Searching in provenance with custom ADQL functions

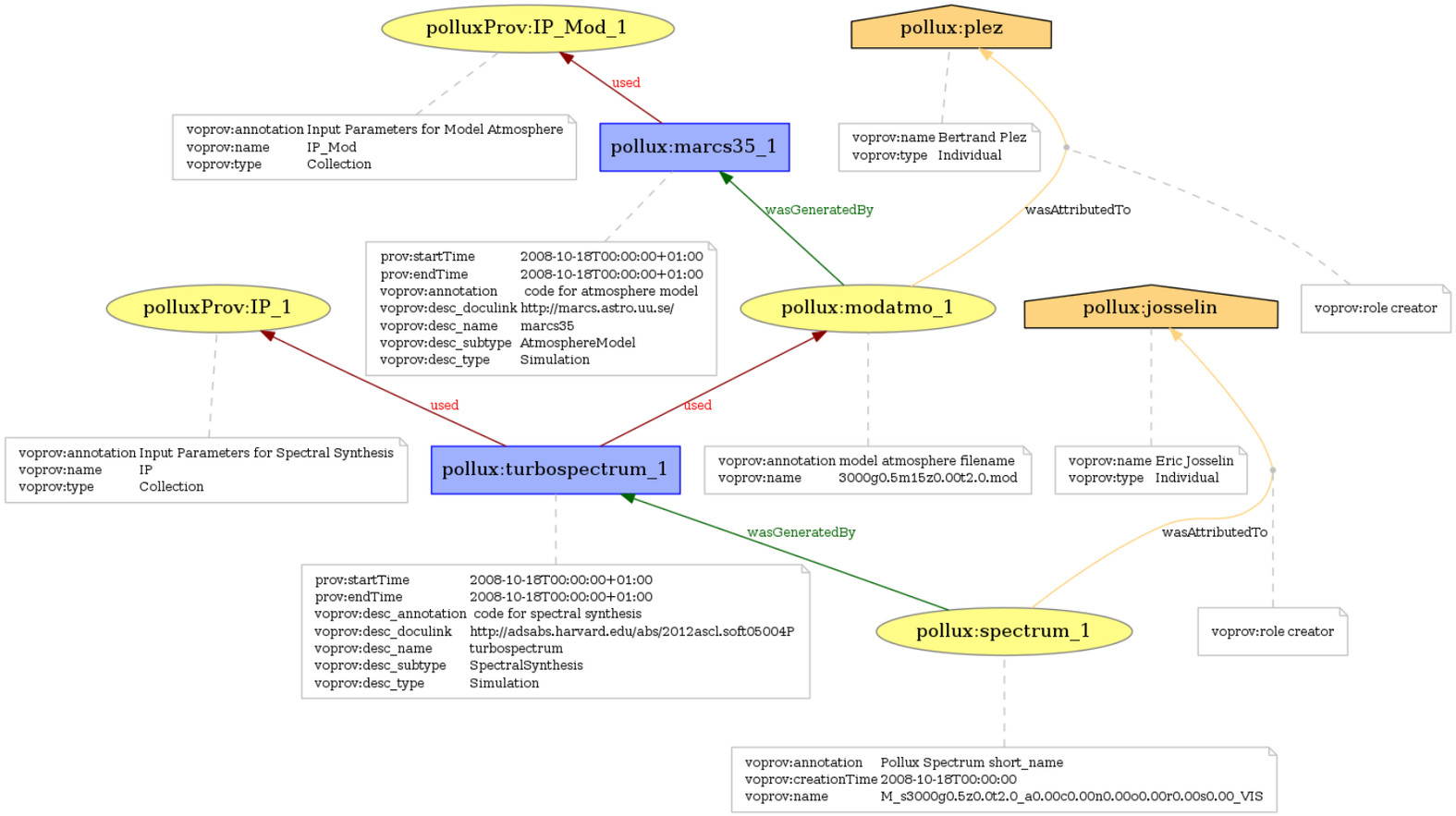
Markus Nullmeier

Zentrum für Astronomie der Universität Heidelberg
Astronomisches Rechen-Institut

mnullmei@ari.uni.heidelberg.de

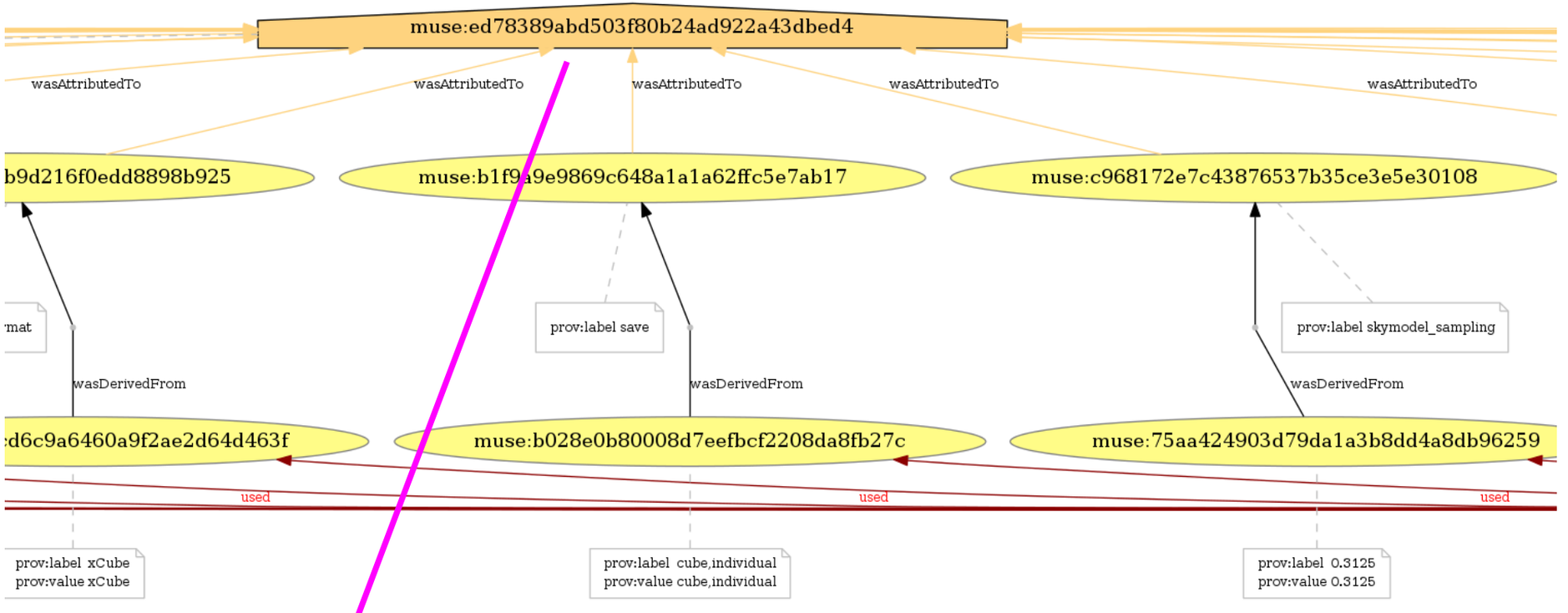
Provenance is about graphs

- **From the proposed IVOA provenance standard:**



Provenance graphs are big and messy

- Closeup



- full graph

Use cases for searching provenance

- **Is the background noise of atmospheric muons still present in this neutrino data sample?**
- **Who was involved in creating that image? Who may be contacted to get this information? Which instrument was used?**
- **Is there a license attached to this data?**
- **Which pipeline version was used?**
- **Show all intermediate processing steps between two datasets**
- **...**

Almost all of these use case require complex traversals of the provenance graph

Options for accessing provenance

- ~~Custom web interfaces~~
  - Not really interoperable
- Specific access protocols → ProvSAP draft
- Relational mapping of the provenance graph → ProvTAP
- IVOA adaption of graph query languages
  - Preferably based on OpenCypher
    - Provenance → property graphs
  - Major effort, especially if as an extension of ADQL
  - Important provenance use cases *not* covered...

