A Declarative Query Language for Data Provenance
(Research Track)

Argyro Avgoustaki
ICS - FORTH
argiro@ics.forth.gr

Giorgos Flouris
ICS - FORTH
fgeo@ics.forth.gr

Dimitris Plexousakis
ICS - FORTH
dp@ics.forth.gr

Abstract
Provenance has been widely studied in several different contexts and with respect to different aspects and applications. Although the problem of determining how provenance should be recorded and represented has been thoroughly discussed, the issue of querying data provenance has not yet been adequately considered. In this paper, we introduce a novel high-level structured query language, named ProvQL, which is suitable for seeking information related to data provenance. ProvQL treats provenance information as a first class citizen and allows formulating queries about the sources that contributed to data generation and the operations involved, about data records with a specific provenance/origins (or with common provenance), and others. This makes ProvQL a useful tool for tracking data provenance information and supporting applications that need to assess data reliability, access control, trustworthiness, or quality.

1 Introduction
Provenance is fundamental for assessing the quality, trustworthiness, reliability and accountability of data. In recent years, provenance has been widely studied in several different contexts, e.g., databases, workflows, distributed systems, Semantic Web, etc., and with respect to different aspects and applications. These studies have resulted to several abstract provenance models such as lineage [7], trio-lineage [2], why and where [3], how [9,11,14], where-how [1], each with a different level of complexity and detail (column, tuple/triple, graph), regarding different operations (queries, updates) and associated with various data models (relational, RDF, etc.). These provenance models have been used to fuel specific implementations of provenance-aware repositories [4,10,15,19,25,26], based on different representation models, such as CIDOC CRMdig [23], W3C PROV [18], or TripleProv [25].

Despite this progress, the issue of retrieving provenance information through queries has received less attention, as, to the best of our knowledge, there are only two query languages for data provenance [22], namely ProQL [15] and PQL [19]. In fact, the typical approach used for retrieving provenance information in most provenance-aware repositories is to use a generic query language (e.g., SPARQL [12]) for querying directly the underlying implementation [4,10,19,25,26] (see Figure 1, bottom).

Although feasible, this approach creates an undesirable bonding between the query formulation process and the implementation details of the provenance-aware repository. This reduces robustness and interoperability, as the application logic is bound to the specific implementation, and thus cannot be migrated easily to alternative implementations.

A well-known method for addressing problems of this type is to develop a standard query language which will abstract provenance retrieval operations from the implementation details of the underlying repository. This approach has been used in all mainstream formalisms for knowledge representation in other contexts (e.g., SQL for relational, SPARQL for RDF), and has been proven to reduce the development effort of applications. The same idea can also improve query performance, as language-specific optimisations may be developed in the underlying repositories to allow faster retrieval times.

Embracing this viewpoint, we propose ProvQL, a declarative, structured, high-level and non-compositional query language for data provenance that allows expressing provenance-enriched queries in a manner independent to the underlying implementation. This way, applications can formulate queries

Figure 1: Querying provenance-aware repositories
using the ProvQL syntax, and these queries will be translated to a query appropriate for the given implementation, in a manner non-specific to the application (see Figure 1, top).

To support this vision, ProvQL is based on a generic model of a provenance-aware repository, described in Section 3. This simply assumes a set of uniquely identifiable data records (over some data model), enriched with provenance information (over some provenance model). The core syntax and semantics of ProvQL are defined based on this generic model (Sections 4 and 5 respectively). Given that different data and provenance models have different expressive power and data retrieval needs, ProvQL foresees the inclusion of modules that extend its core syntax and semantics, and are specific to different data and provenance models. In Sections 4, 5, we describe some indicative modules, used for supporting the data and provenance models described in Section 2.

The syntax of ProvQL borrows features from both SQL and Cypher [6] (see Appendix A.1 for a short introduction to Cypher), allowing the expression of queries in a compact, platform-agnostic and representation-agnostic manner. Formal semantics are defined using the concept of mappings, in a manner similar to SPARQL semantics [20].

ProvQL supports various types of queries, allowing filtering of the required data on the basis of their provenance, the data itself, or both. Possible ProvQL queries include:

- Find the provenance of a given data record
- Identify data records whose provenance is included in the provenance of a given data record
- Identify data records whose provenance contains a specific data record
- Which sources contributed in deriving a data record
- Identify the different ways to construct a given data record

As a proof-of-concept, ProvQL was implemented for RDF data enriched with how-provenance information (Section 6); plans for supporting other data/provenance models are underway. Our implementation maps a provenance-aware repository to a Neo4j graph (Subsection 6.1), and then translates ProvQL queries to appropriate Cypher queries to return the correct results (Subsection 6.2).

Note that we make no claims regarding the appropriateness of the proposed graph-based model for the representation of how-provenance, and this implementation is not proposed as an improvement over alternative ones. Instead, it should be viewed only as a reasonable choice for developing our proof-of-concept. To support ProvQL, provenance-aware repositories should implement (or reuse) a translation of ProvQL queries in their underlying representation, for the specific implementation employed (as in Subsections 6.1, 6.2).

2 Preliminaries

The ProvQL specification considers the RDF and relational data models, as well as the how, why, trio-lineage and lineage provenance models. Due to lack of space, the presentation focuses mainly on RDF and how-provenance, and we present the other models only briefly.

The RDF data model. RDF [16] is a standard model for data interchange on the Web, which represents data in the form of triples (subject, predicate, object), indicating that a certain subject is related to a certain object through some predicate. Formally, RDF employs two disjoint and infinite sets, namely IRIs (I) and literals (L), to form triples which are elements of the set I × I × (I ∪ L).

The how-provenance model. The how-provenance model was first introduced for relational data in [11], but later works [8, 24] explored its applicability to other data models, such as RDF. It is an algebraic model that assumes a set of identifiers (D) and two abstract operators (denoted by ⊕ for the JOIN operator, and ⊗ for the UNION operator), to form a provenance semiring. The identifiers in D are uniquely associated with data records, whereas operators are used to generate algebraic expressions that represent the provenance of a data record, by describing the operations used to construct it.

Table 1 (second column) summarizes the different types of expressions that the how-provenance model allows. An individual provenance expression represents a specific way of generating the underlying data record, whereas a provenance expression represents the different ways that this data record can be generated; as a data record may be generated multiple times at different instances, provenance multisets (or simply provenance) are used to record this fact. The operators ⊕, ⊗ satisfy the properties of semiring operators (transitivity, commutativity, etc). In the rest of the paper we assume that ⊗ has higher precedence than ⊕ operator. We will use the symbol ≡ to denote equivalence between provenance expressions.

Why, trio-lineage and lineage provenance models. Other provenance models (why-provenance, trio-lineage, lineage) allow less fine-grained provenance information. In particular, lineage [7] represents the sources that contributed to the generation of a data record (as a set), but not how they were combined to generate it. Why-provenance [3] encodes the different derivations separately, in a set, but, due to set semantics, derivations that involve the same set of source records are lost; trio-lineage [2] addresses this shortcoming by using multisets. Table 1 shows how different types of provenance expressions manifest themselves in the considered models (for relational/RDF data models).

3 The ProvQL Model

To abstract from the underlying data and provenance models, the core model of ProvQL makes only some generic assumptions regarding these models.

In particular, for the data model, we denote by T the set of all different data records that the data model admits, and C_T
the constants that are used to generate the elements of T (e.g., for RDF, \( \mathbb{C}_T = \mathbb{I} \uplus \mathbb{L} \)), whereas \( \mathbb{T} = \mathbb{I} \times \mathbb{I} \times (\mathbb{I} \uplus \mathbb{L}) \).

