Interoperability for Provenance-aware Databases using PROV and JSON

Xing Niu[1], Raghav Kapoor[1], Dieter Gawlick[2], Zhen Hua Liu[2], Vasudha Krishnaswamy[2], Venkatesh Radhakrishnan[3], Boris Glavic[1]

[1]Illinois Institute of Technology [2]Oracle Corporation [3]Facebook

[1]{xniu7,rkapoor7}@hawk.iit.edu, bglavic@iit.edu [2]{dieter.gawlick,zhen.liu, vasudha.krishnaswamy}@oracle.com [3]rven@fb.com

Abstract

Since its inception, the PROV standard has been widely adopted as a standardized exchange format for provenance information. Surprisingly, this standard is currently not supported by provenance-aware database systems limiting their interoperability with other provenance-aware systems. In this work we introduce techniques for exporting database provenance as PROV documents, importing PROV graphs alongside data, and linking outputs of an SQL operation to the imported provenance for its inputs. Our implementation in the GProM system offloads generation of PROV documents to the backend database. This implementation enables provenance tracking for applications that use a relational database for managing (part of) their data, but also execute some non-database operations.

1. Introduction

The PROV standard [12] enables exchange of provenance information between systems by providing a standardized, extensible representation of provenance graphs. For the first time it is possible to track the provenance of an entity during its entire lifecycle while it (and its dependencies) are processed by multiple provenance-aware systems. However, no relational database system supports tracking of database provenance as well as import and export of provenance in PROV. Thus, applications that load data into a relational database to analyze it cannot benefit from available provenance information for imported data. Furthermore, while a provenance-aware DBMS supports computing the provenance of database operations, these systems are currently not capable of exporting provenance into standardized formats. Even if the system generating imported data and the database are both provenance-aware, it is currently not possible to track the derivation of, e.g., a query output, back to the non-database entities that influenced it indirectly.

We propose to make provenance-aware databases interoperable with other provenance-aware systems through an approach for importing provenance stored as PROV-JSON alongside with data from a relational database, propagating of imported provenance during query processing, and export of database provenance into PROV. PROV-JSON is a JSON (JavaScript Object Notation, a lightweight data-interchange format) serialization of the PROV data model. Our technique uses the database backend (SQL) to construct PROV graphs for database operations on demand. Furthermore, we support provenance tracking for JSON path expressions embedded into SQL. In combination these techniques enable tracking and querying of provenance for the whole lifecycle of entities that were created by a combination of database operations and processes external to the database.

EXAMPLE 1. *Consider an application that predicts demographic information (age, gender, and location) for twitter users from monthly logs of tweets. This application first extracts individual tweets from files storing these monthly logs. Each tweet is passed to a classifier that predicts the poster's demographics and inserts this information into a database relation* `user(state,age,gender)`*. The application then runs a query over the imported data to compute the average age of twitter users per state. Figure 2 shows a simplified PROV graph for this application with three input files (Jan to Mar). The input file content, extracted tuples, result tuples, and query for the application are shown in Figure 1. We use the following node and edge types defined by the PROV standard:* ***entities*** *represent pieces of data and/or physical objects (e.g., a tuple or a file),* ***activities*** *are actions or processes which consume and produce entities (e.g., a query or a process),* ***used*** *edges connect entities to the activities that generated or modified them (e.g., a query returning a result tuple),* ***wasGeneratedBy*** *edges connect activities to the entities they have consumed (e.g., read a process reading from a file),* ***wasDerivedFrom*** *edges represent data flow between entities (e.g., a query output tuple is produced from a query input tuple). In our example, each input file is processed by an extractor task* E_i *which outputs tweets (*tw_1 *to* tw_6*). These tweets are fed into three classifiers (one per input file) that extract tuples* t_1 *to* t_6 *which are inserted into a database. Query* Q *then groups these tuples by state to compute the average age per state (the output tuples* t_{o1} *and* t_{o2} *in the example). Such a provenance graph is useful for, e.g., determining causes of erroneous outputs (by tracing them back to erroneous inputs) or evaluating the quality of an output by understanding how it was derived. For instance, assume that tuple* t_{o2} *represents the query result tuple* $(Illinois, 75)$*. The user, surprised by the high average age of tweeter user, would like to know which input tweets were used to compute this result (and in turn which input files contained these tweets).*

In the above example we have used a provenance graph that covers entities (tuples) and activities (queries and updates) inside the database as well as outside the database system (e.g., the input files and classifiers). Even if we would use a workflow system that tracks provenance to execute the extraction and classification tasks, it would not be possible to create such a graph, because provenance-aware relational database systems have currently no native support for imported provenance and do not enable export of the provenance they generate into PROV. That is while the PROV standard addresses the problem of how to uniformly represent provenance generated by different systems, it does not solve the problem of interoperability. When multiple systems, including relational databases, are involved in the creation of an entity then we need to be able to connect provenance generated by these sys-

	Jan
tw_1	...Arkansas senior...
tw_2	...Chicago...all life...
tw_3	...Bulls...

