

# Querying Data Provenance

Grigoris Karvounarakis<sup>\*</sup>  
LogicBlox ICS-FORTH  
Atlanta, GA, USA Heraklion, Greece  
gregkar@gmail.com

Zachary G. Ives Val Tannen  
University of Pennsylvania  
Philadelphia, PA, USA  
{zives, val}@cis.upenn.edu

## ABSTRACT

Many advanced data management operations (e.g., incremental maintenance, trust assessment, debugging schema mappings, keyword search over databases, or query answering in probabilistic databases), involve computations that look at how a tuple was produced, e.g., to determine its score or existence. This requires answers to queries such as, “Is this data derivable from trusted tuples?”; “What tuples are derived from this relation?”; or “What score should this answer receive, given initial scores of the base tuples?”. Such questions can be answered by consulting the *provenance* of query results.

In recent years there has been significant progress on formal models for provenance. However, the issues of provenance storage, maintenance, and querying have not yet been addressed in an application-independent way. In this paper, we adopt the most general formalism for tuple-based provenance, *semiring provenance*. We develop a query language for provenance, which can express all of the aforementioned types of queries, as well as many more; we propose storage, processing and indexing schemes for data provenance in support of these queries; and we experimentally validate the feasibility of provenance querying and the benefits of our indexing techniques across a variety of application classes and queries.

## Categories and Subject Descriptors

H.2.3 [Database Management]: Languages—*Query languages*;

H.2.4 [Database Management]: Systems—*Query processing*

## General Terms

Languages, Performance, Algorithms

## Keywords

Data provenance, annotation, query language, query processing

## 1. INTRODUCTION

In the sciences, in intelligence, in business, the same adage holds true: data is only as credible as its source. Recently we have begun to see issues like data quality, uncertainty, and authority make

<sup>\*</sup>Work performed while the first author was a Ph.D. candidate at the University of Pennsylvania.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.  
Copyright 2010 ACM 978-1-4503-0032-2/10/06 ...\$10.00.

their way from separate data processing stages, into the very foundations of database systems: data models, mapping definitions, and query languages. Typically, the notion of *data provenance* [12, 18, 29] lies at the heart of assessing authority or uncertainty. Systems like Trio [6] compute provenance or lineage, then use this to derive probabilities associated with answers; systems like ORCHES-TRA [28] record provenance as they propagate data and updates across schema mappings from one database to another, and use provenance to assess trust and authority. Recently [41] provenance has even been shown useful in learning the authority of data sources and schema mappings, based on user feedback over results: a system can learn adjustments to rankings of queries based on feedback over their answers, and it can then propagate this adjustment to the score of one or more relations. Finally, provenance has been used to debug schema mappings [14] that may be imprecise or incorrect: users can see how “bad” data has been produced. (We note that our focus is on *data provenance*, based on declarative mappings, rather than workflow provenance, a separate topic [8, 13, 38].)

Surprisingly, the study of data provenance as a first-class data artifact — worthy of its own data model, query language, and indexing and query processing techniques — has not yet come into the forefront. We believe these topics are of increasing importance, as databases begin to incorporate provenance. There are a variety of reasons why provenance storage and querying support would be advantageous if fully integrated into a DBMS query system.

**Interactive provenance browsers and viewers.** In many applications, ranging from debugging [14] to scientific assessment of data quality [9, 28], users would like to *visualize* the relationship between tuples in different relations, or the derivation of certain results, without being overwhelmed by complexity. This requires a convenient way to (1) explore the (typically large and complex) graph of tuples and derivations, and (2) request and isolate portions of it. Declarative querying is advantageous here: it provides a high-level model for developers of graphical tools to retrieve data, without needing to know the details of its physical representation.

**Developing generalized materialized view support for multiple scoring/ranking models.** Uncertain data has been intensively studied in recent years, with a variety of ranked and probabilistic formulations developed. Such work typically develops a scheme to derive probabilities or scores “on the fly,” based on how extensional (base) tuples are combined. Given a very general tuple-based provenance model such as [29], we can materialize a single view and its provenance — and from this we can efficiently compute any of a variety of scores or annotations through provenance queries.

**Incorporation of generalized trust and confidentiality levels into views.** As materialized data is passed along from system to system, it may be useful to annotate the data with information about the access levels required to see certain portions of it [24, 40]; or,

conversely, to compute from its provenance an *authoritativeness score* to determine how much to *trust* the data [42].

**Efficient indexing schemes for provenance.** Declarative query techniques can benefit from indexing strategies for provenance, and potentially offer better performance than ad hoc primitives.

In Section 2 we show examples of provenance queries and identify a partial list of important use cases for a provenance query language. Our motivation for studying provenance queries comes from developing provenance support within collaborative data sharing systems (CDSSs), a new architecture for data sharing established by the ORCHESTRA [28] and Youtopia [36] systems. In such systems, a variety of sites or *peers*, each with a database, agree to share information. Peers are linked to one another using a network of compositional *schema mappings*, which allow data or updates applied to one peer to be transformed and applied to another peer. A key aspect of such systems is that they support tracking of the provenance of data items as they are mapped from site to site — and they use this provenance to support incremental update propagation (essentially, view maintenance) [28, 36], conflict resolution [42], and ranked result computation [41]. CDSSs use provenance internally, but have, to this point, relied on custom procedural code to perform provenance-based computations. In order to make provenance fully available to users and application developers, we make the following contributions:

- A query language for data provenance, ProQL, useful in supporting a wide variety of applications with derived information. ProQL is based on the more compact graph-based representation [28] of the rich provenance model of [29], and can compute various forms of *annotations*, such as *scores*, for data based on its provenance.
- A general data provenance encoding in relations, which allows storage of provenance in an RDBMS while incurring a modest space overhead.
- A translation scheme from ProQL to SQL queries which can be executed over an RDBMS used for provenance storage.
- Indexing strategies for speeding up certain classes of provenance queries.
- An experimental analysis of the performance of ProQL query processing and the speedup yielded by employing different indexing strategies.

Our work generalizes beyond the CDSS setting, to analogous computations over materialized views in traditional databases. It is also relevant in a variety of problem settings such as computing probabilities for materialized tuples based on event expressions (as in Trio [6]), or to facilitate debugging of schema mappings (as in SPIDER [14]). ProQL was designed for the provenance model of [29], extended to record schema mapping involved in derivations [31]. This model is slightly more general than the models of Trio [6] and Perm [26]. However, a subset of our language could be implemented over such systems, providing them with provenance query support that matches the capabilities of their models.

The rest of the paper is organized as follows. In Section 2 we present our problem setting and some example use cases. In Section 3 we propose the syntax and semantics of ProQL, a language for querying data provenance. In Section 4, we describe a scheme for storing provenance information in relations and evaluating ProQL queries over an RDBMS. Section 5 proposes indexing techniques for provenance that can be used to answer ProQL queries more rapidly. We illustrate the performance of ProQL query processing

and the speedup of these indexing techniques in Section 6. Finally, we discuss related work in Section 7 and conclude and describe future work in Section 8.

## 2. SETTING AND MOTIVATING USE CASES

Our study of provenance comes from the CDSS arena, where different autonomous databases are linked by declarative schema mappings, and data and updates are propagated across those mappings. We briefly describe the main ideas of schema mappings and their relationship to provenance in this section, and also how these ideas generalize to settings with traditional views. Then we provide a set of usage scenarios and use cases for provenance itself — and hence for our provenance query capabilities.

*EXAMPLE 2.1. Suppose we have three data sharing participants,  $P_1, P_2, P_3$ , all interested in information about animals, their sizes, and the various (scientific and common) names by which they may be referred. Let the public schema of  $P_1$  be the relations *Animal* (*id*, *scientificName*, *length*) and *CommonName* (*id*, *name*); the public schema of  $P_2$  be the single relation *Names* (*id*, *name*, *isCanonical*); and the public schema of  $P_3$  be the single relation *Organism* (*name*, *height*, *isAnimal*).*

*For simplicity we will abbreviate the relation names to their first letter, as  $A, C, N$ , and  $O$ . Each of these relations represents the union of data contributed or created locally by each participant, plus data imported by the participant. We can define a local contributions table for each of the relations above, respectively  $A_i, C_i, N_i$ , and  $O_i$ . To copy all data from  $A_i, C_i, N_i$ , and  $O_i$  to the corresponding public schema relations, we can use the following set of Datalog rules:*

$$\begin{aligned} L_1 : A(i, s, l) &:- A_i(i, s, l) \\ L_2 : C(i, n) &:- C_i(i, n) \\ L_3 : N(i, n, c) &:- N_i(i, n, c) \\ L_4 : O(n, h, a) &:- O_i(n, h, a) \end{aligned}$$

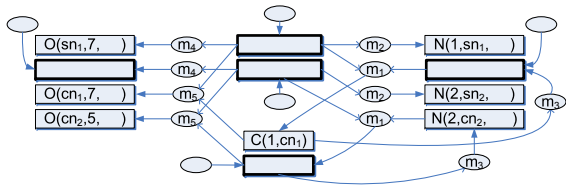
*Finally, we may inter-relate the various public schema relations through a series of schema mappings, also expressed within a superset of Datalog<sup>1</sup>, as the following:*

$$\begin{aligned} m_1 : C(i, n) &:- A(i, s, \_), N(i, n, false) \\ m_2 : N(i, n, true) &:- A(i, n, \_) \\ m_3 : N(i, n, false) &:- C(i, n) \\ m_4 : O(n, h, true) &:- A(i, n, h) \\ m_5 : O(n, h, true) &:- A(i, \_, h), C(i, n) \end{aligned}$$

Observe that each public schema relation is in essence a (possibly recursive) view. Data and updates from each peer are exchanged by materializing this set of views [28, 39]. Our model is in fact a **generalization of one in which multiple views are composed over one another**, and all of the techniques in this paper will apply equally to that setting.

