Open lineage initiative

Purpose

The goal of this initiative is to standardize data processing metadata collection, including lineage, runtime attributes, version, schemas, statistics, …
This metadata can be used for various use cases (observability, governance, privacy, ...) and focuses on modeling the running jobs.
First we're defining a common model (Jobs, Datasets, Runs, described below) followed by optional attributes (also called facets) for the entities in that model.
The open lineage api defines a collection protocol for versioned data lineage and metadata for every run of a job and every update of a dataset. It can be leveraged to collect data for multiple

backends (including but not limited to the Marquez-core backend). The goal here is to decouple the metadata definition from the backend storage.
Integrations for various projects, producing this standard metadata can be contributed to the project, but preferably should be contributed to the project they are integrating.

Plan/Points to agree on

This document is a proposal for discussion.
The following points will be discussed to build a consensus. Once there is an agreement on key entities and their relationship we can have independent discussions on individual facets of metadata enriching those entities:

- **Protocols**: Metadata is collected through a write-only potentially asynchronous protocol.
- **Model:** Name of entities and their relationships. As a first step we are focusing on a generic notion of Jobs, Datasets and Runs. They could be batch or streaming.
- **Lifecycle** of entities and how they get updated or enriched.
- Individual **Facets** of metadata that can be attached to each version of those entities will be discussed independently (see below)

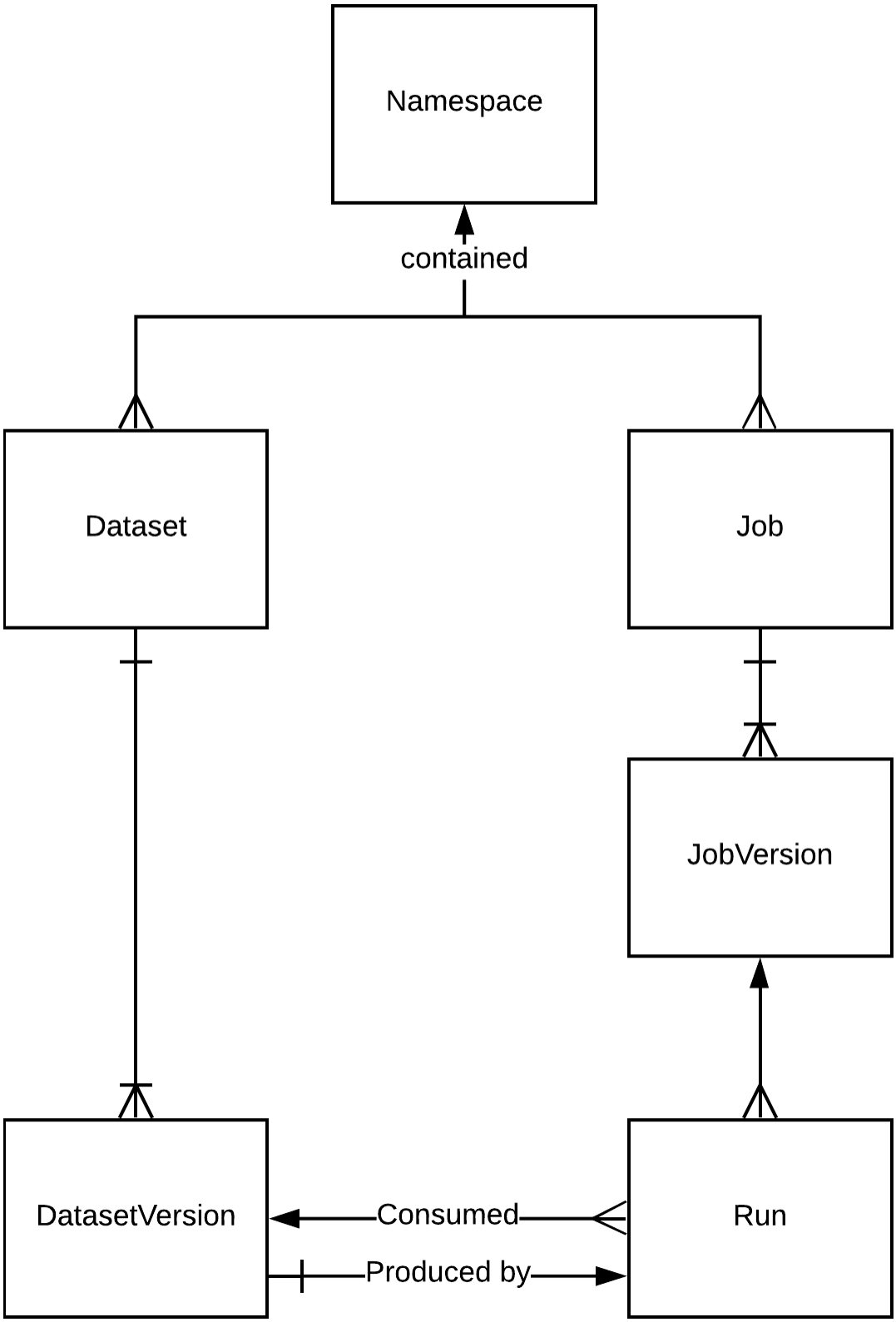
We will discuss those items and reach agreement on github issues in the OpenLineage project

