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

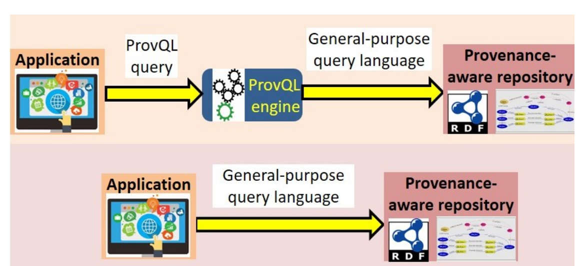

Figure 1: Querying provenance-aware repositories

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:

q_1. Find the provenance of a given data record

q_2. Identify data records whose provenance is included in the provenance of a given data record

q_3. Identify data records whose provenance contains a specific data record

q_4. Which sources contributed in deriving a data record?

q_5. 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 under way. 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[1].

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 ($\mathbb{I}$) and literals ($\mathbb{L}$), to form triples which are elements of the set $\mathbb{I} \times \mathbb{I} \times (\mathbb{I} \cup \mathbb{L})$.

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 ($\mathbb{D}$) and two abstract operators (denoted by $\otimes$ for the JOIN operator, and $\oplus$ for the UNION operator), to form a *provenance semiring*. The identifiers in $\mathbb{D}$ 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 $\oplus$, $\otimes$ satisfy the properties of semiring operators (transitivity, commutativity, etc). In the rest of the paper we assume that $\otimes$ has higher precedence than $\oplus$ operator. We will use the symbol $\equiv$ 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 $\mathbb{T}$ the set of all different data records that the data model admits, and $\mathbb{C}_T$

[1] We omit the description of the well-established relational model.

Terminology	How	Why	Trio-Lineage	Lineage
Individual provenance expression (*ipe*)	$d_1 \otimes \cdots \otimes d_n$	$\{d_1, \ldots, d_n\}$	$\{d_1, \ldots, d_n\}$	d_i
Provenance expression (*pe*)	$ipe_1 \oplus \cdots \oplus ipe_m$	$\{ipe_1, \ldots, ipe_m\}$	$[ipe_1, \ldots, ipe_m]$	$\{ipe_1, \ldots, ipe_m\}$
Provenance multiset (*prov*)	$[pe_1, \ldots, pe_k]$	$[pe_1, \ldots, pe_k]$	$[pe_1, \ldots, pe_k]$	$[pe_1, \ldots, pe_k]$

Table 1: Summary of provenance terminology, for various models

the constants that are used to generate the elements of $\mathbb{T}$ (e.g., for RDF, $\mathbb{C}_T = \mathbb{I} \cup \mathbb{L}$, whereas $\mathbb{T} = \mathbb{I} \times \mathbb{I} \times (\mathbb{I} \cup \mathbb{L})$).

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, $\mathbb{C}_P = \mathbb{D}$, 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 ($\mathbb{V} \cap (\mathbb{C}_T \cup \mathbb{C}_P \cup \mathbb{D}) = \emptyset$).

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 $d \in \mathbb{D}$, $t \in \mathbb{T}$, $prov \in \mathbb{P}$. 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.

	ID	Data record	Provenance Multiset
r_1	d_1	(⟨Mary⟩, ⟨friendOf⟩, ⟨Bob⟩)	$[d_0]$
r_2	d_2	(⟨Bob⟩, ⟨friendOf⟩, ⟨Bill⟩)	$[d_0, d_1 \otimes d_3 \oplus d_1 \otimes d_4]$
r_3	d_3	(⟨John⟩, ⟨knows⟩, ⟨Bill⟩)	$[d_0]$
r_4	d_4	(⟨Mary⟩, ⟨friendOf⟩, ⟨Bill⟩)	$[d_0]$
r_5	d_5	(⟨Mary⟩, ⟨knows⟩, ⟨Bill⟩)	$[d_1 \otimes d_2 \otimes d_3]$
r_6	d_6	(⟨Bill⟩, ⟨knows⟩, ⟨Bob⟩)	$[d_3]$

Table 2: A provenance-aware repository example

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

- $\mathbf{DATA}(d) = t$ if and only if $(d, t, prov) \in \mathbb{R}$
- $\mathbf{PROV}(d) = prov$ 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 (**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 $pe = ipe_1 \oplus \cdots \oplus ipe_n$, we set $\mathbf{IPROV}(pe) = [ipe_1, \ldots, ipe_n]$. We extend the definition for provenance multisets $prov = [pe_1, \ldots, pe_n]$, by setting: $\mathbf{IPROV}(prov) = \bigcup \mathbf{IPROV}(pe_i)$. Finally, abusing notation, for a given identifier $d \in \mathbb{C}_\mathbb{R} \cap \mathbb{D}$, we set: $\mathbf{IPROV}(d) = \mathbf{IPROV}(\mathbf{PROV}(d))$.

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, *prov* includes *prov'* if *prov* consists of at least the same provenance expressions as *prov'* (modulo semiring equivalence). As for contains, we say that *prov* contains *prov'* if each $pe' \in prov'$ is "part of" some $pe \in prov$, i.e., that pe' (or one of its components) is a subexpression of pe (or one of its components). Table 3 contains some examples for illustration.

More formally, for two provenance multisets $prov = [pe_1, \ldots, pe_m]$, $prov' = [pe'_1, \ldots, pe'_n]$, we say that *prov includes prov'*, ($prov \sqsupseteq prov'$) if and only if $n \leq m$ and there exists a renumbering of pe'_i such that $pe_i \equiv pe'_i$ for all $i \leq n$.