Similarly, for the provenance model, we denote by \( \mathbb{P} \) the set of all different multisets of provenance expressions that can be generated, and \( \mathbb{C}_P \) the constants that are used to generate these provenance expressions (e.g., for the how-provenance model, \( \mathbb{C}_P = \mathbb{I} \)), whereas \( \mathbb{P} \) contains all the provenance multisets of semiring expressions generated from \( \mathbb{C}_P \).

We also assume a set \( \mathbb{V} \), representing the variables, which is disjoint from all other constants (\( \mathbb{V} \cap (\mathbb{C}_T \cup \mathbb{C}_P \cup \mathbb{I}) = \emptyset \)).

A provenance-enriched data record is a data record enriched with provenance information and associated with a unique identifier. Thus, for a given pair of data/provenance models and set of identifiers \( \mathbb{D} \), a provenance-enriched data record is a tuple of the form \( r = (d,t,prov) \), where \( d \in \mathbb{D} \), \( t \in \mathbb{T} \), and \( prov \in \mathbb{P} \). A provenance-aware repository (denoted by \( \mathbb{R} \)) is a set of provenance-enriched data records.

Table 2 presents an example of a provenance-aware repository employing the RDF/how-provenance models. Note that the provenance expressions of \( d_1, d_2, d_3, d_4 \) use a special identifier \( d_0 \) which is not assigned to any of the provenance-enriched triples. The identifier \( d_0 \) is a special element of \( \mathbb{D} \), reserved for the provenance of base data records (in the terminology of [11]), i.e., data records whose insertion in the repository was not a result of some operation over other records.

<table>
<thead>
<tr>
<th>ID</th>
<th>Data record</th>
<th>Provenance Multiset</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>( (\text{Mary}, \text{friendOf}, \text{Bob}) )</td>
<td>( \mathbb{d}_{d_1} )</td>
</tr>
<tr>
<td>r2</td>
<td>( (\text{Bob}, \text{friendOf}, \text{Bill}) )</td>
<td>( \mathbb{d}_0 )</td>
</tr>
<tr>
<td>r3</td>
<td>( (\text{John}, \text{knows}, \text{Bill}) )</td>
<td>( \mathbb{d}_0 )</td>
</tr>
<tr>
<td>r4</td>
<td>( (\text{Mary}, \text{friendOf}, \text{Bill}) )</td>
<td>( \mathbb{d}_0 )</td>
</tr>
<tr>
<td>r5</td>
<td>( (\text{Mary}, \text{knows}, \text{Bill}) )</td>
<td>( \mathbb{d}_0 )</td>
</tr>
<tr>
<td>r6</td>
<td>( (\text{Bill}, \text{knows}, \text{Bob}) )</td>
<td>( \mathbb{d}_0 )</td>
</tr>
</tbody>
</table>

Table 2: A provenance-aware repository example

For a given repository \( \mathbb{R} \), we set \( \mathbb{C}_R \) the set of all constants that appear in \( \mathbb{R} \) (obviously, \( \mathbb{C}_R \subseteq \mathbb{C}_T \cup \mathbb{C}_P \cup \mathbb{I} \)). Given some identifier appearing in \( \mathbb{R} \) (say \( d \in \mathbb{C}_R \cap \mathbb{D} \)), we set:

- **DATA** (\( d \)) = \( t \) if and only if \( (d,t,prov) \in \mathbb{R} \)
- **PROV** (\( d \)) = \( prov \) if and only if \( (d,t,prov) \in \mathbb{R} \)

Note that the functions **DATA**, **PROV** are well-defined, as \( d \) uniquely identifies a provenance-enriched record in the context of \( \mathbb{R} \). Also note that these functions are only defined for identifiers that actually appear in \( \mathbb{R} \).

Further, we define a function (\( \text{IPROV} \)) to return the individual provenance expressions for the how-provenance model (the definition for the other models is analogous, using Table 1). In particular, for a provenance expression \( \text{pe} = \text{ipe}_1 \uplus \cdots \uplus \text{ipe}_n \), we set \( \text{IPROV}(\text{pe}) = \{ \text{ipe}_1, \ldots, \text{ipe}_n \} \).

We extend the definition for provenance multisets \( \text{prov} = \{ \text{pe}_1, \ldots, \text{pe}_n \} \), by setting: \( \text{IPROV}(\text{prov}) = \bigcup \text{IPROV}(\text{pe}_i) \).

Finally, abusing notation, for a given identifier \( d \in \mathbb{C}_R \cap \mathbb{D} \), we set: \( \text{IPROV}(d) = \text{IPROV}(\text{IPROV}(d)) \).

We also define two relations among provenance multisets that will prove useful in the following, namely **includes** and **contains**. Again, the definitions are provided for the how-provenance model only, but can be easily adapted for the other models using Table 1. Informally, \( \text{prov} \) includes \( \text{prov}' \) if \( \text{prov} \) consists of at least the same provenance expressions as \( \text{prov}' \) (modulo semiring equivalence). As for contains, we say that \( \text{prov} \) contains \( \text{prov}' \) if each \( \text{pe}' \in \text{prov}' \) is “part of” some \( \text{pe} \in \text{prov} \), i.e., that \( \text{pe}' \) (or one of its components) is a subexpression of \( \text{pe} \) (or one of its components). Table 3 contains some examples for illustration.

More formally, for two provenance multisets \( \text{prov} = \{ \text{pe}_1, \ldots, \text{pe}_m \} \), \( \text{prov}' = \{ \text{pe}_1', \ldots, \text{pe}_n' \} \), we say that \( \text{prov} \) includes \( \text{prov}' \), \( \text{prov} \sqsupseteq \text{prov}' \) if and only if \( n \leq m \) and there exists a renumbering of \( \text{pe}'_i \) such that \( \text{pe}_i \equiv \text{pe}'_i \) for all \( i \leq n \).

For example, in Table 3, \( [d_1 \uplus d_2 \uplus d_3] \) includes \( [d_2 \uplus d_1 \uplus d_3] \), as they are equivalent expressions, but the relation “includes” does not hold among the provenance multisets of the second row of the table, as \( d_1 \uplus d_3 \) does not appear in \( \text{prov} \).

With regards to contains, for two provenance expressions \( x, y \), we say that \( x \) contains \( y \), denoted by \( x \triangleright y \) if and only if any of the following is true:

- \( y \) is an individual provenance expression, and there exists an individual provenance expression \( \text{ipe} \) such that \( y \uplus \text{ipe} \equiv \text{ipe}' \) for some \( \text{ipe}' \in \text{IPROV}(x) \)
- \( \text{IPROV}(x) \sqsubset \text{IPROV}(y) \)

For two provenance multisets \( \text{prov} = \{ \text{pe}_1, \ldots, \text{pe}_m \} \), \( \text{prov}' = \{ \text{pe}_1', \ldots, \text{pe}_n' \} \), we say that \( \text{prov} \) contains \( \text{prov}' \) (\( \text{prov} \triangleright \text{prov}' \)) if and only if for all \( \text{pe}'_i \in \text{prov}' \) there exists \( \text{pe}_j \in \text{prov} \) such that \( \text{pe}_j \equiv \text{pe}'_i \).

