PROVENANCE IN COLLABORATIVE DATA SHARING

GRIGORIOS KARVOUNARAKIS

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2009

Zachary G. Ives
Supervisor of Dissertation

Val Tannen
Supervisor of Dissertation

Jianbo Shi
Graduate Group Chairperson

To my parents Michalis and Argetta,
my brothers Stamatis and Giorgos
and my sister Krystalli

ABSTRACT

Provenance in Collaborative Data Sharing

Grigorios Karvounarakis

Supervisors: Zachary G. Ives and Val Tannen

This dissertation focuses on recording, maintaining and exploiting *provenance* information in Collaborative Data Sharing Systems (CDSS). These are systems that support data sharing across loosely-coupled, heterogeneous collections of relational databases related by declarative schema mappings. A fundamental challenge in a CDSS is to support the capability of *update exchange* — which publishes a participant's updates and then translates others' updates to the participant's local schema and imports them — while tolerating disagreement between them and recording the provenance of exchanged data, i.e., information about the sources and mappings involved in their propagation. This provenance information can be useful *during* update exchange, e.g., to evaluate *provenance-based trust policies*. It can also be exploited *after* update exchange, to answer a variety of user queries, about the *quality, uncertainty or authority* of the data, for applications such as trust assessment, ranking for keyword search over databases, or query answering in probabilistic databases.

To address these challenges, in this dissertation we develop a novel model of *provenance graphs* that is informative enough to satisfy the needs of CDSS users and captures the semantics of query answering on various forms of annotated relations. We extend techniques from data integration, data exchange, incremental view maintenance and view update to define the *formal semantics* of *unidirectional* and *bidirectional* update exchange. We develop *algorithms* to perform update exchange incrementally while maintaining provenance information. We present strategies for implementing our techniques over an RDBMS and experimentally demonstrate their viability in the ORCHESTRA prototype system. We define ProQL, a *query language for provenance graphs* that can be used by CDSS users to combine data querying with provenance testing as well as to compute annotations for their data, based on their provenance, that are useful for a variety of applications. Finally, we develop a prototype implementation ProQL over an RDBMS and *indexing techniques* to speed up provenance querying, evaluate experimentally the performance of provenance querying and the benefits of our indexing techniques.

Acknowledgements

I am indebted to my advisors, Zack Ives and Val Tannen. Zack provided invaluable guidance for my research, that made this thesis possible, and was always available to encourage me and suggest possible avenues to explore, every time I got stuck with some problem along the way. Val introduced me to database theory and taught me that the way to achieve the most profound understanding of a problem — and often to find the best solution to it — involves investigating the underlying mathematical foundations. Both also made direct contributions to my work and helped me learn how to communicate my ideas more effectively. Finally, I would like to thank both of them for their support and understanding with the health problems I had to deal with during my PhD studies.

I am also grateful to the members of my dissertation committee, Peter Buneman, Boon Thau Loo, Susan Davidson, Steve Zdancewic, for their insightful comments, suggestions and criticisms on this dissertation.

I would also like to thank my friends, collaborators and coauthors TJ Green, Nick Taylor, Olivier Biton and Nate Foster for our fruitful discussions as well as their contributions in our joint work. My dissertation work has benefited greatly from comments and discussions during the weekly database group meeting, with these and other members of the database group, including (during my years at Penn): Sarah Cohen-Boulakia, Svilen Mihaylov, Sam Donnelly, Anat Eyal, Yi Chen, Yifeng Zheng, Sören Auer, Shirley Cohen, Mengmeng Liu, Zhuowei Bao. I would also like to thank the DB group as a whole, for giving me the opportunity to attend numerous interesting talks by esteemed visiting researchers or group members, and I was honored with the responsibility of organizing the group meetings for several years.

I have also received great help from the administrative staff of the CIS department. I would especially like to thank Mike Felker, who was always available and willing to help with any administrative issues I came across. I would also like to thank Cheryl Hickey, Jennifer Finley, Kamila Dyjas Mauro and Charity Payne for helping with administrative issues regarding the organization of DB group meetings. Finally, the staff of the business office, including Gail Shannon and Lillian Thomas, also helped make my life at Penn easier.

Last but not least, I am grateful to my friends that have made my life in Philadelphia over the last six years more pleasant: Rebecca Jackson, Dimitris Vytiniotis, Thanassis Geromichalos, Dina Zhabinskaya, Kostas Danas, Eva Zacharaki, Ann-Katherine Carton, Deborah Plummer, Stathis Avdis, Katerina Fragkiadaki, Elpida Tsika, Dimitra Sassaroli, Sotiris Ioannidis, Michalis Agoras, Dimitris Theodorakatos, Vaios Kalpias, Yannis Theoharis, Christos Tsarouchis. Finally, I would like to express my gratitude to my family, to whom this dissertation is dedicated, for the sacrifices they made for my education and their continued love, support and encouragement.

This research has been funded in part by NSF grants IIS-0477972, 0513778, and 0415810, and DARPA grant HR0011-06-1-0016.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The World Wide Web has created opportunities for collaboration among organizations (in the sciences and academia, in businesses etc.), by allowing them to publish and access one another's data. However, practical data sharing has remained an elusive goal for several reasons. An important one is *schema heterogeneity*, i.e., the fact that data from different organizations even in the same "domain" is often structured using different schemas. Data integration [78] (and its variants, e.g., data exchange [45] and warehousing) identified this problem and proposed a "top-down" approach: collaborating organizations need to first agree on a standardized global schema and then map their individual sources to it using *schema mappings*. This approach has been appropriate for cases when the problem domain is well understood and relatively simple and static, e.g., company mergers, integrated E-business websites, etc.

However, for larger-scale communities, coordination and agreement on a common global schema can be difficult and time-consuming. First, the creation and maintenance of this schema requires central administration,[1] which is hard to establish in communities of independent organizations (e.g., research institutes).

[1] Peer data management [1, 12, 64, 72, 83] supports multiple mediated schemas, thus relaxing some aspects of administration, but it assumes data is consistent.

Moreover, it requires collaboration and agreement by all members, which may be difficult to achieve at a global scale, especially for large communities. Even when this agreement is achieved, it essentially "fixes" the participants to those that were involved in the creation of the global schema. Thus, it may be difficult for smaller organizations left out of the standardization process or ones that started publishing after the creation of the "standard" schema to contribute their data. Finally, in communities where source schemas change frequently, the global schema needs to be revised continuously. This requires additional effort from all participating organizations to adapt to such changes and it results in inconsistencies between different repositories, different versions, etc. For instance, in the field of bioinformatics today, after years of effort in data integration, there are multiple schemas that are considered "standard" and being used by different member organizations, due to different research needs, competing groups, and the continued emergence of new kinds of data. The prospect of investing time and effort with questionable effectiveness often hampers collaboration efforts, by discouraging member organizations from taking part in them.

Even more problematic is the fact the data and associations in different participants' databases ("peers") are often contradictory, because e.g., of disagreements between researchers or the use of different scientific methods. In such cases, individual researchers may choose to resolve inconsistencies by selecting data that have originated from sources they consider more "authoritative" or trusted. Each peer may have different perceptions of trust and authority. Such situations cannot be handled by data integration tools since they impose a global consensus on the data, by translating user queries to queries over the actual sources, and they assume that the data is stable, clean, and correct. However, in many large-scale data sharing efforts, particularly in the sciences, data is neither stable nor clean. It is being continuously annotated, corrected, and hand-cleaned by each user — and a major task is not simply to integrate data for querying, but rather to propagate **up-**

dates across interrelated, independently modified databases. Queries could then be posed directly over updated local database instances at each peer, instead of requiring reformulation over remote sources.

However, surprisingly little work in the data integration literature has addressed propagation of updates across databases and the novel challenges they present. Indeed, in order to be able to assess the *quality, uncertainty or authority* of data they receive, users need to know the sources and mappings involved in their propagation. The importance of such *provenance* [21, 35, 27, 11], information has been recognized in data warehousing, but for data sharing we need to capture more details about how source tuples were combined through mappings between heterogeneous schemas during this propagation. In this dissertation, we focus on the problem of capturing the provenance of data, as it is exchanged during update propagation, and exploiting this provenance information for various applications.

1.1 Collaborative Data Sharing Systems

To address the needs of scientists, as presented above, this dissertation adopts the general principles of the new data sharing paradigm of Collaborative Data Sharing Systems (abbreviated CDSS) [68], an extremely flexible scheme for sharing data among different participants. Rather than providing a global view of all data the CDSS instead facilitates import and export among autonomous databases. The CDSS model provides a principled architecture that extends the data integration approach to encompass today's scientific data sharing practices and requirements. In this dissertation, we develop the core formal model and techniques to fulfill the vision of [68]. In particular, we define the formal foundations of the basic capability of *update exchange*, which publishes a participant's updates to "the world" at large, and then translates others' updates to the participant's local schema. As updates are exchanged between peers we capture their *provenance* [21, 35, 27, 11], i.e., information about what sources they originated from and what mappings were involved in their propagation. Provenance can be used *during* update exchange, to enable CDSS operations such as filtering which translated updates to apply according to the local administrator's unique *trust policies* about different peers and mappings. Moreover, it can be exploited by CDSS users *after* update exchange has been performed for a variety of applications, as we explain in Section 1.2.

Data sharing in a CDSS occurs among *loosely* coupled confederations of participants (peers). Each participant controls a local database instance, encompassing all data it wishes to manipulate, including data imported from other participants. Participants can then update both local and imported data, and their edits are logged. Declarative *schema mappings* specify one database's relationships to other participants, much as in peer data management systems [64].

