Apache Beam Proposal

Splittable DoFns for Go SDK

Daniel Oliveira (danoliveira@google.com)
https://s.apache.org/beam-go-sdf
@Dev list discussion

Context

This doc proposes an implementation of Splittable DoFns for the Go SDK. It assumes familiarity with the idea of Splittable DoFns and the concepts involved in implementing it, as laid out in the original Splittable DoFn proposal. This doc also assumes some familiarity with portability in Beam which is how all pipelines are run in the Go SDK. A starting point for learning about portability is the Apache Beam Fn API Overview.

Terminology and Abbreviations

- SDF - Splittable DoFn.
- UDF - User-Defined Function.
- Fn API - The protocol buffer API that enables portable pipelines in Beam.
- Restriction - A description of a portion of the work needed to process an element in an SDF. See the Splittable DoFn proposal for more information.
- Block - An unsplittable portion of work identified by a position within a restriction.

Goal

This document aims to propose a functioning implementation of **bounded** SDF for **batch** pipelines that can later be expanded to support the full set of features. At the time of writing this document, some features of SDF are not yet supported by the Fn API and so cannot be supported. Therefore rather than support a full implementation for SDF, this implementation aims to support a smaller subset of features with the intent to expand it later.

Overview

To implement SDFs in the Go SDK requires changes to three largely independent layers. The changes are described in more detail in later sections, but a rough overview is as follows:

1. User-Facing API: This covers the interface that a Beam user would use to write an SDF using the Go SDK. In addition to describing how a user would write an SDF, it also

covers how that user-defined SDF gets converted to the Go SDKs internal pipeline representation.

2. Serialization: This covers changes that would be made to the process of serializing the Go SDKs internal pipeline representation to the protobufs used in the Fn API pipeline representation. The goal is to correctly serialize SDFs so that runners with SDF support can recognize and handle the SDF.
3. Execution: This covers functionality that needs to be added to the Go SDK Harness to allow it to execute SDFs. This includes correctly deserializing user-defined SDFs and using them to execute the SDF component transforms as defined by URNs in the Beam Runner API.

User-Facing API

Requirements

The user-facing API is the interface allowing users of the Go SDK to write SDFs. Rather than borrowing existing implementations for the user-facing API from the Java SDK or Python SDK, the API for Go is intended to be implemented to fit the expectations of the Go language and remain consistent with the rest of Beam's Go SDK. In order to be complete, the user-facing API must fulfill the requirements below.

Support Existing DoFn Functionality

SDFs are a subset of DoFns and therefore should still support features that all DoFns are expected to have. Primary among these being support for all DoFn methods (Setup, StartBundle, ProcessElement, FinishBundle, Teardown) and optional parameters such as context, window information, or event time.

Support User-Defined Restrictions

The term "restriction" refers specifically to data describing a portion of the work needed for processing an element passed to the SDF. The kind of work that is described by a restriction can vary largely, so the SDF API must allow users to be able to define and use custom restrictions based on users' needs. The only requirements that must be enforced on these restrictions are those required by the SDF model itself as described in the Splittable DoFn proposal.

Support Restriction Tracking

The SDF API must allow users to define behavior for tracking the amount of work that has been processed within a restriction while processing an element (i.e. the current position).

Splitting Restrictions

Users must be able to define behavior for splitting a restriction into two or more additional restrictions, based on desired split points or user-defined initial splits to take advantage of liquid sharding in runners that support it.

Associate a Restriction with a Written SDF

There must exist some way to associate a type of restriction and its associated behavior with an SDF, so that when the SDK harness executes the SDFs it will be able to provide the correct kind of restriction that the UDFs expect.

Omissions

This implementation targets a minimal working implementation for batch pipelines, so several SDF features have been intentionally omitted for now, with plans for later implementation. The most prominent omissions are:

- `ProcessContinuation`: This is unnecessary for the initial implementation. Can later be added as an optional return value in an SDF's `ProcessElement` method.
- Handling Watermarks: Watermarks are currently unimplemented in the Go SDK. Watermarks support is a prerequisite for streaming support for SDFs. The API for watermark support in SDFs is currently undecided.

Approach

Writing user-defined SDFs is done by implementing two interfaces - a restriction tracker and a restriction provider - and writing the processing code in a DoFn. These two interfaces are modeled after the corresponding interfaces in the Python SDK ([1], [2]) and Java SDK ([1], [2]) and therefore should be somewhat familiar to users from those SDKs.