Looking at Table 3 (second row), we note that \( d_1 \uplus d_3 \uplus d_4 \) contains \( d_1 \uplus d_3 \) (and \( d_1 \uplus d_4 \)) so “contains” holds. However, looking at the last row, \( d_3 \uplus d_1 \uplus d_4 \) is not a part of any of the provenance expressions of \( \text{prov} \), because it contains two different individual provenance expressions, but only one of them \( (d_1 \uplus d_4) \) is included in \( d_1 \uplus d_3 \uplus d_1 \uplus d_4 \).
4 Syntax of ProvQL

A ProvQL query is matched against a provenance-aware repository R, and the obtained values are used to construct the result, which can be a provenance expression, a data record, a set of values, or a combination of the above.

Table 4 shows the BNF grammar of ProvQL. In particular, subtable 4a shows the core elements of ProvQL that remain unchanged no matter which data or provenance model is being used. Subtable 4b presents the syntax that varies depending on the data model (RDF, relational), whereas subtable 4b contains the syntactic elements related to the provenance based on the used model (how, why, trio-lineage, lineage). Capitalized words in Table 4 are ProvQL reserved words.

In more details, the general form of a ProvQL query is:

\[ q \leftarrow \text{USING} \text{ dataModel provModel} \]

\[ \text{SELECT} \ selectPattern \text{ WHERE} \ evalPattern \]

As the above general form implies, a ProvQL query consists of two parts. The first part (starting with USING) determines the \textit{query parameters}, i.e., the environment under which the \textit{main part} of the query (starting with SELECT) will run.

The query parameters essentially determine the data and provenance model to assume while interpreting the main part of the query. For brevity, we will omit this part in the examples shown in this paper, and will always assume as default that the RDF data model and the \textit{how-provenance} model are used.

The main part of a query consists of two clauses: the \textit{evaluation patterns} (evalPattern) and the \textit{select patterns} (selectPattern). The evalPattern is responsible for the matching part of the query, providing the filters and conditions that the user wants to impose on the results, whereas the selectPattern describes the values that the user wants to get from the query (query result). These are described in more details below.

\textbf{Evaluation patterns} are used to match provenance and/or data constraints. The exact form of an evalPattern depends on the actual data/provenance model considered, so details on their syntactical form are given in subtables 4b, 4c. An evalPattern can be either a dataEvalPattern, which specifies equality or inequality conditions related to data information (e.g., match a data record with a specific data item), or a provEvalPattern, which specifies conditions on various relations (equality, inequality, includes, contains) over provenance expressions (e.g., find provenance-enriched records whose provenance includes/contains some provenance expression).

\textbf{Select patterns} are used to specify the return values of the query. A query can return provenance expressions (in the form of paths) as defined by a function (PROV, IPROV), a set of data records (using DATA), or items associated to a table 4: Syntax of ProvQL.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
prov & prov & \textbf{\(\equiv\)} & prov \textbf{\(\neq\)} \textbf{\(\neq\)}
\hline
\(d_0, d_1 \otimes d_3 \otimes d_1 \otimes d_4\) & \(d_0\) & \checkmark & \checkmark
\hline
\(d_0, d_1 \otimes d_3 \otimes d_1 \otimes d_4\) & \(d_0, d_1 \otimes d_3, d_1 \otimes d_4\) & \times & \checkmark
\hline
\(d_0, d_1 \otimes d_3 \otimes d_1 \otimes d_4\) & \(d_4 \otimes d_1 \otimes d_1 \otimes d_3\) & \checkmark & \checkmark
\hline
\(d_1 \otimes d_2 \otimes d_3\) & \(d_2 \otimes d_3 \otimes d_1\) & \checkmark & \checkmark
\hline
\(d_2 \otimes d_3 \otimes d_1\) & \(d_3 \otimes d_1\) & \times & \times
\hline
\end{tabular}
\caption{Examples of includes and contains relations}
\end{table}
variable (identifiers, IRIs, literals, etc).

Some examples of supported queries, for the RDF/how-provenance models, follow (the “USING” part is omitted):

q1. Find the provenance of a given data record

```
SELECT PROV(?id) WHERE
DATA(?id) = (s, p, o)
```

q2. Identify data records whose provenance is included in the provenance of a given data record

```
SELECT DATA(?id1), DATA(?id2) WHERE
PROV(?id1) INCLUDES PROV(?id2)
```

q3. Identify data records whose provenance contains a specific data record

```
SELECT DATA(?id1) WHERE PROV(?id1)
CONTAINS ?id2 AND DATA(?id2) = (s, p, o)
```

5 Semantics of ProvQL

5.1 Core semantics

The semantics of ProvQL are based on the idea of mappings, as employed for the SPARQL language [20]. Mappings are used to determine the constant value that a variable should be assigned to. For query answering, we seek “appropriate” mappings, i.e., mappings whose assignments are such that they satisfy the conditions found in the evalPattern for the given provenance-aware repository. Once the “appropriate” mappings are found, we apply them to the selectPattern in order to identify the answers to the query.

For example, in q1 above, an “appropriate” mapping should assign ?id to the specific identifier (say d1) whose associated triple is (s, p, o) in the repository; this satisfies the evalPattern. Then, this specific mapping should be applied to the selectPattern to return the result, which, in this case, should be PROV(d1), since ?id is mapped to d1.

More formally, we define a mapping µ to be a function µ : V ∪ C_R → C_R such that µ(x) = x whenever x ∈ C_R. “Mapping appropriateness” is made precise in Definition 5.1:

**Definition 5.1** Consider an evalPattern EP, a mapping µ and a provenance-aware repository R. Then, µ satisfies EP on R, denoted by µ ⊨_R EP if and only if:

1. For EP an evalPattern of the form “EP1 AND EP2”:
2. For EP an evalPattern of the form “EP1 OR EP2”:
   µ ⊨_R EP iff µ ⊨_R EP1 or µ ⊨_R EP2.

Note that we also need to specify how µ ⊨_R EP is defined for the base case, i.e., when EP is a dataEvalPattern or a provEvalPattern. This definition depends on the actual data/provenance model considered, and is done in the next subsections (Definitions 5.3, 5.5 respectively).

Now we need to define what an answer to a query is:

**Definition 5.2** Take a provenance-aware repository R, and a ProvQL query q whose selectPattern is SP = (SP_1, ..., SP_n) and whose evalPattern is EP. Then, (µ(SP_1), ..., µ(SP_n)) is an answer of q over R if µ ⊨_R EP.

Definition 5.2 also relies on a more precise definition of what µ(SP) is, when SP is a dataSelectPattern or a provSelectPattern. This is done in Definitions 5.4, 5.6, as it depends on the considered data/provenance models.

5.2 Data-dependent semantics

Defining the data-dependent semantics requires two steps. First, we must specify how to determine whether µ ⊨_R EP, for EP a dataEvalPattern as allowed by the data model. Second, we must specify what µ(SP) is, when SP is a dataSelectPattern for the respective model. Definitions 5.3 and 5.4 provide these specifications for the RDF model; repeating this exercise for the relational model is easy and omitted.

**Definition 5.3** Consider a dataEvalPattern EP of the RDF model, a mapping µ and a provenance-aware repository R. Then, µ satisfies EP on R, denoted by µ ⊨_R EP if and only if:

1. For EP of the form DATA(?v) = (s, p, o), where ?v ∈ V, s, p, o ∈ V ∪ C_T: µ ⊨_R EP iff DATA(µ(?v)) = (µ(s), µ(p), µ(o)).
2. For EP of the form DATA(?v) ≠ (s, p, o), where ?v ∈ V, s, p, o ∈ V ∪ C_T: µ ⊨_R EP iff DATA(µ(?v)) ≠ (µ(s), µ(p), µ(o)).
3. For EP of the form DATA(?v) ≠ DATA(?v′), where ?v, ?v′ ∈ V: µ ⊨_R EP iff DATA(µ(?v)) ≠ DATA(µ(?v′)).