	Feb
tw_4	...Chicago...

	Mar
tw_5	Chicago school is horrible! ...
tw_6	Little rock ... great ... 20s. ...

```
Q: SELECT state, avg(age) AS avgAge
   FROM user
   GROUP BY state
```

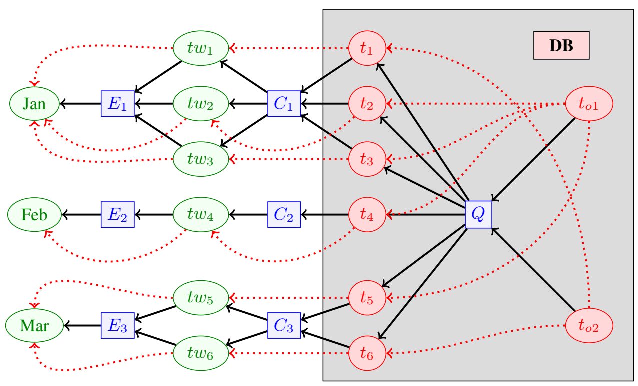
user

	state	age	gender
t_1	Arkansas	80	male
t_2	Illinois	50	female
		...	
t_6	Arkansas	18	male

query result

	state	avgAge
t_{o1}	Arkansas	49
t_{o2}	Illinois	50

Figure 1: Running Example Data and Queries



Key:
■ activities ■ tuple entities ■ file and tweet entities
→ *used* and *wasGeneratedBy* assertions → *wasDerivedFrom* assertions

Figure 2: PROV Graph for the Running Example

tems. In addition to import and export support for PROV graphs this also requires merging of multiple PROV graphs into one coherent description. Provenance-aware database such as GProM [3, 4], Perm [9, 10], or others would be able to determine the *wasDerivedFrom* dependencies between the output tuples of query Q and its inputs. However, to create a provenance graph for the whole application this information needs to be transformed into PROV and connected to the provenance graph fragments associated with the input tuples.

We demonstrate how to make databases interoperable with other provenance systems by implementing export and import of provenance in PROV-JSON and propagation of imported provenance. Our approach translates between database provenance and PROV using SQL. This translation is part of the query computing the provenance. We have implemented our approach in GProM [3, 4], a provenance-aware database middleware which computes provenance for queries and updates under transactional semantics. Our PROV graph generation supports any query (or update) for which GProM can compute provenance.

2. PROV Import and Export

GProM computes provenance for database operations (queries, updates, transactions) on demand using temporal database technologies (maintaining a transaction time history and time travel support for queries) to access past database states if necessary. This functionality is exposed through SQL language extensions, e.g., `PROVENANCE OF (SELECT ...);` computes the provenance of the enclosed query which is returned as a single relation mapping output tuples of the query to input tuples in their provenance. GProM compiles such a query with provenance extensions into SQL code that is evaluated using a regular relational database system.

Export: To support export of such provenance we add an optional `TRANSLATE AS` clause to the `PROVENANCE OF` language construct. This construct is implemented by running several projections over the provenance computation to construct snippets of the PROV-JSON document (e.g., create an entity for each query output tuple), using aggregation to concatenate all snippets of a certain type (e.g., all *used* edges), and a final string concatenation to create the document. Such a query returns a single row with a single column storing the JSON document for the provenance computed by `PROVENANCE OF`.

EXAMPLE 2. *The result of* `PROVENANCE OF` *for query Q from Figure 1 is shown below. Intuitively, each tuple in this result represents one* wasDerivedFrom *assertion, e.g., tuple t_{o1} was derived from tuple t_2.*[1] *Here P denotes a renaming function used to create unique attribute names for attributes in the provenance. We also show which* wasDerivedFrom *edge each tuple corresponds to.*

	Query Result		user *Relation Provenance*		
	state	*avgAge*	*P(state)*	*P(age)*	*P(gender)*
$t_{o1} \rightarrow t_2$	*Illinois*	*50*	*Illinois*	*50*	*female*
		...			
$t_{o2} \rightarrow t_1$	*Arkansas*	*23*	*Arkansas*	*80*	*male*
$t_{o2} \rightarrow t_6$	*Arkansas*	*23*	*Arkansas*	*18*	*male*

If we assume for now that no imported provenance for tuples t_1 to t_6 is available, then generating the JSON serialization of the PROV graph corresponding to query Q is rather straightforward:

1: We create JSON fragments representing the tuple entities using either a system tuple identifier and/or the tuple values to create unique identifiers for these nodes. This can be realized by projecting the result of the provenance computation on either the query result attributes (to create result tuple entities) or the attributes in the provenance (to create input tuple entities). These attribute values are then substituted into a template string for entities.