Example 1. *Figure 1.1 illustrates an example bioinformatics collaborative data sharing system, based on a real application and databases of interest to affiliates of the Penn Center for Bioinformatics. GUS, the Genomics Unified Schema, contains gene expression, protein, and taxon (organism) information; BioSQL, affiliated with the BioPerl project, contains very similar concepts; and a third schema, uBio, establishes synonyms and canonical names for taxa. Instances of these databases contain taxon information that is autonomously maintained but of mutual interest to the others. Suppose that a peer with BioSQL's schema,* $\boldsymbol{P}_{BioSQL}$*, wants to import data from another peer,* $\boldsymbol{P}_{GUS}$*, as shown by the arc labeled* m_1*, but the converse is not true. Similarly, peer* $\boldsymbol{P}_{uBio}$ *wants to import data from* $\boldsymbol{P}_{GUS}$*, along arc* m_2*. Additionally,* $\boldsymbol{P}_{BioSQL}$ *and* $\boldsymbol{P}_{uBio}$ *agree to mutually share some of their data: e.g.,* $\boldsymbol{P}_{uBio}$ *imports taxon synonyms from* $\boldsymbol{P}_{BioSQL}$ *(via* m_3*) and* $\boldsymbol{P}_{BioSQL}$ *uses transitivity to infer new entries in its database, via mapping* m_4*. Finally, each participant may have a certain* trust policy *about what data it wishes to incorporate, according to their provenance: e.g.,* $\boldsymbol{P}_{BioSQL}$ *may only trust data from* $\boldsymbol{P}_{uBio}$ *if it was derived from* $\boldsymbol{P}_{GUS}$ *entries. The CDSS facilitates dataflow among these systems, using mappings and policies developed by the independent participants' administrators.*

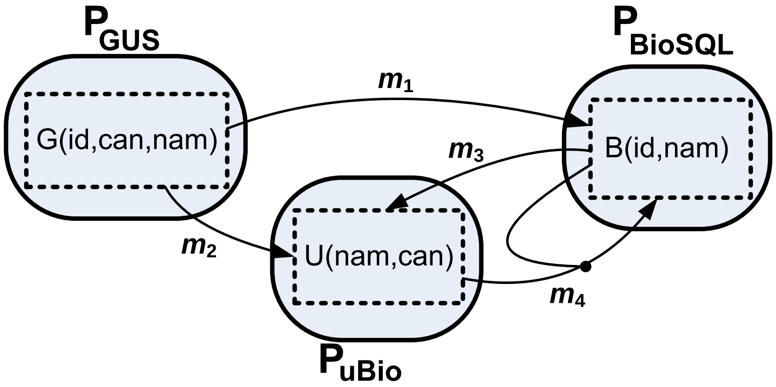


Figure 1.1: Example collaborative data sharing system for three bioinformatics sources. For simplicity, we assume one relation at each participant ($\mathbf{P}_{GUS}, \mathbf{P}_{BioSQL}, \mathbf{P}_{uBio}$). Schema mappings are indicated by labeled arcs.

Each peer operates in "offline" mode for as long as it likes, but it occasionally performs an update exchange operation, which propagates updates to make its database consistent with the others according to the schema mappings and local trust policies. The update exchange process involves *publishing* any updates made locally by the participant, then *importing* new updates from other participants (after mapping them across schemas and filtering them according to trust conditions). Intuitively, this operating mode resembles deferred view maintenance across a set of views — but there are a number of important differences, in part entailed by the fact that instances are related by (paths of) schema mappings rather than views, the fact that we want to record the provenance of exchanged data and the peers' ability to specify local edits and trust policies.

The CDSS records the steps involved in the propagation of the updates, including source tuples and mappings that were involved. This constitutes their provenance or lineage [21, 35, 27, 11], and is useful during update exchange, e.g., in order to assess trust, as well as after update exchange, for a variety of applications akin to provenance querying, that are useful to CDSS users as we explain in Section 1.2. After the update exchange, the participant has an up-to-date data instance, incorporating trusted changes made by participants transitively reachable via schema mappings, and they can query and update that instance and its provenance in an "offline" fashion, until the next time they perform an update exchange operation.

One of the contributions of this thesis is a formal semantics for update exchange and its provenance in a CDSS, that leaves room for disagreement by allowing participants to add to, delete from or modify imported data as well as to filter them according to trust policies. Another contribution involves methods for performing update exchange efficiently, while maintaining the provenance of exchanged data.

1.2 Recording and Exploiting Provenance Information

In the sciences, in intelligence, in business, the same adage holds true: data is only as credible as its source. While data cleaning has been a topic of study for many years in data integration and warehousing, only recently are we beginning to see issues like data quality, uncertainty, and authority make their way into data models, mapping formalisms, and even query languages. In most of these settings, the notion of *data provenance* [21, 35, 58] lies at the heart of dealing with such issues. For update exchange, we record provenance as updates are propagated along schema mappings from one database to another, and use it to assess *trust and authority*; systems like Trio [11] compute provenance or lineage, then use this to derive *probabilities* associated with answers. Recently [94] 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 the 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 [27] that may be imprecise or incorrect: users can see how “bad” data has been produced. (We note that our focus is on *data provenance* in the CDSS context, rather than workflow provenance, a separate topic [26].)

Surprisingly, however, the study of data provenance as a first-class data artifact itself — worthy of its own data model, query language, and indexing and query processing techniques — has not yet come to the forefront. Several provenance models have been proposed for specific applications [21, 35, 11, 27], but none of them is general enough for all the purposes described above. To this end, one of the contributions of this thesis is a rich model of provenance for data derived through compositions of views or schema mappings. For every derived tuple t, a *provenance graph* describes all derivations that have produced it, including source tuples and *how* they were combined through views or mappings as well as possibly other derived tuples that were used to derive t. We explain how provenance graphs can be maintained in a CDSS together with data during update exchange, by extending the datalog programs that perform the latter with appropriate rules. We also show that provenance graphs can be used to compute different forms of data annotations. These include lineage or why-provenance [21, 35, 11], confidentiality policies [50], scores assigned by authority-based ranking schemes [8] or by machine learning based on user feedback about query answers [94], as well as boolean expressions required for query answering in incomplete [66] or probabilistic[52, 98, 11] databases.

Example 2. *Consider the CDSS from Example 1. The graph of Figure 1.2 represents tuple derivations that have been produced during update exchange. In particular, the graph has two kinds of nodes: tuple nodes, shown as rectangles, and mapping nodes, shown as ellipses. Arcs connect tuple nodes to mapping nodes and mapping nodes to tuple nodes. In addition, we have nodes for the local insertions of each peer. This “source” data is annotated with its own id (unique in the system)* $p_1, p_2, \ldots$ *etc., and is connected by an arc*

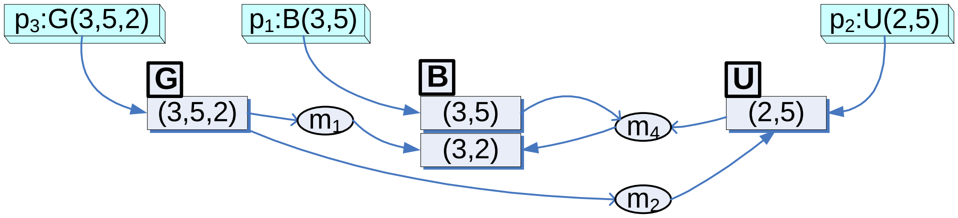


Figure 1.2: A provenance graph for the CDSS of Figure 1.1

to the corresponding tuple entered in the local instance.

From this graph we can analyze the provenance of, say, $B(3, 2)$ by tracing back paths to source data nodes — in this case through (m_4) to p_1 and p_2 and through (m_1) to p_3. As a result, we can determine that there are two alternative "justifications" for $B(3, 2)$. Moreover, if at some time after update exchange we decide that p_1 should not be trusted, e.g., because some error has been discovered in the scientific procedure that produced it, we can determine that $B(3, 2)$ is still valid, since it is still justified by p_3 through (m_1). On the other hand, if p_1 and p_3 are discredited, we can conclude from the provenance graph that $B(3, 2)$ cannot be trusted anymore.

Provenance graphs can often be very large and complex, due to the volume of data at each peer as well as the complexity of compositions of views or paths of mappings. However, CDSS users may be interested in the provenance of specific tuples, parts of certain derivations e.g., not starting from source tuples but from other derived tuples, or derivations of a certain "shape". One of the contributions of this thesis is ProQL, a query language for provenance graphs that can help users navigate through such graphs and focus on parts of interest to them. Moreover, ProQL exploits the generality of the provenance model to compute annotations such as the ones described above. Finally, we describe strategies for implementing ProQL over an RDBMS and investigate indexing techniques to optimize processing of ProQL queries.

1.3 Challenges, Contributions and Outline of Dissertation

To recap, in this dissertation we are addressing the following challenges:

- To define *formal semantics* for *update exchange*, that tolerate disagreement between participants and allow them to filter data according to their own provenance-based trust policies, and develop algorithms to *perform* it.

- To *record* the provenance of data, as updates are propagated across peers during update exchange, and *exploit* it for CDSS operations, e.g., to assess trust.

- To provide tools that allow CDSS users to also *exploit* the provenance of the data in their local instance, after update exchange, e.g., in order to include *provenance testing* in their data querying or *compute annotations* for their data that are useful for a variety of applications.

To address these challenges, in this dissertation we make the following contributions:

- Building upon techniques for exchanging *data* using networks of schema mappings, in Chapter 3 we define unidirectional *update exchange* between participants with heterogeneous schemas: this generalizes incremental view maintenance, peer data management [64] and data exchange [89, 45].

- In Chapter 4 we define a rich model of provenance information for relational algebra and recursive datalog queries based on semirings of *polynomials with coefficients from* $\mathbb{N}$. We extend this model to semirings of *formal power series* with unary functions for the (possibly infinite) provenance of query answers of datalog programs, such as the ones we use to perform update exchange, as we explain in Chapter 5. We show that, even though datalog provenance can be infinite, it can be generated by finite *algebraic systems* of fixed point equations, which imply a *provenance graph model* that is more suitable for visualization and querying, while also capturing relationships between the provenance of derived tuples. We also show that this provenance model generalizes query answering on other forms of annotated relations and can thus be used to compute query answers for different kinds of annotated relations by substituting appropriate values in the provenance expressions (without needing to recompute the queries), including trust annotations for imported tuples, according to users' *trust conditions*.