For example, in Table 3, $[d_1 \otimes d_2 \otimes d_3]$ includes $[d_2 \otimes d_1 \otimes d_3]$, as they are equivalent expressions, but the relation "includes" does not hold among the provenance multisets of the second row of the table, as $d_1 \otimes d_3$ does not appear in *prov*.

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

- y is an individual provenance expression, and there exists an individual provenance expression ipe such that $y \otimes ipe \equiv ipe'$ for some $ipe' \in \mathbf{IPROV}(x)$
- $\mathbf{IPROV}(x) \sqsupseteq \mathbf{IPROV}(y)$

For two provenance multisets $prov = [pe_1, \ldots, pe_m]$, $prov' = [pe'_1, \ldots, pe'_n]$, we say that *prov contains prov'* ($prov \rhd prov'$) if and only if for all $pe'_i \in prov'$ there exists $pe_j \in prov$ such that $pe_j \rhd pe'_i$.

Looking at Table 3 (second row), we note that $d_1 \otimes d_3 \oplus d_1 \otimes d_4$ contains $d_1 \otimes d_3$ (and $d_1 \otimes d_4$) so "contains" holds. However, looking at the last row, $d_3 \oplus d_1 \otimes d_4$ is not a part of any of the provenance expressions of *prov*, because it contains two different individual provenance expressions, but only one of them ($d_1 \otimes d_4$) is included in $d_1 \otimes d_3 \oplus d_1 \otimes d_4$.

prov	*prov′*	*prov* ⊒ *prov′*	*prov* ▷ *prov′*
$[d_0, d_1 \otimes d_3 \oplus d_1 \otimes d_4]$	$[d_0]$	√	√
$[d_0, d_1 \otimes d_3 \oplus d_1 \otimes d_4]$	$[d_0, d_1 \otimes d_3, d_1 \otimes d_4]$	×	√
$[d_0, d_1 \otimes d_3 \oplus d_1 \otimes d_4]$	$[d_4 \otimes d_1 \oplus d_1 \otimes d_3]$	√	√
$[d_1 \otimes d_2 \otimes d_3]$	$[d_2 \otimes d_3 \otimes d_1]$	√	√
$[d_1 \otimes d_2 \otimes d_3]$	$[d_3 \otimes d_1]$	×	√
$[d_0, d_1 \otimes d_3 \oplus d_1 \otimes d_4]$	$[d_3 \oplus d_1 \otimes d_4]$	×	×

Table 3: Examples of includes and contains relations

4 Syntax of *ProvQL*

A *ProvQL* query is matched against a provenance-aware repository $\mathbb{R}$, 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:

$q \leftarrow$ **USING** dataModel provModel
SELECT selectPattern **WHERE** evalPattern

As the above general form implies, a *ProvQL* query consists of two parts. The first part (starting with **USING**) determines the *query parameters*, i.e., the environment under which the *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 *how-provenance* model are used.

The main part of a query consists of two clauses: the *evaluation patterns* (evalPattern) and the *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.

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).

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

(a) Core Syntax

query ::= **USING** dataModel provModel **SELECT** selectPattern **WHERE** evalPattern
dataModel ::= **RDF** | **REL**
provModel ::= **HOW** | **WHY** | **TRIO** | **LIN**
selectPattern ::= (var | dataSelectPattern | provSelectPattern) (‘,’ var | dataSelectPattern) | provSelectPattern)*
evalPattern ::= (‘(’)*dataEvalPattern | provEvalPattern ((**AND** | **OR**) (dataEvalPattern | provEvalPattern))*(‘)’)*
var ::= ‘?’ letter+ num*
letter ::= (a..z | A .. Z)
num ::= (0..9)+
literal ::= ‘ “ ’ letter+ num* ‘ ” ’
iri ::= For the definition see https://bit.ly/3bEPDN2
integer ::= (‘-’)* num
float ::= (‘-’)* num‘,’num
text ::= ‘ “’ literal (WS literal)*
boolean ::= **true** | **false**

(b) Data Model Dependent Syntax

Common Syntax for RDF, Relational
dataSelectPattern ::= dataFunc(var)
dataEvalPattern ::= dataFunc(var) <> dataFunc(var) | dataFunc(var) (= | <>) recordExp
dataFunc ::= **DATA**
RDF
recordExp ::= ‘(’ recordElement ‘,’ recordElement ‘,’ (recordElement | literal) ‘)’
recordElement ::= var | iri
Relational
recordExp ::= ‘(’ tableName (’,’ recordElement)+ ‘)’
recordElement ::= var | literal | integer | float | text | boolean
tableName ::= literal

(c) Provenance Model Dependent Syntax

Common Syntax for How, Why, Trio - lineage, Lineage
provSelectPattern ::= provFunc(var)
provEvalPattern ::= provFunc(var) (= | <> | **INCLUDES** | **CONTAINS**) provFunc(var) | provFunc(var) (**INCLUDES** | **CONTAINS**) provExp
provFunc ::= **PROV** | **IPROV**
How
provExp ::= var (operation var)*
operation ::= **JOIN** | **UNION**
Why
provExp ::= ‘{’‘{’var (‘,’ var)*‘}(’,’ ‘{’var (‘,’ var)*‘}’)*‘}’
Trio-lineage
provExp ::= ‘[’‘{’var (‘,’ var)*‘}(’,’ ‘{’var (‘,’ var)*‘}’)*‘]’
Lineage
provExp ::= ‘{’var (‘,’ var)*’‘}’

Table 4: Syntax of *ProvQL*

variable (identifiers, IRIs, literals, etc).