The restriction tracker is an interface for defining a restriction and tracking its progress, and is intended to cover behavior primarily used when processing elements of the SDF. A restriction tracker can be paired with each element to describe what portion of that element to process and how much has already been processed.

The restriction provider on the other hand is an interface containing behavior needed for an SDF as a whole. This includes how to create a new restriction tracker for an element, as well as how to perform a single split or initial splitting over an existing restriction.

In order to associate these interfaces with a DoFn, there needs to be some method in that DoFn to indicate that the DoFn is splittable, and what RestrictionProvider to use. A DoFn that is marked as splittable must also adhere to various requirements in its implementation in order to function correctly as an SDF, the details of which are described below.

Implementation

A proposed implementation of the interfaces described above is as follows:

```go
package sdf

type RTracker interface {
	TryClaim(pos interface{}) (ok bool, err error)
	TrySplit(fraction float64) (residual RTracker, err error)
	GetProgress() float64
	IsDone() bool
}

type RProvider interface {
	CreateInitialTracker(element interface{}) RTracker
	InitialSplits(element interface{}, rt RTracker) []RTracker
	RestrictionSize(element interface{}, rt RTracker) float64
	RTrackerType() reflect.Type
}

type SplittableDoFn interface {
	RProvider() RProvider
}
```

And to create an SDF a DoFn would be written with methods similar to the following:

```go
func (fn *FooDoFn) RProvider()

func (fn *FooDoFn) ProcessElement(in Foo, restriction RTracker, emit func(out))
```

The following sections will explain the above interfaces in more detail and provide a practical example illustrating their usage.

Restriction Tracker

The `RTracker` interface requires users to implement several methods around their user-defined restrictions in order to fulfill the SDF specification. The underlying user-defined restriction can be any arbitrary internal implementation, assuming that the requirements of restrictions are followed according to the SDF model. Specifically, the blocks of work in a restriction must have a strict ordering so that they can be processed sequentially in monotonically increasing order.

The `RTracker` is also used in the SDF's `ProcessElement` method to ensure that claimed work is tracked. Using the `RTracker` interface to split and track progress also allows the details of multi-threading to be abstracted away from the user by wrapping the user's `RTracker` with another that handles those details. This is described in more detail below.

A restriction coder must be available if defining an `RTracker` to allow for serialization of the underlying restriction. This coder can be either generated or manually defined. The Go SDK already has default methods for generating coders which can be leveraged for this (ex. JSON encoding).

TryClaim

```
TryClaim(pos interface{}) (ok bool, err error)
```

Used in the SDF's `ProcessElement` method to claim a block of work in the restriction before performing that work and emitting outputs. Claims must be performed for every block in a restriction in monotonically increasing order. When a claim fails it indicates that the `ProcessElement` method must stop processing without emitting any more outputs.

The usage of this method is important for keeping the restriction synchronized between the execution of `ProcessElement` and the execution of split requests, which can modify the bounds of the restriction.

This declaration of `TryClaim` uses an empty interface as a position type to allow implementations of `RTracker` to accept whatever type best fits the restriction being used.

TrySplit

```
TrySplit(fraction float64) (residual RTracker, err error)
```

Attempts to split the restriction at a specific fraction of its unclaimed work. This method turns the current `RTracker` into the primary restriction by changing its endpoint as appropriate, and returns a new `RTracker` representing the residual. This method gets called by the SDK harness during a split request, if the split must happen along a currently executing restriction.

GetProgress

```
GetProgress() float64
```

Returns the fraction of work completed on the restriction as a fraction from 0 (no work complete) to 1 (all work complete). This method is used by the SDK harness to check on progress when performing splits or progress reporting.

IsDone

```
IsDone() bool
```

Checks whether all work in the restriction has been claimed. This method is mainly used for pipeline validation, so that incorrectly written `ProcessElement` methods will cause errors for users instead of silently producing incorrect output.

Restriction Provider

The `RProvider` interface allows defining the interactions between the input elements of an SDF and the `RTrackers` associated with those elements. These methods are mainly needed for the SDK harness to be able to execute transforms used by the SDK Harness to execute SDFs. After being defined, an `RProvider` implementation is associated with a DoFn instance, which means that all elements in a DoFn's main input will get passed into the `RProvider` methods when the methods get used.