- In Chapter 5 we describe a general data provenance encoding in relations and show how sets of schema mappings can be translated into *datalog* programs that performs update exchange and records its provenance simultaneously. These techniques form the basis for an implementation of update exchange. We also describe an algorithm that uses provenance information to optimize update exchange, and in particular to detect which tuples are no longer derivable and should be deleted, when a deletion is propagated. Moreover, we introduce a language and semantics for *bidirectional schema mappings* in data and update exchange, that generalizes view update in a CDSS setting and is useful for propagating both data and updates symmetrically among sets of database instances. We show how *update policies* can be expressed along with these mappings, and present an algorithm that uses provenance information to detect and avoid updates that cause *side effects* at *run time*

- We develop a complete implementation of unidirectional and bidirectional update exchange in our ORCHESTRA CDSS prototype, with novel algorithms and encoding schemes to translate updates, maintain provenance, and apply trust conditions and provide a detailed experimental study of the scalability and performance of our implementation of CDSS operations (Chapter 6).

- We define a query language for provenance graphs, ProQL which is useful in supporting a wide variety of applications with derived data (Chapter 7). This language can be used to assess trust and derivability or detect side effects, as required for CDSS operations, as well as to express more complicated provenance queries and, optionally, compute data annotations in particular semirings.
- We develop a prototype implementation of ProQL — for acyclic provenance graphs — over an RDBMS, introduce indexing techniques for speeding up certain classes of ProQL queries and provide a detailed experimental study of the performance of provenance querying in a CDSS as well as the benefits of different indexing techniques (Chapter 8).

Chapter 2 provides background about data exchange, which forms the theoretical foundation for CDSS update exchange. Finally, in Chapter 9 we discuss related work and we present conclusions and future research directions in Chapter 10.

Chapter 2

Background

Update exchange builds upon techniques for *data exchange* using networks of schema mappings. In particular, we use schema mappings to express relationships between multiple schemas in a CDSS, and we propagate updates in order to maintain peer instances that conform to an extension of data exchange solution semantics. For this reason, before presenting its formal semantics, we recall some fundamental data exchange concepts, such as *schema mappings* and how to compute an instance from which all *certain answers* to users' queries can be computed directly.

2.1 Schema Mappings and Data Exchange

In a data exchange [45] setting, we have a source schema $\mathcal{S}$ and a target schema $\mathcal{T}$. For simplicity of the presentation, and without loss of generality, assume that $\mathcal{S} \cap \mathcal{T} = \emptyset$. The relationship between $\mathcal{S}$ and $\mathcal{T}$ is modeled through *schema mappings*, expressed in the form of *tuple generating dependencies* or *tgd*s, i.e., logical expressions of the form:

$$\forall \mathbf{x}, \mathbf{y} \; (\phi(\mathbf{x}, \mathbf{y}) \rightarrow \exists \mathbf{z} \; \psi(\mathbf{x}, \mathbf{z}))$$

where the left hand side (LHS) of the implication, ϕ, is a conjunction of atoms over variables $\bar{x}$ and $\bar{y}$, and the right hand side (RHS) of the implication, ψ, is a conjunction of atoms over variables $\bar{x}$ and $\bar{z}$. A tgd expresses a constraint about the existence of a tuple in the instance on the RHS, given a particular combination of tuples satisfying the constraint of the LHS. Tgds are a popular means of specifying constraints and mappings [39, 45] in data sharing, and they are equivalent to so-called *global-local-as-view* or *GLAV* mappings [51, 64], which in turn generalize the earlier global-as-view and local-as-view mapping formulations [78].

In the case of the data exchange problem, the authors consider only:

- source-to-target tgds, where ϕ and ψ are conjunctions of atoms from $\mathcal{S}$ and $\mathcal{T}$, respectively
- target tgds, in which both ϕ and ψ only contain relational atoms from $\mathcal{T}$.

In [45] the authors also allow the use of target *equality-generating dependencies* or *egds*, in order to specify constraints over the target schema. These are logical expressions of the form:

$$\forall \mathbf{x}\ \phi(\mathbf{x}) \rightarrow x_i = x_j$$

where ϕ only contains relational atoms from $\mathcal{T}$.

Let $\mathcal{M}$ be a set of dependencies of these kinds, and consider the joint schema $\mathcal{S} \cup \mathcal{T}$: its instances are pairs of instances $\langle I, J \rangle$ s.t. I is an instance of $\mathcal{S}$ and J is an instance of $\mathcal{T}$. In this context we can view $\mathcal{M}$ as a (conjunction of) logical assertion(s) over the joint schema $\mathcal{S} \cup \mathcal{T}$. Then, we write $\langle I, J \rangle \models \mathcal{M}$ if the pair of instances I, J satisfies the set of mappings $\mathcal{M}$. Then, given source and target schemas $\mathcal{S}, \mathcal{T}$, a set of dependencies $\mathcal{M}$ and an instance I of $\mathcal{S}$ a *solution* to the data exchange problem is an instance J of $\mathcal{T}$ such that $\langle I, J \rangle \models \mathcal{M}$.

In general, given a set of mappings $\mathcal{M}$ and a source instance I there may be several possible *solutions* to the data exchange problem, i.e., instances J of $\mathcal{T}$ s.t., $\langle I, J \rangle \models \mathcal{M}$. Consider for example the following situation:

Example 3. *Let* $\mathcal{S} = \{S_1(\mathtt{a}, \mathtt{b}), S_2(\mathtt{c}, \mathtt{d})\}$*,* $\mathcal{T} = \{T(\mathtt{e}, \mathtt{f})\}$*,*
$\mathcal{M} = \{d_1 : \forall x, y\ S_1(x, y) \rightarrow \exists z\ T(x, z), \quad d_2 : \forall x, y\ S_2(x, y) \rightarrow \exists z\ T(z, y)\}$ *and*
$I = \{S_1(a_0, b_1), S_2(a_1, b_0)\}$

Observe that the tgds in $\mathcal{M}$ *do not completely specify the target instance. For instance,* d_1 *requires that for every constant value that appears as value of attribute* `a` *of a tuple* $\boldsymbol{t} \in S_1$*, there is a tuple* $\boldsymbol{t}' \in T$ *with the same value for attribute* `e`*. However, it does not specify the value of attribute* `f` *of* $\boldsymbol{t}'$*, or whether there may be multiple tuples in* T *with the same value for attribute* `e` *and multiple different values for attribute* `f`*. In order to express this incomplete information, one can use special values, called* labeled nulls*, which are essentially variables (as opposed to "normal" values, i.e., constants). For example, one target instance for the setting presented above would be:* $J = \{T(a_0, Z_0), T(Z_1, b_0)\}$*, where* Z_0, Z_1 *are labeled nulls. Other solutions would be* $J' = \{T(a_0, b_0)\}$ *and* $J'' = \{T(a_0, Z_0), T(Z_1, b_0), T(Z_2, b_0)\}$*.*

2.2 Computing Instances in Data Exchange

As shown in Example 3, for a given data exchange setting there can be (possibly infinitely) many solutions, raising the natural question about which solution to materialize. For this reason, the authors of [45] identified the special class of *universal* solutions, that have several desirable properties. Before we introduce the definition of a universal solution, we need to clarify the notion of a homomorphism between joint "instances" that can contain labeled nulls:

Definition 2.2.1 (Homomorphism [45]). *Let* $\mathcal{R}$ *be a schema,* $\mathcal{C}$ *be a finite set of constants (the domain of* $\mathcal{R}$*) and* $\mathcal{V}$ *be an infinite set of variables (i.e., the labeled nulls). Moreover, for an instance* K *of* $\mathcal{R}$*, let* $\mathcal{V}(K)$ *be the finite subset of* $\mathcal{V}$*, whose contents are the variables that appear in* K*. Let* K_1 *and* K_2 *be two instances over* $\mathcal{R}$*, with values in* $\mathcal{C} \cup \mathcal{V}$*.*

- *A **homomorphism** $h : K_1 \to K_2$ is a mapping from $\mathcal{C} \cup \mathcal{V}(K_1)$ to $\mathcal{C} \cup \mathcal{V}(K_2)$ that is the identity on constants and for every tuple **t** and every relation $R_i \in \mathcal{R}$ s.t. $R_i(\boldsymbol{t}) \in K_1$, $R_i(h(\boldsymbol{t})) \in K_2$ (where if $\boldsymbol{t} = (t_1, \ldots, t_n)$, then $h(\boldsymbol{t}) = (h(t_1), \ldots, h(t_n))$).*

- *K_1 is homomorphically equivalent to K_2 if there is a homomorphism $h : K_1 \to K_2$ and $h' : K_2 \to K_1$*

- *K_1 is isomorphic to K_2 if there is a bijection h that is a homomorphism from K_1 to K_2 and its inverse h^{-1} is a homomorphism from $K_2 \to K_1$.*

Definition 2.2.2 (Universal solution [45]). *If $\mathcal{M}$ is a mapping and I is a source instance then a **universal solution** for I is an instance J s.t. for every solution J' there is a homomorphism $h : J \to J'$.*

Example 4. *To return to our example above, the solution J turns out to be universal, while J' and J'' are not. For example, the homomorphism $h : J \to J'$ is the identity in constants plus $\{h(Z_0) = b_0, h(Z_1) = a_0\}$. Notice that h is also a homomorphism $J \to J''$ (although it does not map any tuple to $T(Z_2, b_0)$, since it does not have to be surjective).*

Universal solutions turn out to have several useful properties, that make them good candidates for materialization. First, they can be used to compute *certain answers* for unions of conjunctive queries, which is the standard semantics for query answering in data integration [45, 51, 64, 72, 79]. Moreover, a universal solution can be computed by applying the *chase* on the joint instance $\langle I, J \rangle$, instead of on a query, as in the case of equivalent reformulations for query optimization. More precisely, applying the chase on a (joint) instance K proceeds as follows:

- if $d : \forall \mathbf{x}\ \phi(\mathbf{x}) \to \exists \mathbf{y}\ \psi(\mathbf{x}, \mathbf{y})$ is a tgd, the chase with d is applicable if there exists a homomorphism h from $\phi(\mathbf{x})$ to K that cannot be extended to a homomorphism h' from $\phi(\mathbf{x}) \wedge \psi(\mathbf{x}, \mathbf{y})$ to K

- If the chase with d is applicable, let K' be the union of K with the facts produced by: (a) extending h to h' such that each variable in $\mathbf{y}$ is assigned a fresh labeled null (different for each y_i), and (b) taking the image of the atoms of ψ under h'.