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

q_1. Find the provenance of a given data record

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

q_2. 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)
```

q_3. 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 q_1 above, an "appropriate" mapping should assign ?id to the specific identifier (say d_1) 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 $\mathbf{PROV}(d_1)$, since ?id is mapped to d_1.

More formally, we define a *mapping* μ to be a function $\mu : \mathbb{V}\cup\mathbb{C}_{\mathbb{R}} \mapsto \mathbb{C}_{\mathbb{R}}$ such that $\mu(x) = x$ whenever $x \in \mathbb{C}_{\mathbb{R}}$. "Mapping appropriateness" is made precise in Definition 5.1:

Definition 5.1 *Consider an evalPattern EP, a mapping μ and a provenance-aware repository $\mathbb{R}$. Then, μ* satisfies *EP* on *$\mathbb{R}$, denoted by $\mu \models_{\mathbb{R}} EP$ if and only if:*

1. *For EP an evalPattern of the form "EP_1 AND EP_2": $\mu \models_{\mathbb{R}} EP$ iff $\mu \models_{\mathbb{R}} EP_1$ and $\mu \models_{\mathbb{R}} EP_2$.*
2. *For EP an evalPattern of the form "EP_1 OR EP_2": $\mu \models_{\mathbb{R}} EP$ iff $\mu \models_{\mathbb{R}} EP_1$ or $\mu \models_{\mathbb{R}} EP_2$.*

Note that we also need to specify how $\mu \models_{\mathbb{R}} EP$ 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 $\mathbb{R}$, and a ProvQL query q whose selectPattern is $SP = (SP_1,\dots,SP_n)$ and whose evalPattern is EP. Then, $(\mu(SP_1),\dots,\mu(SP_n))$ is an* answer *of q* over *$\mathbb{R}$ iff $\mu \models_{\mathbb{R}} EP$.*

Definition 5.2 also relies on a more precise definition of what $\mu(SP_i)$ is, when SP_i 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 $\mu \models_{\mathbb{R}} EP$, for *EP* a dataEvalPattern as allowed by the data model. Second, we must specify what $\mu(SP)$ 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 $\mathbb{R}$. Then, μ* satisfies *EP* on *$\mathbb{R}$, denoted by $\mu \models_{\mathbb{R}} EP$ if and only if:*

1. *For EP of the form $\mathbf{DATA}(?v) = (s,p,o)$, where $?v \in \mathbb{V}$, $s,p,o \in \mathbb{V}\cup\mathbb{C}_T$: $\mu \models_{\mathbb{R}} EP$ iff $\mathbf{DATA}(\mu(?v)) = (\mu(s),\mu(p),\mu(o))$.*
2. *For EP of the form $\mathbf{DATA}(?v) \neq (s,p,o)$, where $?v \in \mathbb{V}$, $s,p,o \in \mathbb{V}\cup\mathbb{C}_T$: $\mu \models_{\mathbb{R}} EP$ iff $\mathbf{DATA}(\mu(?v)) \neq (\mu(s),\mu(p),\mu(o))$.*
3. *For EP of the form $\mathbf{DATA}(?v) \neq \mathbf{DATA}(?v')$, where $?v,?v' \in \mathbb{V}$: $\mu \models_{\mathbb{R}} EP$ iff $\mathbf{DATA}(\mu(?v)) \neq \mathbf{DATA}(\mu(?v'))$.*

Definition 5.4 *For a dataSelectPattern of the form $\mathbf{DATA}(?v)$ in the RDF model, we set $\mu(\mathbf{DATA}(?v)) = \mathbf{DATA}(\mu(?v))$.*

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 $\mu \models_{\mathbb{R}} EP$, for *EP* a provEvalPattern of the respective model; (b) what $\mu(SP)$ 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 why, trio-lineage and lineage is analogous and omitted.

To simplify Definition 5.5, we abuse notation and use μ to apply to provenance expressions, by "pushing" μ inside the expression, i.e., $\mu(d_{11}\otimes\cdots\otimes d_{1n_1}\oplus\cdots\oplus d_{m1}\otimes\cdots\otimes d_{mn_m}) = \mu(d_{11})\otimes\cdots\otimes\mu(d_{1n_1})\oplus\cdots\oplus\mu(d_{m1})\otimes\cdots\otimes\mu(d_{mn_m})$.

Definition 5.5 *Consider a provEvalPattern of the* how-provenance *model EP, a mapping μ and a provenance-aware repository $\mathbb{R}$. Then, μ* satisfies *EP* on *$\mathbb{R}$, denoted by $\mu \models_{\mathbb{R}} EP$, if and only if:*