The process of executing the set of extended-Datalog rules provided above is an instance of *data exchange* [21], and produces a set of materialized data instances that form a *canonical universal solution*. In this solution, as with any materialized view in set-semantics, each tuple in the view may have been derivable in multiple ways, e.g., due to a projection within a mapping, or a union of data from two different mappings. The set of such derivations

<sup>1</sup>To capture the full generality of standard “tuple generating dependency” or GLAV schema mappings [30], Datalog must be extended to support *Skolem functions* that provide a mechanism for creating special *labeled null* values that may represent *the same value* in multiple data instances. These details are not essential to the understanding of this paper, and hence we omit them in our discussion, though our implementation fully supports such mappings.



**Figure 1: Example provenance graph (rectangles are tuples, ellipses are derivations, and ovals with '+' represent original base data, also shown as boldface).**

are what we term the *provenance* of each tuple, and they can be described in terms of *base data* (EDBs), other derived tuples (for recursive derivations), and mappings (using the name that we have assigned to each rule). Moreover, a tuple may be the result of *composition* of mappings (which may also involve joins): e.g., a tuple may be derived in instance  $O$  as a result of applying  $m_5$  to data in  $C$  that was mapped from  $A$  and  $N$  along  $m_1$ . Our goal is to record *how data was derived through the mappings*.

Figure 1 illustrates the result of data exchange over the mappings of Example 2.1 along with the relationship between tuples and their derivations. This *provenance graph* has two types of nodes: tuples (represented in rectangles, labeled with the values of the tuples) and derivations (represented as ellipses, labeled with the names of the mappings). This graph describes the relative derivations of tuples in terms of one another; local contribution tables for each relation contain the tuples indicated by boldface, and their presence is indicated by oval nodes with a '+'. Given a tuple node in the provenance graph, we can find its *alternate direct derivations* by finding the set of derivation nodes that have directed edges pointing to it. (These represent union.) In turn, each derivation has a set of  $m$  source tuples that are joined — the set of tuple nodes with edges going to the derivation node — and a set of  $n$  consequents — the set of tuple nodes that have edges pointing to them from the derivation node. The unique properties of our graph model will provide a desideratum for our provenance query language semantics: whenever we return a derivation node in the output, we will *also* want all  $m$  source nodes and  $n$  target nodes, to maintain the meaning of the derivation. This contrasts with the graph data models of [1, 16, 22, 32].

### Use Cases for Provenance Graph Queries

Given this provenance graph, there are many scenarios where a user (especially through a graphical tool) may want to retrieve and browse a portion of the graph. Based on our discussions with scientific users, and on previous work in the data integration community, we consider several query use cases.

**Q1. The ways a tuple was derived.** A scientist, intelligence analyst, or author of mappings [14] may want to visualize the different ways a tuple can be derived — including the source tuple values and the combination of mappings used. This is essentially a projection of the provenance graph, containing all base tuples from which the tuple of interest is derivable, as well as the derivations themselves, including the mappings involved and intermediate tuples that were produced. This graph may be visualized for the user.

**Q2. Relationships between tuples.** One may also be interested in restricting the set of derivations to those involving tuples from a certain source or derived relation or set of relations, e.g., if that relation is known to be authoritative [9].

**Q3. Results derivable from a given mapping or view.** The above use cases started with a tuple and considered its provenance. Conversely, we can query the provenance for tuples derived using a particular mapping (as is useful in [14]) or from a particular source.

### Q4. Identifying tuples with common/overlapping provenance

As data is propagated along different paths in a CDSS, it may be useful to be able to determine at a given time whether tuples at two different peers have some common provenance. For instance, suppose we are trying to assess trustworthiness of information according to the number of peers in which it appears *independently* [20]. In that case, it is important to be able to identify when information came from the same peer or source.

## 2.1 From Provenance to Tuple Annotations

In the previous section and in our actual storage model, we focus on provenance as a graph. However, formally this graph encodes a (possibly recursively defined) set of *provenance polynomials* in a *provenance semiring* [29] (also called *how-provenance*). This correspondence is useful in computing *annotations* like scores, probabilities, counts, or derivability of tuples in a view.

Suppose we are given a provenance graph such as that of Figure 1, and that every EDB tuple node is annotated with a *base value*: perhaps the Boolean true value if we are testing for derivability, a real-valued tuple weight if we are performing approximate keyword search over the tuples, etc. Then we can compute annotations for the remaining nodes in a *bottom-up* fashion: for any derivation node whose source tuple nodes have all been given annotations, we combine the source tuple nodes' annotation values with an appropriate *abstract product operation*: we AND Boolean values for derivability, or sum tuple weights in the keyword search model. When we reach a tuple node whose derivation nodes have all been given scores, we apply an *abstract sum operation* to determine which annotation to apply to the tuple node: we OR Boolean values from the mappings for derivability, or compute the MIN annotation of the different derivation nodes' weights in the keyword search model. Finally, mappings themselves can affect the resulting annotation, e.g., an untrusted mapping may produce false on all inputs. We repeat the process until all nodes have been annotated.

We can get different types of annotations for different use cases, based on how we instantiate the *base value*, *abstract product operation*, and *abstract sum operation*. The work of [29, 31] provides a formal definition of the properties that must hold for these values and operations, namely that they satisfy the constraints of a *semiring*; but we summarize some useful cases (including a few novel ones) in Table 1. Each row in the table represents a particular use case, and its semiring implementation.

The **derivability** semiring assigns true to all base tuples, and determines whether a tuple (whose annotation must also be true can be derived from them. **Trust** is very similar, except that we must check each EDB tuple to see whether it is trusted — annotating it with true or false. Moreover, each mapping may be associated with the neutral function **Nm**, returning its input value unchanged, or the distrust function **Dm**, returning false on all inputs. Any derived tuples with annotation true are trusted. The **confidentiality level** semiring [24] assigns a confidentiality access level to a tuple derived by joining multiple source tuples: for any join, it assigns the highest (most secure) level of any input tuple to the result; for any union, it assigns the lowest (least secure) level required. The **weight/cost** semiring is useful in ranked models where output tuples are given a cost, evaluating to the sums of the individual scores or weights of atoms joined (and to the lowest cost of different alternatives in a union). This semiring can be used to produce ranked results in keyword search [41] or to assess data quality. The **probability** semiring represents probabilistic event expressions that can be used for query answering in probabilistic databases.<sup>2</sup> The **lin-**

<sup>2</sup>As observed in [19], computing actual probabilities from these event expressions is in general a #P-complete problem. Techniques

**Table 1: Useful mappings of base values and operations in evaluating provenance graphs.**

Use case	base value	product $R \otimes S$	sum $R \oplus S$
Derivability	true	$R \wedge S$	$R \vee S$
Trust	trust condition result	$R \wedge S$	$R \vee S$
Confidentiality level	tuple confidentiality level	more_secure ( $R, S$ )	less_secure ( $R, S$ )
Weight/cost	base tuple weight	$R + S$	$\min(R, S)$
Lineage	tuple id	$R \cup S$	$R \cup S$
Probability	tuple probabilistic event	$R \cap S$	$R \cup S$
Number of derivations	1	$R \cdot S$	$R + S$

**eage** semiring corresponds to the set of all base tuples contributing to some derivation of a tuple. The **number of derivations** semiring counts the number of ways each tuple is derived, as in the bag relational model.

**Cycles (recursive mappings).** For provenance graphs containing cycles (due to recursive mappings) there are certain limitations. The first 5 semirings of Table 1 have idempotence and absorption properties guaranteeing they will remain finite in the presence of cycles (if evaluation is done in a particular way); for the number of derivations semiring, the annotations may not converge to a final value (i.e., we can have infinite counts). In this paper we develop a query language capable of handling cycles (as can occur in a CDSS such as ORCHESTRA, where participants independently specify schema mappings to their individual databases instances). However, we focus our initial implementation on the acyclic case.

#### Use Cases for Tuple Annotation Computation

Within data integration and exchange settings, there are a variety of cases where we would like to assign an annotation to each result in a materialized view, based on its provenance.

