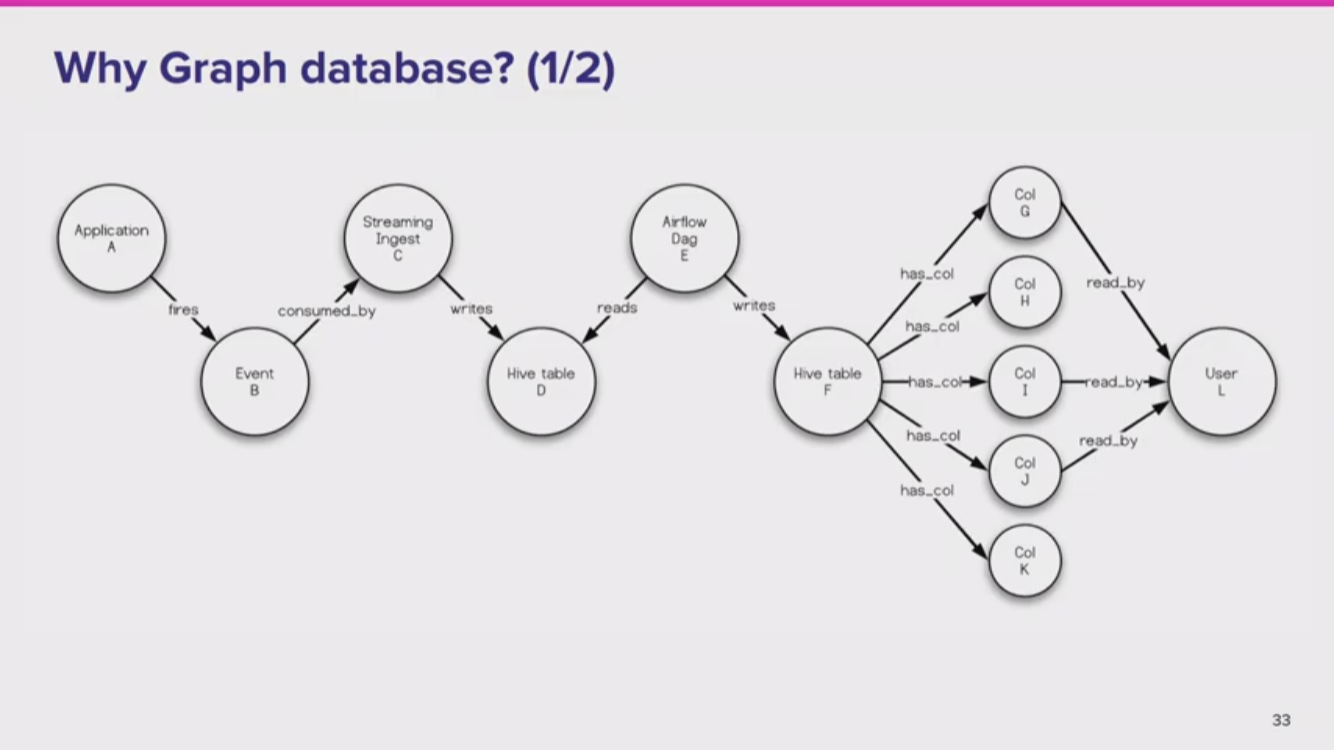
[Amundsen RFC]  
Data Lineage

kort linje

# Why?

The goal of Amundsen Data Lineage is to enable more users to quickly gain an understanding of how data flows across their organization.

Examples of questions lineage can help give an answer to are: What transformations did the data go through? Where did it come from? Who else is using derivatives, of “me”, a dataset upstream? What caused this error in the data? Who should I contact for the column that was propagated?



# Features

The current plan is to implement lineage focusing on table lineage, with a view to expanding with column level granularity as a later add on. This includes:

* A lineage DAG (directed acyclic graph) view showing either **Tables**, **Applications/Jobs** or **Both** with zoom, pan, filtering on the frontend
  + When you select a **Table** in the graph viz there must be an option to navigate to the Amundsen Table details page for it
  + On select of an **Application/Job** node you can navigate to its associated URL (an example for this would be navigation to an Airflow DAG)

## Non-goals

Not in scope for this RFC, but when we have the high priority frontend UX and metadata storage in place a critical area is to develop methods of easy automated data propagation for “sources that matter the most” to the community.

Likewise, as mentioned above, column level lineage is not in scope for now.

Also not in scope currently is adding search (on the front page) for **Applications/Jobs**. Should that eventually become a priority a **Job** detail page is probably also needed (unless we could just find and ask the metadata service for the DAG showing “this”).

# User Interface

## Simple

A simple (to implement) way to present lineage in the UI is just to add it in list or tabular form to the `/table\_detail` pages.

