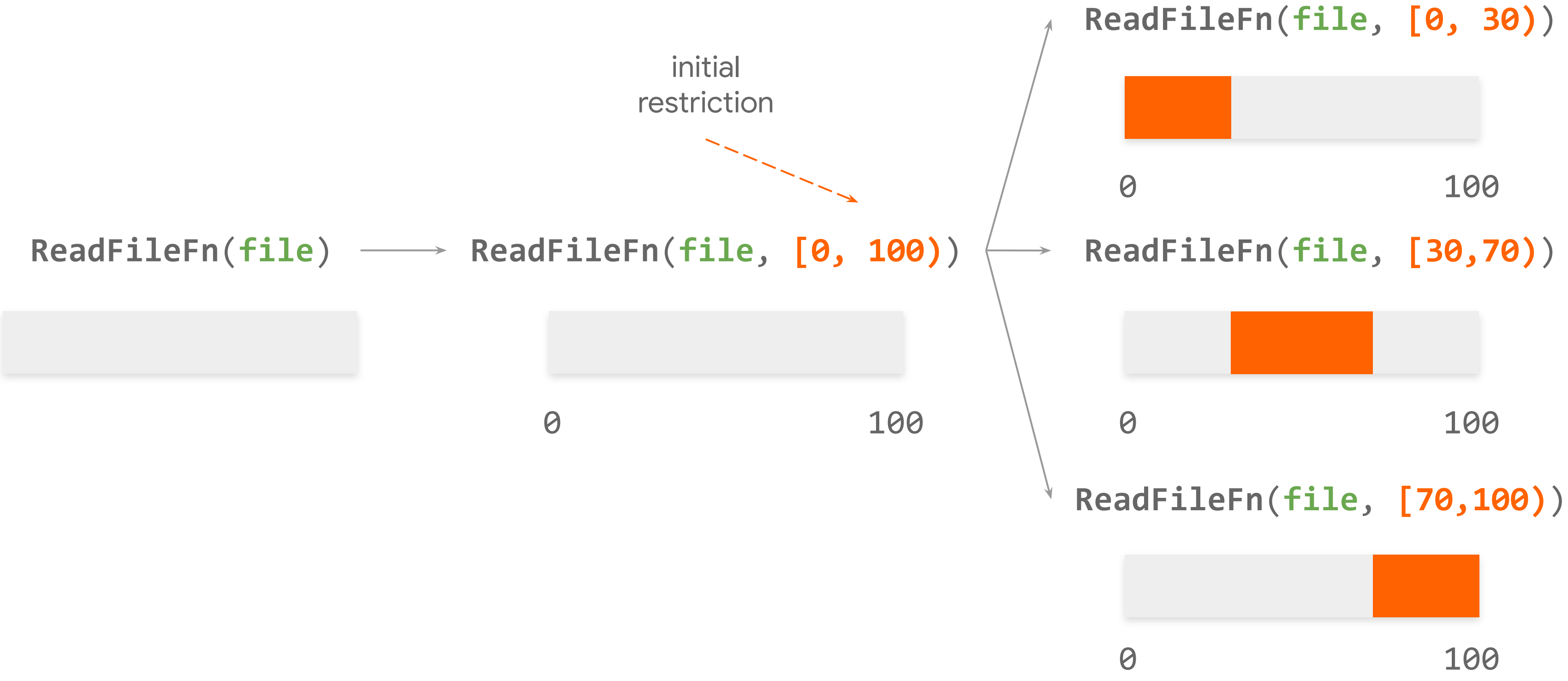
1. Each element is paired with a restriction (e.g. filename is paired with offset range representing the whole file).
2. For each element the initial restriction is split into smaller restrictions.
3. The runner distributes element and restriction pairs to several workers.
4. Element and restriction pairs are processed in parallel. At this point, the runner can decide to further split any restriction being processed.

How does a SDF work?