The algorithm that computes the universal solution starts with a joint instance $\langle I, \emptyset \rangle$, where I is a source instance and $\emptyset$ is a target instance, and applies the chase to it. For CDSS update exchange we only consider sets of tgds; in this case the chase cannot fail (as is the case in the presence of egds). Moreover, if the set of tgds is *weakly acyclic* [40, 45], the chase is guaranteed to terminate. In [45] the authors define the result of the chase (when it terminates) as the *canonical universal solution*. However, the chase is not deterministic i.e., considering the dependencies in different order may produce different terminating chase sequences. As a result, in the general case the universal solution that is computed by the chase is not unique. The authors of [80, 65] identified a slightly different form of canonical universal solution which can be defined deterministically and is also appropriate for computing certain answers.

In a CDSS, we perform update exchange by maintaining peer instances that correspond to the canonical universal solution (of [80, 65]) of the data exchange setting that involves the source data and the schema mappings at each peer. As a result, each peer can use their local instance to compute the certain answers for any union of conjunctive queries. As we show in Chapter 5 such solutions can be maintained incrementally by translating schema mappings into appropriate datalog programs, whose execution involves similar steps to the chase procedure. For this reason, in order to record the provenance of update exchange we need a model that captures the provenance of datalog programs, such as the one we present in Chapter 4.

In the general case, including target tgds and egds, the chase may also fail or diverge. As it was proved in [45], if the chase terminates, its result is a universal solution, while if there exists some failing finite chase starting with $\langle I, \emptyset \rangle$, then there is no universal solution. Moreover, they identified a sufficient condition for the mapping to guarantee termination of the chase, namely that it is a union of a *weakly acyclic*[1] set of tgds with a set of egds. Finally, observe that in the general case the chase is not deterministic, i.e., considering the dependencies in the mapping in different order may produce different terminating chase sequences. As a result, in the general case the universal solution that is computed by the chase is not unique.

2.2.1 Certain Answers

An important reason for materializing a target instance, especially for data integration applications, is to be able to answer queries over the target schema using this target instance directly. To address this problem, we first need to define what it is we expect to get as "relevant" answers by evaluating queries over a target schema, given a source schema and instance and schema mappings. The answer would be simple if for every I there was a single J s.t. $\langle I, J \rangle \models \mathcal{M}$, namely $Q(J)$. However, this is not the case, and in general there are many Js, instances of $\mathcal{T}$, s.t. $\langle I, J \rangle \models \mathcal{M}$. This situation is reminiscent of the problem of querying *incomplete databases* [97, 66], as it was first observed in [2] for a particular case of mappings. For this reason, the authors of [2] suggested the set of *certain answers*, which has been adopted by most researchers as the desired set of answers to the problem of answering queries across mappings.

Definition 2.2.3 (Certain answers). *A tuple* $\mathbf{t}$ *is a certain answer of* $Q_{\mathcal{T}}$ *across* $\mathcal{M}$ *if for every* J *such that* $\langle I, J \rangle \models \mathcal{M}$, $\mathbf{t} \in Q_{\mathcal{T}}(J)$. *As a result, for every source instance* I, *the set*

[1] This notion refers to a different kind of mapping graph and is not to be confused with acyclicity of mappings in the context of Peer Data Management Systems [64]. A more detailed analysis of weak acyclicity is outside the scope of this dissertation, but can be found in [40, 45]

of all certain answers of a query Q across a mapping $\mathcal{M}$:

$$certain_{\mathcal{M},I}(Q_{\mathcal{T}}) \triangleq \bigcap_{J:\langle I,J\rangle \models \mathcal{M}} Q_{\mathcal{T}}(J)$$

It turns out that not all data exchange solutions are appropriate for computing certain answers. Fortunately, as it was proved in [45], universal solutions can be used to compute the *certain answers* to (unions of) conjunctive queries over $\mathcal{T}$; in particular, one can compute these answers by evaluating such queries over the universal solution, treating labeled nulls as constants, and dropping tuples with labeled nulls from the result.

As we explain in the next chapter, in a CDSS we use schema mappings to express relationships between multiple schemas and perform update exchange in a way that maintains the canonical universal solution of the data exchange setting involving the source data and the schema mappings at each peer. As a result, each peer can use their local instance to compute the certain answers for any union of conjunctive queries.

Chapter 3

Introduction to CDSS Update Exchange

The CDSS model builds upon the fundamentals of data exchange, data integration and peer data management [64], but adds several novel aspects. As in a PDMS, the CDSS contains a set of *peers*, each representing an autonomous domain of control. Each peer's administrator has full control over a local DBMS, its schema, the conditions under which the peer trusts data and the schema mappings relating its schema with that of other peers. However, the main mode of operation is different. PDMSs allow users to write queries over their own schema, and translate them over the schemas of other peers, in order to be executed over their instances. In a CDSS we want to deal with *updates* at each peer, as well as to leave room for disagreement between peers, without requiring a global consensus on the data. For this reason, each peer maintains a local instance containing their local data as well as data imported through mappings, together with their provenance. Imported data can be filtered according to each peer's *trust policies* or curated, possibly taking their provenance into account; however, these changes are not imposed on the sources from which this data was imported (who may, in fact, disagree with them).[1] Then, CDSS users can pose their queries over, or perform further updates to, this local instance. We describe CDSS operation in Section 3.1 with an example showing update translation, provenance tracking and trust conditions. Section 3.2 formalizes these operations.

3.1 CDSS Operation

Each peer **P** in a CDSS represents an autonomous domain with its own unique schema and associated *local data instance*. The users located at **P** query and update the local instance in an "offline" fashion. Their updates are recorded in a *local edit log*. Periodically, upon the initiative of **P**'s administrator, **P** requests that the CDSS perform an *update exchange* operation. This *publishes* **P**'s local edit log — making it globally available via central or distributed storage [95], as shown at the top part of Figure 3.1. This also subjects **P** to the effects of the updates that the other peers have published (since the last time **P** participated in an update exchange). To determine these effects, the CDSS performs *update translation* (overview in Section 3.1.1), using the schema mappings to compute corresponding updates over **P**'s schema. As the updates are being translated, they are also filtered based on **P**'s *trust* conditions (overview in Section 3.1.2) that use the *provenance* of the data in the updates. As a result, only trusted updates are applied to **P**'s database, whereas untrusted data is *rejected*. Additional rejections are the result of manual curation: If a local user deletes data that was *not* inserted by **P**'s users (and hence must have arrived at **P** via update exchange), then that data remains rejected by **P** in future update exchanges of the CDSS. These steps are illustrated at the bottom part of Figure 3.1.

[1] In Chapter 5.3 we introduce update exchange over *bidirectional* schema mappings, for peers that require closer collaboration, e.g., involving the propagation of such changes back to their sources. To distinguish with them, we sometimes refer to update exchange as described in this chapter as *unidirectional*.

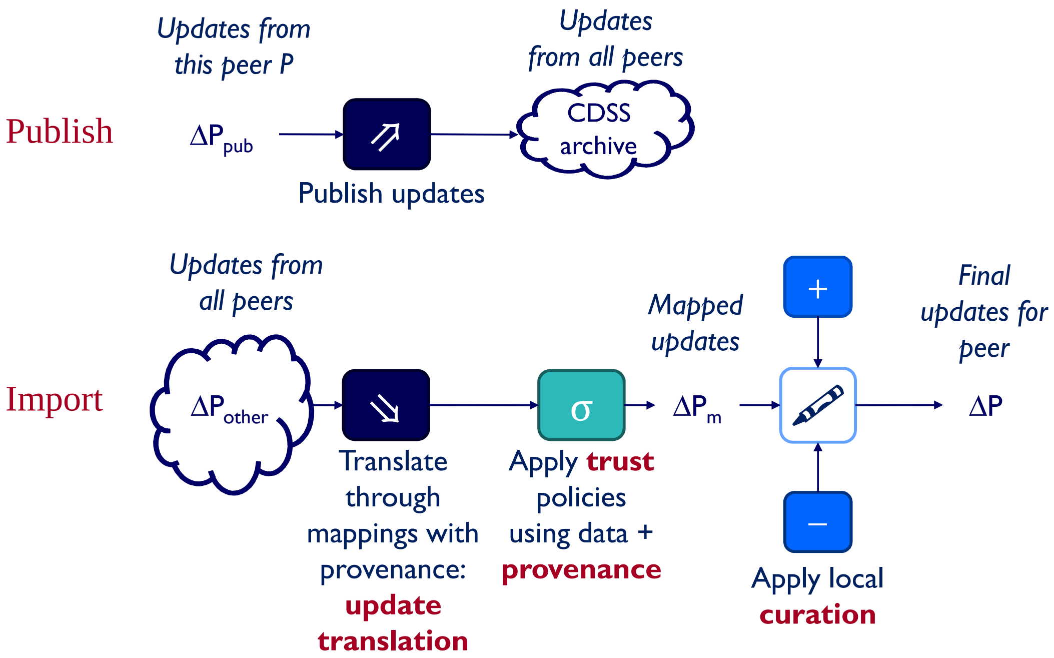


Figure 3.1: High-level description of CDSS operation

After update exchange, **P**'s local instance will intuitively contain data mapped from other peers, "overlaid" by any updates made by **P**'s users and recorded in the local edit log. If **P**'s instance is subsequently updated locally, then **P**'s users will see the effects of these edits. Other peers in the CDSS will only see data that resulted from the **last** update exchange, i.e., they will not see the effects of any unpublished updates at **P**. This situation continues until **P**'s next update exchange.

Intuitively, this operating mode resembles deferred view maintenance across a set of views — but there are a number of important differences, in part entailed by the fact that instances are related by schema mappings rather than views, and in part entailed by peers' ability to specify local edits and trust policies.

Application of updates. The result of update translation, once trust policies have been enforced over provenance and data values, is a set of updates to the local instances and their provenance. In this thesis, we assume that these updates are mutually compatible. A more realistic approach would treat them as *candidate* updates and further use the prioritization and conflict reconciliation algorithm of [95] to determine which updates to apply. In fact, we do so in our ORCHESTRA prototype implementation, but for simplicity of presentation we ignore this aspect in the model formalization that follows.