2: We create a constant string representing the query activity.

3: Edges are generated in the same fashion as entities. For example, to generate *wasDerivedFrom* edges, we combine the query output attributes (respective input attributes) into identifiers for result (respective input) tuples and substitute them into a template.

4: The results of steps 1 to 3 are combined into a single JSON document using an aggregation function which concatenates strings (e.g., create a string representing all entities) and string concatenation to combine the aggregated fragments with fixed "glue" strings.

These operations are expressible in SQL as long as an aggregation function concatenating strings is available. We implement the provenance computation and translation in a single query. The translation code consists of multiple branches, each accessing the result of the provenance computation and outputting a single text value. These values are then combined using cross product.[2] Appendix A shows the generated SQL code for the running example. Note while this query may look surprisingly complicated, the part of the code generating the PROV graph consists of a fixed number of aggregations over the output of the provenance computation. This code only depends on the schemas of input relations of the query for which provenance is computed, and is otherwise independent of this query. Thus, the overhead of PROV graph construc-

[1] This relational encoding [10] encodes provenance polynomials.

[2] Using cross-products is unproblematic in this case, because the inputs to each crossproduct contain only a single row.

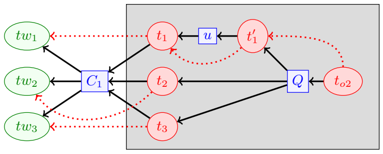


Figure 3: Example PROV Graph with Update

tion is linear in the size of the generated PROV document and the number of input relations.

Import: We provide a language construct `IMPORT PROV FOR ...` to import PROV for an existing relation `R`. This construct is used to import available PROV graphs for imported tuples and store them alongside the data. We add three columns to each table to store imported provenance. For for each imported tuple, we store a PROV-JSON snippet representing its provenance in the `prov_doc` attribute. The `prov_eid` attribute indicates which of the entities in this snippet represents the imported tuple. Column `prov_time` stores a timestamp as of the time when the tuple was imported. We use this column to correctly model provenance of tuples subjected to updates (see Section 3). These attributes are useful for querying imported provenance and, even more important, to correctly include it in exported provenance (as explained below).

EXAMPLE 3. *Below we show the relation* `user` *with imported provenance. Attribute value* d *is the PROV graph from Figure 2 without the query and query outputs.*

	state	*age*	*gender*	*prov_doc*	*prov_eid*	*prov_time*
t_1	*Arkansas*	*80*	*male*	d	t_1	*2015-...*
				...		
t_6	*Arkansas*	*18*	*male*	d	t_6	*2015-...*

Alternative Storage Organization: An obvious disadvantage of the default storage scheme explained above is that PROV graphs may be replicated if more than one tuple was created by the same process. We allow users to use their own storage scheme for imported provenance, e.g., a normalized format which stores the imported PROV-JSON documents in a separate relation. However, we require that the user makes the system aware of how to connect a tuple to its PROV document. For example, we may store the example PROV graph d in a relation `import(did, doc)` and use a foreign key to did instead of the actual document in relation `user`. Another option is to use database-side deduplication techniques to avoid replicated storage of PROV graphs (e.g., Oracle's Advanced LOB Deduplication feature).

Using Imported Provenance During Export: To compute the provenance of a query that accesses a relation with imported provenance, we have to propagate imported provenance to connect it to provenance produced by the query. Unless the user requests export of provenance, we treat the imported provenance in the same fashion as we treat provenance generated by database operations. To export the provenance of a query over data with imported provenance, we include the imported provenance as bundles in the generated PROV graph and connect the entities representing input tuples in the imported provenance to the query activity and output tuple entities. Bundles [12] enable nesting of PROV graphs within PROV graphs, treating a nested graph as a new entity. Whenever we need to refer to the identifier of an input tuple entity (e.g., for *used* edges) we use the identifier stored in the `prov_eid` attribute.

3. Handling Updates

So far we have assumed that tuples with imported provenance are never modified. If a tuple is modified, e.g., by running an SQL `UPDATE` statement, then this should be reflected when provenance is exported. For instance, assume the user has run an update to correct tuple t_1's age value (setting age to 70) before running the query. This update should be reflected in the exported provenance as follows: 1) there should be an activity, say u, that represents this update; 2) there should be two versions of the tuple t_1 entity in the graph. Figure 3 shows part of a PROV graph for the example reflecting this update. Since PROV supports versioned entities, the main challenge in supporting updates for export is how to track the provenance of updates under transactional semantics. This problem has been recently addressed in GProM [3] using the novel concept of reenactment queries. Using GProM the user can request the provenance of an past update, transaction, or set of updates executed within a given time interval. To export provenance for updated tuples we use GProM to generate a provenance representation similar to the one for queries where tuples versions in the provenance are represented in the same fashion as shown in Example 2. We then apply the same techniques as for queries to create the entities and edges to create a PROV document. Since it may not be feasible to export the whole derivation history of tuples that have been imported a long time ago, we let the user decide how far to trace back.