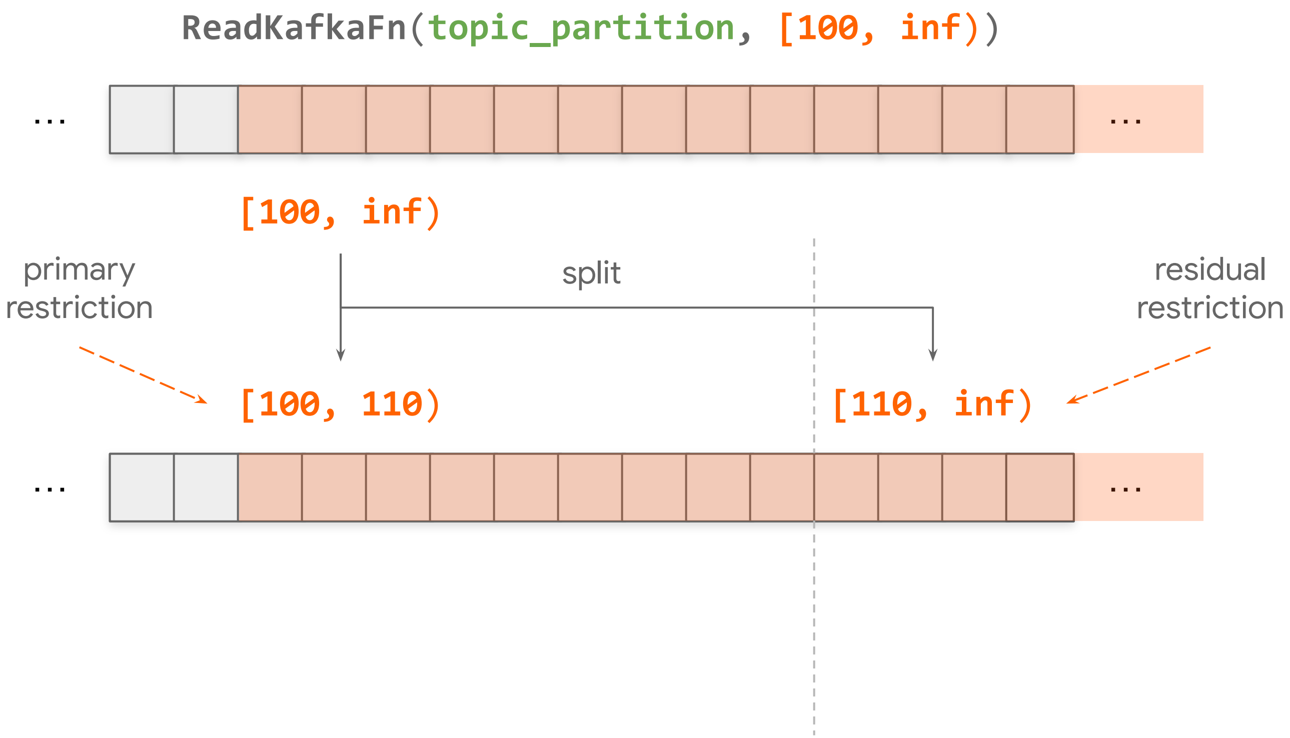
Processing with restrictions

Bounded source

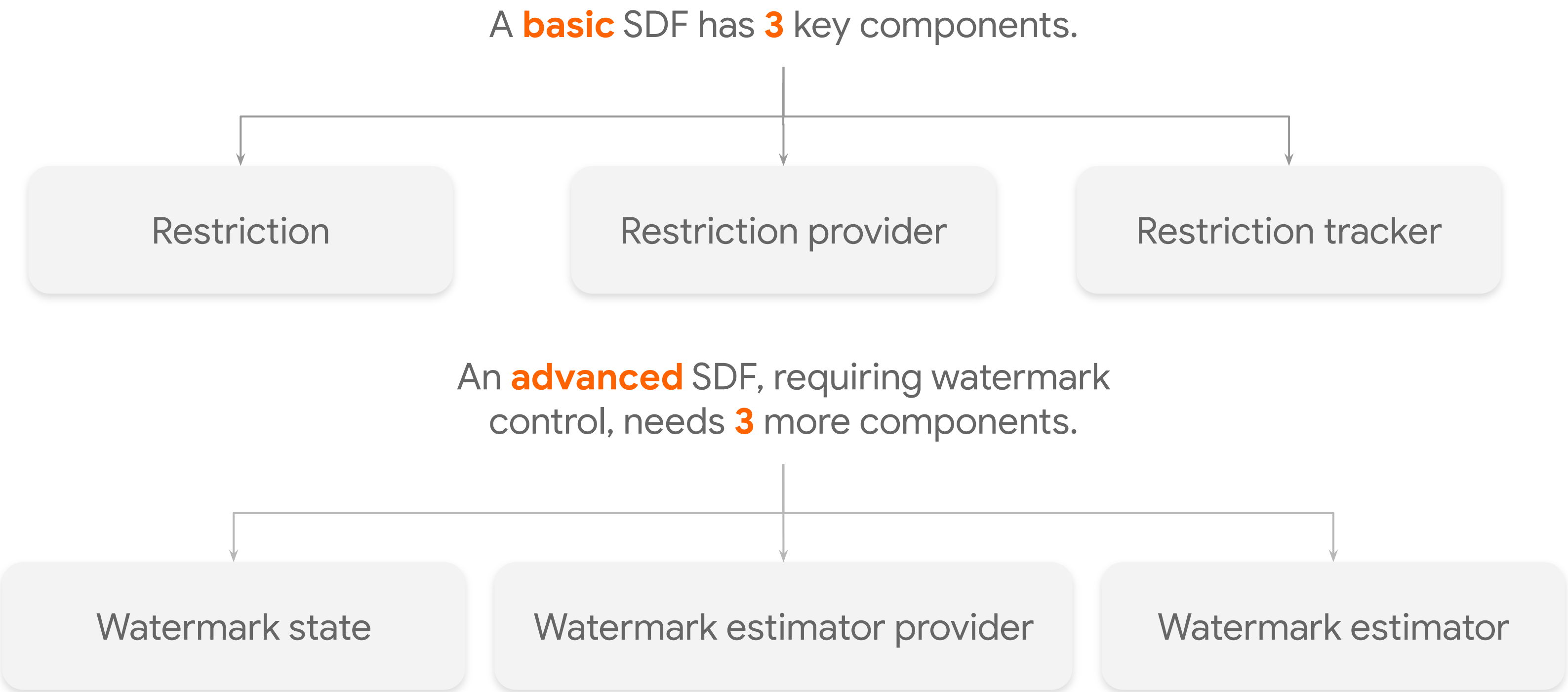

initial
restriction
ReadFileFn(file)
ReadFileFn(file, [0, 100))
0
100
ReadFileFn(file, [0, 30))
0
100
ReadFileFn(file, [30,70))
0
100
ReadFileFn(file, [70,100))
0
100

Processing with restrictions

Unbounded source



SDF components



A **basic** SDF has **3** key components.

- Restriction
- Restriction provider
- Restriction tracker

An **advanced** SDF, requiring watermark control, needs **3** more components.

- Watermark state
- Watermark estimator provider
- Watermark estimator

SDF components

Basic SDF

Restriction

- It represents a subset of work for a given element.
- No specific class needs to be implemented to represent a restriction.

Restriction provider

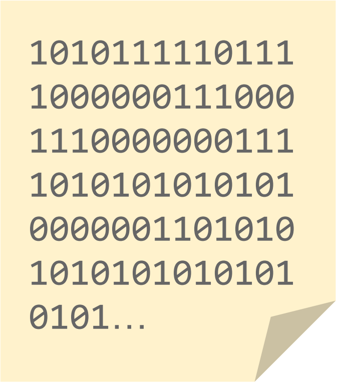
- It lets developers override default implementations used to generate and manipulate restrictions.
- It extends from the `RestrictionProvider` base class.