1. *For EP of the form $provFunc(?v_1) = provFunc(?v_2)$: $\mu \models_{\mathbb{R}} EP$ iff $provFunc(\mu(?v_1)) = provFunc(\mu(?v_2))$.*

2. *For EP of the form* $provFunc(?v_1) \neq provFunc(?v_2)$*:* $\mu \models_{\mathbb{R}} EP$ *iff* $provFunc(\mu(?v_1)) \neq provFunc(\mu(?v_2))$.
3. *For EP of the form* $provFunc(?v_1)$ *INCLUDES* $provFunc(?v_2)$*:* $\mu \models_{\mathbb{R}} EP$ *iff* $provFunc(\mu(?v_1)) \sqsupseteq provFunc(\mu(?v_2))$.
4. *For EP of the form* $provFunc(?v_1)$ *CONTAINS* $provFunc(?v_2)$*:* $\mu \models_{\mathbb{R}} EP$ *iff* $provFunc(\mu(?v_1)) \triangleright provFunc(\mu(?v_2))$.
5. *For EP a provEvalPattern of the form* $provFunc(?v)$ *INCLUDES provExp:* $\mu \models_{\mathbb{R}} EP$ *iff* $provFunc(\mu(?v_1)) \sqsupseteq \mu(provExp)$.
6. *For EP a provEvalPattern of the form* $provFunc(?v)$ *CONTAINS provExp:* $\mu \models_{\mathbb{R}} EP$ *iff* $provFunc(\mu(?v_1)) \triangleright \mu(provExp)$.

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

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 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 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 path 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.

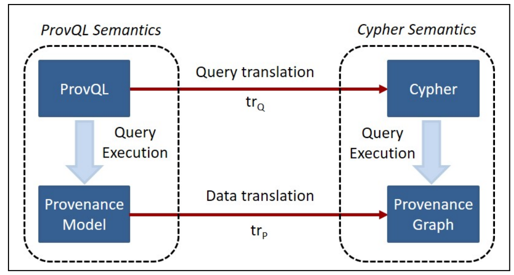

Figure 2: Implementation using translations

6.1 Data Translation (tr_P)

The data translation function (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 $\mathcal{G} = (W, E)$ *consists of:*

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

The idea of the translation under 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 multiset 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, $\otimes$-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 $r_2 = (d_2, t_2, prov_2)$ 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 $prov_2$. More specifically, $prov_2$ consists of two provenance expressions $pe_1 = d_0$ and $pe_2 = d_1 \otimes d_3 \oplus d_1 \otimes d_4$. For each provenance expression we

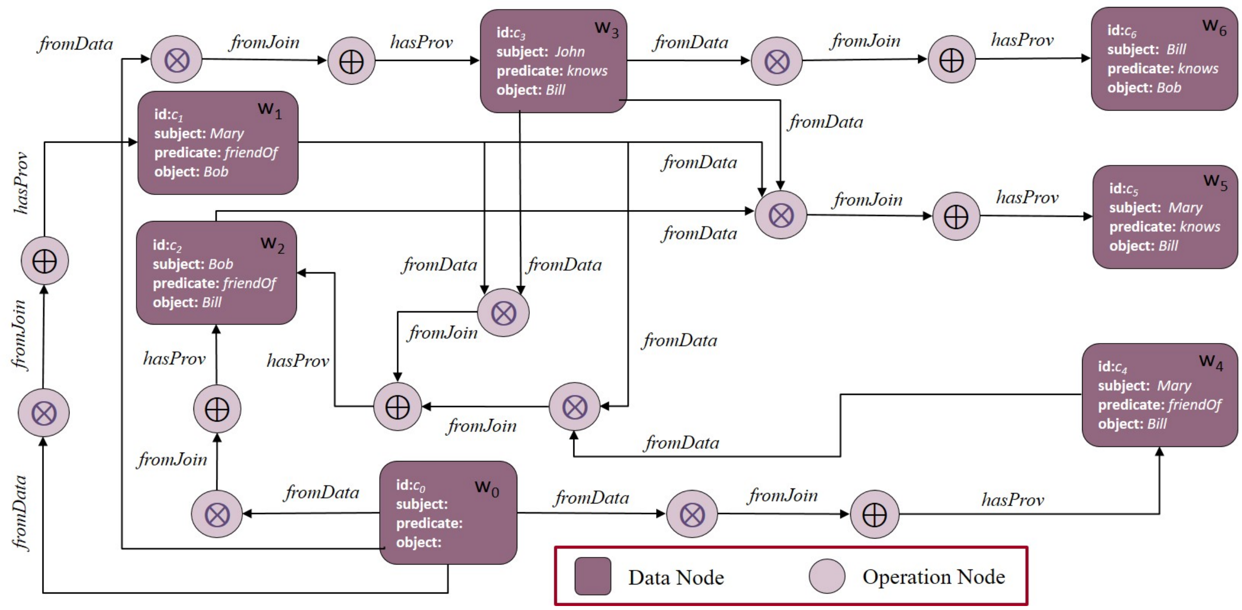

Figure 3: Example of a provenance graph

construct a ⊕-node and connect it to the data node w_2 with a directed edge, labelled as "hasProv" (lines 5-7). Then, we create a ⊗-node for each individual provenance expression of pe_1 and pe_2 (line 10), and connect it to the corresponding ⊕-node (line 11). Finally, for each individual provenance expression, we connect the data nodes that contribute to the JOIN operation to that ⊗-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 = 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 = new DataNode(d_i, t_i)
2: **if** $w_i \notin W_{data}$ **then**
3: $\quad W_{data} = W_{data} \cup \{w_i\}$
4: **return** w_i

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

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 = new Edge("start", "dest", "l")
2: $E = E \cup \{e\}$
3: **return** e

Algorithm 4 Provenance Graph Construction Algorithm

Require: A provenance-aware repository $\mathbb{R}$, $r_i(d_i, t_i, prov_i) \in \mathbb{R}$
Ensure: A provenance graph $\mathcal{G}(W, E)$
1: w_0 = new DataNode(d_0)
2: **for all** $r_i \in \mathbb{R}$ **do**
3: $\quad w_i = $ CREATEDATANODE(d_i, t_i)
4: **for all** $r_i \in \mathbb{R}$, $w_i \in W$ **do**
5: $\quad$ **for all** $pe_k \in prov_i$ **do**
6: $\quad\quad$ currNode = CREATEOPNODE("union")
7: $\quad\quad$ CREATEEDGE(currNode, w_i, "hasProv")
8: $\quad\quad$ **for all** $ipe_m \in pe_k$ **do**
9: $\quad\quad\quad$ previousNode = currNode
10: $\quad\quad\quad$ currNode = CREATEOPNODE("join")
11: $\quad\quad\quad$ CREATEEDGE(currNode,previousNode, "fromJoin")
12: $\quad\quad\quad$ **for all** $d_j \in ipe_m$ **do**
13: $\quad\quad\quad\quad$ CREATEEDGE(w_j,currNode, "fromData")