4. Querying Provenance

Since we treat provenance computations as queries, SQL can be used to query provenance. This has been demonstrated to be quite effective for querying relational provenance. To query imported PROV graphs, however, we would want to be able to access their internal structure. If the database supports JSON path expressions embedded in SQL or extraction of relational data from JSON, (e.g., the SQL/JSON standard supported by Oracle [11] and DB2) then we use this functionality to express queries that span database and imported provenance.

EXAMPLE 4. *Assume we want to know how the tweets in the provenance of the example query result* $(Illinois, 80)$ *are distributed over the input files of monthly twitter logs. This query can be implemented in, e.g., Oracle, by computing the provenance of* Q*, extracting the* wasDerivedFrom *edges as relational data from the propagated imported PROV documents, filtering out tuples from the query input based on these* wasDerivedFrom *edges, and counting the number of such tuples grouped by input file.*

5. Provenance for JSON Path Expressions

So far we have assumed that the output of our twitter analysis workflow is represented as relational data that can directly be loaded into a database system for analysis. However, this assumption may not hold, i.e., the output of the classification may only be available in a common data exchange format such as XML or JSON [11]. Most commercial and open-source DBMS support extracting of relational content from these semistructured data formats. For instance, the SQL/XML standard defines the `XMLTABLE` construct for this purpose and analogously the SQL/JSON standard [11] defines `JSON_TABLE`. Both constructs are table functions which use a row path expression to match a set of nodes within the semi-structured document and a set of column path expressions which assemble a tuple from a node matching the row path expression by extracting attribute values from child elements of the matched node.

EXAMPLE 5. *For instance, in a variation of our running example, the user would import a single JSON document (shown in Figure 4) storing the results of the classifiers into the database and then use the DBMS to extract tuples* t_1 *to* t_6*. JSON supports nesting of arrays and objects (represented by* `[]` *respective* $\{\}$*). The example document contains an array of objects - each representing one*

```
[
    {
        "state": "Arkansas",
        "age": "80",
        "gender": "male"
    },
    {
        "state": "Illinois",
        "age": "50",
        "gender": "female"
    }
    ...
]
```

Figure 4: Running Example JSON Input File

classification result. If we treat the JSON documents as opaque values then we would only be able to track back the provenance of a `user` *tuple to this imported JSON document. That is even if the PROV graph for this JSON document is available we would lose the information on which tweets an imported tuple depend on.*

To keep track of dependencies between tuples and the part of a semistructured document they were extracted from, we support tracking the provenance of JSON path expressions embedded in SQL following an approach similar to [7].

EXAMPLE 6. *For example, we can use the JSON path expression* `$[*].state` *to extract all state values from the JSON document shown in Figure 4 (here* `$` *represents the root of the JSON document and* `[*]` *is a wildcard that matches any element of the outer array containing the classification result objects). The provenance of each extracted state value in this example consists of the value itself and the path leading to this entity in JSON document (e.g.,* `$[0].state` *for* (Arkansas,80,male)*)).*

A detailed explanation of our approach is beyond the scope of this paper. Similar to provenance for SQL operations we compute provenance requests for JSON path expressions on-demand by compiling them into SQL/JSON code. Intuitively, the provenance of a path expression for an input JSON document d consists of a set of JSON fragments paired with paths. Each such pair represents one binding of the path expression to a subdocument of d and the path that leads from the root of d to this subdocument.

6. Implementation and Experiments

6.1 Implementation

We have implemented the proposed approach in GProM [4]. Provenance export for queries is fully functional while import of PROV is done manually for now. Exporting of propagated imported provenance is supported, but we only support the default storage layout. While GProM already supports provenance computation for updates and transactions, our current prototype does not support the PROV translation described in Section 3 yet. We plan to add support for the import statement to GProM's parser and user defined storage layouts for imported provenance in the near future.

6.2 Experiments

We ran a small suite of experiments to evaluate the performance of provenance export and propagation of imported provenance compared to computing database provenance without translating it into PROV-JSON. We used TPC-H [14] benchmark datasets with scale factors from 0.01 to 10 (~10MB up to ~10GB size). Experiments were run on a machine with 2 x AMD Opteron 3.3Ghz Processors, 128GB RAM, and 4 1 TB 7.2K RPM disks configured in RAID 5.