Restriction tracker

- It tracks for which parts of the restriction processing have been completed.
- It extends from the `RestrictionTracker` base class.

SDF components

Basic SDF



There are built-in classes in the SDK that can be leveraged when restrictions can be represented as an offset range. This is useful when working with files.

OffsetRange
(Restriction)

OffsetRestrictionTracker
(RestrictionTracker)

SDF Components

Restriction provider

You must provide an implementation of a restriction extending from `RestrictionProvider`.

It is mandatory to override the following methods:

- **`inital_restriction`**`(self, element)`

  It returns the initial restriction for the given element.

- **`create_tracker`**`(self, restriction)`

  It returns a new tracker for the given restriction.

- **`restriction_size`**`(self, element, restriction)`

  It returns the size of the given restriction. It must be a non-negative value.

SDF Components

Restriction provider

Other methods, that have default implementations could be overridden if necessary:

- `restriction_coder(self)`

  It returns a coder for restrictions. Only required if it cannot be inferred at runtime.

- `split(self, element, restriction)`

  It enables runners to perform initial splits to increase parallelism. It returns an iterator of restrictions.

- `split_and_size(self, element, restriction)`

  It does the same as the split method but additional returns the size of each of the splits.

- `truncate(self, element, restriction)`

  It truncates the provided restriction into a restriction representing a finite amount of work when the pipeline is draining.

SDF Components

Restriction tracker

You must provide a restriction tracker extending from `RestrictionTracker` and overriding at least the following methods:

- **`current_restriction`**`(self)`

  It returns the restriction that the `DoFn.process()` call will be doing. It is subject to variations as the runner might have concurrently split the work to be done. (See `try_split` method in the next slide).

- **`try_claim`**`(self, position)`

  It must be used from within the `DoFn.process` method to notify that there is more work to process. If the given position fits into the current restriction boundaries , it is marked as processed in the tracker and returns `True`. If not, it returns `False` and the `DoFn.process` call must immediately return.

- **`check_done`**`(self)`

  It checks whether the restriction has been fully processed. If so, it must return True. If not, it must raise a `ValueError` error with an informative message.

- **`is_bounded`**`(self)`

  It returns `True` if the current restriction represents a finite amount of work and `False` otherwise.

SDF Components

Restriction tracker

In addition to this, for streaming scenarios, overrides need to be provided for:

- **`try_split`**`(self, fraction_of_remainder)`

  If possible, it splits the current restriction into a primary one and a residual one. Once the split is done:

  - The **primary restriction** becomes the current restriction for the `DoFn.process()` invocation.
  - The **residual restriction** is left to be executed by a separate `DoFn.process()` invocation (most likely in a different process).

  The `fraction_of_remainder` parameter provides an indication of the percentage of the work left to process that the primary restriction should represent.

  Note — Although not mandatory, it is also recommended to provide an override for this method in batch scenarios.

SDF Components

Restriction tracker

Finally, it is also advisable to provide an implementation for the following method:

- **`current_progress`**`(self)`

  It returns a `RestrictionProgress` object detailing the progress made processing the piece of work represented by the current restriction. This information helps the runner make a better job at parallel processing.

How do we code it?

To denote a `DoFn` as splittable the `DoFn.process()` method should have exactly one parameter whose default value is an instance of `RestrictionParam`.