The list could just be placed as multiple sidebar items, similar to how the existing GENERATED BY component works. If you want to view multiple attributes on **XXX**

RFC recommendation **XXX - later - or not at all?**

## DAG visualization

A couple of options that could be adopted are:

* <https://quantumblacklabs.github.io/kedro-viz/>. It is already packaged up as a React npm component. The kedro viz UI already has filtering to either show nodes, jobs or both. Filter on tags (which we already have on Table nodes) - In addition you can style black/white and toggle nodes to be either icons or text boxes [square jobs] and (round data).

*A Proof of Concept UI integration has been “hacked together using palm leaves and thumbnails”. It showcases (some) navigation between table view and the Kedro-viz UI - a 1 minute first-take video is at* [*https://www.youtube.com/watch?v=-Po\_CC74x34*](https://www.youtube.com/watch?v=-Po_CC74x34) *.*

kedro-viz is Apache 2 licensed. Some underlying libs seem to be MIT which should be fine as well. Kedro-viz builds on dagrejs (actually *not* the -d3 companion linked here. <https://github.com/dagrejs/dagre-d3/wiki>)

* <https://getdbt.com> -> [Example](https://www.getdbt.com/example-documentation/#!/overview) A (Global view) has some nice filtering “query language” or [Example B](https://www.getdbt.com/example-documentation/#!/model/model.snowplow.snowplow_sessions) (local-on a single table) Try expanding to full view and the right-click menu.

Apparently built on the `vis-network` package from <https://visjs.org> - Examples [e.g. this](https://visjs.github.io/vis-network/examples/network/layout/hierarchicalLayoutUserdefined.html). It’s MIT/Apache dual licensed. There’s a fairly popular React wrapper of it as well: <https://github.com/crubier/react-graph-vis>

# API Interface

Atlas has this interface: <http://atlas.apache.org/api/v2/resource_LineageREST.html>

If we pick kedro-viz in the frontend they have this input format for a “pipeline” <https://github.com/quantumblacklabs/kedro-viz/blob/develop/src/utils/data/demo.mock.js> The kedro-viz format can be expressed as a subset of the Atlas format.

A new **/lineage** endpoint will be defined that supports a GET. The request data should contain the following:

* **direction** (string] optional) - BOTH | IN | OUT - default BOTH
* **levels** (integer, optional) - - default 3

The response should contain:

* **entities** (list, required)
  + **resource\_name** (string, required)
  + **entity\_type** (string, required) job | dataset
    - [subtype]/params?? **XXX** later
  + **text\_XXX** (string, optional) for display - if
  + **tags** (list[string], optional)
* **relations** (list, required) --- must match with entifies to form a DAG
  + **source** (string/id??, required)
  + **target** (string/id??, required)
    - … type/etc?/kind … later! Used for?
* **xxx**OPTIONAL: **tags**/attributes/meanings to filter on (“tags” is Kedro’s naming for this. I assume “attributes” in the Atlas spec covers a similar concept)

For relations we need to decide on if the responsibility to filter out relations **xxx**- I’m leaning towards front-end because another frontend might be interested in it (could e.g. be to highlight what’s missing to aid data corrections/augmentation)

This design is loosely based on Atlas and Kedro-viz APIs mentioned above. This ensures low effort to add support to the Atlas proxy - that minimal info is present to render (and lineage viz **xxx** )

Appendix

## Exploration of UX alternatives

To support the goals of a user discovering lineage … different “prior art” has done, if it’s open source is it suitable to adopt (or interface to) for Amundsen- ….. what where & how in the screen flow. Must first and foremost not overwhelm but at the same time be at your fingertips (gradual reveal of details).

* **List:** Should be small and non-obtrusive on the user - just remind or “hint” about the possibility that exists to explore lineage. Could just show a partial view, like 1-3 Node “hops” up or downstream from where you are.
* **Table:** For displaying more attributes xxx - might be an option to include in **XXX**. A variation of this is an (expandable) tree view, but this can quickly get “old school enterprisey” tiring.
* **DAG Graphics:** To apply a classic WoW metaphor this can work as `M` key “switch to map” full screen dialog. For big DAGs with zoom a birds-eye world map … it can also have it’s own little search/filtering (like the graph mini-query-language in DBT - or, the different, one in the sidebar in the kedro-viz UI)

For DAG graphics if size/number of nodes becomes so big it hinders usefulness we can consider to implement some kind of namespace scoping/prefiltering . For Tables in schema the schema is a good Filter xxx… the trigger points for when a reduced scope is applied. Just maybe, should we add xxx…

**TODO** figure out if we want UI mockups or if the kedro-viz and existing sidebar Application icon is just what we go with.