In the remainder of this section, we provide a high-level overview of the characteristic aspects of update exchange: (1) how updates are translated along mappings between different peer schemas, (2) how trust conditions can filter updates based on their provenance and values.

3.1.1 Update Translation and Query Answers

The "source" or "base" data in a CDSS, as seen by the users, are the local edit logs at each peer. These edit logs describe local data creation and curation in terms of insertions and deletions/rejections. Of course, a local user submitting a query expects answers that are fully consistent with the local edit log. With respect to the *other* peers' edit logs the user would expect to receive all *certain* answers inferable from the schema mappings and the tuples that appear in the other peers' instances [64]. Indeed, the certain answers semantics has been validated by over a decade of use in data integration and data exchange [45, 51, 64, 72, 79].

Queries are answered in a CDSS using only the local peer instance. Hence, the content of this instance must be such that all and only answers that are certain (as explained above) and consistent with the local edit log are returned. As was shown in the work on data exchange [45, 80, 65], the certain answer semantics can in fact be achieved through a form of "data translation", building peer instances called *canonical universal solutions*. In our case, the source data consists of edit logs so we generalize this to *update translation*. As we explained in Chapter 2, *canonical universal solutions* can be computed using the chase procedure. This consists of a series of applications of tgds from $\mathcal{M}$ to derive more tuples, until a fixpoint is reached. Then, the provenance of an imported tuple essentially records all these chase steps that were involved in deriving the tuple.

A key aspect of the canonical universal solutions is the *placeholder values* or *labeled nulls* for unknown values that are nonetheless needed in order to validate mappings with existentials (such as m_3 in Example 5). The labeled nulls are internal bookkeeping (e.g., queries can join on their equality), but tuples with labeled nulls are discarded in order to produce certain answers to queries. (We can additionally return tuples with labeled nulls, i.e., a superset of the certain answers, which may be desirable for some applications.)

Example 5. *Having defined the main concepts of data exchange in Chapter 2, we can now explain the CDSS of Figure 1.1 in more detail. To do so, we first need to establish some notation. We use Σ for the* ***union*** *of all peer schemas and $\Sigma(\boldsymbol{P})$ for the schema of peer $\boldsymbol{P}$. We use $\mathcal{M}$ for the set of all mappings, which we can think of as logical constraints on Σ. When we refer to mappings we will use the notation of tgds. For readability, we will omit the universal quantifiers for variables in the LHS. When we later refer to queries, including queries based on mappings, we will use the similar notation of datalog (which, however, reverses the order of implication, specifying the output of a rule on the left).*

Peers $\boldsymbol{P}_{GUS}, \boldsymbol{P}_{BioSQL}, \boldsymbol{P}_{uBio}$ have one-relation schemas describing taxa IDs, names, and canonical names:$\Sigma(\boldsymbol{P}_{GUS}) = \{G(id, can, nam)\}$, $\Sigma(\boldsymbol{P}_{BioSQL}) = \{B(id, nam)\}$, $\Sigma(\boldsymbol{P}_{uBio}) = \{U(nam, can)\}$. Among these peers are mappings $\mathcal{M} = \{m_1, m_2, m_3, m_4\}$, shown as arcs in the figure. The mappings are:

$$
\begin{array}{ll}
(m_1) & G(i, c, n) \rightarrow B(i, n) \\
(m_2) & G(i, c, n) \rightarrow U(n, c) \\
(m_3) & B(i, n) \rightarrow \exists c \, U(n, c) \\
(m_4) & B(i, c) \wedge U(n, c) \rightarrow B(i, n)
\end{array}
$$

Observe that m_3 has an existential variable: the value of c is unknown (and not necessarily unique). The first three mappings all have a single source and target peer, corresponding *to the LHS and the RHS of the implication. In general, relations from multiple peers may occur on either side, as in mapping m_4, which defines data in the BioSQL relation based on its own data combined with tuples from uBio.*

Continuing our example, assume that the peers have the following local edit logs (where '$+$' signifies insertion):

ΔG

$+$	1	2	3
$+$	3	5	2

ΔB

$+$	3	5

ΔU

$+$	2	5

The update translation constructs local instances that contain:

G

1	2	3
3	5	2

B

3	5
3	2
1	3
3	3

U

2	5
3	2
5	c_1
2	c_2
3	c_3

Examples of (traditional) certain answers query semantics at $\boldsymbol{P}_{BioSQL}$:

- *query* $ans(x,y)$:- $U(x,z), U(y,z)$ *returns* $\{(2,2),(3,3),(5,5)\}$;
- *query* $ans(x,y)$:- $U(x,y)$ *returns* $\{(2,5),(3,2)\}$.

Moreover, if the edit log ΔB would have also contained the curation deletion $(\,-\mid 3 \quad 2\,)$ then after update translation, B would not only be missing $(3,2)$, but also $(3,3)$; and U would be missing $(2,c_2)$.

Finally, this example suggests that the set semantics is not telling the whole story. For example the tuple $U(2,5)$ has two different "justifications": it is a local insertion as well as the result of update translation via (m_2). The tuple $B(3,2)$ comes from two different update translations, via (m_1) and via (m_4). These correspond to alternative derivations *in the provenance of $B(3,2)$.*

The challenge in the CDSS model is that peer instances cannot be computed merely from schema mappings and data instances. The ability of all peers to do curation deletions and trust-based rejections requires a new formalization that takes edit logs and trust policies into account. We outline in Section 3.2 how we can do that and still take advantage of the techniques for building canonical universal solutions.

3.1.2 Trust Policies and Provenance

In addition to schema mappings, which specify the relationships between data elements in different instances, the CDSS supports *trust policies*. These express, for each peer **P**, what data from update translation should be trusted and hence accepted. The trust policies consist of *trust conditions* that refer to the other peers, to the schema mappings, and even to selection predicates on the data itself. Different trust conditions may be specified separately by each peer, and we discuss how these compose in Section 4.8.1.

Example 6. *Some possible trust conditions in our CDSS example:*

- *Peer $\mathbf{P}_{BioSQL}$ distrusts any tuple $B(i, n)$ if the data came from $\mathbf{P}_{GUS}$ and $n \geq 3$, and trusts any tuple from $\mathbf{P}_{uBio}$.*
- *Peer $\mathbf{P}_{BioSQL}$ distrusts any tuple $B(i, n)$ that came from mapping (m_4) if $n \neq 2$.*

Adding these trust conditions to the update exchange in Example 5 we see that $\mathbf{P}_{BioSQL}$ will reject $B(1, 3)$ by the first condition. As a consequence, $\mathbf{P}_{uBio}$ will not get $U(3, c_3)$. Moreover, the second trust condition makes $\mathbf{P}_{BioSQL}$ reject $B(3, 3)$. Note that formulations like "comes from $\mathbf{P}_{GUS}$" need a precise meaning. We give this in Chapter 4.

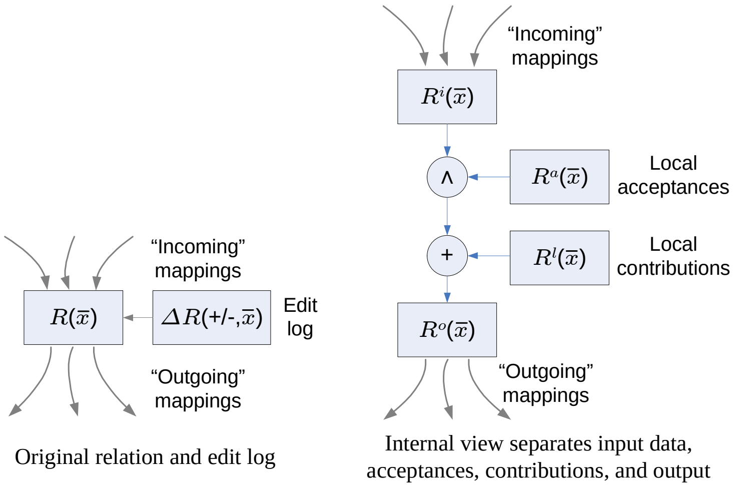
Since the trust conditions refer to other peers and to the schema mappings, the CDSS needs a precise description of how these peers and mappings have contributed to a given tuple produced by update translation. Information of this kind is commonly called *data provenance*. As we saw at the end of Example 5, provenance can be quite complex in a CDSS. In particular, we need a more detailed provenance model than why-provenance [21] and lineage [35] (including the extended model recently proposed in [11]). We discuss their limitations more thoroughly in Section 4.3, but informally, we need to know not just from which tuples a tuple is derived, but also *how* it is derived, including separate alternative derivations through different mappings.

3.2 Update Exchange Formalized

After the informal overview in the previous section, we now provide a formal discussion of how local edits and schema mappings work together in a CDSS. After we also present our model of provenance in Chapter 4 we explain how this framework can be extended to incorporate trust conditions. In particular, we extend the model proposed in the data exchange literature [45], which specifies how to compute peer instances, given data at other peers. We discuss how to incorporate updates into the computation of data exchange solutions, which we simply term *update translation*.

We first explain how the system automatically expands the user-level schemas and mappings into "internal" schemas and mappings. These support data exchange and additionally capture how edit log deletions and trust conditions are used to reject data translated from other peers. First, we state two fundamental assumptions we make about the form of the mappings and the updates.

We allow the set of mappings in the CDSS to only form certain types of *cycles* (i.e., mappings that recursively define relations in terms of themselves). In general, query answering is undecidable in the presence of cycles [64], so we restrict the topology of schema mappings to be at most *weakly acyclic* [42, 45]. Mapping (m_3) in Example 5 completes a cycle, but the set of mappings is weakly acyclic.



Original relation and edit log

Internal view separates input data, acceptances, contributions, and output

Figure 3.2: To capture the effects of the edit log on each relation (left), we internally encode them as four relations (right), representing incoming data, local acceptances and local contributions, and the resulting ("output") table.

We also assume that within the set of updates published at the same time by a peer, no data dependencies exist (perhaps because transient operations in update chains were removed [68]). These updates are stored in an *edit log*. For each relation $R(\bar{x})$ in the local instance we denote by $\Delta R(d, \bar{x})$ the corresponding edit log. ΔR is an ordered list that stores the results of manual curation at the peer, namely the inserted tuples whose d value is '$+$' and the deleted tuples whose d value is '$-$'.