This `RestrictionParam` instance can either be constructed:

- Explicitly passing an instance of a `RestrictionProvider`
- Not passing anything, in which case the `DoFn` will have to extend from `RestrictionProvider` and provide overrides for the required methods.

```python
import beam
import os

class FileToWordsRestrictionProvider(beam.transforms.core.RestrictionProvider):

    def initial_restriction(self, file_name):
        return OffsetRange(0, os.stat(file_name).st_size)

    def create_tracker(self, restriction):
        return beam.io.restriction_trackers.OffsetRestrictionTracker()

    def restriction_size(self, file_name, restriction)
        return restriction.end - restriction.start

class FileToWordsFn(beam.DoFn):

    def process(
        self,
        file_name,
        tracker=beam.DoFn.RestrictionParam(FileToWordsRestrictionProvider())):
        # TODO
```

```python
import beam
import os

class FileToWordsFn(beam.DoFn, beam.transforms.core.RestrictionProvider):

    def initial_restriction(self, file_name):
        return OffsetRange(0, os.stat(file_name).st_size)

    def create_tracker(self, restriction):
        return beam.io.restriction_trackers.OffsetRestrictionTracker()

    def restriction_size(self, element, restriction):
        return restriction.end - restriction.start

    def process(self,
        file_name,
        tracker=beam.DoFn.RestrictionParam()):
        # TODO
```

How do we code it?

The next is to proceed to the implementation of the `DoFn.process` method

1. Recover the current restriction using the parameter of type `RestrictionParam` passed as argument.
2. Try to claim / lock the position.
3. Proceed to the processing associated to that element and restriction pair.

⚠ Things to take into consideration when writing the DoFn.process method:

- If the amount of work performed per input element is unbounded (e.g. reading messages from a Kafka partition) the function needs to be annotated with with the decorator `beam.DoFn.unbounded_per_element`.
- The current restriction can be modified in parallel in another thread, so it is not advised to store its state locally.
- Only after successfully claiming a position should we produce any output and / or perform side effects.

```python
import beam
import os

class FileToWordsRestrictionProvider(beam.transforms.core.RestrictionProvider):

    def initial_restriction(self, file_name):
        return OffsetRange(0, os.stat(file_name).st_size)

    def create_tracker(self, restriction):
        return beam.io.restriction_trackers.OffsetRestrictionTracker()

    def restriction_size(self, file_name, restriction)
        return restriction.end - restriction.start

class FileToWordsFn(beam.DoFn):

    def process(self,
        file_name,
        tracker=beam.DoFn.RestrictionParam(FileToWordsRestrictionProvider())):

        with open(file_name) as file_handle:
            file_handle.seek(tracker.current_restriction.start())
            while tracker.try_claim(file_handle.tell()):
                yield read_next_record(file_handle)
```

INITIAL SPLITS

```
class FileToWordsRestrictionProvider(beam.transforms.core.RestrictionProvider):

    def initial_restriction(self, file_name):
        return OffsetRange(0, os.stat(file_name).st_size)

    def create_tracker(self, restriction):
        return beam.io.restriction_trackers.OffsetRestrictionTracker()

    def restriction_size(self, element, restriction):
        return restriction.end - restriction.start

    def split(self, file_name, restriction):
        split_size = 64 * (1 << 20)
        i = restriction.start
        while i < restriction.end - split_size:
            yield OffsetRange(i, i + split_size)
        i += split_size
        yield OffsetRange(i, restriction.end)
```

We split the file in blocks of 64 MiB to increase parallelism

How do runners use sizing information?

The may use it:

- **Before processing an element and restriction**

  To choose who processes the restrictions and how they are processed so optimal balancing and parallelization can be achieved.

- **During the processing of an element and restriction**

  To choose which restrictions to split and who should process them.

SIZING

```
class FileToWordsRestrictionProvider(beam.transforms.core.RestrictionProvider):

    def initial_restriction(self, file_name):
        return OffsetRange(0, os.stat(file_name).st_size)

    def create_tracker(self, restriction):
        return beam.io.restriction_trackers.OffsetRestrictionTracker()

    def restriction_size(self, element, restriction):
        return restriction.end - restriction.start
```

All restrictions have a cost proportional to file size

</>

SIZING

```
class FileToWordsRestrictionProvider(beam.transforms.core.RestrictionProvider):

    def __init__(self, weights = {})
        self.weights = weights

    def initial_restriction(self, file_name):
        return OffsetRange(0, os.stat(file_name).st_size)

    def create_tracker(self, restriction):
        return beam.io.restriction_trackers.OffsetRestrictionTracker()

    def restriction_size(self,filename, restriction)

        base_name, extension = os.path.splittext(file_name)
        weight = self.weights[extension] if extension in self.weights else 1
        return weight * (restriction.end - restriction.start)
```

The processing of files with certain extensions is computationally more expensive so we reflect that in the restriction size

What if we are stuck?