**Q5. Whether a tuple remains derivable.** During incremental view maintenance or update exchange, when a base tuple is derived, we need to determine whether existing view tuples remain derivable. Provenance can speed up this test [28].

**Q6. Lineage of a tuple.** During view update or bidirectional update exchange [33] it is possible to determine at run-time whether update propagation can be performed without side effects based on the derivability test of Q5 and the *lineages* [18] of tuples — i.e., the set of all base tuples each can be derived from, without distinguishing among different derivations.

**Q7. Whether to trust a tuple.** In CDSS settings, a set of *trust policies* is used to assign trust/distrust and authority levels to different data sources, views, and mappings — resulting in a *trust level* for each derived tuple based on its provenance [28, 42].

**Q8. A tuple’s rank or score.** In keyword query systems over databases, it is common to represent the data instance or the schema as a graph, where edges represent join paths (e.g., along foreign keys) between relations. These edges may have different *costs* depending on similarity, authority, data quality, etc. These costs may be assigned by the common TF/IDF document/phrase similarity metric, by ObjectRank and similar authority-based schemes [4], or by machine learning based on user feedback about query answers [41]. The score of each tuple is a function of its provenance. If we are given a materialized view in this setting, we may wish to store the provenance, rather than the ranking, in the event that costs over the same edges might be assigned differently based on the user or the query context [41].

**Q9. A tuple’s associated probability.** In Trio [6], a form of provenance (called lineage in [6], though more general than that of [18]) is computed for query results, and then probabilities are from probabilistic databases [19] can be used to compute them more efficiently; this is outside the scope of this paper.

assigned based on this lineage. In similar fashion, we can compute probabilities from a materialized representation of provenance.

**Q10. Computing confidentiality/access control levels for data.** Recent work [24] has shown how provenance can be used to assign access control levels to different tuples in a database. If the tuples might represent “shredded XML,” i.e., a relational representation of an XML document, then the access control level of a tuple (XML node) should be the strictest access control level of any node along the path from the XML root. In relational terms, the access control level of a tuple represents the strictest level of any tuple in a join expression corresponding to path evaluation.

In the next section, we describe a general language for expressing a wide variety of provenance queries, including these use cases.

### 3. A QUERY LANGUAGE FOR PROVENANCE

To address the provenance querying needs of CDSS users, as expressed in the use cases of the previous section, we propose a language, ProQL (for **Provenance Query Language**). We noted previously that our use cases can be divided into ones that (1) help a user or application determine the relationship between sets of tuples, or between mappings and tuples; (2) provide a *score/rank*, *access control level* or *assessment of derivability* or *trust* for a tuple or set of tuples. Consequently, ProQL has two core constructs. The first defines *projections of the provenance graph*, typically with respect to some tuple or tuples of interest. The second specifies how to evaluate a returned subgraph as an expression under a specific semiring, to *compute an annotation* from that semiring for each tuple.

#### 3.1 Core ProQL Semantics

A ProQL query takes as its input a provenance graph  $G$ , like the one of Figure 1. The *graph projection* part of the query:

- Matches parts of the input graph according to *path expressions* (possibly filtering them based on various predicates).
- Binds variables on tuple and derivation nodes of matched paths.
- Returns an output provenance graph  $G'$ , that is a *subgraph* of  $G$  and is composed of the set of paths returned by the query. For each derivation node, every tuple node to which it is related is also returned in  $G'$ , maintaining the arity of the mapping.
- Returns tuples of bindings from distinguished query variables to nodes in  $G'$ , henceforth called *distinguished nodes*.

Note that provenance is a record of how data was related through mappings and data exchange; it does not make sense to be able to independently “create new provenance” within a provenance query language. Hence, unlike GraphLog [16], Lorel [1] or StruQL [22] — but similarly to XPath — ProQL cannot create new nodes or graphs, but always returns a subgraph of the original graph. Moreover, provenance graphs are different from the graph models of those languages, in containing two kinds of nodes (tuple and derivation nodes, where, as previously described, a derivation node is in some sense “inseparable” from the set of tuple nodes it relates).

If the ProQL query only consists of a graph projection part, it returns the subgraph described above, together with sets of bindings for the distinguished variables. The set of bindings accompanying the graph projection is especially useful for the optional next stage: ProQL queries can also support *annotation computation* for the nodes referenced in the binding tuples, using a particular semiring. This is a unique feature of ProQL compared to other graph query languages, that is enabled by the fact that provenance graphs can be used to compute annotations in various semirings, as explained in Section 2.1. The annotation computation part of a ProQL query specifies an assignment of values from a particular semiring (e.g., trust value, Boolean, score) to some of the nodes in  $G'$  and computes the values in that semiring for the distinguished nodes. The result is a set of tuples consisting of pairs (*distinguished node id*, *semiring annotation value*) for each bound variable output by the query.

Due to space limitations, this paper focuses on a single ProQL query block, but our design generalizes to support nested graph projection and annotation computation queries. For the latter, we need to retain both the annotations and the subgraph over which they were computed, in order to evaluate the outer query.

## 3.2 ProQL Syntax

As we explained above, ProQL queries can have two main components, graph projection and annotation computation. The graph projection part can be used independently, if one only needs to compute a projection of a provenance graph. The annotation computation part can apply an assignment to a provenance graph and compute values for its distinguished nodes in the corresponding semiring. To simplify the presentation, we explain the two core constructs of ProQL and their basic clauses, separately. An EBNF grammar for our language can be found in [31].

### 3.2.1 Graph Projection

Unlike the graphs typically considered in semi-structured data, our provenance graph is not rooted. We adopt a path expression syntax where the individual “steps” consist of traversals from a node representing a tuple in a relation, through a node representing a derivation through a mapping, to another node representing a tuple. We refer to the actual nodes in the provenance graph as *tuple nodes* and *derivation nodes*, respectively. Within the path expression, we may restrict the tuple nodes to belong to a certain relation, or the derivation nodes to belong to a certain mapping. We may also bind variables to either type of node. We use the syntax:

[*relation-name variable*]

to indicate tuple nodes (where both *relation-name* and *variable* are optional), and one of the three forms:

<- | < *mapping-name* | < *variable*

to indicate derivation nodes (belonging to the corresponding mapping). A schema mapping  $M$  in general may have  $m$  source atoms and  $n$  target atoms. Thus, in contrast to other graph models and query languages, even if a path expression includes one source and/or target atom, any matched derivation node corresponding to  $M$  will have  $n$  tuple nodes to its left and  $m$  tuple nodes to its right. We also allow for arbitrary paths (compositions of multiple steps) between nodes, using the notation <-+ for paths of length one or more. Paths may not be bound to variables.

Given this path notation, we outline our basic ProQL syntax, comprising 4 basic clauses (see [31] for further detail).

**FOR:** This clause binds variables (whose names are prefixed with the \$ character) to sets of tuple and/or derivation nodes in the graph, through path expressions.

**WHERE:** This clause is used to specify filtering conditions on the

variables bound in the FOR clause. Conditions on tuple nodes may be expressed over the attributes of the tuple, or over the name of the relation in which it belongs. Derivation nodes may be tested for their mapping name. If path expressions are included in the WHERE clause they are evaluated as existential conditions.

**INCLUDE PATH:** For each set of bound variables satisfying the WHERE clause, this clause specifies the nodes and paths to be copied to the output graph. If a derivation node variable is output, its source and target tuple nodes are also output. At the end of query execution, the output graph unifies all nodes and paths that have been copied through INCLUDE PATH operations.

**RETURN:** In addition to returning a graph, it is essential that we be able to identify specific nodes in this graph. The RETURN clause specifies the set of distinguished variables whose bindings are to be returned together as result tuples.

Using these clauses, we can express ProQL queries for the first four use cases of Section 1.

**Q1.** Given the setting of Figure 1, return the subgraph containing all derivations of tuples in  $O$  from base tuples:

```
FOR [O $x]
INCLUDE PATH [$x] <-+ []
RETURN $x
```

Note the use of the path wildcard (<-+) specifying all paths from all nodes that derive any \$x node.

**Q2.** Return the part of derivations of tuples in  $O$  that involve tuples in relation A.

```
FOR [O $x] <-+ [A $y]
INCLUDE PATH [$x] <-+ [$y]
RETURN $x
```

**Q3.** Find tuples that can be derived through mappings  $m_1$  or  $m_2$  and return all one-step derivations from those tuples.

```
FOR [$x] <$p [], [$y] <- [$x]
WHERE $p = m1 OR $p = m2
INCLUDE PATH [$y] <- [$x]
RETURN $y
```

Note the comparison as to whether \$p is from mappings  $m_1$  or  $m_2$ . Reusing \$x in the second path expression is a syntactic shortcut implying a join between paths matched by the two path expressions.