**Definition 5.4** For a dataSelectPattern of the form DATA(?v) in the RDF model, we set µ(DATA(?v)) = DATA(µ(?v)).

5.3 Provenance-dependent semantics

As with the case of data-dependent semantics, defining the provenance-dependent semantics requires defining: (a) how to determine whether µ ⊨_R EP, for EP a provEvalPattern of the respective model; (b) what µ(SP) is, when SP is a provSelectPattern of the respective model. Definitions 5.5 and 5.6 below provide these specifications for the patterns used in the how-provenance model; repeating this exercise for the rationale model is easy and omitted.

**Definition 5.5** Consider a provEvalPattern of the how-provenance model EP, a mapping µ and a provenance-aware repository R. Then, µ satisfies EP on R, denoted by µ ⊨_R EP, if and only if:

1. For EP of the form provFunc(?v_1) = provFunc(?v_2):
   µ ⊨_R EP iff provFunc(µ(?v_1)) = provFunc(µ(?v_2)).
2. For EP of the form \( \text{provFunc}(?v_1) \neq \text{provFunc}(?v_2) \): 
\[
\mu \models E \iff \text{provFunc}(\mu(?v_1)) \neq \text{provFunc}(\mu(?v_2)).
\]
3. For EP of the form \( \text{provFunc}(?v_1) \) INCLUDES \( \text{provFunc}(?v_2) \): 
\[
\mu \models E \iff \text{provFunc}(\mu(?v_1)) \supseteq \text{provFunc}(\mu(?v_2)).
\]
4. For EP of the form \( \text{provFunc}(?v_1) \) CONTAINS \( \text{provFunc}(?v_2) \): 
\[
\mu \models E \iff \text{provFunc}(\mu(?v_1)) \supset \text{provFunc}(\mu(?v_2)).
\]
5. For EP a provEvalPattern of the form \( \text{provFunc}(?v) \) 
INCLUDES \( \text{provExp}: \mu \) \( \models E \iff \text{provFunc}(\mu(?v)) \supseteq \mu(\text{provExp}). \)
6. For EP a provEvalPattern of the form \( \text{provFunc}(?v) \) 
CONTAINS \( \text{provExp}: \mu \models E \iff \text{provFunc}(\mu(?v)) \supset \mu(\text{provExp}). \)

**Definition 5.6** For a provSelectPattern of the form \( \text{provFunc}(?v) \), we set \( \mu(\text{provFunc}(?v)) = \text{provFunc}(\mu(?v)) \).

## 6 Implementing ProvQL

As a proof of concept, we implemented ProvQL for a specific choice of data and provenance models (namely, RDF and how-provenance). Instead of implementing a native provenance-aware repository, and then implementing ProvQL on top of that, we chose the indirect route of representing a provenance-aware repository as a Neo4j graph database and using appropriate Cypher queries to access it (see Figure 2).

Towards this aim, we defined a data translation function (called \( \text{tr}_P \) – see Subsection 6.1), which determines the exact nodes and edges to create in the Neo4j graph to represent a given provenance-aware repository. Then, a query translation function (called \( \text{tr}_Q \) – see Subsection 6.2) maps a ProvQL to an appropriate Cypher query [6], which is executed over the Neo4j database. The results are then transformed into an appropriate format (e.g., JSON) to be presented to the user. Both translation functions are carefully defined to respect the ProvQL semantics, i.e., to ensure that the result of the generated Cypher query over the respective Neo4j graph database are the ones that the ProvQL semantics dictates.

The choice of a graph database as our implementation substrate was based on the fact that ProvQL queries (and especially those requiring access to provenance information) often require complex path traversals to evaluate the algebraic expressions that express how-provenance. In this respect, graph databases seem an obvious choice, as they excel in graph traversals. Neo4j in particular is an open-source, NoSQL graph database that provides great advantages regarding schema flexibility, query expressivity and data scalability. Cypher was developed to be used in Neo4j. It allows expressing simple and complex traversals and paths, and is very efficient in evaluating path traversal queries. We omit details on Cypher, but a brief tutorial can be found in Appendix A.1.

![Figure 2: Implementation using translations](image-url)

### 6.1 Data Translation (\( \text{tr}_P \))

The data translation function (\( \text{tr}_P \)) is used to map a provenance-aware repository into a Neo4j graph database, which we call provenance graph:

**Definition 6.1** A provenance graph \( G = (W, E) \) consists of:
- A set of nodes, \( W = W_{\text{data}} \cup W_{\text{op}} \), where:
  - \( W_{\text{data}} \) is the set of data nodes, containing identifier-triple pairs, i.e., \( W_{\text{data}} = \{(d_1, t_1), \ldots, (d_k, t_k)\} \), for \( k \geq 0 \), \( d_i \in D \), \( t_i \in \mathbb{I} \times \mathbb{L} \)
  - \( W_{\text{op}} \) is the set of operation nodes, and contains \( \oplus \)-nodes and \( \otimes \)-nodes, i.e., \( W_{\text{op}} = \{\oplus_1, \ldots, \oplus_n\} \cup \{\otimes_1, \ldots, \otimes_m\} \) for \( n \geq 0 \), \( m \geq 0 \)
- A set of directed labelled edges \( E \subseteq W \times W \times \{\text{fromData}, \text{fromJoin}, \text{hasProv}\} \)

The idea of the translation under \( \text{tr}_P \) is visualised in Figure 3, which shows the provenance graph corresponding to the provenance-aware repository of Table 2. In particular, each provenance-enriched data record corresponds to one data node in the provenance graph, which contains (as attributes) the record’s identifier and values (subject, predicate and object). Then, for each provenance expression in the provenance multi-set of the given record, we create a fresh \( \oplus \)-node and associate it with the respective data node using a “hasProv” edge. Note that labelled edges are used to optimize the Cypher query execution. Different individual provenance expressions of a given provenance expression are represented using fresh \( \otimes \)-nodes that are connected with the respective \( \oplus \)-node using a “fromJoin” edge. Finally, \( \oplus \)-nodes are connected with the respective data nodes (that compose the individual provenance expression) using “fromData” edges. Note that an extra dummy node (with empty triple attributes) represents \( d_0 \).

Algorithm 4 describes the above process in more details. We will explain Algorithm 4 using the provenance-enriched data record \( r_2 = (d_2, t_2, prov_{r_2}) \) of Table 2 (see also Figure 3). As a first step, we create the data node \( w_2 \) (lines 2-3), which contains information about the identifier \( d_2 \) and triple \( t_2 \). Then, we create the required \( \oplus \)-nodes and \( \otimes \)-nodes and paths to represent the provenance \( prov_{r_2} \). More specifically, \( prov_{r_2} \) consists of two provenance expressions \( pe_1 = d_0 \) and \( pe_2 = d_1 \otimes d_3 \odot d_1 \otimes d_4 \). For each provenance expression we
construct a $\oplus$-node and connect it to the data node $w_2$ with a directed edge, labelled as “hasProv” (lines 5-7). Then, we create a $\otimes$-node for each individual provenance expression of $pe_1$ and $pe_2$ (line 10), and connect it to the corresponding $\oplus$-node (line 11). Finally, for each individual provenance expression, we connect the data nodes that contribute to the JOIN operation to that $\otimes$-node; in our example, we have that $pe_1$ contains one individual provenance expression (namely $d_0$, so we connect it with $w_0$), whereas $pe_2$ contains two individual provenance expressions (namely, $d_1 \otimes d_3$, $d_1 \otimes d_4$, which are connected to $w_1$, $w_3$ and $w_1$, $w_4$ respectively).