14: **return** $\mathcal{G}(W, E)$

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 *ProvQL* 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/*how-provenance* case.

During translation, we verify that the *ProvQL* 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 $\mathbb{D}$), or to some element of the data record (subject, predicate, object), but not both.

For the translation, we map each variable in the *ProvQL* 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 *ProvQL* query contributes to the generation of the RETURN_Pattern in Cypher (and sometimes also 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 *ProvQL* query translates into a clause in the Cypher query. Note that, as *ProvQL* is a specialised query language, the translation into a generic language (Cypher in our case) generates a much more complex query, as expected.

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

Cypher	`MATCH (b:Operation) -[:hasProv] ->`
	`(a:Data) WITH a,b MATCH h = (c:Data)`
	`-[:fromData] -> (d:Operation) -`
	`[:fromJoin]-> (b)`
	`WHERE c.subject = s AND`
	`c.predicate = p AND c.object = o`
	`RETURN a.{subject,predicate,object}`
	`as data_a`

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 RDFProv [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 *ProvQL* 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 *ProvQL*, a query language for data enriched with provenance information. *ProvQL* 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 *ProvQL*, 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 *ProvQL* on top of that, and comparing the performance between the two different approaches.

References

[1] Argyro Avgoustaki, Giorgos Flouris, Irini Fundulaki, and Dimitris Plexousakis. Provenance management for evolving rdf datasets. In *European Semantic Web Conference*, pages 575–592. Springer, 2016.

[2] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. Uldbs: Databases with uncertainty and lineage. Technical report, Stanford, 2005.

[3] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In *International conference on database theory*, pages 316–330. Springer, 2001.

[4] Artem Chebotko, Shiyong Lu, Xubo Fei, and Farshad Fotouhi. Rdfprov: A relational rdf store for querying and managing scientific workflow provenance. *Data & Knowledge Engineering*, 69(8):836–865, 2010.

[5] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4), 2009.

[6] Neo4j Corp. Neo4j's graph query language: An introduction to cypher.

[7] Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. *VLDB Journal*, 12(1):41–58, 2003.

[8] Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. Provenance for sparql queries. In *International Semantic Web Conference*, pages 625–640. Springer, 2012.

[9] Giorgos Flouris, Irini Fundulaki, Panagiotis Pediaditis, Yannis Theoharis, and Vassilis Christophides. Coloring rdf triples to capture provenance. In *International Semantic Web Conference*. Springer, 2009.

[10] Boris Glavic and Gustavo Alonso. The perm provenance management system in action. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1055–1058, 2009.

[11] Todd J Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40, 2007.

[12] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. Technical report, W3C, 2013.

[13] Olaf Hartig. Provenance information in the web of data. In *Linked Data On the Web*, 2009.

[14] Grigoris Karvounarakis, Irini Fundulaki, and Vassilis Christophides. Provenance for linked data. In *In Search of Elegance in the Theory and Practice of Computation*, pages 366–381. Springer, 2013.

[15] Grigoris Karvounarakis, Zachary G Ives, and Val Tannen. Querying data provenance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 951–962, 2010.