Export: We computed the provenance of a three way join between relations `customer`, `order`, and `nation` with additional selection

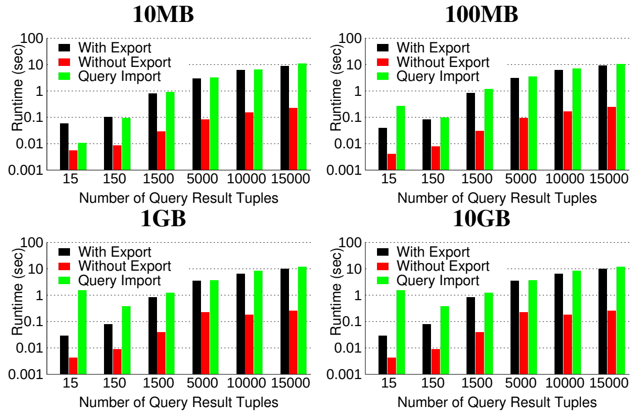


Figure 5: Provenance Computation W/WO PROV-JSON Export

conditions to control selectivity (and, thus, the size of the exported PROV-JSON document). Every result tuple of this query depends on exactly three input tuples (one from each relation). We compare performance of provenance computation and provenance computation plus translation of the generated provenance into PROV-JSON. Figure 5 shows the runtime of these experiments averaged over 100 runs for database sizes from 10MB to 10GB varying the number of result tuples (by changing the selection condition in the query) between 15 and 15K tuples. Generating the PROV-JSON document comes at some additional cost over just computing the provenance. However, this cost is linear in the size of the generated document and independent on the size of the database. To stress test the export mechanism we also computed the provenance of TPC-H query Q13 which produces large provenance graphs. The approach still scales linearly up to scale factor 0.1 (~280MB of exported provenance). The runtime for 1GB is roughly 20 times higher than for 100MB.

Propagating imported provenance: For the next experiment we stored imported PROV-JSON documents alongside every tuple in the `customer` relation. Each customer is associated with a unique small PROV-JSON document that we generated based on a few handcrafted templates. Performance results for exporting provenance for the queries from the previous experiment are shown in Figure 5. Export runtime increases linearly in the size of the imported PROV graphs. The unexpected spike for the query with 15 result tuples stems from the fact that the database chooses an suboptimal execution plan. Our preliminary experiments demonstrate the feasibility of implementing provenance import and export using SQL and integrating it with provenance computation for queries.

7. Related Work

The introduction of the PROV standard marks an important step towards interoperability between provenance systems. However, a common exchange format for provenance does not solve all provenance interoperability problems. Gehani et al. [8] study the problem of identifying nodes in two provenance graphs that represent the same real world entity, activity, or actor and discuss how to integrate such provenance graphs. Some approaches try to address the interoperability problem between database and other provenance-aware systems by introducing a common model [1, 2, 6] for both types of provenance or by monitoring database access to link database provenance with other provenance systems [5, 13]. With PROV we also rely on a common model for provenance, but instead of requiring a central authority for monitoring and provenance recording, we support interoperability through import and export of provenance in PROV. Our approach for tracking provenance of JSON path expressions is similar to work on tracking the provenance of path expressions in XML query languages [7].

8. Conclusions

We integrated import and export of provenance represented as PROV-JSON into provenance-aware databases. Our approach uses the DBMS to construct a PROV graph representing the fine-grained provenance of a database operation on the fly. If the underlying database system supports SQL JSON then this capability can be used to query PROV graphs imported into the DBMS. This enables tracking the provenance of data that has been derived by multiple provenance-aware system where one of the involved systems is a database. Our approach uses imported provenance for tuples in the provenance of a query result to construct one comprehensive PROV graph that represents the whole derivation history of an entity even before it was imported into the database. In addition to extending the implementation of our approach as outlined in Section 6.1, it would be interesting to investigate de-duplication techniques to handle redundancy in imported provenance automatically (e.g., Oracle's securefiles feature or existing provenance specific techniques) and investigate methods for automatic detection of common elements in independently imported provenance graphs.

References

[1] U. Acar, P. Buneman, J. Cheney, J. van den Bussche, N. Kwasnikowska, and S. Vansummeren. A graph model of data and workflow provenance. In *TaPP*, 2010.

[2] Y. Amsterdamer, S. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *PVLDB*, 5(4):346–357, 2011.

[3] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Reenacting Transactions to Compute their Provenance. Technical report, Illinois Institute of Technology, 2014.

[4] B. Arab, D. Gawlick, V. Radhakrishnan, H. Guo, and B. Glavic. A generic provenance middleware for database queries, updates, and transactions. In *TaPP*, 2014.

[5] F. Chirigati and J. Freire. Towards integrating workflow and database provenance. In *IPAW*, pages 11–23, 2012.