Internal peer schemas. For each relation R in Σ, the user-level edit log ΔR and the local instance R are implemented internally by four different relations, all with the same attributes as R. We illustrate these relations in Figure 3.2. Their meaning is as follows:

- R^i is the peer's *input table*. It contains tuples produced by update translation, via mappings, from data at other peers.
- R^ℓ, the peer's *local contributions table*, contains the tuples inserted locally, unless the edit log shows they were later deleted.

- R^α, the peer's *acceptance table*, contains tuples that were imported from other peers and have not been rejected through a local curation deletion. (Deletions of local contributions are dealt with simply by removing tuples from R^ℓ).

- R^o is the peer's curated table and also its *output table*. After update exchange, it will contain the local contributions as well as the input tuples that are not rejected. This table is the source of the data that the peer exports to other peers through outgoing mappings. This is also the table that the peer's users query, called the local instance in Section 3.1.1 and Example 5.

Internal schema mappings. Along with expanding the original schema into the internal schema, the system transforms the original mappings $\mathcal{M}$ into a new set of tgds $\mathcal{M}'$ over the internal schema, which are used to specify the effects of local contributions and rejections.

- For each tgd m in $\mathcal{M}$, we have in $\mathcal{M}'$ a tgd m' obtained from m by replacing each relation R on the LHS by R^o and each relation R on the RHS by R^i;

- For each R in Σ, $\mathcal{M}'$ is extended with rules to remove tuples in R^r and add those in R^ℓ:

$$\begin{array}{ll} (i_R) & R^i(\bar{x}) \land R^\alpha(\bar{x}) \to R^o(\bar{x}) \\ (\ell_R) & R^\ell(\bar{x}) \to R^o(\bar{x}). \end{array}$$

Then, the results of [45, 80, 65] generalize to this set of tgds:

Theorem 3.2.1. *Let $\mathcal{M}$ be weakly acyclic and I be an instance of all local contributions and acceptance tables.Then $\mathcal{M}'$ is also weakly acyclic, hence chase$_{\mathcal{M}'}(I)$ terminates in polynomial time, and moreover it yields an instance of the of input and output tables that is a canonical universal solution for I with respect to $\mathcal{M}'$.*

To recompute the relation input and output instances based on the extensional data in the local contributions and acceptance tables, we use a procedure similar to the α-chase of [65], as described in Chapter 5. For the provenance of derived tuples we want to record the sequence of steps of this chase procedure that were involved in each of its (possibly multiple) derivations; for each step, this includes the mapping that was used and the tuples that it was applied on.

Definition 3.2.1 (Consistent system state)**.** *An instance (I, J) of Σ', where I is an instance of the local acceptances and contributions tables and J of the input and output tables, is* ***consistent*** *if J is a canonical universal solution for I with respect to $\mathcal{M}'$.*

(In [57] we provided an alternative formalization uses a *rejections table*, R^r, to record all tuples rejected by the local curator at every peer. Although the intended semantics is the same, we need to use negation in internal schema mappings in order to express the application of rejection (i.e., the operation $R^i - R^r$). This is a *safe* form of negation, since the negated atoms occur only on the LHS of the implication and every variable in the LHS also occurs in a positive atom there. As a result, this form of negation does not have adverse effects w.r.t. termination of the computation of a solution, and in fact the chase procedure produces the same instances for the input and output tables at every peer, as with the formalization presented above. In fact, the two semantics are equivalent if all the tgds are *full* (i.e., they contain no existentially quantified variables) or when the rejection tables are all empty. However, if there are tuples with the labeled nulls R^i or R^r, their interaction with negation can cause problems; in fact, there are cases where no universal solution exists.[2])

To recap, *publishing* a peer **P**'s edit log means producing a new instance of **P**'s local acceptances and contributions tables, while holding the other peers' local ac-

[2] A similar phenomenon was pointed out for mappings containing a more general form of negation in [41].

ceptances and contributions tables the same[3]. *Recomputing* **P**'s instance means computing a new instance of **P**'s input and output tables that is a canonical universal solution with respect to the internal mappings $\mathcal{M}'$. By definition, after each update exchange the system must be in a consistent state. (The initial state, with empty instances, is trivially consistent.)

The semantics presented above deal with translating updates over schema mappings and allowing curation of imported data. However, update exchange also involves tracking of provenance and evaluation of trust policies over it. In order to incorporate these in our formal model, we first need to define our provenance model. We do so in the next chapter, and present our the necessary extensions to our formal update exchange semantics in Section 4.8.2.

[3] This easily generalizes to updates over multiple instances.

Chapter 4

Provenance Model

For the kinds of provenance applications described in Section 1.2 we need, for every imported tuple, to be able to determine the source tuples it originated from, as well as *how* they were combined and the mappings involved in its propagation. Existing provenance models, such as why-provenance [21] and lineage [35] (including the extended model recently proposed in [11]) focus on identifying the set of source tuples involved but fail to represent some aspects of the views involved in the derivation: whether there is a single or multiple derivations, how the tuples were combined in each of them and which specific queries were involved. Essentially, we need a data provenance model that captures the abstract form of tuple derivations produced by steps of applications of mappings during the computation of the canonical universal solution that is the result of update exchange. We first define a model for the provenance of positive relational algebra ($\mathcal{RA}^+$) queries, since the body of each mapping is such a query.

Intuitively, for every derived tuple we want to represent all the ways in which it was derived as expressions over source tuples and the mappings involved, while distinguishing alternative derivations. Similar requirements have appeared in the context of *annotated relations*, where query answering involves generalizing the relational algebra ($\mathcal{RA}$) to perform corresponding operations on the annotations of input tuples, in order to compute correct annotations for output tuples. One such example of query answering over annotated relations comes from the seminal paper in incomplete databases [66], which generalized $\mathcal{RA}^+$ to c-tables, where relations are annotated with Boolean formulas. In probabilistic databases, [52] and [98] generalized $\mathcal{RA}^+$ to event tables, also a form of annotated relations. In data warehousing, [35] and [36] compute lineages for tuples in the output of queries, in effect generalizing $\mathcal{RA}^+$ to computations on relations annotated with sets of contributing tuples. Finally, $\mathcal{RA}^+$ on bag semantics can be viewed as a generalization to annotated relations, where a tuple's annotation is a number representing its multiplicity.

In [58] we observe that in all four cases, the calculations with annotations are strikingly similar. However, the final annotations do not encode all the provenance information we need, because they discard some of it during the computation in order to simplify the final annotation. For example, in c-tables, as we explain in Section 4.1, the annotations are boolean expressions, and boolean absorption is used to simplify them along the way. This suggests looking for an algebraic structure on annotations that captures the above as particular cases. We propose using commutative semirings for this purpose in Section 4.2. In fact, we can show that the laws of commutative semirings are *forced* by certain expected identities in $\mathcal{RA}^+$. Having identified commutative semirings as the right algebraic structure, we argue in Section 4.3 that a symbolic representation of semiring calculations is just what is needed to record, document, and track $\mathcal{RA}^+$ querying from input to output for applications which require rich *provenance* information. It is a standard philosophy in algebra that such symbolic representations form *the most general* such structure. In the case of commutative semirings, the symbolic representation is that of polynomials. We therefore propose to use polynomials to capture *provenance*. To deal with sets of mappings that may contain cycles, we then extend our approach in Section 4.4 to model to recursive datalog queries, where we also

A B C	
$a\ b\ c$	?
$d\ b\ e$	?
$f\ g\ e$	?

(a)

A B C	
$a\ b\ c$	b_1
$d\ b\ e$	b_2
$f\ g\ e$	b_3

(b)

$$\left\{\emptyset, \begin{array}{|c|}\hline a\ c\\\hline\end{array}, \begin{array}{|c|}\hline d\ e\\\hline\end{array}, \begin{array}{|c|}\hline f\ e\\\hline\end{array}, \begin{array}{|c|}\hline a\ c\\ a\ e\\ d\ c\\ d\ e\\\hline\end{array}, \begin{array}{|c|}\hline d\ e\\ f\ e\\\hline\end{array}, \begin{array}{|c|}\hline a\ c\\ f\ e\\\hline\end{array}, \begin{array}{|c|}\hline a\ c\\ a\ e\\ d\ c\\ d\ e\\ f\ e\\\hline\end{array}\right\}$$

(c)

Figure 4.1: A maybe-table and a query result

A C	
$a\ c$	$(b_1 \wedge b_1) \vee (b_1 \wedge b_1)$
$a\ e$	$b_1 \wedge b_2$
$d\ c$	$b_1 \wedge b_2$
$d\ e$	$(b_2 \wedge b_2) \vee (b_2 \wedge b_2) \vee (b_2 \wedge b_3)$
$f\ e$	$(b_3 \wedge b_3) \vee (b_3 \wedge b_3) \vee (b_2 \wedge b_3)$

(a)

A C	
$a\ c$	b_1
$a\ e$	$b_1 \wedge b_2$
$d\ c$	$b_1 \wedge b_2$
$d\ e$	b_2
$f\ e$	b_3

(b)

Figure 4.2: Result of Imielinski-Lipski computation

record the name of the mapping used at each step. Finally, in Section 4.8 we show how we can use provenance to compute query answers in different commutative semirings, and in particular in order to evaluate simple trust conditions in update exchange.

4.1 Queries on Annotated Relations

We illustrate the general form of query answering on annotated relations by considering three important examples of such relations and highlighting the similarities between them.