[16] Frank Manola and Eric Miller. Rdf primer. Technical report, W3C, 2004.

[17] Luc Moreau. The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2-3), 2010.

[18] Luc Moreau and Paolo Missier. PROV-DM: the PROV data model. W3C Recommendation, 2013.

[19] Vicky Papavasileiou, Ken Yocum, and Alin Deutsch. Ariadne: Online provenance for big graph analytics. In *Proceedings of the 2019 International Conference on Management of Data*, pages 521–536. ACM, 2019.

[20] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.

[21] Sherif Sakr, Marcin Wylot, Raghava Mutharaju, Danh Le Phuoc, and Irini Fundulaki. *Linked Data: Storing, Querying, and Reasoning*. Springer, 2018.

[22] Wang Chiew Tan. Provenance in databases: Past, current, and future. *IEEE Data Eng. Bull.*, 30(4), 2007.

[23] Maria Theodoridou, Yannis Tzitzikas, Martin Doerr, Yannis Marketakis, and Valantis Melessanakis. Modeling and querying provenance by extending CIDOC CRM. *Distributed and Parallel Databases*, 27(2):169–210, 2010.

[24] Yannis Theoharis, Irini Fundulaki, Grigoris Karvounarakis, and Vassilis Christophides. On provenance of queries on semantic web data. *IEEE Internet Computing*, 15(1):31–39, 2010.

[25] Marcin Wylot, Philippe Cudre-Mauroux, and Paul Groth. TripleProv: Efficient processing of lineage queries in a native RDF store. In *Proceedings of the 23^{rd} International Conference on World Wide Web*, 2014.

[26] Marcin Wylot, Philippe Cudre-Mauroux, and Paul Groth. Executing provenance-enabled queries over web data. In *WWW*, 2015.

[27] Jun Zhao, Carole Goble, Robert Stevens, and Daniele Turi. Mining taverna's semantic web of provenance. *Concurrency and Computation: Practice and Experience*, 20(5):463–472, 2008.

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 $-->$ between two nodes, e.g. $(n)-->(m)$, 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".*
  ```
  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.*
  ```
  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).

Appearance of $?v$	**Type** ($TypeVar(?v)$)
$dataFunc(?v)$	ID
$provFunc(?v)$	ID
Anywhere in a provExp	ID
$dataFunc(t) = (?v,x,y)$	$SUB(t)$
$dataFunc(t) = (x,?v,y)$	$PRED(t)$
$dataFunc(t) = (x,y,?v)$	$OBJ(t)$

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 7: Translations (tr_Q)

ProvQL Expression	Translation to Cypher		
	MATCH_Pattern	WHERE_Pattern	RETURN_ Pattern
?v, where $TypeVar(?v) = ID$			var.id
?v, where $TypeVar(?v) = SUB(t)$			RETURN t.subject
?v, where $TypeVar(?v) = PRED(t)$			RETURN t.predicate
?v, where $TypeVar(?v) = OBJ(t)$			RETURN t.object
DATA(?v) when DATA(?v) appears in the selectPattern	MATCH (v:Data) WITH v		RETURN v.{subject, predicate, object} as data_v
DATA(?v) when DATA(?v) appears in the evalPattern	MATCH (v:Data) WITH v		
PROV(?v) when PROV(?v) appears in the selectPattern	MATCH (v_n3:Operation)-[:hasProv]-> (v:Data) WITH v_n3, v MATCH v_p=(v_n1:Data)-[:fromData]-> (v_n2:Operation)-[:fromJoin]-> (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		RETURN v_p
PROV(?v) when PROV(?v) appears in the evalPattern	MATCH (v_n3:Operation)-[:hasProv]-> (v:Data) WITH v_n3, v MATCH v_p=(v_n1:Data)-[:fromData]-> (v_n2:Operation)-[:fromJoin]-> (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		

Continued on next page

ProvQL Expression	Translation to Cypher		
	MATCH_Pattern	WHERE_Pattern	RETURN_ Pattern
IPROV(?v) when IPROV(?v) appears in the selectPattern	MATCH (v_n2:Operation)-[:fromJoin]-> (v_n3:Operation)-[:hasProv]-> (v:Data) WITH v, v_n2,v_n3 MATCH v_p=(v_n1:Data)-[:fromData]-> (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		RETURN v_p