**Algorithm 1 CreateOpNode Algorithm**

**Require:** The “type” of the operation node (union, join)

**Ensure:** An operation node $x$ of the given type

1: $op = \text{new OperationNode(“type”)}$
2: $W_{op} = W_{op} \cup \{x\}$
3: return $x$

**Algorithm 2 CreateDataNode Algorithm**

**Require:** A triple $t_i$ (subject, predicate, object) and its identifier $d_i$

**Ensure:** A data node $w_i(d_i, t_i)$

1: $w_i = \text{new DataNode}(d_i, t_i)$
2: if $w_i \notin W_{data}$ then
3: $W_{data} = W_{data} \cup \{w_i\}$
4: return $w_i$

**Algorithm 3 CreateEdge Algorithm**

**Require:** A start node “start”, a destination node “dest”, a label “l”

**Ensure:** A directed edge $e$ ("start", "dest", "l")

1: $e = \text{new Edge(“start”, “dest”, “l”)}$
2: $E = E \cup \{e\}$
3: return $e$

**Algorithm 4 Provenance Graph Construction Algorithm**

**Require:** A provenance-aware repository $R$, $r_i(d_i,t_i,prov_i) \in R$

**Ensure:** A provenance graph $G(W,E)$

1: $w_0 = \text{new DataNode}(d_0)$
2: for all $r_i \in R$ do
3: $w_i = \text{CREATEDATA}(d_i)$
4: for all $r_i \in R$, $w_i \in W$ do
5: for all $pe \in prov_i$ do
6: $\text{currNode} = \text{CREATEOPNODE(“union”)}$
7: $\text{CREATEEDGE(currNode, w_i, “hasProv”)}$
8: for all $ipe \in pe$ do
9: previousNode = currNode
10: $\text{currNode} = \text{CREATEOPNODE(“join”)}$
11: $\text{CREATEEDGE(currNode, previousNode, “fromJoin”)}$
12: for all $d_j \in ipe$ do
13: $\text{CREATEEDGE(w_j, currNode, “fromData”)}$
14: return $G(W,E)$

**6.2 Query Translation ($tr_Q$)**

The query translation process aims at rewriting the $ProvQL$ query into an appropriate Cypher query, which will run over the provenance graph generated by the application of $tr_P$ on the provenance-aware repository. The translation process is quite complex and we only present the basic ideas here; additional (formal) details appear in Appendix A.2.

A Cypher query has the following general form:

MATCH MATCH_PATTERN
WHERE WHERE_PATTERN
RETURN RETURN_PATTERN

A MATCH_PATTERN defines the nodes or paths that will
be used in the evaluation whereas WHERE_Pattern “filters” these nodes/paths with the required conditions; the RETURN_Pattern is used to identify the returned values.

As explained in Section 4, the first part of a ProQL query (query parameters) determines the data and provenance model to be used. This part is not involved in the translation, but determines how to interpret the query while translating it. Here, we focus on the RDF/Prov-case.

During translation, we verify that the ProQL query is syntactically valid, and analyse the use of each variable to ensure that all variables appearing in the selectPattern also appear in the evalPattern. Moreover we ensure, based on the position that each variable appears in the query, that a variable is used to refer to an identifier (from $\mathcal{D}$), or to some element of the data record (subject, predicate, object), but not both.

For the translation, we map each variable in the ProQL query to a unique Cypher variable that represents either a node or a path in the Cypher query. Each pattern appearing in the selectPattern of a ProQL query contributes to the generation of the RETURN_Pattern in Cypher (and sometimes MATCH_Pattern), whereas filter conditions and constraints of the evalPattern correspond to a MATCH_PATTERN (which may contain other nested MATCH_Patterns) and a WHERE_Pattern. Table 5 illustrates the translation process through an example, using a color code to show how each clause in the ProQL query translates into a clause in the Cypher query. Note that, as ProQL is a specialised query language, the translation into a generic language (Cypher in our case) generates a much more complex query, as expected.

Table 5: Translating query $q_3$ (from Section 4) to Cypher

<table>
<thead>
<tr>
<th>ProQL</th>
<th>Cypher</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT DATA(?id1)</td>
<td>MATCH (b:Operation) -[:hasProv] -&gt; (a:Data) WITH a,b MATCH h = (c:Data)</td>
</tr>
<tr>
<td>WHERE PROV(?id1) CONTAINS {?id2}</td>
<td>-[:fromData] -&gt; (d:Operation) - [:fromJoin] -&gt; (b)</td>
</tr>
<tr>
<td>AND DATA(?id2) = (s,p,o)</td>
<td>WHERE c.subject = s AND c.predicate = p AND c.object = o</td>
</tr>
<tr>
<td></td>
<td>RETURN a.{subject,predicate,object} as data_a</td>
</tr>
</tbody>
</table>

Table 5: Translating query $q_3$ (from Section 4) to Cypher

## 7 Related Work

Provenance has been widely studied in the literature [5, 17, 22] with respect to many different aspects, contexts and granularities, and various provenance models have been proposed [1, 3, 9, 11, 13, 14]. In this context, W3C supported the creation of a widely used workflow provenance model, namely PROV [18], in an effort to standardise provenance representation and querying. The features of PROV model were exploited in [26] to associate RDF data resulting from SPARQL queries with provenance information. Moreover, there are provenance-aware systems, such as RDFFProv [4] and Taverna [27], which support provenance querying and management for Semantic Web data using the PROV model. In spite of our common motivation to query provenance related to RDF data, there is a great difference between our works. Our underlying provenance models concern data provenance information that provides a detailed view on the origin of a piece of data, whereas PROV regards workflow provenance that describes procedural data processing and involves operations that are treated as black boxes [21].

Another popular provenance-aware system for relational data is Perm [10], where tuples are annotated with provenance represented in relational form and, hence, can be queried, stored and optimized using standard relational database techniques. ARIADNE [19] introduced PQL, a declarative query language that is able to capture and query provenance on Big Graph analytics in Vertex-Centric graph processing systems. PQL differs from ProQL as it addresses online provenance querying for a newly introduced provenance model and its data model is a graph instead of algebraic expressions.

A query language for data provenance, namely ProQL, was proposed in [15]. Similar to our approach, the authors represented data provenance as a directed graph that contains two types of nodes (tuples and derivations) connected through labeled edges. In contrast to our language, ProQL is model-dependent as it supports only the how-provenance and relational models. Furthermore, ProQL has been translated into SQL, which is less efficient for path traversals, data-relations questions and path expressions than Cypher.

## 8 Conclusions and Future Work

This paper introduced ProQL, a query language for data enriched with provenance information. ProQL is by design modular, to support different types of data and provenance models, and to be adaptable to different implementations of these models. We presented the syntax and semantics of the language for the RDF and relational data models, and for the how, why, trio-lineage and lineage provenance models. We also presented a graph-based implementation of ProQL, for RDF data enriched with how-provenance information.

An experimental evaluation of our implementation on large provenance-aware repositories is currently under way. Additional future plans include the enrichment of the language with features such as distinction or aggregation, implementing support for the other data/provenance models, and incorporation of high-level features to support typical uses of provenance, such as trustworthiness and access control assessment. Furthermore, we plan to implement a native provenance-aware repository that would support the execution of ProQL on top of that, and comparing the performance between the two different approaches.
References


A Appendix

A.1 Cypher Query Language

Cypher is a declarative graph query language that was developed to be used in Neo4j. It is based on the property graph data model, which represents directed graphs with labels on nodes and edges, associated with (key,value) pairs.