[6] D. Deutch, Y. Moskovitch, and V. Tannen. A provenance framework for data-dependent process analysis. *PVLDB*, 7(6), 2014.

[7] J. N. Foster, T. J. Green, and V. Tannen. Annotated XML: Queries and Provenance. In *PODS*, pages 271–280, 2008.

[8] A. Gehani and D. Tariq. Provenance integration. In *TaPP*, 2014.

[9] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, pages 174–185, 2009.

[10] B. Glavic, R. J. Miller, and G. Alonso. Using SQL for Efficient Generation and Querying of Provenance Information. In *In Search of Elegance in the Theory and Practice of Computation*, pages 291–320. Springer, 2013.

[11] Z. H. Liu, B. Hammerschmidt, and D. McMahon. JSON data management: supporting schema-less development in RDBMS. In *SIGMOD*, pages 1247–1258, 2014.

[12] P. Missier, K. Belhajjame, and J. Cheney. The W3C PROV family of specifications for modelling provenance metadata. In *EDBT*, pages 773–776, 2013.

[13] Q. Pham, T. Malik, B. Glavic, and I. Foster. LDV: Light-weight Database Virtualization. In *ICDE*, pages 1179–1190, 2015.

[14] TPC. TPC-H Benchmark Specification, 2009.

A. Example Provenance Export

In this section we show the queries and results of applying our approaches to generate the PROV-JSON representation for the provenance of the running example query. We discuss the case where no imported provenance for the workflow that generated the user relation tuples is available. We use the following query to request provenance for the example query and translate it into PROV-JSON:

```
PROVENANCE OF (
  SELECT state, avg(age) FROM usr GROUP BY state
) TRANSLATE AS JSON;
```

Figure 8 shows the query generated by GProM for computing the provenance of example query Q (which generates the result shown in Example 2). Figure 6 shows a visualization of the PROV graph based on this PROV-JSON document (generated using the PROV translator available at `https://provenance.ecs.soton.ac.uk/validator/view/translator.html`). Query Q groups the 6 input tuples by state to compute the average age per state. Thus, the output contains the two tuples shown on the bottom of the visualization. For instance, result tuple `(Illinois,50)` was derived from the 4 left-most input tuples and tuple `(Arkansas,49)` was derived from the 2 right-most input tuples. In the JSON serialization these two tuples are represented as entities with the following identifiers:

```
"tupQ\(Arkansas|49\)"
"tup\(Illinois|50\)"
```

Note that identifiers like these are self-explanatory, but not necessarily unique. For instance, the input relation `usr` may contain duplicates. We will support additional options for the `TRANSLATE AS JSON` clause to include additional distinguishing information in the generated identifiers such as a system row identifier and a version timestamp, the database name, and the database servers IP address. However, for simplicity we will use the simple value-based identifiers in this example. The PROV-JSON document created by running the query is shown in Figure 7. The JSON serialization of PROV uses a top-most object where each field corresponds to all elements of a certain type (e.g., *used* edges). Field *prefix* defines the namespaces used in this PROV-JSON document. Field *entity* stores all entities (tuples in our example) and *activity* stores all activities (the query in our example). For example query Q, there are 8 entities and 1 activity. The following parts store edges between nodes in the PROV graph. A separate field is used for every type of edge. For example, the first element in *wasDerivedFrom* models that the entity `"rel:tupQ\(Illinois|50\)"` was derived from the entity `"rel:tup_USR\(Illinois|31|female\)"`. The first element in *wasGeneratedBy* indicates that entity `"rel:tupQ\(Illinois|50\)"` was generated by the activity `"rel:Q"`. The first element in *used* indicates that entity `"rel:tup_USR\(Illinois|31|female\)"` was used by the activity (`"rel:Q"`).

