Beam and OpenLineage

Proposal by Charles Nguyen (phucnh402@gmail.com)
Created: Oct 14, 2025 Updated: Oct 20, 2025

Context

Overview

Data lineage helps users understand how their data came to be. Beam currently has limited support for it, and with little to no documentation on the feature. We want to address these gaps by adding broader data lineage support for Beam's local runners and integration with OpenLineage's open standard [5] for lineage collection.

Background

Currently, what Beam supports for data lineage is asset level lineage which tracks the relationships between entire datasets. In particular, the lineage information is emitted as metrics from a few supported IO Transforms [6]. However, the capability to construct and visualize the lineage graph from these metrics depends on the runner backend, and is currently done in other services such as Google Dataflow and Dataplex for Dataflow runner.

Note that Beam does have the functionality to render and export pipeline graphs (see Beam Playground). Specifically, the Python SDK has `PipelineRender` and the Java SDK has `PortablePipelineDotRenderer` that takes in a portable pipeline proto and produces Graphviz DOT graph object. However, we're more interested in asset level lineage: how to use these lineage metrics to build out the relationship between IO sources and sinks, and not concern so much about individual PTransforms in the graph.

This feature is originally scoped across several Github issues #33980, #339381, #339382 [1]. This proposal evolves from a previous proposal [10] that was written for this same feature but in a different context. It is rewritten/updated here to be more relevant.

Goal

We want to support asset level lineage for one or a few Beam's local runners, namely Python and Java Direct Runner, and most notably Prism Runner.

As the lineage metrics are collected from the IOs in the pipeline, we build out a lineage graph by traversing the job graph, using this lineage information and forming pairs of sources and sinks.

The next step is to integrate Beam with OpenLineage and expose this lineage graph according to OpenLineage's format. Any observability system that accepts OpenLineage format can consume Beam's lineage graph to visualize it. To showcase this work end-to-end, Marquez and/or Google Cloud Dataplex will be used to collect and view the lineage graph [11][12].

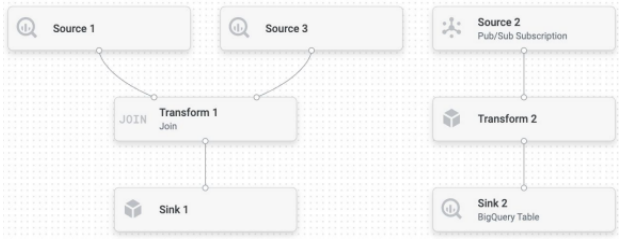
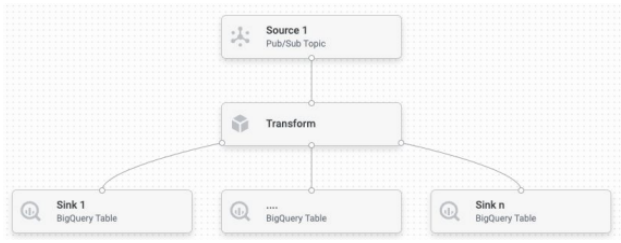
Main

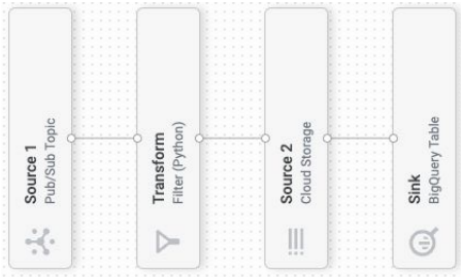
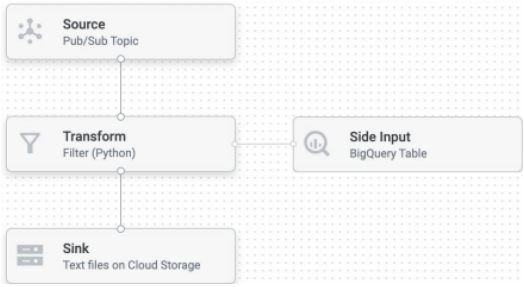
Lineage Graph

Python and Java SDKs have Lineage API ([8], [9]) that are used in IO Transforms to add lineage information as a metric. In particular, this asset lineage information is added as a BoundedTrie metric (used to be StringSet) that uses these respective data structures to store. See for instance [7] for how BigQuery IO uses this API.

Like any other metric, lineage metric is scoped to the transform that emits it. To build out a lineage graph, Dataflow backend does this by traversing the pipeline graph, linking the emitted lineage info together to form source-sink pairs. The proposal is for Beam's similar capability to track data lineage in a local runner(s).

There are a few special instances of Beam's job graph that we have to be aware of ([2]):

Cases	Beam Job Examples
Multi Sources and Sinks We have some fixed number of sources and sinks, but together they form several connected components.	
Dynamic Destinations We have a variable number of tables that the pipeline can write to, e.g. depending on the events received from PubSub subscription, we may write to BigQuery tables 1, 2, ..., n.	

Dynamic Reads The data read from IO source can change, leading to a dynamic pipeline. Consider the following pipeline that listens to events from PubSub subscription, gets some file name of interest from the event data, reads the file from Cloud Storage bucket and stores the data in BigQuery table. Note that the Cloud Storage IO source is a middle node!	
Side Inputs Beam supports a few side input patterns, where a `PTransform` can make calls to some external service/IO to enrich data, before writing it to a sink.	

These cases should influence how we traverse the job graph, e.g. since an IO source can be a middle node, the traversal should therefore start at every sink and find all sources along the path. Due to the fact that the job graph can be dynamic, the lineage graph can therefore change throughout the lifetime of the job as well. This is more relevant if we do decide to support capturing the live status of the pipeline in the lineage graph.