It made sense to make these methods separate from `RTracker` because the methods in `RProvider` are more dependent on a user's specific needs compared to those in `RTracker`. `RTracker` implementations can often be generalized while `RProvider` implementations are would often be tailored to the specific SDFs being used.

Due to the lack of generics in Go, the elements in these methods are all declared as empty interfaces. Type assertion must be used to match the type with that of the DoFn's input. It is the pipeline author's responsibility to avoid type mismatch errors by ensuring that the type assertions match the actual input types being used. The following is an example of using type assertions in a `RProvider` expecting string inputs:

```
func (*FooRProvider) InitialSplits(element interface{}, tracker RTracker)
[]RTracker {
        stringElement :=  element.(*string)
        ...
}
```

There will most likely be differences between the elements passed in here compared to the same elements passed to the DoFn methods like `ProcessElement`. This is due to the fact that the methods in `RProvider` lack the runtime analysis performed on `ProcessElement` in order to detect the input types. Any differences a user may encounter between the elements input to `RProvider` vs `ProcessElement` must be well-documented for users.

CreateInitialTracker

Creates an `RTracker` for an entire element. This method gets called on all inputs to a Splittable DoFn before `ProcessElement` gets called.

InitialSplits

Allows users to split the initial restrictions created by `CreateInitialTracker` before they get processed. This is used for optimizing performance on runners that support liquid sharding, as it allows the processing of an element to immediately be split between several restrictions and be sent to different workers to process in parallel without having to wait for a split request from the runner.

RestrictionSize

Returns an estimate of the amount of work involved in processing a restriction. This size estimate is used as a weight when distributing work on runners that support liquid sharding with sized restrictions. Sizes have no unit of measurement, they only represent abstract weights relative to each other (ex. 200 means twice as much work as a 100).

RTrackerType

Returns the `reflect.Type` of the RTracker produced by the various methods in the `RProvider`. This method is used by the Go SDK and SDK harness to get the restriction coder used in an SDF. This method can be easily implemented similarly to the following example implementation:

```
return reflect.TypeOf((*FooRTracker)(nil))
```

Splittable DoFns

SplittableDoFn is an interface used for identifying DoFns as splittable and associating them with an RProvider. A DoFn is considered a Splittable DoFn if it implements all the methods of this interface and fulfills the following requirements:

1. The ProcessElement method has an RTracker parameter after the inputs.
2. The ProcessElement method only produces output via emit functions and not a return value.

An example of a valid DoFn:

```
type FooDoFn struct {
}

func (fn *FooDoFn) Splittable() RProvider {
        return new(FooRProvider)
}

func (fn *FooDoFn) ProcessElement(s string, tracker RTracker, emit func(out)) {
        fooTracker := tracker.(*FooRTracker)
        ...
}
```

Changes would need to be made to DoFn validation (in the following files: [1], [2]) to support checking whether a DoFn is splittable and, if so, checking that it is a validly constructed SDF.

Example SDF

This section presents an example SDF illustrating all the elements of the user-facing API that were described above. The example will focus on showing the usage of SDF API elements and omit or simplify code not directly relevant to SDF, such as file IO or error checking. Therefore, this example should not be used in actual pipelines.

The following code is an example of an SDF which will read in filenames of text files and output every line of that file as a string. The splitting will be done at byte offsets within the file.

Restriction Tracker

```
// OffsetRangeRTracker tracks a restriction  that can be represented as a range of
// integer values, for example for byte offsets in a file, or indices in an array.
type OffsetRangeRTracker struct {
        start, end int64 // Half-closed interval with boundaries [start, end).
        claimed    int64 // Tracks the last claimed position.
        stopped    bool  // Tracks whether TryClaim has already indicated to stop
                         // processing elements for any reason.
}

func NewOffsetRangeRTracker(start, end int64) *OffsetRangeRTracker {
        return &OffsetRangeRTracker{
                start: start,
                end: end,
                claimed: start - 1,
                stopped: false,
        }
}
```

`OffsetRangeRTracker` is a very basic restriction tracker that represents a restriction as a range of integer values. For this example the integer values will represent byte offsets in a file. Since a block of work in this SDF is one line of text, but the restriction measures byte offsets, it is likely that the range of a restriction will not always line up perfectly with blocks of work, which is something that must be accounted for when using this restriction tracker in an SDF.