**Q4.** Select tuples from  $O$  and  $C$  that have common provenance (called “join using provenance” in [13]), and return their derivations:

```
FOR [O $x] <-+ [$z], [C $y] <-+ [$z]
INCLUDE PATH [$x] <-+ [], [$y] <-+ []
RETURN $x, $y
```

Observe that there are two variables in the RETURN clause of the query above. As a result, this query returns pairs of bindings to tuple nodes in the provenance graph that have common provenance.

### 3.2.2 Annotation Computation

We now consider how to take returned subgraphs and use them to compute semiring annotations for sets of tuples — matching the needs of our remaining use cases. For this situation, we add two new clauses to ProQL.

**EVALUATE SEMIRING OF:** This clause is used to specify the semiring for which we want to evaluate the graph returned by the nested graph projection query. Semirings built into our implementation include those presented in Table 1, and we expect that future implementers of ProQL may wish to add additional semirings to match their domain requirements.

**ASSIGNING EACH:** To compute annotations in particular semirings, one needs to assign values from that semiring to *leaf* nodes, i.e., EDB tuple nodes in the original graph or tuple nodes that have no incoming derivations in projected subgraphs; as well as to define appropriate unary functions for the mappings. The ASSIGNING EACH clause can be used to specify such assignments similarly to a *switch* statement in C or Java: first, we define a variable that iterates over the set of leaf nodes from the query’s projected provenance

subgraph, and then we list cases and the value to assign a node, should the case be met.<sup>3</sup> In these conditions one can check membership in a relation or express selections on values of particular attributes of the corresponding tuples. Finally, there is an optional DEFAULT statement, if none of the CASE statements is satisfied. If there is no DEFAULT statement, all leaf nodes not matching any CASE are assigned the identity element for the  $\cdot$  operation of the semiring.

Similarly, a second ASSIGNING EACH clause can be used to define unary mapping functions in each semiring. In this case, one can specify conditions over the name of a mapping as well as the semiring value of its single parameter. The default value for mappings, if no DEFAULT statement is provided, is the identity function. Function definitions are restricted in two key ways [31]: one cannot specify an assignment that returns a non-zero value when the input is 0 and mapping application must commute with (finite and infinite) sums.

Any (or both) of these two kinds of ASSIGNING EACH clauses may be specified in a query, depending on whether a user wants to “customize” their value assignment for leaf nodes and/or mappings or they are satisfied with default values. We illustrate the usage of the ASSIGNING EACH clause(s) in the following queries for use cases Q5-Q10 of Section 1.

**Q5.** Determine derivability of the tuples in  $U$  from base tuples (the default assignment is sufficient in this case).

```
EVALUATE DERIVABILITY OF {
  FOR [O $x]
  INCLUDE PATH [$x] <-+ []
  RETURN $x
}
```

**Q6.** Same as above, but substitute the word “LINEAGE” for “DERIVABILITY”.

**Q7.** Assuming peer  $O$  distrusts any tuple  $O(n, h, a)$  if the data came from  $A(i, n, h)$  and  $h \geq 6$ , trusts any tuple from  $C$  and distrusts  $m_4$  while trusting all other mappings if their input is trusted, determine what set of tuples in  $O$  is trusted:

```
EVALUATE TRUST OF {
  FOR [O $x]
  INCLUDE PATH [$x] <-+ []
  RETURN $x
} ASSIGNING EACH leaf_node $y {
  CASE $y in C : SET true
  CASE $y in A and $y.height >= 6 : SET false
  DEFAULT : SET true
} ASSIGNING EACH mapping $p($z) {
  CASE $p = m4 : SET false
  DEFAULT : SET $z
}
```

**Q8-Q10.** These are similar to Q7, using the “WEIGHT”, “PROBABILITY” and “CONFIDENTIALITY” semirings, respectively, and assigning appropriate base values for each semiring.

## 4. STORING & PROCESSING PROVENANCE

In this section we describe our prototype implementation of the core operations of the language, as presented earlier. Our implementation is built on top of the standalone ORCHESTRA engine [28], that creates a complete replica of all data and provenance in the CDSS at each peer, to accommodate disagreements among peers, and uses each peer’s relational DBMS for provenance storage and querying. To this end, we describe the core aspects of our provenance encoding in relations, and our query execution strategy that exploits a relational DBMS engine. The next section discusses how we enhance this basic engine with indexing techniques.

### 4.1 Provenance Storage in Relations

Extending the ORCHESTRA implementation of [28], we store provenance in a set of relations in an RDBMS. Intuitively, we would

<sup>3</sup>if multiple CASE statements match, the first one is followed

$$\begin{array}{ll}
 P_2(i, n, true) :- A(i, n, l) & P_3(i, n, false) :- C(i, n) \\
 P_4(n, h, true) :- A(i, n, h) & \\
 \\
 P_1(i, n) & P_5(i, n) \\
 \begin{array}{|c|c|} \hline 1 & cn_1 \\ \hline 2 & cn_2 \\ \hline \end{array} & \begin{array}{|c|c|} \hline 1 & cn_1 \\ \hline 2 & cn_2 \\ \hline \end{array}
 \end{array}$$

**Figure 2: Relations corresponding to Figure 1, assuming the key of  $A$  is  $id$ , that of  $C$  and  $N$  is the pair  $(id, name)$  and the key of  $O$  is  $name$ . Provenance relations  $P_2, P_3, P_4$  are superfluous because the mappings are projections over  $A$  and  $C$ , hence they are replaced with views.**

like a scheme resembling the *edge relation* encoding of a tree or graph (i.e., a relation in which tuples contain source and destination attributes). Of course, mappings in our setting are not strictly binary relationships — we can map from  $m$  source tuples to  $n$  target tuples. We observe that each relation connected by provenance can be identified by its key. Hence we encode a single mapping derivation in a relation containing the keys of all  $m$  source and  $n$  target relations. For compactness, we only store one copy of any set of attributes that are constrained by the mapping to be the same (e.g., attributes joined on equality, or copied from input to output relations). Each tuple in the provenance relation exactly represents a derivation node and its outgoing edges, as in Figure 1. Each tuple node is simply a tuple in one of the database relations. Our scheme differs from that of [28], in that each derivation is represented by a single tuple: in that work, there were certain cases (specifically with Skolem functions) where that was not the case.

**Superfluous Provenance Relations.** If we refer back to Example 2.1, we see that mapping  $m_2$  computes  $N$  by projecting over the attributes of  $A$ , and adding a constant true. Here  $m_2$ ’s provenance relation would contain the key attributes of  $A$ , and we can add the remaining (constant) attribute for  $N$  simply by knowing the definition of  $m_2$ . Hence we consider this provenance relation to be superfluous: rather than materializing a table for  $m_2$ , we define it as a virtual view over  $A$ .

**Combining Provenance Relations.** Unfortunately, a general problem that arises using a relational encoding is that there are many potential path traversals through different combinations of provenance relations; and the result is a large number of queries (for alternate paths) with multiple joins (representing multiple nodes on a path). A natural question is how best to *combine* provenance relations to improve performance. In [28], we established that it was more effective to take all source and target tuples’ key attributes for a single schema mapping and store these in their own provenance table — as opposed to storing data from multiple derivations with the same target relation in a combined table that used disjoint union. Hence we build upon this idea, and we show in Section 5 how we can index combinations of such provenance tables to optimize path traversals.

**EXAMPLE 4.1.** For our running example of Figure 1, suppose that the key of  $A$  is  $id$ , that of  $C$  and  $N$  is the pair  $(id, name)$  and the key of  $O$  is  $name$ . Figure 2 shows the provenance relations (where  $P_i$  corresponds to mapping  $m_i$ ).

### 4.2 Translating ProQL to SQL

In this section, we describe our strategy for executing ProQL queries that return projections of the provenance graph or compute annotations based on a provenance graph. ProQL queries may include conditions in the WHERE clause specifying a *set of tuples* of interest. For instance, perhaps we have a screenful of tuples from

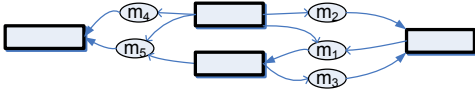


Figure 3: Provenance schema graph for running example

some relation  $R$  for which we wish to compute rankings. Rather than compute a ProQL query over *all* tuples in  $R$ , we would like to perform *goal-directed* computation such that we only evaluate provenance for the selected tuples, as well as only for the paths matching the path expressions in the query. Intuitively, this resembles pushing selections through joins in relational algebra queries.

We assume that provenance graphs are stored in an RDBMS, according to our relational encoding of the previous section. Thus, our approach relies on converting ProQL queries into SQL queries (or, generally, sets of SQL queries) that can ultimately be executed over an underlying RDBMS. More precisely, we break the query answering process into several stages:

- Convert the schema mappings into a *provenance schema graph* (this is common for all queries).
- Match the ProQL query against the provenance schema graph to identify nodes that match path expressions.
- Create a Datalog program based on the set of schema mappings and provenance relations that correspond to the schema graph nodes, as well as the source relations whose EDB data is to be included.
- Execute the program in an SQL DBMS, in a goal-directed fashion, based on tuples and mappings of interest.