Python & Java Direct Runners

Lineage API uses the underlying existing Metrics API to add lineage information as metric. In the context of Direct Runner, the Metrics API's implementation is not runner-SDK agnostic and therefore each combination of runner + SDK requires separate implementation to query metrics. To get a good mental model around the implementation, refer to the Metrics Architecture [3] and Metrics API [3] (Note that this is prior to Fn API). Consequently, the lineage graph traversal implementation for Direct Runner would require duplicating efforts for each SDK.

Lineage metrics are keyed by step (transform) name and metric name. The lineage graph traversal for Python and Java Direct Runners will be implemented with this logic in order to properly link lineage info together.

Prism Runner

Prism Runner is built based on Beam's portability framework. As part of that framework, new work was done on adding/reporting metrics over Fn API layer [4] and how to query them over Job API [4].

Metric is now defined in `MonitoringInfo` proto and scoped according to its key-value labels (which includes transform id, metric name, etc...). The lineage graph traversal will operate on the pipeline proto and will be implemented with this logic in order to properly link lineage info together. There are two approaches on where this implementation will be...

Implement on SDK-side

The `GetJobMetrics` RPC (defined as part of Job API) is used to query metrics (and by extension, the lineage information) from Prism job server. Each SDKs has client implementation that makes a call to this RPC, and at the same time also has the pipeline proto. Our implementation can go here on the SDK-side, but that would obviously defeat the purpose of the portability framework and require duplicating efforts across the SDKs.

Implement on Runner-side

Our implementation can reside in `sdks/go/runners/prism/internal`, and lineage graph can perhaps be exposed through a new RPC defined as part of Job API.

Prism Runner is in active development, and this body of work may require iterations to gain more experience, which may be best done for Direct Runners as a start. But as we transition to Prism Runner being the default local runner, the ultimate goal of this work is to support Prism Runner as well.

Beam + Open Lineage

OpenLineage [5] is the open standard for data lineage collection and analysis, providing both specification and APIs for collecting lineage metadata. It offers a standard model with facets (pieces of metadata) as building blocks to define the running pipeline and its events. The model is as follow:

- Job: a process that consumes and produces datasets (which are defined as the Job's inputs and outputs). In the context of Beam, a Job will be the "edge" of the source-sink pairs in the lineage graph. It will represent all the transforms applied from the source to the sink. The input and output will therefore be the IO source and sink. See [5] for OpenLineage Job facet.

- Run: an instance of a Job. It is used to observe changes in Jobs between their instances. In the context of Beam, this will probably not be relevant much if we are not capturing the live status of the running pipeline. See [5] for OpenLineage Run facet.

- Dataset: a representation of a collection of data. In the context of Beam, this corresponds to the IOs. If the IO PTransform produces a PCollection with schema, then it can be specified as well in the facet for Dataset. See [5] for various OpenLineage Dataset facets.

In terms of implementation, these facets are merely JSON specs and OpenLineage provides the client library with APIs to build these facets. Once the lineage graph is built from the traversal and according to the specs, we create an `OpenLineageClient` client instance to emit lineage events. OpenLineage has an ecosystem of consumers that come built-in with dashboards for visualization.

Using the open-source consumer Marquez and its dashboard is a canonical example of showcasing this end-to-end integration [11]. Once an HTTP client is set up to POST these OpenLineage events, Marquez service running locally can then collect the events and visualize them as a lineage graph. Google Cloud Dataplex is also a natural option that allows either making HTTP requests or RPC calls to post OpenLineage-compatible events to the Dataplex service [12].

Note that OpenLineage officially offers Python and Java client libraries only. In the case of Prism Runner, there's a Go client library [13] that is not maintained by the project.

Documentation

More documentation on the feature is part of the goal. SDKs already offer a way for IO to emit lineage information, but the fact is currently not documented officially on Beam's site aside from a Beam Summit Talk [2] (Dataflow/Dataplex is currently the only backend that consumes this information, and is more mentioned there). As we onboard Beam's own local runner to support lineage tracking, the intention is to have more visibility into this feature through Beam's site documentation and/or blog write-up.

References

[1] Feature's GitHub Issues - issues/33980, issues/33981 and issues/33982

[2] Beam Summit Talk on Data Lineage - https://beamsummit.org/sessions/2024/data-lineage/

[3] Metrics Architecture & API Before Fn API - https://cwiki.apache.org/confluence/display/BEAM/Metrics+architecture+inside+the+runners and User Defined Metrics API

[4] Metrics Architecture & API After Fn API - Apache Beam Fn API : Defining and adding SDK Metrics and Apache Beam Fn API: Get Metrics API: Metric Extraction via proto RPC API.

[5] OpenLineage - https://openlineage.io/docs

[6] Supported IO Transforms - https://cloud.google.com/dataflow/docs/guides/lineage#limitations

[7] Lineage API Example - https://github.com/apache/beam/pull/32116/files

[8] Lineage API Python SDK - https://github.com/apache/beam/blob/1cd03d36a2581135adf1eb7f77c010532a7535c4/sdks/python/apache_beam/metrics/metric.py#L337

[9] Lineage API Java SDK - https://github.com/apache/beam/blob/1cd03d36a2581135adf1eb7f77c010532a7535c4/sdks/java/core/src/main/java/org/apache/beam/sdk/metrics/Lineage.java

[10] Old Proposal - Enhance Lineage Support in Beam - GSoC Proposal

[11] OpenLineage Getting Started Example - https://openlineage.io/getting-started/

[12] Google Cloud Dataplex with OpenLineage - https://cloud.google.com/dataplex/docs/open-lineage

[13] OpenLineage Go Client Library - https://github.com/ThijsKoot/openlineage-go