`NewOffsetRangeRTracker` is a helper method that will be used below, for quickly creating a properly-initialized `OffsetRangeRTracker`.

```
// TryClaim accepts an int64 position and successfully claims it if that position
// is greater than the previously claimed position and within the restriction.
// The tracker considers a restriction finished processing once the user tries to
// claim a position at or past the end of the restriction.
func (tracker *OffsetRangeRTracker) TryClaim(rawPos interface{}) (bool, error) {
	if tracker.stopped == true {
		return false, errors.New(
		    "Cannot claim more work after TryClaim returns false.")
	}

	pos := rawPos.(int64)
	if pos <= tracker.claimed {
		tracker.stopped = true
		return false, errors.New(
		    "Claim must be greater than previously claimed position.")
	}

	tracker.claimed = pos
	if pos >= tracker.end {
		tracker.stopped = true
		return false, nil // Done processing restriction, not an error.
	}
	return true, nil
}
```

This implementation of `TryClaim` accepts positions as int64 values to be compatible with `OffsetRangeRTrackers`. This example illustrates error checking for requirements of the `TryClaim` method; returning errors if the position claimed is not monotonically increasing, or if a claim attempt occurs after TryClaim indicates to stop. Allowing `TryClaim` to return errors is intended to let users differentiate between stopping due to successfully processing a restriction versus stopping because an error was encountered. It also lets users add in-depth error checking to their SDFs.

This `TryClaim` implementation expects users to continue claiming positions until one is claimed at or past the end of the restriction, at which point it indicates to finish processing. This approach is not mandated by the requirements of `TryClaim` but selected for this implementation for ease of use. In other implementations it may make more sense to have different ways to detect when a restriction is finished processing.

```
// TrySplit splits at the nearest integer greater than the given fraction of the
// remainder.
func (tracker *OffsetRangeRTracker) TrySplit(fraction float64) (RTracker, error) {
	if tracker.stopped || tracker.IsDone() {
		return nil, nil
	}
	if fraction < 0 || fraction > 1 {
```

```
                return nil, errors.New("Fraction must be in range [0, 1]")
        }

        splitPt := tracker.start + int64(fraction * float64(tracker.end -
tracker.start))
        if splitPt == tracker.end {
                return nil, nil
        }
        residual := NewOffsetRangeRTracker(splitPt, tracker.end)
        tracker.end = splitPt
        return residual, nil
}
```

The `TrySplit` method is straightforward, splitting the restriction at the given fraction of the remainder.

```
// GetProgress reports progress as the claimed fraction of the restriction.
func (tracker *OffsetRangeRTracker) GetProgress() float64 {
        fraction := float64(tracker.claimed - tracker.start) /
                float64(tracker.end - tracker.start)
        return math.Min(fraction, 1.0)
}
```

`GetProgress` is straightforward, returning the fraction of work remaining (with some clamping to keep the fraction between 0 and 1).

```
// IsDone returns true if the most recent claimed element is past the end of
// the restriction.
func (tracker *OffsetRangeRTracker) IsDone() bool {
        return tracker.claimed >= tracker.end
}
```

`IsDone` takes the simple approach and checks if the entire range of the restriction has been claimed. In comparison, less simple `RTracker` implementations could check whether each individual block of work has been claimed.

Restriction Provider

```
type ReadLinesRProvider struct {
}

// CreateInitialTracker takes a string filename of a text file and returns an
// RTracker representing the entirety of that file as a size in bytes.
```

```go
func (provider *ReadLinesRProvider) CreateInitialTracker(element interface{})
RTracker {
        filename := element.(string)
        stat, _ := os.Stat(filename)
        size := stat.Size()
        return NewOffsetRangeRTracker(0, size)
}
```

`CreateInitialTracker` is straightforward. Of note is the usage of a type assertion to access the concrete element, which is necessary for elements and `RTrackers` passed to `RProvider` methods.

```go
// InitialSplits splits each initial restriction in two.
func (provider *ReadLinesRProvider) InitialSplits(element interface{}, rt RTracker)
[]RTracker {
        rtImpl := rt.(*OffsetRangeRTracker)
        mid := ((rtImpl.end-rtImpl.start) / 2) + rtImpl.start

        splitRt := make([]RTracker, 2)
        splitRt = append(splitRt, NewOffsetRangeRTracker(rtImpl.start, mid))
        splitRt = append(splitRt, NewOffsetRangeRTracker(mid, rtImpl.end))
        return splitRt
}
```

