Authoring Apache Beam IO Transforms

JIRA issue: [BEAM-1025] (User guide - How to create Beam IO Transforms)
Status: in progress - this doc is still being written

This document has been superseded by the pull request to add this document to the apache beam site.
https://github.com/apache/beam-site/pull/196

Document TODOs:

- discuss other mutation types (ie, Delete) - it is basically a write
- discuss more detailed design patterns, incl reading byte ranges in files
- discuss BigQueryIO "export to file" design pattern (and when does that make sense)
- Write section is bare bones

Introduction

This is a guide to implementing read and write transforms in the beam model. These transforms are the way that Beam pipelines import data so it can be processed and then write data to a store.

Reading/writing data in beam is just a parallel task like everything else. Sometimes you can assemble it from ParDo's, GBK's etc.; rarely you will need the more specialized Source and Sink classes for specific features. There are changes coming soon (SplittableDoFn, BEAM-65) that will allow Source to no longer be necessary.

As you work on your IO Transform, be aware that there are many examples, helper classes implementing common things like reading from files, and the Beam community is excited to help those building new IO Transforms.

Examples to follow

Currently, the implemented IO transforms show a variety of different styles. These are the suggested examples to follow:

- DatastoreIO - ParDo based database read, write, and good example of transform API

- BigtableIO - good test examples, demonstrates Dynamic Work Rebalancing

Suggested steps for implementers

1. Check out this guide and come up with your design. If you'd like, you can email the beam dev mailing list with any questions you might have.
2. If you are planning to contribute your IO transform to the beam community, you'll be going through the normal beam contribution life cycle - see the Apache Beam Contribution Guide for more details.
3. As you implement your tests, consult the test guidelines TODO: link to the testing doc or incorporate it into this doc.
4. As you're working on your IO transform, the Beam IO Transform Style Guide will provide specific information on many implementation details of transform. TODO: link to PTransform style guide. (for now, email the beam mailing list if you have questions.)

Read transforms

Read transforms take data from outside of the beam pipeline and produce PCollections of data.

For some data stores or files types where the data can be read in parallel, you can actually think of the process as a mini-pipeline. This will often consist of two steps: (1) Splitting the data into parts to be read in parallel and then (2) reading from each of those parts. Each of those steps will be a ParDo, with a GroupByKey in between. The reason for the Group By Key is an implementation detail but for most runners it allows the runner to use different numbers of workers for splitting (which will likely occur on very few workers) and reading (which will likely benefit from more workers.) For runners which support Dynamic Work Rebalancing, the GBK will also allow that to occur. TODO: the last 2 sentences will need to be discussed and I'm excited for suggestions :)

Here are some examples of read transform implementations using the "reading as a mini-pipeline" model to use when data can be read in parallel:

- **Reading from chunks of a file** - if you have a file format that can be sub-divided, the structure will look like:
    - Determine Byte Ranges ParDo: As input, takes in a file name. Emit a PCollection of byte ranges inside of the file.
    - Read Byte Range ParDo: Given the PCollection of byte ranges, and reads each byte range and reads the data in that byte range, emitting a PCollection of records.
- **Reading from a file glob** (eg "~/data/**") -

    - Get File Paths ParDo: As input, take in a file glob. Emit a PCollection of strings, each of which is a file path.
    - Reading ParDo: Given the PCollection of file paths, read each one, emitting a PCollection of records.
- **Reading from a NoSQL Database** (eg Bigtable or HBase) - these databases often allow reading from ranges in parallel.
    - Determine Key Ranges ParDo: As input, receive connection information for the database and the key range to read from. Output a PCollection of key ranges that can be read in parallel efficiently.
    - Read Key Range ParDo: Given the PCollection of key ranges, read the key range, emitting a PCollection of records.

For data stores or files where reading cannot occur in parallel, this is a simple task that can be accomplished with a single ParDo. For example:

- **Reading from a database query** - traditional SQL database queries often can only be read in sequence. The ParDo in this case would establish a connection to the database and read batches of records, emitting a PCollection of those records.
- **Reading from a gzip file** - a gzip file has to be read in order, so it cannot be parallelized. The ParDo in this case would open the file and read in sequence, emitting a PCollection of records from the file.

When do you need the Source class?

The above discussion is in terms of ParDos - this is because Sources have proven to be tricky to implement. **At this point in time, it's not recommended that you implement a Source if you can get away with a ParDo.** If you're trying to decide between the two, feel free to email the beam dev mailing list and we can discuss the specific pros and cons of your case.

In some cases implementing a Source may be necessary or result in better performance.

- ParDos will not work for reading from unbounded sources - they do not support checkpointing (TODO: maybe now possible with state API?) and don't support mechanism like de-duping that have proven useful for streaming data sources.
    - TODO: double check this information.
- ParDos cannot provide hints to runners about the size of data they are reading or progress - without size estimation of the data or progress on your read, the runner doesn't have any way to guess how large your read will be, and thus if it attempts to dynamically allocate workers, it does not have any clues as to how many workers you may need for you pipeline.
- ParDos do not support Dynamic Work Rebalancing - these are features used by some readers to improve the processing speed of jobs (but may not be possible with your data source.)

- ParDos do not receive 'desired_bundle_size' as a hint from runners when performing initial splitting.

Write transforms

Write transforms are responsible for the work of taking the contents of a PCollection and transferring that data outside of the Beam Pipeline.

A few examples of write transforms:

- **Writing records to a database** - a single ParDo: write each record, 1 record per RPC.
    - TODO: discuss connection management (setup/teardown), batching, exactly-once, etc…
- **Writing bounded data to files**, multiple records per file. Use FileBasedSink.

TODO: this needs more thought/discussion

When do you need the Sink class?

You are strongly discouraged from using the Sink class unless you are creating a FileBasedSink. Most of the time, a simple ParDo is all that's necessary.