We explain each of these stages in more detail below.

#### 4.2.1 Provenance Schema Graph

While paths in the provenance graph exist at the instance (tuple) level, in fact these tuples belong to specific relations that are connected through mappings defined at the schema level. Hence, it makes sense to abstract the set of possible provenance relationships among tuples into a set of potential derivations among relations — in essence to define a schema for the provenance. Intuitively similar to a Dataguide [27] over the provenance, this graph is useful as a basis for matching patterns and ultimately defining queries.

We term this graph among relations and mappings a *provenance schema graph*, constructed as follows. First, we create one node for each relation (a *relation node*, labeled with the name of the relation) and one *mapping node* for each mapping (labeled with the mapping name). Then, we add directed edges from the mapping node to a relation node if the mapping has a target atom matching the relation node’s label. Finally, we add directed edges from a relation node to the mapping node if the mapping has a source atom matching the relation node’s label. The result looks like Figure 3, where we show relation nodes with rectangles and mapping nodes with ellipses.

#### 4.2.2 Matching ProQL Patterns

The next step is to determine which subgraphs of the provenance schema graph match the ProQL patterns. We start with the distinguished reference nodes of the ProQL query: these nodes can range over all relations or may be restricted to a single relation, if specified by the query. For each path expression in the FOR clause, our algorithm traverses the schema graph from each node that can match the “originating” node of the path, using a nondeterministic-state-machine-based scheme to find paths that match the pattern. (We prevent paths from cycling back upon themselves.) The ultimate result is a set of mapping nodes and relation nodes.

#### 4.2.3 Creating a Datalog Program

As an intermediate step towards creating the ultimate SQL queries to return answers, we first create a Datalog program based on the set of mapping and relation nodes returned by the pattern-match.<sup>4</sup> This process is fairly straightforward. For each mapping node returned from the matching step, we add the corresponding mapping to the program. For every relation node matched in the schema graph, we also add rules to test if we have reached a local contribution relation (containing leaf nodes of the provenance graph).

EXAMPLE 4.2. *For our running example, suppose we want to evaluate a query returning all derivations of tuples in  $O$  from tuples in  $A$  and  $N$ . From the provenance schema graph of Figure 3 the matching step will return  $m_4$ , which defines  $O$  in terms of  $A$ , as well as  $m_1$ , which derives tuples in  $C$  from  $A$  and  $N$ , which can then be combined with  $A$  through  $m_5$ , to derive tuples in  $O$ . Then, the Datalog program contains rules for these mappings involving the corresponding provenance relations; e.g., for  $m_5$  this rule is:*  
 $O(n, h, true) :- P_5(i, n), A(i, \_, h), C(i, n)$   
*Moreover, the Datalog program contains rules  $L_1, L_3, L_4$  from Example 2.1, in order to test whether tuples of interest may be derived from the local contributions of one of the matched relations.*

In order to represent the returned graph of a ProQL query we create a set of output tables — one for each relevant provenance relation — and populate them with the edges in the output subgraph. Queries also return a relational result containing the tuple keys (possibly paired with an annotation from some semiring) for the bindings in the RETURN clause.

#### 4.2.4 Executing the Program

We now consider how to execute the Datalog version of our ProQL query over a provenance graph stored in an RDBMS. Recall the contents of the provenance relations for our running example, as shown in Figure 2. In order to reconstruct partial or complete derivations of a tuple — as described in path expressions in the graph projection part of ProQL queries — we need to combine tuples from multiple provenance relations. Moreover, to execute ProQL queries with an annotation computation component, we need to identify complete derivations from leaf nodes, for which an assignment of semiring values is given in the query.

For acyclic provenance graphs, each tuple can only have a finite number of *distinct derivation tree shapes*. For each of those derivation shapes, we can compute a conjunctive rule that reconstructs them from the one-step derivations stored in the provenance relations, by *recursively unfolding* the rules of the Datalog program of Section 4.2.3. The result is a union of conjunctive rules over provenance relations and base data “reachable” from them.

EXAMPLE 4.3. *Continuing our running example, in the body of the rule shown in Example 4.2, tuples in  $A$  can only be derived locally (from  $A_1$ ) while tuples in  $C$  can be derived either from  $C_1$  or through  $m_1$  ( $m_3$  does not match since the query only asked for derivations from tuples in  $A$  and  $N$ ). Then, one (breadth-first) unfolding step yields the rules:*

$O(n, h, true) :- P_5(i, n), A_1(i, \_, h), C_1(i, n)$   
 $O(n, h, true) :- P_5(i, n), A_1(i, \_, h), P_1(i, n), A(i, s, \_), N(i, n, false)$   
*We repeat this process (using only rules matching the ProQL pattern) until all body atoms in all rules are either provenance relation atoms or local contribution relation atoms.*

During this unfolding we can create a semiring expression corresponding to this derivation tree shape. This expression can then

<sup>4</sup>This program can be recursive for cyclic provenance graphs. However, in this paper we focused on ProQL evaluation over acyclic provenance graphs, for which this program is not recursive.

be used to compute annotations, by “plugging in” annotations for leaf nodes and combining them with the appropriate semiring multiplication operation at intermediate tree nodes.

Of course, each conjunctive rule only computes a subset of the tuples and their provenance — specifically the tuples and provenance values for one potential derivation tree. We convert each conjunctive rule into SQL (adding an additional attribute for the provenance expression evaluation). Then, we take the resulting SQL SELECT.FROM.WHERE blocks and combine their output using SQL UNION ALL. Finally, we evaluate an aggregation query over the combined output, in which we GROUP BY the values of the tuples, then combine the provenance attributes using an aggregation function, and finally threshold the results with a HAVING expression. Referring to Table 1, for the first two semirings (derivability and trust), we can SUM the annotations (assuming we represent **true** as 1 and **false** as 0), then add a HAVING clause testing for a non-zero annotations. The next two expressions can be evaluated using MIN; and the number of derivations can be SUMmed.

These components form a baseline implementation of ProQL, providing all the required functionality. However, more can be done to improve its performance. In the next section we introduce indexing techniques that can be used to speed up processing of provenance queries.

## 5. INDEXING DATA PROVENANCE

The main challenge in answering ProQL queries lies in navigating through graph-structured data, according to unrooted path expressions. As we explained in Section 4.2, such path traversals are translated into joins among provenance relations, each representing a one-step derivation. Such paths in provenance graphs can often be long, and their translation produces unfolded rules containing multi-way joins, whose execution can be expensive. Moreover, different unfolded rules may contain overlapping paths, meaning that multiple rules may contain common join subexpressions.

A natural question to ask is whether one could optimize ProQL queries by precomputing the shared joins, i.e., *indexing* paths in a provenance graph. Then, queries involving those paths can start at one node and find sets of nodes reachable within a certain number of hops directly from this index, without needing to join individual provenance relations. Ideally, such an index structure could be retrofitted into a relational DBMS engine, so that our SQL-based strategy could benefit from it.

Among a variety of path indices that have been studied in the literature [17, 27, 35, 37], the most natural indexing technique to adapt for our provenance query scheme is the *access support relation* [35] (ASR) originally developed for object-oriented databases. An ASR is an  $n$ -ary relation among sets of objects connected through paths that can be used to speed up queries involving path expressions in object-oriented query languages. Unlike the other types of path indices, ASRs can be emulated using conventional relational tables, which reference the base tables on (B-Tree) indexed attributes. This provides very similar performance to having built-in support for ASR structures, while having the virtue that it will run on any off-the-shelf RDBMS.

In the case of object-oriented databases, each object has a unique object identifier (OID) and the ASR is an auxiliary structure known to the DBMS, consisting of tuples with references to objects by their OIDs. Clearly, in our case we neither have objects nor OIDs. Moreover, our patterns have some subtle differences from paths in the object-oriented sense. However, one can take most of the basic principles of the ASR and extend them to match our setting.

In particular, we can define ASRs for paths in provenance graphs by creating *materialized views* for joins among provenance rela-

tions that correspond to paths of mappings along some derivations. These views can also be stored as relations in the RDBMS, together with the provenance relations. Then, rewriting unfolded rules to take advantage of such ASRs amounts to a case of answering queries using materialized views [30]. Moreover, we can define relational indices on key columns of the ASRs to provide efficient lookup of specific rows (corresponding to paths in particular derivations) as well as to optimize queries that involve longer paths (and, therefore, need to join multiple ASRs).

In the rest of this section we explore different options regarding how to adapt ASRs so that they can be combined with our relational storage of provenance to speed up processing of ProQL queries. These options also determine the appropriate schema for the relational storage of the resulting ASRs.

### 5.1 ASR Design Choices

To index paths in a provenance graph, we need to materialize the results of joins among provenance relations: each relation represents an edge traversal, and an index represents a traversal of multiple edges. However, as we index a path within an ASR, we have several choices about whether to also index some or all of its subpaths. In this section, we discuss these options and their likely advantages and disadvantages. Later we discuss their implementation and experimentally compare them.