Here `InitialSplits` always splits each restriction in two in order to serve as an example implementation of this method. Actual implementations can take more dynamic approaches, such as splitting restrictions so that each restriction is no larger than a specified size. Alternatively, if it is known that the job will not be on a runner that supports liquid sharding, then this method could merely be a stub and return the restriction unchanged.

```go
// RestrictionSize returns a size estimate of the restriction as the size of the
// restriction in bytes.
func (provider *ReadLinesRProvider) RestrictionSize(element interface{}, rt
RTracker) float64 {
        rtImpl := rt.(*OffsetRangeRTracker)
        return float64(rtImpl.end - rtImpl.start)
}
```

For this example the size estimate returned by `RestrictionSize` is simply the size in bytes of the restriction. This is reasonable for an SDF reading a text file where the amount of work involved is a linear correlation to the size of the file. But in other SDFs, the work may not linearly correlate with the size of the restriction, and in those cases the implementation of this method would be more complex to get a more accurate estimation.

```
// Returns the reflect.Type of the RTrackers used by this RProvider.
func (provider *ReadLinesRProvider) RTrackerType() reflect.Type {
	return reflect.TypeOf((*OffsetRangeRTracker)(nil))
}
```

`RTrackerType` is very straightforward and shows the suggested approach of calling `reflect.TypeOf` on a nil pointer to the `RTracker` being used in the `RProvider`.

DoFn

```
// A splittable DoFn that reads a text file and outputs each line.
type ReadLinesDoFn struct {
}

// Mark this DoFn as splittable with an RProvider for reading text files.
func (fn *ReadLinesDoFn) RProvider() RProvider {
	return &ReadLinesRProvider{}
}

// For each filename, open the file and read it, outputting each line.
func (fn *ReadLinesDoFn) ProcessElement(filename string, rt RTracker, emit func(out
string)) error {
	rtImpl := rt.(*OffsetRangeRTracker)

	file, _ := os.Open(filename)
	var scanner *bufio.Scanner
	var pos int64
	if rtImpl.start > 0 {
		// Find the start of the first line within the restriction.
		file.Seek(rtImpl.start - 1, 0)
		scanner = bufio.NewScanner(file)
		scanner.Scan()
		pos = rtImpl.start - 1 +  int64(len(scanner.Bytes()))
	} else {
		// Restriction starts at the beginning of a file, no seek needed.
		scanner = bufio.NewScanner(file)
		pos = 0
	}

	var ok bool
	var err error
	for ok, err = rtImpl.TryClaim(pos); ok == true; {
		scanner.Scan()
		emit(scanner.Text())
		pos += int64(len(scanner.Bytes()))
```

```
        }
        if err != nil {
                return err
        }
        return nil
}
```

ReadLinesDoFn is where all the code above comes together and actually forms an SDF. The DoFn is marked as splittable by giving it the RProvider method and indicating our previously written provider as the one to use.

The ProcessElement method is where the actual splittable work occurs. Of particular note is the way that block length is handled. Blocks of work begin at the first byte of a line, so if a restriction begins anywhere other than the start of the file, the method must seek to a spot slightly before the restriction and use the scanner to seek to the beginning of the next line. Meanwhile, the method keeps track of what position to claim with the pos variable, updating it as lines are read.

The loop for claiming and processing work follows all the requirements of using TryClaim. It always attempts to claim work before producing any output, only processes claimed blocks, and immediately stops work and exits ProcessElement once a claim fails, which is expected to happen when the loop attempts to claim a block beyond the end of the restriction.

Open Questions

There are still some open questions in the API proposed above, either because the issues in question are not vital for an initial implementation, or because additional feedback is needed before making a final decision.

Merged or Split Restriction Trackers

Should the API split restriction trackers and restrictions as concepts, or keep them merged?

In the current proposal, RTrackers encompass both the concept of a restriction (a range of work, with start and end boundaries) and the restriction tracker (behavior that tracks the claims made on work inside a restriction). This differs from the other SDKs which make a distinction between the two.

Merged: The current merged approach was selected largely because it simplified the user-facing API, despite being less consistent with the SDF model semantically. The downside of this approach is that an optimal restriction coder would need to avoid encoding non-restriction fields (i.e. only encode the start and end boundaries). It is also slightly more difficult to explain to users how the RTracker is used since the tracking parts and the restriction are used differently.