**TODO** reach out to kedro-UI to figure out how to collaborate. (We can offer to

# Lineage data sources and Extractors

A big factor in good, cheap, fast lineage is getting hold of lineage DAG data. Unlike metadata which is quite easily automated, at least for something like standards compliant SQL databases, if you want lineage info there are no standard interfaces, and you often end up with your data flowing across several tools and storage locations. This means we quickly need Extractors to interface to many tools.

## Airflow

<https://airflow.apache.org/lineage.html> has a (young) mechanism with an API to populate a metadata store, currently Apache Atlas. Maybe the Amundsen Metadata store could simply implement a similar REST API for the Neo4j backend?

The

## Extracting integrity info from SQL databases

Some SQL databases have <https://www.postgresql.org/docs/7.4/catalog-pg-depend.html> - Compared to table metadata this doen’t appear to be as well standardised as `infor,ation\_schema` (<https://en.wikipedia.org/wiki/Information_schema>) so we need some adaptation in different Extractors

Similarly for Redshift there’s <https://github.com/awslabs/amazon-redshift-utils/blob/master/src/AdminViews/v_view_dependency.sql>

And for SQL Server there is/was a [sp\_depends](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-depends-transact-sql?view=sql-server-ver15) stored procedure which is now deprecated - and being replaced by [sys.dm\_sql\_referencing\_entities](https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-sql-referencing-entities-transact-sql?view=sql-server-ver15) and [sys.dm\_sql\_referenced\_entities](https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-sql-referenced-entities-transact-sql?view=sql-server-ver15) instead

Parsing SQL transforms

FF. ….

Lyft has SQL parser supporting Presto SQL dialect. Implemented based on a Python 2 ANTLR4 based grammar.

Query Parser … Haskell, looks capable, and the language and/or approach taken in the implementation appears to support the problem space well. <https://eng.uber.com/queryparser/> implemented for Presto, Hive and Vertica.

## Instrumenting Spark transforms

Would this be a possible open source lineage candidate?

<https://absaoss.github.io/spline/>

Yes, if you want/need to follow lineage of Spark transformations. But as I understand it it doesn’t do much of you are already doing your transformations in (non-Spark) SQL or something else like Python or R ETL/data science code.

It’s possible you can “borrow” the visualization part from it, not sure how feasible that is though?

## Manual entry

Manual/textual input of DAGs would also be an option <https://github.com/dagrejs/dagre-d3/wiki> simply uses notation from dot

We want to simplify and infer how the edges are named based on the type of Application nodes - so we don’t have to spell it out all the time

JSON schema to CSV

Relations:

A node

Upstream: looking at inlet towards the source:

Indsæt din tekst her <https://medium.com/hashmapinc/apache-atlas-using-the-v2-rest-api-6f9be1c256ae> example Atlas REST DAG nodes, relations/ with curl

Indsæt din tekst her Indsæt din tekst her Indsæt din tek

## Configuration

A new config variable NOTIFICATIONS\_ENABLED will be added with a default of False.

Notifications will be sent via email, and this feature will require a custom implementation of base\_mail\_client to be configured on the MAIL\_CLIENT Flask configuration.

**Frontend default: both | tables | jobs**

# Evaluering

Indsæt din tekst her

Plan steps:

#### Databuilder sample data

* + Transform nodes, edges, (tags) … no just PUT it - might never need to go to SQL/CSV …
  + Model - transforms to CSVs - loads to Neo4j
    - *Future:*

Xxx xxxxx SQL

SELECT \* | col1, col3 FROM table

INSERT/MERGE 42 as col1 IN table

transform -> (GEN) -> column(s) - (IN)-> table

Xxx xxxxx SQL

transform -> (GEN) -> \* equivalent - (IN) -> table

<https://getmanta.com/manta-flow-demo/> and is offering column level visualizations

#### Metadata Service sample data

* + PUT/POST j nodes, edges, (tags)

Lineage - as in-page simple relation steps tabular/tree info:

* Updating the existing **Application** component in **Table Details** to xxx MAYBE NOT DO THIS KEDRO/??? works for both - so JUST use LINEAGE link…multiple hops in/out

Non-scope:

Likewise, column level lineage is not in scope for now.., unle**xxx for now the “hope” is that if column granularity is available the** jobs **simply link to those instead of** tables **(and if table granularity is needed that’s one extra hop “behind” the column, already exists) or they are also linked to the jobs in parallel to columns**

API - augment - for simple - non-graphical inpage viz:

Simplest possible augmentation on the existing /table\_detail API **XXX (is that simply to send Lineage-not even it’s derived - different from how Application - is feed in...)**

# Evaluering

Indsæt din tekst her Indsæt din tekst her Indsæt din tekst her Indsæt din tekst her Indsæt din tekst her Indsæt din tekst her Indsæt din tekst her Indsæt din tekst her Indsæt din tekst her.

kort tankestreg