The choice of whether to materialize only the complete path or (some or all of) its subpaths impacts how we join the provenance relations in forming the ASR. In particular, for a two-step ASR, an inner join among provenance relations represents a complete path, a left outerjoin results in a path and its prefixes (padded by NULLs in the resulting ASR), a right outerjoin represents a path and its suffixes, and a full outerjoin represents a path and all its subpaths.

To include paths and subpaths within a longer (e.g., 3-step) ASR, we may need to union together the results of multiple queries. Suppose we have a path through provenance tables  $P_3 \leftarrow P_2 \leftarrow P_1$ . Naively outerjoining multiple steps, e.g., some set of linked provenance tables  $P_3 \bowtie P_2 \bowtie P_1$ , might result in ASR tuples containing entries from  $P_3$  and  $P_1$ , with NULLs in place of  $P_2$  (since there might not exist an edge connecting these steps). Instead, we can index all subpaths in this case by unioning a pair of joins:

$$P_{(3,2,1)} = P_3 \bowtie P_2 \bowtie P_1 \cup P_3 \bowtie P_2 \bowtie P_1$$

In the rest of this paper, we use the terms *subpath ASR*, *prefix ASR* and *suffix ASR* to refer to ASRs based on these operations, which index a path as well as all its subpaths, prefixes or suffixes, respectively, and *complete path ASR* for the ASR that only contains the inner join of all mappings. We note that inner joins can be expressed as Datalog rules and thus can easily be maintained incrementally, together with regular provenance relations [28]. Incremental maintenance of outerjoins is more complicated and we intend to explore it in future work.

### 5.2 Using ASRs in ProQL Query Evaluation

In order to take advantage of ASRs, we need to rewrite the rules in the Datalog program of Section 4.2.4 — replacing provenance relation atoms with ASRs that contain those provenance relations. In essence, this is a matter of substituting materialized views, which we cannot always depend on from an underlying RDBMS.

One factor that can significantly complicate this rewriting process is the existence of overlapping ASR definitions, i.e., when different ASRs may index overlapping (sub)paths. Here, in order to produce a minimal rewriting (i.e., one with the smallest possible number of atoms) we would need to follow an expensive dynamic programming approach, considering the ASRs in all possible orders. We note that this rewriting needs to be performed at execution

### Algorithm unfoldASRs

**Input:** set of rules  $R$ , set of ASRs  $A$

**Output:** set of rules with ASRs  $S$

```
1.  $S \leftarrow \emptyset$ 
2. for every rule  $r$  in  $R$ 
3.   do repeat
4.      $didSomething \leftarrow false$ 
5.     for every ASR  $a$  in  $A$ 
6.       do  $foundUnfolding \leftarrow false$ 
7.          $P \leftarrow$  paths indexed by  $a$  listed in inverse order of
           length
8.         for every path  $p$  in  $P$ 
9.           do if ( $!foundUnfolding$ )
10.            then  $foundUnfolding \leftarrow \text{unfoldPath}(r,p)$ 
11.              if ( $foundUnfolding$ )
12.                then  $didSomething \leftarrow true$ 
13.         until ( $!didSomething$ )
14.        $S \leftarrow S \cup \{r\}$ 
15. return  $S$ 
```

### Algorithm unfoldPath

**Input:** rule  $r$  (modified if unfolding is found), rule  $p$  (representing a path in an ASR)

**Output:** true if unfolding was found, false otherwise

```
1.  $h \leftarrow \text{findHomomorphism}(r,p)$ 
2. if ( $h \neq \emptyset$ )
3.   then for each variable  $x$  in the head and body of  $p$ 
4.     do replace  $x$  with  $h(x)$ 
5.   for each atom  $a$  in the body of  $p$ 
6.     do remove  $a$  from the body of  $r$ 
7.   add the head of  $p$  to the body of  $r$ 
8.   return true
9. else
10.  return false
```

### Algorithm findHomomorphism

**Input:** rule  $r$ , rule  $p$

**Output:** a homomorphism from  $r$  to  $p$  (i.e., set of mappings from variables in  $p$  to variables and constants in  $r$ ) or  $\emptyset$ , if no homomorphism exists elided for brevity

Figure 4: ASR Rewriting Algorithm

time for each ProQL query, so being able to perform it efficiently is crucial for overall query performance.

For this reason, we chose to allow only non-overlapping ASR definitions, for which a minimal unfolding can always be produced by the greedy algorithm **unfoldASRs** of Figure 4. This algorithm considers each path contained in an ASR in inverse order of length. If there are no overlapping ASRs, this guarantees that the resulting unfolding is minimal, since (shorter) subpaths are only unfolded if it was impossible to unfold any of their (longer) superpaths.

In step 10, **unfoldASRs** employs algorithm **unfoldPath**, which first looks for a *homomorphism* from the body of the path  $p$  to that of the rule  $r$ , i.e., a mapping from variables in  $p$  to variables and constants in  $r$  such that each atom in the body of  $p$  is mapped to an atom in the body of  $r$ . If such a homomorphism is found, it replaces those mapped atoms in the body of  $r$  with the image of the head of  $p$  (i.e., an ASR atom “selecting” the part of the ASR representing this subpath) under the homomorphism (replacing variables with the values to which they are mapped).

**EXAMPLE 5.1.** In our running example, if we define an ASR  $P_{(5,1)}$  for the path of  $m_1$  followed by  $m_5$ , the unfolding algorithm would replace the  $P_5$  and  $P_1$  atoms in the second rule of Example 4.3 with a  $P_{(5,1)}$  atom, producing the following rule which contains one join fewer than the original one:

$O(n, h, true) :- P_{5,1}(i, n), A_l(i, -, h), A(i, s, -), N(i, n, false)$

## 6. EXPERIMENTAL EVALUATION

Given the lack of established provenance query systems and benchmarks, we developed microbenchmarks for provenance queries. We

investigate the performance of path traversal queries, which are at the core of any provenance query, and the optimization benefits of ASRs for such queries on CDSS settings with different mapping topologies. First, we consider a simple topology, where all peers are connected through mappings that form a *chain*, as shown in the provenance schema graph of Figure 5, in order to focus on specific factors that affect performance of provenance querying and illustrate ASR optimization opportunities. Next, we experiment with a more realistic *branched* topology, as shown in Figure 6, and investigate the performance and scalability of provenance query processing, for different numbers of peers and amounts of data at each peer. Finally, we consider grouping mappings along paths in ASRs and analyze the performance benefits of ASRs of different types and lengths.

### 6.1 Experimental Setup

Our ProQL prototype, including parsing, unfolding and translation to SQL queries was implemented as a Java layer running atop a relational DBMS engine. We used Java 6 (JDK 1.6.0\_07) and Windows Server 2008 on a Xeon ES5440-based server with 8GB RAM. Our underlying DBMS was DB2 UDB 9.5 with 8GB of RAM.

#### 6.1.1 Settings and Terminology

Due to the lack of real-world data sharing settings that are sufficiently large and complex to test our system at scale, we created synthetic workloads based on bioinformatics schemas and data from the SWISS-PROT protein database [3]. We generate peer schemas and mappings by partitioning the 25 attributes in the SWISS-PROT universal relation into two relations and adding a shared key to preserve losslessness. Then, each mapping has a join between two such relations in the body and another join between two relations in the head.

In typical bioinformatics CDSS settings, one would expect most of the data to be contributed by a small subset of authoritative peers; thus, in most of our experiments we consider settings with relatively few peers with local data, while the remaining peers import data along incoming mappings, edit them according to their trust policies, and propagate them further along outgoing mappings. In our first experiment we also explore the scalability of provenance querying in a setting with local data at all peers, as a stress test.

Both of the topologies we experimented on have a *target* peer, which is the one that all mappings are propagating data to, directly or indirectly. This does not imply that we expect real-world settings to form rooted trees. In fact, these topologies should not be interpreted as a complete CDSS setting, but rather as a projection of the complete mapping graph that only contains peers from which our target peer of interest is reachable. However, this projection allows us to focus on the extreme case, where all peers and mappings propagate data to this particular target peer. Typically, in a CDSS, there will be many peers and mappings that do not propagate data to this peer (e.g., other peers that import data from common authoritative sources) but those mapping paths do not affect the evaluation or the result of the provenance queries whose performance we measure.

We generate local data for each peer by sampling from the SWISS-PROT database and generating a new key by which the partitions may be rejoined. For these experiments, we substituted integer hash values for each large string in the SWISS-PROT database, modeling the amount overhead taken by CLOBs in a real bioinformatics database. We refer to the *base size* of a workload to mean the number of SWISS-PROT entries inserted locally at each peer and propagated to the other peers before provenance queries were executed.