A.1.1 Cypher Building Blocks

The building blocks of Cypher are the following:

- **Nodes**
  A node is used to represent an entity. Nodes contain properties, which are (key,value) pairs and they can have zero or more edges connecting them to other nodes. They are denoted with a parentheses, e.g. (node).

- **Relationships**
  A relationship represents a relation between two nodes. Relationships have a direction indicated by a start and an end node. Like nodes, relationships can contain properties. Relationships are denoted with an arrow \( \rightarrow \) between two nodes, e.g. (n) \( \rightarrow \) (m), while additional information can be placed inside of the arrow. This information can be:
  1. a relationship type \( -[:KNOWS|:LIKE]-> \)
  2. a variable name \( -[rel:KNOWS]-> \)
  3. additional properties \( -[since:2010]-> \)
  4. structural information for paths of variable length \( -[:KNOWS*..4]-> \)

- **Properties**
  A property is a (key,value) pair that describes nodes and relationships.

- **Labels**
  Labels are associated with a set of nodes or relationships to denote a role or a type \( (n:Data) \).

A.1.2 Cypher Clauses

- **MATCH**
  This clause searches the graph for data with a specified pattern.
  
  **Example A.1** The following query searches for nodes with relationships pointing to nodes with label “Data”.
  
  ```cypher
  MATCH (n) - (m:Data)
  ```

- **WITH**
  This clause is used to chain query parts together, piping the results from one to be used as starting points or criteria in the next.

- **WHERE**
  This clause is used to add constraints to the patterns in a MATCH clause or filters the results of a WITH clause.

- **RETURN**
  This clause specifies what to include in the query result set.

  **Example A.2** The following query searches for nodes that are connecting with other nodes through a relationship \( r \). The property strength of \( r \) should be satisfy the constraint \( r\.strength > 0.5 \). The query will return the values of \( n, r, \) and \( m \).
  
  ```cypher
  MATCH (n)-[r]-(m)
  WHERE r\.strength > 0.5
  RETURN n, r, m
  ```

A.2 Details on the implementation of ProvQL

The type of a variable \(?v\) is denoted by \( TypeVar(?v) \). Table 6 shows how \( TypeVar(?v) \) is detected. In particular, when a variable appears as an argument in a dataFunc, a provFunc or in a provExp then it has to be an identifier \( d_i \). In any other case, the type of the variable depends on its position in a triple (e.g. subject, predicate or object).

<table>
<thead>
<tr>
<th>Appearance of (?v)</th>
<th>Type (TypeVar(?v))</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataFunc(?v)</td>
<td>ID</td>
</tr>
<tr>
<td>provFunc(?v)</td>
<td>ID</td>
</tr>
<tr>
<td>Anywhere in a provExp</td>
<td>ID</td>
</tr>
<tr>
<td>dataFunc(t) = (?v,x,y)</td>
<td>SUB(t)</td>
</tr>
<tr>
<td>dataFunc(t) = (x,?v,y)</td>
<td>PRED(t)</td>
</tr>
<tr>
<td>dataFunc(t) = (x,y,?v)</td>
<td>OBJ(t)</td>
</tr>
</tbody>
</table>

Table 6: Determining a variable’s type (TypeVar)

Table 7 shows the translation of each element of ProvQL to a proper element in Cypher. The idea of the table is that each ProvQL expression contributes to the formulation of a Cypher query, by imposing the addition of some content (string) in the MATCH_Pattern, WHERE_Pattern and/or RETURN_Pattern. In the table, the leftmost column contains the ProvQL expression to be translated, whereas the other three contain the string that should be added to the MATCH_Pattern, WHERE_Pattern and RETURN_Pattern of the Cypher query respectively.