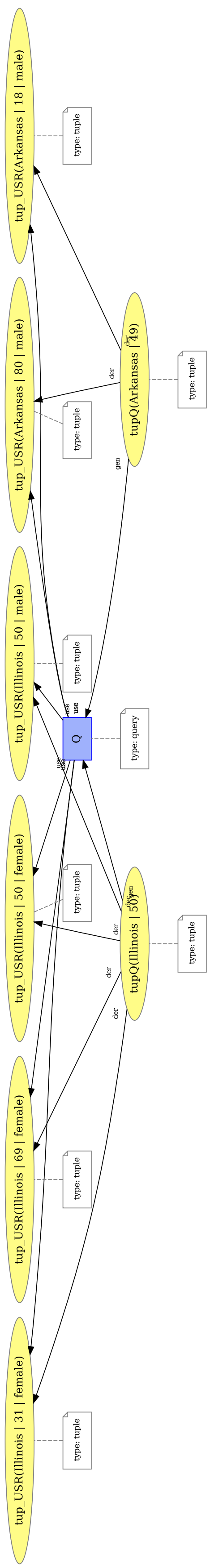


Figure 6: Visualization of the PROV graph for Q


```
{
    "prefix": {
        "rel": "http://relational-model.org"
    },
    "entity": {
        "rel:tupQ\(Illinois|50\)": {
            "prov:type": "rel:tuple"
        },
        "rel:tupQ\(Arkansas|49\)": {
            "prov:type": "rel:tuple"
        },
        "rel:tup_USR\(Illinois|31|female\)": {
            "prov:type": "rel:tuple"
        },
        "rel:tup_USR\(Illinois|69|female\)": {
            "prov:type": "rel:tuple"
        },
        "rel:tup_USR\(Arkansas|80|male\)": {
            "prov:type": "rel:tuple"
        },
        "rel:tup_USR\(Illinois|50|female\)": {
            "prov:type": "rel:tuple"
        },
        "rel:tup_USR\(Illinois|50|male\)": {
            "prov:type": "rel:tuple"
        },
        "rel:tup_USR\(Arkansas|18|male\)": {
            "prov:type": "rel:tuple"
        }
    },
    "activity": {
        "rel:Q": {
            "prov:type": "rel:query"
        }
    },
    "wasDerivedFrom": {
        "_:wdb_USR\(Illinois|50|Illinois|31|female\)": {
            "prov:usedEntity": "rel:tup_USR\(Illinois|31|female\)",
            "prov:generatedEntity": "rel:tupQ\(Illinois|50\)"
        },
        "_:wdb_USR\(Illinois|50|Illinois|69|female\)": {
            "prov:usedEntity": "rel:tup_USR\(Illinois|69|female\)",
            "prov:generatedEntity": "rel:tupQ\(Illinois|50\)"
        },
        "_:wdb_USR\(Arkansas|49|Arkansas|80|male\)": {
            "prov:usedEntity": "rel:tup_USR\(Arkansas|80|male\)",
            "prov:generatedEntity": "rel:tupQ\(Arkansas|49\)"
        },
        "_:wdb_USR\(Illinois|50|Illinois|50|female\)": {
            "prov:usedEntity": "rel:tup_USR\(Illinois|50|female\)",
            "prov:generatedEntity": "rel:tupQ\(Illinois|50\)"
        },
        "_:wdb_USR\(Illinois|50|Illinois|50|male\)": {
            "prov:usedEntity": "rel:tup_USR\(Illinois|50|male\)",
            "prov:generatedEntity": "rel:tupQ\(Illinois|50\)"
        },
        "_:wdb_USR\(Arkansas|49|Arkansas|18|male\)": {
            "prov:usedEntity": "rel:tup_USR\(Arkansas|18|male\)",
            "prov:generatedEntity": "rel:tupQ\(Arkansas|49\)"
        }
    },
    "wasGeneratedBy": {
        "_:wgb\(Illinois|50\)": {
            "prov:activity": "rel:Q",
            "prov:entity": "rel:tupQ\(Illinois|50\)"
        },
        "_:wgb\(Arkansas|49\)": {
            "prov:activity": "rel:Q",
            "prov:entity": "rel:tupQ\(Arkansas|49\)"
        }
    },
    "used": {
        "_:wub_USR\(Illinois|31|female\)": {
            "prov:activity": "rel:Q",
            "prov:entity": "rel:tup_USR\(Illinois|31|female\)"
        },
        "_:wub_USR\(Illinois|69|female\)": {
            "prov:activity": "rel:Q",
            "prov:entity": "rel:tup_USR\(Illinois|69|female\)"
        },
        "_:wub_USR\(Arkansas|80|male\)": {
            "prov:activity": "rel:Q",
            "prov:entity": "rel:tup_USR\(Arkansas|80|male\)"
        },
        "_:wub_USR\(Illinois|50|female\)": {
            "prov:activity": "rel:Q",
            "prov:entity": "rel:tup_USR\(Illinois|50|female\)"
        },
        "_:wub_USR\(Illinois|50|male\)": {
            "prov:activity": "rel:Q",
            "prov:entity": "rel:tup_USR\(Illinois|50|male\)"
        },
        "_:wub_USR\(Arkansas|18|male\)": {
            "prov:activity": "rel:Q",
            "prov:entity": "rel:tup_USR\(Arkansas|18|male\)"
        }
    }
}
```