Separate: Keeping the restriction tracker and restriction separate are more semantically correct and easier to communicate to users. The downside of this approach is that it would add some complexity to the API. Users would need to be able to associate restriction trackers/objects to their SDF, and be able to add and retrieve restriction objects from restriction trackers. The SDK harness would also gain the responsibility of creating the correct restriction trackers from restriction objects and providing those trackers to DoFns (currently this occurs implicitly). Ultimately this somewhat increases the complexity of the API, but would make the API more semantically consistent with the model.

Thread-safe Restriction Tracker Wrapper

How to go about implementing a wrapper around RTrackers to handle concurrency?

`RTracker` needs to be thread-safe, especially with the `TryClaim`, `TrySplit`, and `GetProgress` methods. `TrySplit` is expected to be called while `ProcessElement` uses `TryClaim` in a separate thread. The Java SDK currently handles this concurrency behind the scenes by wrapping restriction trackers to add thread safety and then delegate to the user-written methods. This also allows additional correctness checks such as checking that no output is produced after `TryClaim` returns false, or that there isn't a subsequent call to `TryClaim` after one that returns false. This can greatly reduce boilerplate error checks in user code.

That approach is desirable, but more difficult in the current API because users expect to be able to cast `RTrackers` to their concrete type to view the restriction boundaries.

Possible solutions:

1. Add the thread-safe wrapper to the API and pass the user's tracker already wrapped into `ProcessElement`, informing users that they are required to unwrap their tracker to inspect it.

```
func (fn *FooDoFn) ProcessElement(i int, rt RTracker, emit func(out int)) error {
        rtImpl := rt.(*WrappedRTracker).Unwrap().(*FooRTracker)
```

2. Make thread-safety the user's responsibility. Add an optional thread-safe wrapper to the API, but don't automatically wrap the trackers for the user. User must wrap trackers themselves in `CreateInitialTracker`.
3. Split the restriction object and restriction tracker so that users no longer need to cast restriction trackers to inspect restriction boundaries. This is the approach the Java SDK takes.

Alternative Considered

Because of the lack of generics in Go, the implementation above requires users to cast the elements and restriction trackers used as inputs to the UDFs. A similar issue was avoided for the inputs to DoFns by dynamically validating the method names and signatures on any interface that a user is using as a DoFn. This essentially allows any arbitrary struct to be used as a DoFn with method signatures containing the already cast inputs.

A similar approach was considered for the user-facing SDF API. With that approach no more casting would be required. For example:

Proposed Implementation:

```
func (*FooRProvider) InitialSplits(element interface{}, tracker RTracker)
[]RTracker {
        fooTracker := tracker.(*FooRTracker)
        stringElement :=  element.(*string)

        ...
}
```

Alternative:

```
func (*FooRProvider) InitialSplits(element string, tracker FooRTracker)
[]FooRTracker {
        ...
}
```

Essentially this approach would treat the restriction provider similarly to a DoFn. Any struct could be used as a restriction provider, and the method names and signatures would be analyzed to ensure that types are consistent everywhere, including within any DoFns that the restriction provider is used with.

This approach is more user-friendly and consistent with the current state of the Go SDK, however it requires a far more complex implementation to perform the dynamic analysis and validation, such as what currently exists for DoFns ([1], [2]). This approach would take longer to implement, would require additional testing due to its complexity, and would be less performant by default due to requiring reflection or code generation to execute methods.

Ultimately the simpler implementation was preferred because it allows for a faster implementation of SDF, and if it is later decided that this alternative is better, it is easier to go from a simple implementation to a more complex one than vice versa.

Serialization

Overview

SDFs must be identifiable to runners as splittable in order for runners to correctly treat them as SDFs. This is done by serializing SDFs as ParDos and setting the following fields in ParDoPayload: `splittable` and `restriction_coder_id`. In addition to those fields, the contents of the `do_fn` field in SDFs can differ from those in non-splittable ParDos if desired.

Implementation

Serialization of pipelines from the Go SDK representation to the proto representation occurs in translate.go. The translation of ParDos will need to be modified to check if the DoFn within the ParDo is splittable, and if so set the necessary fields when serializing it.

The DoFn can be identified as splittable by performing a type assertion with the DoFn used in the ParDo to check if it fulfills the `Splittable` interface. This can be done either when creating the edge with the SDF and storing the result, or by checking during the translation into protos.