The first example comes from the study of *incomplete databases*, where a simple representation system is the *maybe-table* [93, 59], in which optional tuples are an-

A B C	
$a\ b\ c$	2
$d\ b\ e$	5
$f\ g\ e$	1

(a)

A C	
$a\ c$	$2 \cdot 2 + 2 \cdot 2 = 8$
$a\ e$	$2 \cdot 5 = 10$
$d\ c$	$2 \cdot 5 = 10$
$d\ e$	$5 \cdot 5 + 5 \cdot 5 + 5 \cdot 1 = 55$
$f\ e$	$1 \cdot 1 + 1 \cdot 1 + 5 \cdot 1 = 7$

(b)

Figure 4.3: Bag semantics example

notated with a '?', as in the example of Figure 4.1(a). Such a table represents a set of *possible worlds*, and the answer to a query over such a table is the set of instances obtained by evaluating the query over each possible world. Thus, given a query like

$$q(R) \stackrel{\Delta}{=} \pi_{\text{AC}}\big(\pi_{\text{AB}}R \bowtie \pi_{\text{BC}}R \cup \pi_{\text{AC}}R \bowtie \pi_{\text{BC}}R\big)$$

the query result is the set of possible worlds shown in Figure 4.1(c). Unfortunately, this set of possible worlds cannot itself be represented by a maybe-table, intuitively because whenever the tuples (a, e) and (d, c) appear, then so do (a, c) and (d, e), and maybe-tables cannot represent such a dependency.

To overcome such limitations, Imielinski and Lipski [66] introduced c-*tables*, where tuples are annotated with Boolean formulas called *conditions*. A maybe-table is a simple kind of c-table, where the annotations are distinct Boolean variables, as shown in Figure 4.1(b). In contrast to weaker representation systems, c-tables are expressive enough to be *closed* under $\mathcal{RA}^+$ queries, and the main result of [66] is an algorithm for answering $\mathcal{RA}^+$ queries on c-tables, producing another c-table as a result. On our example, this algorithm produces the c-table shown in Figure 4.2(a), which can be simplified to the c-table shown in Figure 4.2(b); this c-table represents exactly the set of possible worlds shown in Figure 4.1(c).

Another kind of table with annotations is a *multiset* or *bag*. In this case, the annotations are natural numbers which represent the multiplicity of the tuple in

A B C	
$a\ b\ c$	X
$d\ b\ e$	Y
$f\ g\ e$	Z

E	Pr
X	0.6
Y	0.5
Z	0.1

(a)

A C	
$a\ c$	X
$a\ e$	X $\cap$ Y
$d\ c$	X $\cap$ Y
$d\ e$	Y
$f\ e$	Z

(b)

Figure 4.4: Probabilistic example

the multiset. (A tuple not listed in the table has multiplicity 0.) Query answering on such tables involves calculating not just the tuples in the output, but also their multiplicities.

For example, consider the multiset shown in Figure 4.3(a). Then $q(R)$, where q is the same query from before, is the multiset shown in Figure 4.3(b). Note that for projection and union we add multiplicities while for join we multiply them. There is a striking similarity between the arithmetic calculations we do here for multisets, and the Boolean calculations for the c-table.

A third example comes from the study of *probabilistic databases*, where tuples are associated with values from $[0, 1]$ which represent the probability that the tuple is present in the database. Answering queries over probabilistic tables requires computing the correct probabilities for tuples in the output. To do this, Fuhr and Röllecke [52] and Zimányi [98] introduced *event tables*, where tuples are annotated with probabilistic events, and they gave a query answering algorithm for computing the events associated with tuples in the query output.[1]

Figure 4.4(a) shows an example of an event table with associated *event probabilities* (e.g., X represents the event that (a, b, c) appears in the instance, and X, Y, Z are assumed independent). Considering again the same query q as above, the Fuhr-

[1] The Fuhr-Röllecke-Zimányi algorithm is a general-purpose *intensional* algorithm. Dalvi and Suciu [37] give a sound and complete algorithm which returns a *safe query plan*, if one exists, which may be used to answer the query correctly via a more efficient *extensional* algorithm. Their results do not apply to our example query.

Rölleke-Zimányi query answering algorithm produces the event table shown in Figure 4.4(b). Note again the similarity between this table and the example earlier with c-tables. The probabilities of tuples in the output of the query can be computed from this table using the independence of X and Y.

4.2 Positive Relational Algebra

In this section we attempt to unify the examples above by considering generalized relations in which the tuples are annotated (*tagged*) with information of various kinds. Then, we will define a generalization of the positive relational algebra ($\mathcal{RA}^+$) to such tagged-tuple relations. The examples in Section 4.1 will turn out to be particular cases.

We use here the named perspective [3] of the relational model in which tuples are functions $t : U \to \mathbb{D}$ with U a finite set of attributes and $\mathbb{D}$ a domain of values. We fix the domain $\mathbb{D}$ for the time being and we denote the set of all such U-tuples by U-Tup. (Usual) relations over U are subsets of U-Tup.

A notationally convenient way of working with tagged-tuple relations is to model tagging by a function on all possible tuples, with those tuples not considered to be "in" the relation tagged with a special value. For example, the usual set-theoretic relations correspond to functions that map U-Tup to $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ with the tuples in the relation tagged by true and those not in the relation tagged by false.

Definition 4.2.1. *Let K be a set containing a distinguished element 0. A K**-relation** over a finite set of attributes U is a function $R : U$-$\mathsf{Tup} \to K$ such that its **support** defined by $\mathsf{supp}(R) \triangleq \{t \mid R(t) \neq 0\}$ is finite.*

In generalizing $\mathcal{RA}^+$ we will need to assume more structure on the set of tags. To deal with selection we assume that the set K contains two distinct values $0 \neq 1$

which denote "out of" and "in" the relation, respectively. To deal with union and projection and therefore to combine different tags of the same tuple into one tag we assume that K is equipped with a binary operation "$+$". To deal with natural join (hence intersection and selection) and therefore to combine the tags of joinable tuples we assume that K is equipped with another binary operation "$\cdot$".

Definition 4.2.2. *Let $(K, +, \cdot, 0, 1)$ be an algebraic structure with two binary operations and two distinguished elements. The definitions of the operations of the **positive algebra** are shown in Table 4.1.*

Proposition 4.2.1. *The operations of $\mathcal{RA}^+$ preserve the finiteness of supports therefore they map K-relations to K-relations. Hence, Definition 4.2.2 gives us an algebra on K-relations.*

This definition generalizes the definitions of $\mathcal{RA}^+$ for the motivating examples we saw. Indeed, for $(\mathbb{B}, \vee, \wedge, \mathsf{false}, \mathsf{true})$ we obtain the usual $\mathcal{RA}^+$ with set semantics. For $(\mathbb{N}, +, \cdot, 0, 1)$ it is $\mathcal{RA}^+$ with bag semantics.

For the Imielinski-Lipski algebra on c-tables we consider the set of Boolean expressions over some set B of variables which are *positive*, i.e., they involve only disjunction, conjunction, and constants for true and false. Then we identify those expressions that yield the same truth-value for all boolean assignments of the variables in B.[2] Denoting by $\mathsf{PosBool}(B)$ the result and applying Definition 4.2.2 to the structure $(\mathsf{PosBool}(B), \vee, \wedge, \mathsf{false}, \mathsf{true})$ produces exactly the Imielinski-Lipski algebra. Finally, for $(\mathcal{P}(\Omega), \cup, \cap, \emptyset, \Omega)$ we obtain the Fuhr-Rölleke-Zimányi $\mathcal{RA}^+$ on event tables.

These four structures are examples of *commutative semirings*, i.e., algebraic structures $(K, +, \cdot, 0, 1)$ such that $(K, +, 0)$ and $(K, \cdot, 1)$ are commutative monoids, $\cdot$ is

[2] in order to permit simplifications; it turns out that this is the same as transforming using the axioms of *distributive lattices* [34]

empty relation For any set of attributes U, there is $\emptyset : U\text{-}\mathsf{Tup} \to K$ such that $\emptyset(t) = 0$.

union If $R_1, R_2 : U\text{-}\mathsf{Tup} \to K$ then $R_1 \cup R_2 : U\text{-}\mathsf{Tup} \to K$ is defined by

$$(R_1 \cup R_2)(t) \ \triangleq\ R_1(t) + R_2(t)$$

projection If $R : U\text{-}\mathsf{Tup} \to K$ and $V \subseteq U$ then $\pi_V R : V\text{-}\mathsf{Tup} \to K$ is defined by

$$(\pi_V R)(t) \ \triangleq \sum_{t=t' \text{ on } V \text{ and } R(t') \neq 0} R(t')$$

(here $t = t'$ on V means t' is a U-tuple whose restriction to V is the same as the V-tuple t; note also that the sum is finite since R has finite support)

selection If $R : U\text{-}\mathsf{Tup} \to K$ and the selection predicate $\mathbf{P}$ maps each U-tuple to either 0 or 1 then $\sigma_{\mathbf{P}} R : U\text{-}\mathsf{Tup} \to K$ is defined by

$$(\sigma_{\mathbf{P}} R)(t) \ \triangleq\ R(t) \cdot \mathbf{P}(t)$$

Which $\{0, 1\}$-valued functions are used as selection predicates is left unspecified, except that we assume that false—the constantly 0 predicate, and true—the constantly 1 predicate, are always available.

natural join If $R_i : U_i\text{-}\mathsf{Tup} \to K \ \ i = 1, 2$ then $R_1 \bowtie R_2$ is the K-relation over $U_1 \cup U_2$ defined by

$$(R_1 \bowtie R_2)(t) \ \triangleq\ R_1(t_1) \cdot R_2(t_2)$$

where $t_1 = t$ on U_1 and $t_2 = t$ on U_2 (recall that t is a $U_1 \cup U_2$-tuple).

renaming If $R : U\text{-}\mathsf{Tup} \to K$ and $\beta : U \to U'$ is a bijection then $\rho_\beta R$ is a K-relation over U' defined by

$$(\rho_\beta R)(t) \ \triangleq\ R(t \circ \beta)$$

Table 4.1: Definitions of $\mathcal{RA}^+$ operations on K-relations

distributive over $+$ and $\forall a,\ 0 \cdot a = a \cdot 0 = 0$. Further evidence for requiring K to form such a semiring is given by

Proposition 4.2.2. *The following $\mathcal{RA}$ identities:*

- *union is associative, commutative and has identity $\emptyset$;*

- *join is associative, commutative and distributive over union;*
- *projections and selections commute with each other as well as with unions and joins (when applicable);*
- $\sigma_{\textsf{false}}(R) = \emptyset$ *and* $\sigma_{\textsf{true}}(R) = R$.

hold for the positive algebra on K-relations if and only if $(K, +, \cdot, 0, 1)$ is a commutative semiring.