#### 6.1.2 Provenance Queries

The main goal of these experiments is to evaluate the performance of the path traversal component of ProQL, with or without

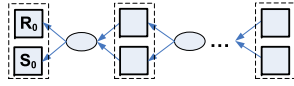


Figure 5: Chain topology

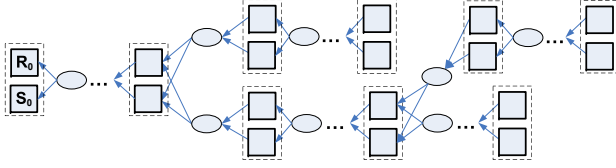


Figure 6: Branched topology

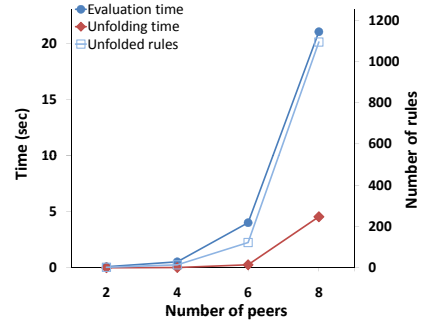


Figure 7: Query processing times and unfolded rules for chain of varying length with data at every peer

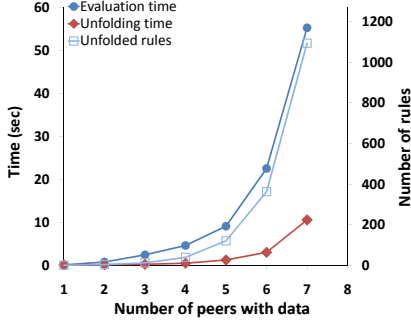


Figure 8: Query processing times and unfolded rules for chain of 20 peers with varying number of peers with data

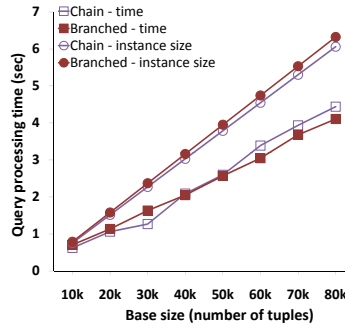


Figure 9: Query processing times and instance size for chain and branch topologies of 20 peers and varying base sizes

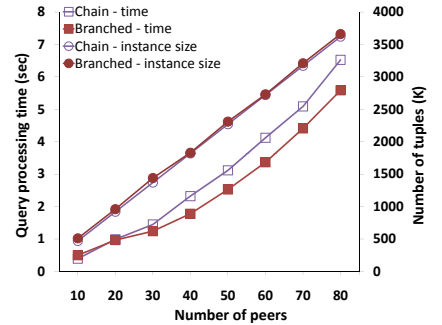


Figure 10: Query processing times and instance size for chain and branch topologies of varying numbers of peers

the use of ASRs. As a result, for our experiments, we used queries of the form (hereby called *target* query):

```
FOR [R0 $x]
INCLUDE PATH [$x] <+> []
RETURN $x
```

where  $R_0$  is a relation at the target peer of the corresponding topology. Such queries traverse all the paths in the mapping graphs up to their end, and thus are ideal in order to evaluate path traversal.

We also experimented with similar queries involving annotation computation similar to **Q7** from Section 3.2. Perhaps surprisingly, we found that the execution time for queries involving such annotation computations was very similar to that of their graph projection component, i.e., the graph projection component dominates execution time. Thus, for simplicity, in the experiments below we focus on graph projection queries without annotation computation.

### 6.1.3 Experimental Methodology

Each experiment was repeated seven times, with the best and worst results discarded, and the remaining five numbers averaged. In all of our experiments, the results were very similar among these five runs, and thus the confidence intervals were too small to be visible on the graphs.

## 6.2 Number of Peers with Local (Base) Tables

In the experiments of this section we use the chain topology of Figure 5. For the first experiment, we perform a “stress-test” by assuming that all peers have local data and investigate the performance of the target query shown above. Figure 7 shows that, in this case, the number of unfolded rules grows exponentially with the number of peers. Intuitively, this is because every tuple at every peer may either be inserted locally or derived from some peer further “downstream” in the graph of mappings, and the unfolding needs to cover all these possible derivations. Moreover, for every join we need to consider all combinations for each side of the join. Thus, as also shown in Figure 7, unfolding time and evaluation time

for the unfolded rules also grow exponentially, remaining efficient (sub-20 sec.) for up to 8 peers.

To isolate the effect of the number peers with *local data* we repeated this experiment for a fixed total number of peers, varying the number of local contribution relations. Figure 8 shows that the number of unfolded rules, as well as unfolding and evaluation times, also grow exponentially with the number of peers supplying local data, for a setting with 20 total peers.

## 6.3 Number of Peers and Base Size

As we explained earlier, in real-world bioinformatics settings it is more likely that only a small number of authoritative sources will contribute local data, that is then propagated along (possibly long) paths of mappings. For this reason, in the next experiments, we consider CDSS settings that have data at a few of the peers near the right-hand side of the topologies of Figures 5 and 6. Figure 9 shows that the size of the instances produced as a result of the propagation of local data at the grows linearly with the base size. Query processing time (i.e., the sum of unfolding and evaluation times) also grows linearly up to a few seconds, even for a base size of 80k tuples per peer relation.

In Figure 10 we show that the size of the instance that results from the propagation of 10k tuples inserted locally at a few (2-3) of the peers grows linearly with the total number of peers through which they are propagated. Query processing time also grows at a roughly linear rate for both topologies, although a bit faster for the branched topology. Moreover, it is within a few seconds, even for topologies of 80 peers. This implies that our implementation could scale to at least a few hundreds of peers, but we were unable to run experiments for settings with more than 80 peers because the resulting SQL queries were too large for DB2 (as each unfolded rule can contain up to  $n$ -way joins, where  $n$  is roughly equal to the number of nodes in a derivation tree of the target peer).

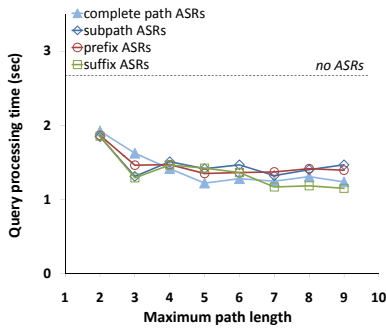


Figure 11: Query processing times for different ASR types and lengths, for chain of 20 peers few of which have local data

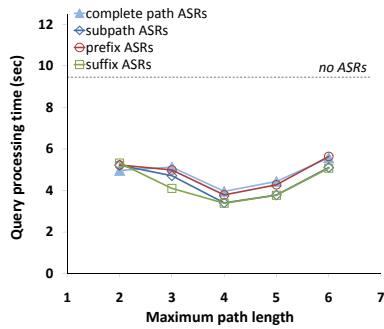


Figure 12: Query processing time for different ASR types and lengths, for chain of 8 peers, half of which have local data

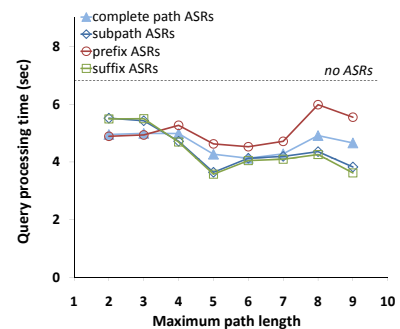


Figure 13: Query processing time for different ASR types and lengths, for branched topology of 20 peers

## 6.4 Different ASR Sizes and Types

Even though we showed that query processing is fairly efficient — perhaps surprisingly so, given the size and complexity of the unfolded rules in some cases — there is a lot of room for optimization, using ASRs to materialize joins that appear in many of these unfolded rules. As we discussed earlier, there are several options about which (sub)paths to store in an ASR. Moreover, there are different options in terms of the *length* of the paths to index in each case. In this section, we investigate — for the topologies presented above — the effect of different options to query processing times.

First, we consider a chain topology of 20 peers, 2 of which have local data, with a base size of 50k for each peer. For each *maximum path length*, we essentially “split” the chain into paths up to this length, and possibly store the remaining mappings in a shorter ASR, if the number of mappings is not a multiple of this path length. Figure 11 shows the total query processing time (i.e., the sum of evaluation and unfolding times) for the target query for different maximum lengths of all ASR types. The dashed line indicates the processing time for this query without using any ASRs. We observe that all ASR types provide a significant performance improvement, which increases with path length. Intuitively, in this topology and for all lengths shown in the graph, the paths covered by ASRs are subsumed completely by the paths traversed by the query. Thus we can take full advantage of the ASRs, even in if they only contain the complete path. Moreover, the use of longer ASRs results in SQL queries with fewer joins that are faster to evaluate.