Protocols

The lineage metadata collected is sent over a potentially asynchronous protocol and does not require synchronous access to an API. The backend protocol is abstracted out behind a write-only api which could be a Kafka topic or a REST API for example.
Metadata is standardised by defining an OpenAPI spec (PUT/POST/PATCH operations on a path with a request payload in json and no response). This doesn't preclude having a corresponding binary representation as necessary.

Model

The model defines the following standardized entities. Elements marked TBD are not yet defined. This is based on Marquez model.


Namespace
contained
Dataset
Job
JobVersion
DatasetVersion
Consumed
Produced by
Run

Namespace: unique name

Used to create unique names for jobs and datasets

Versioned Jobs: a Job has a unique name within a namespace. A UUID is created for each version. Those can be batch or streaming jobs

JobVersion -> Job

Versioned Datasets: a dataset has a unique name within a Namespace. A UUID is created for each version

A dataset can be a streaming or at-rest dataset and optionally have a schema.

DatasetVersion -> Dataset

Runs: and attempts

Run -> JobVersion

TBD: Run -> BatchID

This run was an attempt for a given batch (as defined by a time interval or batchid)

A run can also represent the run of a streaming job (for example tracking versions of the job running from a starting time until it was stopped, for example to upgrade it)

Lineage

As defined by tracking the versions of datasets that were consumed or produced by a given run of a given version of a job.

Run -> JobVersion

Run -(Input)-> DatasetVersion

Run -(Output)-> DatasetVersion

TBD: Versioned partitions

What partitions where consumed in what dataset version

Job to job dependency

This reflects dependencies defined when scheduling a workflow

JobVersion -> JobVersion

Job grouping

This reflects higher level entities (ex: airflow dag, ...)

TBD: JobVersion -> Dag tag

Lifecycle

The model gets updated as jobs run:

Run start:

- Update Input Datasets (This will track if schemas have been changed outside of instrumented jobs)
  - Schema
- Update job

    - Code location, version control (git sha, ...)
    - inputs/outputs
- Create a new Run UUID
    - Points to current job version
    - Point to the batch id/logical time this run is an attempt for
    - Optionally Link to partitions consumed or produced
- Mark run as started (timestamp)

Run update

- Optionally patch the running job with new information
    - Query profile

Run complete:

- Update Output Datasets:
    - Schema
    - UUID of the run that modified it
    - Optionally: statistics,...
- Mark run as complete (or failed)

Facets

Facets are optional pieces of metadata that can be attached to the core model entities. The current Marquez model defines some of this metadata already. Here is a list of facets we may want to define. A JSON schema would be defined for each of those.

Dataset level

Each Dataset version can have the following facets

Data stats

At the Partition level or dataset level.

Per column:

- min/max,
- [approx] distribution,
- # nulls,
- [approx] # distinct,
- [approx] top N,
- Row count

- data size

Data version

Actual version of the underlying dataset if they support it (ex: iceberg, delta lake, ...)

Column level lineage

For each column in the output, what column from the input it derives from.
what the expression is.

Schema

TBD: Type systems

SQL
Avro
...

Job

Source

If self contained the source (code) of the job (ex: SQL, ...)

Dependencies

(libraries/environment versions)

Parameters

The parameters that may influence the job runtime

Source control version/location

Ex: the git/github/svn/... URL with sha/version where the job is located

Query Plan/Query Profile

A standardized representation of a query plan (dag of logical operators with names and attributes) and profile (dag of physical operators with name, parallelism, input/output volume, ...)

Run

Logical time

For time-based incremental jobs, this the logical partition produced in output datasets

Batch id

For batch triggered jobs, this can be an id tracked through multiple pipelines in cascading order.

All

Custom

The ability to define non-standard facets using a namespacing mechanism. Ex: X:{vendor}:{name}