In some scenarios it can happen that the actual data necessary to complete the processing of an element and restriction pair is not ready.

It is quite frequent with unbounded restrictions, but it can also happen with bounded ones if the data is not yet ready.

A few examples are:

- We are reading messages from a Kafka topic partition and no new messages have been published.
- We are watching a directory for new files and none have been added.
- The source system that we are reading data from is throttling.

What should I do???

What if we are stuck?

The `DoFn.process` method should return signaling that the processing current restriction is not done, optionally suggesting a time to resume at (*). This will improve resource utilization as execution will continue for restrictions with work available.

(*) The runner will try to honor the time to resume at, but without offering any guarantees.

```
class MySplittableDoFn(beam.DoFn):
  def process(self,
      element,
      restriction_tracker=beam.DoFn.RestrictionParam(MyRestrictionProvider())):
    current_position = restriction_tracker.current_restriction.start()
    while True:
      try:
        records = external_service.fetch(current_position)
        if records.empty():
          restriction_tracker.defer_remainder(timestamp.Duration(second=10))
          return
        for record in records:
          if restriction_tracker.try_claim(record.position):
            current_position = record.position
            yield record
          else:
            return
      except TimeoutError:
        restriction_tracker.defer_remainder(timestamp.Duration(seconds=60))
        return
```

SDF Components

Advanced SDF - Watermark control

Watermark state

- It is a user-defined object. In its simplest form it could just be a timestamp.

Watermark estimator

- It tracks the watermark state when the processing of an element and restriction pair is in progress.
- It extends from the `WatermarkEstimator` base class.

Watermark estimator provider

- It lets developers define how to initialize the watermark state and create a watermark estimator.
- It extends from the `WatermarkEstimatorProvider` base class.

Controlling the watermark

Default behaviour

- Provide no watermark estimation.
- The runner computes the output watermark as the minimum of all upstream watermarks.

Controlling the watermark

Advanced behaviour

- The `WatermarkEstimatorProvider` returns an initial estimation for the watermark state that an element and restriction pair will produce.
- The `WatermarkEstimator` updates the estimation based on the processing time, timestamp of output records or manual modifications done in the `DoFn.process` call.
- The runner computes the output watermark by taking the minimum over:
  - All upstream watermarks.
  - The estimation reported by each element and restriction pair.
- The reported watermark must monotonically increase for each element and restriction pair across bundle boundaries.
- When an element and restriction pair stops processing its watermark, it is no longer considered part of the calculation.

Controlling the watermark

Watermark estimators

ManualWatermarkEstimator

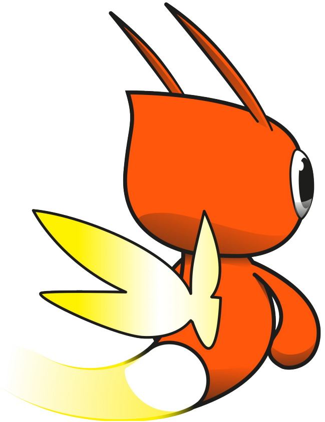
It gives the possibility to provide an estimation of the watermark state manually in the `DoFn.process` method.

MonotonicWatermarkEstimator

It take as estimation of the watermark state the timestamp of the output record and assumes that the value is monotonically increasing.

WalltimeWatermarkEstimator

It uses processing time as the estimated watermark state.



Now it’s time to try it all out. Let’s go!!