IPROV(?v) when IPROV(?v) appears in the evalPattern	MATCH (v_n2:Operation)-[:fromJoin]-> (v_n3:Operation)-[:hasProv]-> (v:Data) WITH v, v_n2, v_n3 MATCH v_p=(v_n1:Data)-[:fromData]-> (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		
DATA(?v1) <> DATA(?v2)	[[DATA(?v1)]] [[DATA(?v2)]]	WHERE v1.id <> v2.id	
DATA(?v) = recordExp if recordExp = (s,p,o) and $s, p \in \mathbb{I}, o \in \mathbb{I} \cup \mathbb{L}$	[[DATA(?v)]]	WHERE v.subject = s AND v.predicate = p AND v.object = o	
DATA(?v) <> recordExp if recordExp = (s,p,o) and $s, p \in \mathbb{I}, o \in \mathbb{I} \cup \mathbb{L}$	[[DATA(?v)]]	WHERE v.subject = s OR v.predicate = p OR v.object = o	
PROV(?v1) <> PROV(?v2)	[[PROV(?v1)]] [[PROV(?v2)]], v1,v1_p	WHERE v1.id <> v2.id AND (apoc.util.md5(v1_t1) <> apoc.util.md5(v2_t1) OR apoc.util.md5(v1_t2) <> apoc.util.md5(v2_t2) OR apoc.util.md5(v1_t3) <> apoc.util.md5(v2_t3))	
PROV(?v1) = PROV(?v2)	[[PROV(?v1)]] [[PROV(?v2)]], v1,v1_p	WHERE v1.id <> v2.id AND apoc.util.md5(v1_t1) = apoc.util.md5(v2_1) AND apoc.util.md5(v1_t2) = apoc.util.md5(v2_t2) AND apoc.util.md5(v1_t3) = apoc.util.md5(v2_t3)	

Continued on next page

ProvQL Expression	Translation to Cypher		
	MATCH_Pattern	WHERE_Pattern	RETURN_ Pattern
PROV(?v1) <> IPROV(?v2)	[[PROV(?v1)]] [[IPROV(?v2)]], v1,v1_p	WHERE v1.id <> v2.id AND (apoc.util.md5(v1_t1) <> apoc.util.md5(v2_t1) OR apoc.util.md5(v1_t2) <> apoc.util.md5(v2_t2))	
PROV(?v1) = IPROV(?v2)	[[PROV(?v1)]] [[IPROV(?v2)]], v1,v1_p	WHERE v1.id <> v2.id AND apoc.util.md5(v1_t1) = apoc.util.md5(v2_1) AND apoc.util.md5(v1_t2) = apoc.util.md5(v2_t2)	
IPROV(?v1) <> IPROV(?v2)	[[IPROV(?v1)]], v1,v1_p [[IPROV(?v2)]], v1,v1_p	WHERE v1.id <> v2.id AND (apoc.util.md5(t1) <> apoc.util.md5(s1) OR apoc.util.md5(t2) <> apoc.util.md5(s2))	
IPROV(?v1) = IPROV(?v2)	[[IPROV(?v1)]], v1,v1_p [[IPROV(?v2)]], v1,v1_p	WHERE v1.id <> v2.id AND apoc.util.md5(t1) = apoc.util.md5(s1) AND apoc.util.md5(t2) = apoc.util.md5(s2)	
PROV(?v1) INCLUDES PROV(?v2)	[[PROV(?v1)]], collect(v1_p) as v1_path, size(()-[]->()-[]->(v1_n3)) as v1_s [[PROV(?v2)]], collect(v2_p) as v2_path, size(()-[]->()-[]->(v2_n3)) as v2_s	WHERE ALL(d IN v2_path WHERE d IN v1_path AND v1.id <> v2.id AND v1_s=v2_s)	
PROV(?v1) INCLUDES IPROV(?v2)	[[PROV(?v1)]], collect(v1_p) as v1_path, size(()-[]->()-[]->(v1_n3) as v1_s [[IPROV(?v2)]], collect(v2_p) as v2_path size(()-[]->(v2_n2)) as v2_s	WHERE ALL(d IN v2_path WHERE d IN v1_path AND v1.id <> v2.id AND v1_s=v2_s)	
IPROV(?v1) INCLUDES PROV(?v2)	[[IPROV(?v1)]], collect(v1_p) as v1_path, size(()-[]->>(v1_n2) as v1_s [[PROV(?v2)]], collect(v2_p) as v2_path size(()-[]->()->(v2_n3)) as v2_s	WHERE ALL(d IN v2_path WHERE d IN v1_path AND v1.id <> v2.id AND v1_s = v2_s)	
IPROV(?v1) INCLUDES IPROV(?v2)	[[IPROV(?v1)]], collect(v1_p) as v1_path , size(()-[]->>(v1_n2) as v1_s [[IPROV(?v2)]], collect(v2_p) as v2_path size(()-[]->(v2_n2)) as v2_s	WHERE ALL(d IN v2_path WHERE d IN v1_path AND v1.id <> v2.id AND v1_s=v2_s)	
PROV(?v1) CONTAINS PROV(?v2)	[[PROV(?v1)]], collect(v1_p) as v1_path1,[[IPROV(?v1)]] as v1_path2 [[PROV(?v2)]], collect(v2_p) as v2_path	WHERE ALL(d IN v2_path WHERE (d IN v1_path1 OR d IN v1_path2) AND v1.id <> v2.id)	

Continued on next page

ProvQL Expression	**Translation to Cypher**		
	MATCH_Pattern	**WHERE_Pattern**	**RETURN_ Pattern**

`PROV(?v1)`
`CONTAINS`
`IPROV(?v2)`
	`[[PROV(?v1)]], collect(v1_p) as v1_path1,[[IPROV(?v1)]] as v1_path2 [[IPROV(?v2)]], collect(v2_p) as v2_path`	`WHERE ALL(d IN v2_path WHERE (d IN v1_path1 OR d IN v1_path2) AND v1.id <> v2.id)`	

`IPROV(?v1)`
`CONTAINS`
`IPROV(?v2)`
	`[[IPROV(?v1)]], collect(v1_p) as v1_path, [[IPROV(?v2)]], collect(v2_p) as v2_path`	`WHERE ALL(d IN v2_path WHERE (d IN v2_path IN v1_path) AND v1.id <> v2.id)`	