Graph databases

- **Graph databases, e. g.,**
  - 
  - **Agens Graph (for PostgreSQL, hybrid) – `www.bitnine.net`**
  - **Microsoft SQL server (since 2017)**
  - **...**
- **→ are more efficient because of**
  - **storage structures are adapted to graphs**
  - **graph-specific indexing**
  - **search algorithms in backend, not in query language**

ProvTAP: traversal queries in “relationalisised” graphs come down to SQL joins

```
with new_g as (select array_agg(CAST(row(wgb_entity_id, wgb_activity_id) as p_edge))
                as y from provenance.wasgeneratedby
               join (select unnest((x).e)) as en
                   ON (provenance.wasgeneratedby.wgb_entity_id = en.unnest) ),
     new_u as (select array_agg(CAST(row(u_activity_id, u_entity_id) as p_edge))
                as y from provenance.used
               join (select unnest((x).a)) as en
                   ON (provenance.used.u_activity_id = en.unnest) ),
     new_t as (select array_agg(CAST(row(wat_entity_id, wat_agent_id) as p_edge))
                as y from provenance.wasattributedto
               join (select unnest((x).e)) as en
                   ON (provenance.wasattributedto.wat_entity_id = en.unnest) ),
     new_s as (select array_agg(CAST(row(waw_activity_id, waw_agent_id) as p_edge))
                as y from provenance.wasassociatedwith
               join (select unnest((x).a)) as en
                   ON (provenance.wasassociatedwith.waw_activity_id = en.unnest) )
select CAST(row(
              (select y from new_g), (select y from new_u), (select y from new_t),
              (select y from new_s))
     as ed_list);
```

- **The user had better get this right…**
- **No *recursive* queries possible: many use cases still out of reach**

Pragmatic solution:
custom ADQL functions for ProvTAP

- a. k. a. “user defined” ADQL functions
  - Implemented by TAP servers

- prov_search_precursor_nodes(result_nodes, node_search_pattern);
- prov_search_result_nodes(start_nodes, node_search_pattern);
- prov_traverse_nodes(start_nodes, traverse_rule, node_s_pattern);
- prov_linking_graph(result_nodes, start_nodes, filter_pattern);
- prov_search_pattern(result_nodes, start_nodes, search_pattern);
- ...

Implementing those via SQL CTEs, benefits:

- not bound to a particular RDBMS
  - Portability of code in, e.g., IVOA implementation notes
- may be used internally in ADQL implementations
- use the same DB for both catalogs, ... and provenance
- “transitional solution”, available today
  – until graph+relational DBMS are ubiquitous(10+yrs?)
- Requires array-like data structures
  - Possible with JSON data types in all relevant SQL RDBMS
  - But PostgreSQL even features first-class arrays :-)

Implementation details (I)

- *→ traversal of the provenance graph*
- 1 traversal step = 1 join over node ids (entities, ...)
- regular ADQL query: >1 traversal step impractical
- network latency → slow global traversals :-(

- *Search everything with a single query, e. g.:*
- `SELECT to_prov(find_prov_precursors(max_depth, entities, activities));`
- → returns subgraph of all specified nodes' precursors in PROV-N format (inside VOTable)
- → or just return specific node or edge types of the precursor graph:
  `SELECT entity_ids(find_prov_precursors(…));`
  `SELECT was_generated_by(find_prov_precursors(…));`

• Implementation details (II)

actual implementation for PostgreSQL:

```
CREATE FUNCTION find_prov_precursors(max_depth INTEGER, entities TEXT[] = null,
                                     activities TEXT[] = null, agents TEXT[] = null)
RETURNS graph_list AS $$
WITH RECURSIVE prov_precursors(depth, start_nodes, nodes, next_edges, edges) AS (
   (WITH input_n AS (SELECT CAST(ROW(entities, activities, agents) AS id_list) AS i_nodes)
    SELECT max_depth, i_nodes, i_nodes, new_edges(i_nodes), empty_egdes() FROM input_n)
UNION ALL
    (WITH z AS (SELECT * FROM prov_precursors), new AS (
     SELECT new_target_nodes(next_edges, nodes) AS nodes FROM z)
     SELECT z.depth - 1, new.nodes, n_union(nodes, new.nodes), new_edges(new.nodes),
                                                    e_union(z.edges, z.next_edges)
     FROM z, new WHERE z.depth <> 0 AND NOT empty(new.nodes)))
SELECT CAST(ROW(nodes, edges) AS graph_list) FROM prov_precursors ORDER BY depth LIMIT 1;
$$ LANGUAGE SQL;
```