In some cases, the computation is performed recursively; in such cases, the respective column contains a symbol of the form “[[ X ]]”, which is used to denote that a recursive computation has to take place with regards to element X. For example, when translating DATA(?v1) <> DATA(?v2) the MATCH_Pattern will contain whatever results from the translation of DATA(?v1) and DATA(?v2); this is denoted by [[DATA(?v1)]] [[DATA(?v2)]] respectively.
<table>
<thead>
<tr>
<th>ProvQL Expression</th>
<th>MATCH_Pattern</th>
<th>WHERE_Pattern</th>
<th>RETURN_Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>(?v, where TypeVar(?v) = ID)</td>
<td></td>
<td></td>
<td>var.id</td>
</tr>
<tr>
<td>(?v, where TypeVar(?v) = SUB(t))</td>
<td></td>
<td></td>
<td>RETURN t.subject</td>
</tr>
<tr>
<td>(?v, where TypeVar(?v) = PRED(t))</td>
<td></td>
<td></td>
<td>RETURN t.predicate</td>
</tr>
<tr>
<td>(?v, where TypeVar(?v) = OBJ(t))</td>
<td></td>
<td></td>
<td>RETURN t.object</td>
</tr>
<tr>
<td>DATA(?v) when DATA(?v) appears in the select-Pattern</td>
<td>MATCH {v:Data} WITH v</td>
<td></td>
<td>RETURN v.{subject, predicate, object} as data_v</td>
</tr>
<tr>
<td>DATA(?v) when DATA(?v) appears in the evalPattern</td>
<td>MATCH {v:Data} WITH v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROV(?v) when PROV(?v) appears in the select-Pattern</td>
<td>MATCH {v_n3:Operation}-[:hasProv]-&gt; {v:Data} WITH v_n3, v MATCH v_p=(v_n1:Data)-[:fromData]-&gt; {v_n2:Operation}-[:fromJoin]-&gt; {v_n3:Operation} WITH v, v_n2, v_n3, collect (properties(v_n1)) as v_t1, collect (v_n2.type) as v_t2, collect (v_n3.type) as v_t3, v_p</td>
<td></td>
<td>RETURN v_p</td>
</tr>
<tr>
<td>PROV(?v) when PROV(?v) appears in the evalPattern</td>
<td>MATCH {v_n3:Operation}-[:hasProv]-&gt; {v:Data} WITH v_n3, v MATCH v_p=(v_n1:Data)-[:fromData]-&gt; {v_n2:Operation}-[:fromJoin]-&gt; {v_n3:Operation} WITH v, v_n2, v_n3, collect (properties(v_n1)) as v_t1, collect (v_n2.type) as v_t2, collect (v_n3.type) as v_t3, v_p</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>ProvQL Expression</th>
<th>MATCH_PATTERN</th>
<th>WHERE_PATTERN</th>
<th>RETURN_PATTERN</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPROV(?v) when IPROV(?v) appears in the select-Pattern</td>
<td>MATCH (v_n2:Operation)-[:fromJoin]-&gt; (v_n3:Operation)-[:hasProv]-&gt; (v:Data) WITH v, v_n2, v_n3 MATCH v_p=(v_n1:Data)-[:fromData]-&gt; (v_n2) WITH v, v_n2, v_n3, collect (properties(v_n1)) as v_t1, collect (v_n2.type) as v_t2, collect (v_n3.type) as v_t3, v_p</td>
<td>WHERE</td>
<td>RETURN v_p</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPROV(?v) when IPROV(?v) appears in the evalPattern</td>
<td>MATCH (v_n2:Operation)-[:fromJoin]-&gt; (v_n3:Operation)-[:hasProv]-&gt; (v:Data) WITH v, v_n2, v_n3 MATCH v_p=(v_n1:Data)-[:fromData]-&gt; (v_n2) WITH v, v_n2, v_n3, collect (properties(v_n1)) as v_t1, collect (v_n2.type) as v_t2, collect (v_n3.type) as v_t3, v_p</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA(?v1) &lt;&gt; DATA(?v2)</td>
<td>[DATA(?v1)] [DATA(?v2)]</td>
<td>WHERE v1.id &lt;&gt; v2.id</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA(?v) = recordExp if recordExp = (s,p,o) and s, p ∈ I, o ∈ I∪L</td>
<td>[[DATA(?v)]]</td>
<td>WHERE v.subject = s AND v.predicate = p AND v.object = o</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA(?v) &lt;&gt; recordExp if recordExp = (s,p,o) and s, p ∈ I, o ∈ I∪L</td>
<td>[[DATA(?v)]]</td>
<td>WHERE v.subject = s OR v.predicate = p OR v.object = o</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROV(?v1) &lt;&gt; PROV(?v2)</td>
<td>[[PROV(?v1)]] [[PROV(?v2)]]</td>
<td>WHERE v1.id &lt;&gt; v2.id AND (apoc.util.md5(v1_t1) &lt;&gt; apoc.util.md5(v2_t1) OR apoc.util.md5(v1_t2) &lt;&gt; apoc.util.md5(v2_t2) OR apoc.util.md5(v1_t3) &lt;&gt; apoc.util.md5(v2_t3))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROV(?v1) = PROV(?v2)</td>
<td>[[PROV(?v1)]] [[PROV(?v2)]]</td>
<td>WHERE v1.id &lt;&gt; v2.id AND apoc.util.md5(v1_t1) = apoc.util.md5(v2_t1) AND apoc.util.md5(v1_t2) = apoc.util.md5(v2_t2) AND apoc.util.md5(v1_t3) = apoc.util.md5(v2_t3)</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>ProvQL Expression</th>
<th>MATCH_Pattern</th>
<th>WHERE_Pattern</th>
<th>RETURN_Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROV(?v1) &lt;&gt; IPROV(?v2)</td>
<td>PROV(?v1)] [[IPROV(?v2)]}, v1,v1_p</td>
<td>WHERE v1.id &lt;&gt; v2.id AND (apoc.util.md5(v1_t1) &lt;&gt; apoc.util.md5(v2_t1) OR apoc.util.md5(v1_t2) &lt;&gt; apoc.util.md5(v2_t2))</td>
<td></td>
</tr>
<tr>
<td>PROV(?v1) = IPROV(?v2)</td>
<td>PROV(?v1)] [[IPROV(?v2)]}, v1,v1_p</td>
<td>WHERE v1.id &lt;&gt; v2.id AND apoc.util.md5(v1_t1) = apoc.util.md5(v2_t1) AND apoc.util.md5(v1_t2) = apoc.util.md5(v2_t2)</td>
<td></td>
</tr>
<tr>
<td>IPROV(?v1) &lt;&gt; IPROV(?v2)</td>
<td>[[IPROV(?v1)]}, v1,v1_p</td>
<td>WHERE v1.id &lt;&gt; v2.id AND (apoc.util.md5(s1) &lt;&gt; apoc.util.md5(t1) OR apoc.util.md5(s2) &lt;&gt; apoc.util.md5(t2))</td>
<td></td>
</tr>
<tr>
<td>IPROV(?v1) = IPROV(?v2)</td>
<td>[[IPROV(?v1)]}, v1,v1_p</td>
<td>WHERE v1.id &lt;&gt; v2.id AND apoc.util.md5(s1) = apoc.util.md5(t1) AND apoc.util.md5(s2) = apoc.util.md5(t2)</td>
<td></td>
</tr>
<tr>
<td>PROV(?v1) INCLUDES PROV(?v2)</td>
<td>PROV(?v1)]}, collect(v1_p) as v1_path, size(()-[]-&gt;()-[]-&gt;(v1_n3)) as v1_s [[IPROV(?v2)]}, collect(v2_p) as v2_path, size(()-[]-&gt;()-[]-&gt;(v2_n3)) as v2_s</td>
<td>WHERE ALL(d IN v2_path WHERE d IN v1_path AND v1.id &lt;&gt; v2.id AND v1_s=v2_s)</td>
<td></td>
</tr>
<tr>
<td>PROV(?v1) INCLUDES IPROV(?v2)</td>
<td>PROV(?v1)]}, collect(v1_p) as v1_path, size(()-[]-&gt;()-[]-&gt;(v1_n3)) as v1_s [[IPROV(?v2)]}, collect(v2_p) as v2_path, size(()-[]-&gt;()-&gt;(v2_n2)) as v2_s</td>
<td>WHERE ALL(d IN v2_path WHERE d IN v1_path AND v1.id &lt;&gt; v2.id AND v1_s=v2_s)</td>
<td></td>
</tr>
<tr>
<td>IPROV(?v1) INCLUDES PROV(?v2)</td>
<td>[[IPROV(?v1)]}, collect(v1_p) as v1_path, size(()-[]-&gt;()-&gt;(v1_n2)) as v1_s [[PROV(?v2)]}, collect(v2_p) as v2_path, size(()-[]-&gt;()-&gt;(v2_n3)) as v2_s</td>
<td>WHERE ALL(d IN v2_path WHERE d IN v1_path AND v1.id &lt;&gt; v2.id AND v1_s=v2_s)</td>
<td></td>
</tr>
<tr>
<td>IPROV(?v1) INCLUDES IPROV(?v2)</td>
<td>[[IPROV(?v1)]}, collect(v1_p) as v1_path, size(()-[]-&gt;()-&gt;(v1_n2)) as v1_s [[IPROV(?v2)]}, collect(v2_p) as v2_path, size(()-[]-&gt;()-&gt;(v2_n2)) as v2_s</td>
<td>WHERE ALL(d IN v2_path WHERE d IN v1_path AND v1.id &lt;&gt; v2.id AND v1_s=v2_s)</td>
<td></td>
</tr>
<tr>
<td>PROV(?v1) CONTAINS PROV(?v2)</td>