`IPROV(?v1)`
`CONTAINS`
`PROV(?v2)`
	`[[IPROV(?v1)]], collect(v1_p) as v1_path, [[PROV(?v2)]], collect(v2_p) as v2_path`	`WHERE ALL(d IN v2_path WHERE (d IN v2_path IN v1_path) AND v1.id <> v2.id)`	

`PROV(?v)`
`INCLUDES`
`provExp`
	`[[PROV(?v)]], v_n2,`
	`size(()-[]->(v_n2)) as v_s`
	`MATCH (m1)-[r1:fromData]->(v_n2)`
	`WITH v_n2, v_s, v, r1`
	`MATCH (m2)-[r2:fromData]->(v_n2)`
	`WITH v_n2, v_s, v, r1,r2`
	`...`
	`WITH v_n2, v_s, v, r1,r2, ... rn-1`
	`MATCH (mn)-[rn:fromData]->(v_n2)`
		`WHERE ID(r_1) <> ID(r_2) AND ID(r_1) <> ID(r_3) AND ... ID(r_1) <> ID(r_n) AND ID(r_2) <> ID(r_3) AND ... ID(r_{n-1}) <> ID(r_n) AND v_s=n`
		
		if provExp = $m_1 \otimes m_2 \otimes ... \otimes m_n$	

	`[[PROV(?v)]], v_n3,`
	`size(()-[:fromData]->(v_n2)-[:fromJoin]->(v_n3)) as v_s`
	`MATCH (m11)-[r11:fromData]->(k11)-[:fromJoin]->(v_n3)`
	`WITH v_n2, v_s, v, r11,k11,`
	`size(()-[]->(k1)) as m11_size`
	`MATCH (m12)-[r12:fromData]->(k12)-[:fromJoin]->(v_n3)`
	`...`
		WHERE $ID(r_{1,1})$ <> $ID(r_{1,2})$ *AND* $ID(r_{1,1})$ <> $ID(r_{1,3})$ AND ... $ID(r_{1,1})$ <> $ID(r_{1,n})$ AND $ID(r_{2,1})$ <> $ID(r_{2,2})$ AND ... $ID(r_{k,1})$ <> $ID(r_{k,2})$ AND ... $ID(r_{k,n-1})$ <> $ID(r_{k,n})$ AND ... AND $v_s = n$ *AND* $v_d = k$
		
		if provExp = $m_{1,1} \otimes m_{1,2} \otimes ... m_{1,n} \oplus m_{2,1} \otimes ... m_{2,n} \oplus ... \oplus m_{k,1} \otimes ... m_{k,n}$	

`PROV(?v)`
`INCLUDES`
`provExp`
	`[[PROV(?v)]], v_n2,`
	`size(()-[]->(v_n2)) as v_s`
		`WHERE v_s=1`
		
		if provExp = m_1	

`PROV(?v)))`
`CONTAINS`
`provExp`
	`[[PROV(?v)]], v_n2`
	`MATCH (m1)-[r1:fromData]->(v_n2)`
	`WITH v_n2, v_s, v, r1`
	`MATCH (m2)-[r2:fromData]->(v_n2)`
	`WITH v_n2, v_s, v, r1,r2`
	`...`
	`WITH v_n2, v_s, v, r1,r2, ... rn-1`
	`MATCH (mn)-[rn:fromData]->(v_n2)`
		WHERE $ID(r_1)$ <> $ID(r_2)$ *AND* $ID(r_1)$ <> $ID(r_3)$ *AND* ... $ID(r_1)$ <> $ID(r_n)$ *AND* $ID(r_2)$ <> $ID(r_3)$ *AND* ... $ID(r_{n-1})$ <> $ID(r_n)$
		
		if provExp = $m_1 \otimes m_2 \otimes ... \otimes m_n$	

Continued on next page

ProvQL Expression	Translation to Cypher		
	MATCH_Pattern	WHERE_Pattern	RETURN_Pattern
	[[PROV(?v)]], v_n3, v_n2 MATCH (m11)-[r11:fromData]->(v_n2) -[:fromJoin]->(v_n3) WITH v_n2, v_s, v, r11 MATCH (m12)-[r12:fromData]->(v_n2) -[:fromJoin]->(v_n3) WITH v_n2, v_s, v, r11,r12 ... MATCH (mkn)-[rkn:fromData]->(v_n2) -[:fromJoin]->(v_n3)	WHERE $ID(r_{1_1}) <> ID(r_{1,2})$ *AND* $ID(r_{1,1}) <> ID(r_{1,3})$ *AND* ... $ID(r_{1,1}) <> ID(r_{1,n})$ *AND* $ID(r_{2,1}) <> ID(r_{2,2})$ *AND* ... $ID(r_{k,1}) <> ID(r_{k,2})$ *AND* ... $ID(r_{k,n-1}) <> ID(r_{k,n})$ *AND* if provExp = $m_{1,1} \otimes m_{1,2} \otimes ... m_{1,n}$ $\oplus\, m_{2,1} \otimes ... m_{2,n}$ $\oplus ... \oplus\, m_{k,1} \otimes ... m_{k,n}$	
	[[PROV(?v)]] if provExp = m_1		
IPROV(?v) INCLUDES provExp	[[IPROV(?v)]], v_n2, size(()-[]->(v_n2)) as v_s MATCH (m1)-[r1:fromData]->(v_n2) WITH v_n2, v_s, v, r1 MATCH (m2)-[r2:fromData]->(v_n2) WITH v_n2, v_s, v, r1,r2 ... WITH v_n2, v_s, v, r1,r2, ... rn-1 MATCH (mn)-[rn:fromData]->(v_n2)	WHERE $ID(r_1) <> ID(r_2)$ *AND* $ID(r_1) <> ID(r_3)$ *AND* ... $ID(r_1)$ $<> ID(r_n)$ *AND* $ID(r_2) <> ID(r_3)$ *AND* ... $ID(r_{n-1}) <> ID(r_n)$ *AND* $v_s = n$ if provExp = $m_1 \otimes m_2 \otimes ... \otimes m_n$	
IPROV(?v) CONTAINS provExp	[[IPROV(?v)]], v_n2, MATCH (m1)-[r1:fromData]->(v_n2) WITH v_n2, v_s, v, r1 MATCH (m2)-[r2:fromData]->(v_n2) WITH v_n2, v_s, v, r1,r2 ... WITH v_n2, v_s, v, r1,r2, ... rn-1 MATCH (mn)-[rn:fromData]->(v_n2)	WHERE $ID(r_1) <> ID(r_2)$ *AND* $ID(r_1) <> ID(r_3)$ *AND* ... $ID(r_1)$ $<> ID(r_n)$ *AND* $ID(r_2) <> ID(r_3)$ *AND* ... $ID(r_{n-1}) <> ID(r_n)$ if provExp = $m_1 \otimes m_2 \otimes ... \otimes m_n$	