Figure 7: PROV-JSON Document Produced For Query Q

```sql
-- Compute database provenance
WITH temp_view_of_0 AS (
    SELECT F0."GROUP_0" AS STATE,
           F0."AGGR_0" AS "AVG(AGE)",
           F1."_P_SIDE_GROUP_0" AS PROV_USR_STATE,
           F1.PROV_USR_AGE AS PROV_USR_AGE,
           F1.PROV_USR_GENDER AS PROV_USR_GENDER
    FROM
          (SELECT avg(F0.AGE) AS "AGGR_0",
                  F0.STATE AS "GROUP_0"
          FROM usr F0
          GROUP BY F0.STATE) F0
       JOIN
          (SELECT F0.STATE AS "_P_SIDE_GROUP_0",
                  F0.AGE AS PROV_USR_AGE,
                  F0.GENDER AS PROV_USR_GENDER
          FROM usr F0
          WHERE F0.AGE = F0.AGE
                AND F0.GENDER = F0.GENDER
                AND F0.STATE = F0.STATE) F1
       ON F0."GROUP_0" = F1."_P_SIDE_GROUP_0"
          OR (F0."GROUP_0" IS NULL
              AND F1."_P_SIDE_GROUP_0" IS NULL)
)
-- Assemble final PROV-JSON document
SELECT ((((((((('{"prefix": {"rel": "http://relational-model.org"}, "entity": { '
             || F0."allEntities")
             || '}, "activity": { "rel:Q": {"prov:type":"query"} }, "wasDerivedFrom": {')
             || F1."allWdb")
             || '}, "wasGeneratedBy": {')
             || F2."allWgb")
             || '}, "used": {')
             || F3."allUsed")
             || '}}') AS "jExport"
FROM (((((
          -- Create entities nodes
          SELECT replace(rtrim(xmlagg(xmlcdata(F0."entity")).getclobval(),','),
                                     chr(38) || 'quot;','"') AS "allEntities"
          FROM (((
                SELECT ('"rel:tupQ\(' || (F0.STATE || ('|'
                              || (F0."AVG(AGE)" || '\)": {"prov:type":"tuple"},'))))
                             AS "entity"
                FROM ((temp_view_of_0) F0))
                UNION ALL (
                SELECT ('"rel:tup_USR\(' || (F0.PROV_USR_STATE || ('|'
                              || (F0.PROV_USR_AGE || ('|' || (F0.PROV_USR_GENDER
                              || '\)": {"prov:type":"tuple"},')))))) AS "entity"
                FROM ((temp_view_of_0) F0))) F0)) F0)
           CROSS JOIN ((
           -- Create wasDerivedFrom edges
           SELECT DISTINCT replace(rtrim(xmlagg(xmlcdata(((((((((((((((((((((((('"_:wdb_USR\(' || F0.STATE)
                         || '|') || F0."AVG(AGE)") || '|')
                         || F0.PROV_USR_STATE) || '|') || F0.PROV_USR_AGE)
                         || '|') || F0.PROV_USR_GENDER)
                         || '\)": { "prov:usedEntity": "rel:tup_USR\(')
                         || F0.PROV_USR_STATE) || '|') || F0.PROV_USR_AGE)
                         || '|') || F0.PROV_USR_GENDER) || '\)"')
                         || ',"prov:generatedEntity":"')
                         || 'rel:tupQ\(') || F0.STATE) || '|') || F0."AVG(AGE)")
                         || '\)"},')))).getclobval(),','),chr(38)
                         || 'quot;','"') AS "allWdb"
           FROM ((temp_view_of_0) F0)) F1))
           CROSS JOIN ((
           -- Create wasGeneratedBy edges
           SELECT replace(rtrim(xmlagg(xmlcdata(((((((((((('"_:wgb\(' || F0.STATE) || '|')
                         || F0."AVG(AGE)")
                         || '\)" : { "prov:activity": "rel:Q", "prov:entity": "')
                         || 'rel:tupQ\(') || F0.STATE) || '|') || F0."AVG(AGE)")
                         || '\)')
                         || '"},')))).getclobval(),','),chr(38)
                         || 'quot;','"') AS "allWgb"
           FROM ((temp_view_of_0) F0)) F2))
           CROSS JOIN ((
           -- Create used edges
           SELECT replace(rtrim(xmlagg(xmlcdata((((((((((((('"_:wub_USR\(' || F0.PROV_USR_STATE)
                         || '|') || F0.PROV_USR_AGE) || '|')
                         || F0.PROV_USR_GENDER)
                         || '\)" : { "prov:activity": "rel:Q", "prov:entity": "rel:tup_USR\(')
                         || F0.PROV_USR_STATE) || '|') || F0.PROV_USR_AGE)
                         || '|') || F0.PROV_USR_GENDER)
                         || '\)"},')))).getclobval(),','),chr(38)
                         || 'quot;','"') AS "allUsed"
           FROM ((temp_view_of_0) F0)) F3)
);
```

Figure 8: Generating Provenance for Example Query Q and Translating the Result into PROV-JSON