We also experimented with a topology with fewer peers, more of which have local data. In particular, we used a chain of 8 peers, 4 of which have a base size of 50k. As illustrated in Figure 12, for all ASR types and lengths we again get a significant benefit. In this case there is a larger number of unfolded rules involving combinations of subpaths of the chain. As a result, subpath, prefix and suffix ASRs generally perform better than complete path ASRs. We note that suffix ASRs perform better than prefix ASRs. This is due to the fact that our target query is looking for paths starting from any node but ending in a specific node. For queries, e.g., returning all tuples that can be derived from a particular base tuple, prefix ASRs would provide a larger benefit. Finally, we note that the performance benefit of ASRs increases for greater lengths up to 4 but then decreases for longer paths. This is both due to the increased unfolding cost, for longer subpath/prefix/suffix ASRs, and due to the fact that longer complete path ASRs can be utilized by fewer unfolded rules because the complete paths they index are not subsumed by the paths in the rules.

Finally, we performed the same experiment on a branched topology of 20 peers, 4 of which have a base size of 50k. In this topology, the target query is translated to 40 unfolded rules, each containing

paths along combinations of these branches. As illustrated in Figure 13, this branching raises challenges for some ASR types. In particular, because the unfolded rules contain paths along different branches, complete path and prefix ASRs that cross the boundaries of the branches in the topology can be exploited by fewer of the unfolded rules and thus yield a smaller performance benefit. Still, for up to medium lengths (6 steps), complete path ASRs provide a significant benefit, because of the existence of short subpaths in the topology with no branches and no local data. On the other hand, subpath and suffix ASRs provide an even larger benefit for greater lengths, since the different subpaths they contain can be used for path traversals along different branches.

## 6.5 Overall Conclusions

Our final conclusion from these experiments is that ProQL query processing can be performed within the requirements of target CDSS applications, i.e., with execution times under a minute for various mapping topologies of tens of peers. In particular, query processing times are in the scale of seconds or tens of seconds, even for settings with tens of peers, and the main obstacle for scaling to larger settings comes from limitations of the underlying DBMS, regarding the size and complexity of the generated SQL queries. Moreover, ASRs yield significant performance benefits for path traversal queries, and the speedup is often higher for longer ASRs, especially in the case of subpath and suffix ASRs.

## 7. RELATED WORK

This paper focuses on querying data provenance, as captured in systems like [6, 10, 26, 28]. However, some of our provenance querying use cases have been influenced by work on workflow provenance querying [8, 13, 38] and business process querying [5]. Despite the shared motivations and use cases with workflow provenance querying, there is a fundamental difference with our work: our underlying model of data provenance deals with declarative queries involving operations such as union and join, while workflow provenance models, such as [38], typically describe procedural workflows and involve operations that are treated as black boxes. Identifying workflows whose runs can be described declaratively or designing a more user-friendly *visual* layer over ProQL — as the one presented in [5] for BPEL processes — are interesting directions for future work.

The design of ProQL has been influenced by graph query languages, such as GraphLog [16], Lorel [1], StruQL [22] and RQL [32]. However, as we explained in Section 3.1, there are fundamental differences between provenance graphs and the underlying graph models of these languages, as well as between the semantics of provenance querying and those of graph query languages. Some simple query languages have been proposed for models of data an-

notations that are (explicitly or implicitly) computed through user queries [7, 15, 25]. Finally, [11] studied the expressive power of languages that manipulate annotations explicitly, and compared them with the implicit where-provenance associated with a query or update. However, none of these models can facilitate a variety of annotation computations — supported by how-provenance — that are crucial for applications such as those discussed in Sections 1–2.

Our encoding of the provenance graph in relations leverages many ideas from [28] and resembles the approach of [23], where *edge* relations are used to store XML in an RDBMS. In terms of indexing to improve evaluation of path expressions, a wide variety of techniques have been studied in the literature for different data models, ranging from semi-structured data [27, 37] to objects [35] to XML [17, 34]. However, virtually all XML index techniques are based on the notion of a distinguished document root, while our queries can have multiple relation nodes of interest. Moreover, XML index techniques have been designed for tree- or DAG-shaped data, and most do not generalize to graphs with cycles.

## 8. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed ProQL, a query language for provenance graphs which is useful in supporting a wide variety of applications with derived data. In particular, we showed that ProQL queries can be used to assess trust and derivability or detect side effects of possible updates, as required in basic CDSS operations, as well as to express more complicated provenance queries and, optionally, compute data annotations in particular semirings. We developed a prototype implementation of ProQL over an RDBMS, introduced indexing techniques for speeding up ProQL queries that involve path traversals and provided a detailed experimental study of the performance of provenance query processing in a variety of CDSS settings and the benefits of our indexing techniques.

In future work, we intend to deploy ProQL in real-world bioinformatics data sharing applications in order to assess its impact in practice, and possibly identify new use cases to be handled by future ProQL extensions. Moreover, we plan to develop an alternative ProQL implementation scheme that can handle cyclic provenance graphs and run over our distributed ORCHESTRA engine [43]. One possible approach — that may also provide better performance if there are large numbers of possible derivations for each tuple — is to execute the set of rules in bottom-up fashion, materializing the intermediate results. Finally, we would like to explore automated index selection techniques for generating appropriate ASR definitions automatically, for a given mapping topology and workload of ProQL queries over a stored provenance graph. In particular, we plan to investigate whether techniques such as [2], can be applied in our case, directly or with some extensions and combined with cost estimates from the optimizer of the underlying RDBMS.

## Acknowledgments

This work was funded by NSF grants IIS-0447972, 0713267, and 0513778, and CNS-0721541. We thank Peter Buneman, Steve Zdancewic, Susan Davidson, Boon Thau Loo, the members of the Penn Database Group, and the anonymous reviewers for their feedback and suggestions.

## 9. REFERENCES

- [1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *IJDL*, 1(1), April 1997.
- [2] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, 2000.
- [3] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL. *Nucleic Acids Research*, 28, 2000.
- [4] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: Authority-based keyword search in databases. In *VLDB*, 2004.
- [5] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes. In *VLDB*, 2006.
- [6] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.
- [7] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *VLDB*, 2004.
- [8] O. Biton, S. C. Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In *ICDE*, 2008.
- [9] S. C. Boulakia, O. Biton, S. B. Davidson, and C. Froidevaux. BioGuideSRS: querying multiple sources with a user-centric perspective. *Bioinformatics*, 2007.
- [10] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD Conference*, 2006.
- [11] P. Buneman, J. Cheney, and S. Vansummeren. On the expressiveness of implicit provenance in query and update languages. In *ICDT*, 2007.
- [12] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.
- [13] A. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *SIGMOD*, 2008.
- [14] L. Chiticariu and W.-C. Tan. Debugging schema mappings with routes. In *VLDB*, 2006.
- [15] L. Chiticariu, W. C. Tan, and G. Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In *SIGMOD*, 2005.
- [16] M. P. Consens and A. O. Mendelzon. Expressing structural hypertext queries in GraphLog. In *HYPERTEXT*, 1989.
- [17] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *VLDB*, 2001.
- [18] Y. Cui. *Lineage Tracing in Data Warehouses*. PhD thesis, Stanford Univ., 2001.
- [19] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.
- [20] X. L. Dong, L. Berti-Equille, and D. Srivastava. Integrating conflicting data: The role of source dependence. *PVLDB*, 2(1), 2009.
- [21] R. Fagin, P. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. *Theoretical Computer Science*, 336:89–124, 2005.
- [22] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a web-site management system. In *SIGMOD*, 1998.
- [23] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), Sept. 1999.
- [24] J. N. Foster, T. J. Green, and V. Tannen. Annotated XML: queries and provenance. In *PODS*, 2008.
- [25] F. Geerts, A. Kementsietsidis, and D. Milano. Mondrian: Annotating and querying databases through colors and blocks. In *ICDE*, 2006.
- [26] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, 2009.
- [27] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.
- [28] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007. Amended version available as Univ. of Pennsylvania report MS-CIS-07-26.
- [29] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.
- [30] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.
- [31] G. Karvounarakis. *Provenance for Collaborative Data Sharing*. PhD thesis, University of Pennsylvania, July 2009.
- [32] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: a declarative query language for RDF. In *WWW*, 2002.
- [33] G. Karvounarakis and Z. G. Ives. Bidirectional mappings for data and update exchange. In *WebDB*, 2008.
- [34] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, 2002.
- [35] A. Kemper and G. Moerkotte. Access support in object bases. In *SIGMOD*, 1990.
- [36] L. Kot and C. Koch. Cooperative update exchange in the Youtopia system. In *Proc. VLDB*, 2009.
- [37] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.
- [38] Open provenance model. <http://wiki.ipaw.info/bin/view/Challenge/OPM>, 2008.
- [39] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating web data. In *VLDB*, 2002.
- [40] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *S&P*, May 2008.
- [41] P. P. Talukdar, M. Jacob, M. S. Mehmood, K. Crammer, Z. G. Ives, F. Pereira, and S. Guha. Learning to create data-integrating queries. In *VLDB*, 2008.
- [42] N. E. Taylor and Z. G. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD*, 2006.
- [43] N. E. Taylor and Z. G. Ives. Reliable storage and querying for collaborative data sharing systems. In *ICDE*, 2010.