The `restriction_coder_id` field is somewhat more difficult to serialize since it requires retrieving the SDF's associated `RProvider` and then retrieving the associated `RTracker`'s type. This is easily done by calling the `RTrackerType` method on the `RProvider` and is the reason for the existence of said method. Once the type is retrieved, it can be used to retrieve the restriction coder followed by the coder ID.

Finally, the `do_fn` field can optionally have a different payload if needed for SDFs, but as of yet no changes are needed and the current payload for standard DoFns should continue to work for SDFs.

Execution

Overview

The execution of SDFs is the responsibility of the SDK harness. After an SDF has been serialized and sent to the runner as part of a pipeline description, the runner may expand the transform in a number of ways and then send the expanded SDF to the runner harness as part of bundles. These expanded transforms are identified via URNS from `SplittableParDoComponents` in the Beam Runner API.

In order to support the execution of SDFs, the Go SDK harness must implement transforms for all those URNs that are required by SDF-capable runners, and support splitting and progress reporting on elements currently being processed.

SplittableParDoComponents URNs

At the time of writing this document, the URNs under `SplittableParDoComponents` includes some soon-to-be deprecated URNs that may not be present in the initial implementation. The following URNs will be targeted for the initial implementation:

- `PAIR_WITH_RESTRICTION`: This transform uses `CreateInitialTracker` from the `RProvider` to pair incoming elements to the SDF with their initial restrictions.
- `SPLIT_AND_SIZE_RESTRICTIONS`: This transform uses `InitialSplits` and `RestrictionSize` from the `RProvider` to perform initial splits on the initial restrictions and get size estimates for each of the split restrictions.
- `PROCESS_SIZED_ELEMENTS_AND_RESTRICTIONS`: This transform actually processes an SDF, similarly to how a ParDo transform would process a non-splittable DoFn. This transform is the only one where split requests from another thread may call `TrySplit` on `RTrackers` being processed in the execution thread.
- `SPLIT_RESTRICTION`: This transform is simply a version of `SPLIT_AND_SIZE_RESTRICTIONS` listed above, but without size estimation. This transform is likely to be deprecated, but will need to be implemented if the initial implementation is done before the deprecation.

Implementation

URN Execution

The URNs needed for SDFs must be handled in translate.go similarly to how Combines are currently handled ([1], [2]). If the transform is detected to have one of the SDF URNs, then the payload must be read to retrieve the data needed to perform the SDF (in particular to be able to construct restriction providers), and an executor created to handle the actual execution of the URNs.

`PAIR_WITH_RESTRICTION` and `SPLIT_AND_SIZE_RESTRICTIONS` make use of the user-written `RProvider`, so the `RProvider` method in the SDF must be used to create an instance of the user's `RProvider`, and then its methods may be called.

`PROCESS_SIZED_ELEMENTS_AND_RESTRICTIONS` functions nearly identically to an executor for ParDos with the only addition being the handling of the `RTracker` parameter. The `RTrackers` are already paired with input elements due to the previously used transforms, so the only additional work the executor needs to do is extracting the `RTracker` from the KV and passing it into the DoFn's `ProcessElement` method.

Split Requests

`ProcessBundleSplitRequests` from the runner are already partially supported in the Go SDK. The Go SDK harness can currently split along element boundaries, but cannot yet split within an element due to lacking SDF support. The split request handling code is found in harness.go, plan.go, and datasource.go, all three of which may require some changes.

To expand the existing code to support SDFs, first the calculation to determine the split point must also be made to check if the split point lies within the currently processing element and at what remainder of that element. Then, the executor for the data source must somehow be able to retrieve the restriction tracker of the currently processing element and call `TrySplit` on it.

The elements in an SDF are processed in a different thread than the one handling the split request, so an important part of supporting SDF splits is the implementation of some form of thread-safety on the `RTracker`. A more in-depth discussion on this can be found above.

Progress Reporting

`ProcessBundleProgressRequests` from the runner are already supported in the Go SDK for element counts. With SDFs, this implementation must be expanded to also cover sub-element progress. This existing code is, just like the splitting code, found in harness.go, plan.go, and datasource.go.

To get sub-element progress, the methods involved will need to retrieve the restriction tracker of the currently processing element and call `GetProgress` on it to retrieve the fraction of work completed.

Similarly to the splitting code, some form of thread-safety should be present on the `RTracker` when retrieving progress.