<td>PROV(?v1)]}, collect(v1_p) as v1_path1,[[IPROV(?v1)]} as v1_path2 [[PROV(?v2)]}, collect(v2_p) as v2_path</td>
<td>WHERE ALL(d IN v2_path WHERE d IN v1_path AND v1.id &lt;&gt; v2.id AND v1_s=v2_s)</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>ProvQL Expression</th>
<th>MATCH_PATTERN</th>
<th>WHERE_PATTERN</th>
<th>RETURN_PATTERN</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROV(?v1) CONTAINS IPROV(?v2)</td>
<td>[[PROV(?v1)], collect(v1_p) as v1_path1, [[IPROV(?v1)]] as v1_path2, [[IPROV(?v2)]]], collect(v2_p) as v2_path</td>
<td>WHERE ALL(d IN v2_path WHERE (d IN v1_path1 OR d IN v1_path2) AND v1.id &lt;&gt; v2.id)</td>
<td></td>
</tr>
<tr>
<td>IPROV(?v1) CONTAINS PROV(?v2)</td>
<td>[[IPROV(?v1)], collect(v1_p) as v1_path, [[IPROV(?v1)]]], collect(v2_p) as v2_path</td>
<td>WHERE ALL(d IN v2_path WHERE (d IN v2_path IN v1_path) AND v1.id &lt;&gt; v2.id)</td>
<td></td>
</tr>
<tr>
<td>PROV(?v) INCLUDES provExp</td>
<td>[[PROV(?v)], v_n2, size(()-[r:fromData]-&gt;(v_n2)) as v_s MATCH (m1)-[r1:fromData]-&gt;(v_n2) WITH v_n2, v_s, v, r1 MATCH (m2)-[r2:fromData]-&gt;(v_n2) WITH v_n2, v_s, v, r1,r2 ... WITH v_n2, v_s, v, r1,r2, ... rn-1 MATCH (mn)-[rn:fromData]-&gt;(v_n2) WHERE ID(r_1) &lt;&gt; ID(r_2) AND ID(r_1) &lt;&gt; ID(r_3) AND ... ID(r_1) &lt;&gt; ID(r_n) AND ID(r_2) &lt;&gt; ID(r_3) AND ... ID(r_{n-1}) &lt;&gt; ID(r_n) AND v_s=n if provExp = m_1 ⊕ m_2 ⊕ ... ⊕ m_n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROV(?v) INCLUDES provExp</td>
<td>[[PROV(?v)], v_n3, size(()-[r:fromData]-&gt;(v_n2)-[:fromJoin]-&gt;(v_n3)) as v_s MATCH (m1)-[r1:fromData]-&gt;(v_n2)-[:fromJoin]-&gt;(v_n3) WITH v_n2, v_s, v, r1,k MATCH (m2)-[r2:fromData]-&gt;(v_n2)-[:fromJoin]-&gt;(v_n3) WITH v_n2, v_s, v, r1,k,1 MATCH (m11)-[r11:fromData]-&gt;(v_n2)-[:fromJoin]-&gt;(v_n3) WITH v_n2, v_s, v, r1,k,11, size(()-[r1]:fromData)-&gt;(v_n3) MATCH (m2)-[r2:fromData]-&gt;(v_n2)-[:fromJoin]-&gt;(v_n3) MATCH (m11)-[r11:fromData]-&gt;(v_n2)-[:fromJoin]-&gt;(v_n3) MATCH (m12)-[r12:fromData]-&gt;(v_n2)-[:fromJoin]-&gt;(v_n3) MATCH (mn)-[rn:fromData]-&gt;(v_n2) WHERE v_s=n AND v_s=k if provExp = m_{1,1} ⊕ m_{1,2} ⊕ ... ⊕ m_{1,n} ⊕ m_{2,1} ⊕ ... ⊕ m_{2,n} ⊕ ... ⊕ m_{k,1} ⊕ ... ⊕ m_{k,n}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROV(?v) INCLUDES provExp</td>
<td>[[PROV(?v)], v_n2, size(()-[r:fromData]-&gt;(v_n2)) as v_s WHERE v_s=1 if provExp = m_1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROV(?v1) CONTAINS provExp</td>
<td>[[PROV(?v1)], v_n2 MATCH (m1)-[r1:fromData]-&gt;(v_n2) WITH v_n2, v_s, v, r1 MATCH (m2)-[r2:fromData]-&gt;(v_n2) WITH v_n2, v_s, v, r1,r2 ... WITH v_n2, v_s, v, r1,r2, ... rn-1 MATCH (mn)-[rn:fromData]-&gt;(v_n2) WHERE ID(r_1) &lt;&gt; ID(r_2) AND ID(r_1) &lt;&gt; ID(r_3) AND ... ID(r_1) &lt;&gt; ID(r_n) AND ID(r_2) &lt;&gt; ID(r_3) AND ... ID(r_{n-1}) &lt;&gt; ID(r_n) AND ID(r_n) AND ID(r_2) AND ID(r_3) AND ... ID(r_{n-1}) &lt;&gt; ID(r_n) AND v_s=n if provExp = m_1 ⊕ m_2 ⊕ ... ⊕ m_n</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>ProvQL Expression</th>
<th>MATCH_Pattern</th>
<th>WHERE_Pattern</th>
<th>RETURN_Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROV(?v)</td>
<td>[[[PROV(?v)]], v_n3, v_n2</td>
<td>WHERE ID(r1) &lt;&gt; ID(r1, 2) AND ID(r1, 1) &lt;&gt; ID(r1, 3) AND ... ID(r1, 1) &lt;&gt; ID(r1, n) AND ID(r2, 1) &lt;&gt; ID(r2, 3) AND ... ID(r2, 1) &lt;&gt; ID(r2, n) AND ID(r3, n-1) &lt;&gt; ID(r3, n) AND ... if provExp = m1_1 ⊘ m1_2 ⊘ ... m1_n</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MATCH (m1)-[r1:fromData]-&gt;(v_n2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-[:fromJoin]-&gt;(v_n3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WITH v_n2, v_s, v, r1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MATCH (m2)-[r2:fromData]-&gt;(v_n2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-[:fromJoin]-&gt;(v_n3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WITH v_n2, v_s, v, r1, r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MATCH (mkn)-[rk:fromData]-&gt;(v_n2)</td>
<td>WHERE ID(r_{1}) &lt;&gt; ID(r_{1, 2}) ... ID(r_{1}, n) AND ID(r_{2}) &lt;&gt; ID(r_{2, 3}) ... ID(r_{2}, n) AND ID(r_{3, n-1}) &lt;&gt; ID(r_{3, n}) ... if provExp = m1_1 ⊘ m1_2 ⊘ ... m1_n</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-[:fromJoin]-&gt;(v_n3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if provExp = m1_1 ⊘ m1_2 ⊘ ... m1_n</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IPROV(?v)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if provExp = m1_1 ⊘ m1_2 ⊘ ... m1_n</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INCLUDES provExp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MATCH (m1)-[r1:fromData]-&gt;(v_n2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-[:fromJoin]-&gt;(v_n3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WITH v_n2, v_s, v, r1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MATCH (m2)-[r2:fromData]-&gt;(v_n2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-[:fromJoin]-&gt;(v_n3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WITH v_n2, v_s, v, r1, r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MATCH (mkn)-[rk:fromData]-&gt;(v_n2)</td>
<td>WHERE ID(r_{1}) &lt;&gt; ID(r_{2}) ... ID(r_{n}) AND ... ID(r_{1}) &lt;&gt; ID(r_{2}) ... ID(r_{n}) AND ... if provExp = m_{1} ⊘ m_{2} ⊘ ... m_{n}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-[:fromJoin]-&gt;(v_n3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if provExp = m_{1} ⊘ m_{2} ⊘ ... m_{n}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IPROV(?v)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if provExp = m_{1} ⊘ m_{2} ⊘ ... m_{n}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONTAINS provExp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MATCH (m1)-[r1:fromData]-&gt;(v_n2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-[:fromJoin]-&gt;(v_n3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WITH v_n2, v_s, v, r1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MATCH (m2)-[r2:fromData]-&gt;(v_n2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-[:fromJoin]-&gt;(v_n3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WITH v_n2, v_s, v, r1, r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MATCH (mkn)-[rk:fromData]-&gt;(v_n2)</td>
<td>WHERE ID(r_{1}) &lt;&gt; ID(r_{2}) ... ID(r_{n}) AND ... ID(r_{1}) &lt;&gt; ID(r_{2}) ... ID(r_{n}) AND ... if provExp = m_{1} ⊘ m_{2} ⊘ ... m_{n}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-[:fromJoin]-&gt;(v_n3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if provExp = m_{1} ⊘ m_{2} ⊘ ... m_{n}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>