Glaringly absent from the list of relational identities are the idempotence of union and of (self-)join. Indeed, these fail for the bag semantics, an important particular case of our general treatment.

Any function $h : K \to K'$ can be used to transform K-relations to K'-relations simply by applying h to each tag (note that the support may shrink but never increase). Abusing the notation a bit we denote the resulting transformation from K-relations to K'-relations also by h. The $\mathcal{RA}$ operations we have defined work nicely with semiring structures:

Proposition 4.2.3. *Let $h : K \to K'$ and assume that K, K' are commutative semirings. The transformation given by h from K-relations to K'-relations commutes with any $\mathcal{RA}^+$ query (for queries of one argument) $q(h(R)) = h(q(R))$ if and only if h is a semiring homomorphism.*

4.3 Polynomials for Provenance

Lineage and why-provenance were defined in [35, 36, 21] as ways of relating the tuples in a query output to the tuples in the query input that “contribute” to them. The lineage of a tuple t in a query output is in fact the *set* of all contributing input tuples. The why-provenance is a set of sets, each of which corresponds to the contributing input tuples of a single derivation (but multiple derivations with the

A B C	
$a\ b\ c$	p
$d\ b\ e$	r
$f\ g\ e$	s

(a)

A C	
$a\ c$	$\{p\}$
$a\ e$	$\{p, r\}$
$d\ c$	$\{p, r\}$
$d\ e$	$\{r, s\}$
$f\ e$	$\{r, s\}$

(b)

A C	
$a\ c$	$2p^2$
$a\ e$	pr
$d\ c$	pr
$d\ e$	$2r^2 + rs$
$f\ e$	$2s^2 + rs$

(c)

Figure 4.5: Lineage and provenance polynomials

same input tuples are still "collapsed" to the same set, i.e., the fact that there can be many of them is ignored.)

Computing the lineage for queries in $\mathcal{RA}^+$ turns out to be exactly Definition 4.2.2 for the semiring $(\mathcal{P}(X), \cup, \cup, \emptyset, \emptyset)$ where X consists of the ids of the tuples in the input instance. For example, we consider the same tuples as in relation R used in the examples of Section 4.1 but now we tag them with their own ids p,r,s, as shown in Figure 4.5(a). The resulting R can be seen as a $\mathcal{P}(\{p, r, s\})$-relation by replacing p with $\{p\}$, etc. Applying the query q from Section 4.1 to R we obtain according to Definition 4.2.2 the $\mathcal{P}(\{p, r, s\})$-relation shown in Figure 4.5(b).

This example illustrates the limitations of lineage (also recognized in [27]). For example, in the query result in Figure 4.5(b) (f, e) and (d, e) have the same lineage, the input tuples with id r and s. However, the query can also calculate (f, e) from s alone and (d, e) from r alone. In a provenance application in which one of r or s is perhaps less trusted or less usable than the other the effect can be different on (f, e) than on (d, e) and this cannot be detected by lineage. It seems that we need to know not just *which* input tuples contribute but also *how* they contribute.[3]

The semiring for why-provenance is more complicated, as described in [18], and is informative enough for boolean trust computations, such as the ones in the example above, but is still not rich enough for more complicated trust computa-

[3] In contrast to why-provenance, the notion of provenance we propose could justifiably be called **how-provenance**.

tions, such as the ones described at the end of Section 4.8.1 and in Chapter 7. On the other hand, by using the different operations of the semiring, Definition 4.2.2 appears to fully "document" how an output tuple is produced. To record the documentation as tuple tags we need to use a semiring of symbolic expressions. In the case of semirings, like in ring theory, these are the *polynomials*.

Definition 4.3.1. *Let X be the set of tuple ids of a (usual) database instance I. The* ***positive algebra provenance semiring*** *for I is the semiring of polynomials with variables (a.k.a. indeterminates) from X and coefficients from $\mathbb{N}$, with the operations defined as usual*[4]*: $(\mathbb{N}[X], +, \cdot, 0, 1)$.*

Example 7. *Start again from the relation R in Figure 4.5(a) in which tuples are tagged with their own id. R can be seen as an $\mathbb{N}[p, r, s]$-relation. Applying to R the query q from Section 4.1 and doing the calculations in the provenance semiring we obtain the $\mathbb{N}[p, r, s]$-relation shown in Figure 4.5(c). The provenance of (f, e) is $2s^2 + rs$ which can be "read" as follows: (f, e) is computed by q in three different ways; two of them use the input tuple s* twice*; the third uses input tuples r and s. We also see that the provenance of (d, e) is different and we see* how *it is different!* □

The following standard property of polynomials captures the intuition that $\mathbb{N}[X]$ is as "general" as any semiring:

Proposition 4.3.1. *Let K be a commutative semiring and X a set of variables. For any valuation $v : X \to K$ there exists a unique homomorphism of semirings*

$$\textsf{\textit{Eval}}_v : \mathbb{N}[X] \to K$$

such that for the one-variable monomials we have $\textsf{\textit{Eval}}_v(x) = v(x)$.

As the notation suggests, $\textsf{Eval}_v(P)$ evaluates the polynomial P in K given a valuation for its variables. In calculations with the integer coefficients, na where

[4] These are polynomials in commutative variables so their operations are the same as in middle-school algebra, except that subtraction is not allowed.

$n \in \mathbb{N}$ and $a \in K$ is the sum in K of n copies of a. Note that $\mathbb{N}$ is embedded in K by mapping n to the sum of n copies of 1_K.

Using the Eval notation, for any $P \in \mathbb{N}[x_1, \ldots, x_n]$ and any K the **polynomial function** $f_P : K^n \to K$ is given by:

$$f_P(a_1, \ldots, a_n) \triangleq \mathsf{Eval}_v(P) \quad v(x_i) = a_i, i = 1..n$$

Putting together Propositions 4.2.3 and 4.3.1 we obtain Theorem 4.3.1 below, a conceptually important fact that says, informally, that the semantics of $\mathcal{RA}^+$ on K-relations for any semiring K **factors** through the semantics of the same in provenance semirings.

Indeed, let K be a commutative semiring, let R be a K-relation, and let X be the set of tuple ids of the tuples in $\mathsf{supp}(R)$. There is an obvious valuation $v : X \to K$ that associates to a tuple id the tag of that tuple in R.

We associate to R an "abstractly tagged" version, denoted $\bar{R}$, which is an $X \cup \{0\}$-relation. $\bar{R}$ is such that $\mathsf{supp}(\bar{R}) = \mathsf{supp}(R)$ and the tuples in $\mathsf{supp}(\bar{R})$ are tagged by their own tuple id. For example, in Figure 4.7(d) we show an abstractly-tagged version of the relation in Figure 4.7(b). Note that as an $X \cup \{0\}$-relation, $\bar{R}$ is a particular kind of $\mathbb{N}[X]$-relation.

To simplify notation we state the theorem for queries of one argument (but the generalization is immediate):

Theorem 4.3.1. *For any $\mathcal{RA}^+$ query q we have*

$$q(R) \;=\; \mathsf{Eval}_v \circ q(\bar{R})$$

To illustrate an instance of this theorem, consider the provenance polynomial $2r^2 + rs$ of the tuple (d, e) in Figure 4.5(c). Evaluating it in $\mathbb{N}$ for $p = 2, r = 5, s = 1$ we get 55 which is indeed the multiplicity of (d, e).

4.3.1 Recording mapping names in provenance

In a CDSS, apart from the base tuples we also need to record the mappings that were involved in a derivation. This can be achieved by extending semirings with a set of unary functions, one for each mapping (henceforth called mapping functions).

Definition 4.3.2 ($\mathcal{M}$-semiring). *Let $\mathcal{M} = \{m_1, m_2, \ldots, m_n\}$ be a set of mapping names. An $\mathcal{M}$-semiring is an algebraic structure $(K, +, \cdot, 0, 1, \mathcal{M})$ such that $(K, +, \cdot, 0, 1)$ is a semiring and $\forall m \in \mathcal{M}, m : K \to K$ is a unary function such that:*

- $\forall m \in \mathcal{M} \quad m(0) = 0$
- $\forall m \in \mathcal{M} \quad \forall a, b \in K \; m(a + b) = m(a) + m(b)$

The first property is essential in order to guarantee the finite support of relations (otherwise, the mappings could be applied to the infinitely many non-existent tuples of a K-relation, i.e., the ones annotated with 0 ...). The second property is justified by commutativity of union with mapping application.

We write $(\mathbb{N}, \mathcal{M})[X]$ for the $\mathcal{M}$-semiring of "polynomials" with coefficients in $\mathbb{N}$, with monomials such as: $m_1(a \cdot b), m_1(a \cdot m_2(c^2 \cdot d)), 2 \cdot m_1(c \cdot d) \cdot (m_3(e))^3$ etc. We can show that Propositions 4.2.3 and 4.3.1 can be extended for $\mathcal{M}$-semirings. In particular:

Proposition 4.3.2. *Let $h : K \to K'$ and assume that K, K' are commutative $\mathcal{M}$ and $\mathcal{M}'$-semirings, respectively. The transformation given by h from K-relations to K'-relations commutes with any $\mathcal{RA}^+$ query (for queries of one argument) $q(h(R)) = h(q(R))$ if and only if h is an $\mathcal{M}$-semiring homomorphism.*

Proposition 4.3.3. *Let K be a commutative $\mathcal{M}'$-semiring and X a set of variables. For any valuation $v : X \to K$ there exists a unique homomorphism of $\mathcal{M}$-semirings*

$$\textsf{\textit{Eval}}_{v,\mathcal{M}} : (\mathbb{N}, \mathcal{M})[X] \to K$$

$$Q(x,y) \text{ :- } R(x,z), R(z,y)$$

(a)

$a\ a$	2
$a\ b$	3
$b\ b$	4

(b)

$a\ a$	$2 \cdot 2 = 4$
$a\ b$	$2 \cdot 3 + 3 \cdot 4 = 18$
$b\ b$	$4 \cdot 4 = 16$

(c)

Figure 4.6: Datalog with bag semantics

such that for the one-variable monomials we have $\textit{\textsf{Eval}}_{v,\mathcal{M}}(x) = v(x)$*.*

As a result, Theorem 4.3.1 can be extended for $\mathcal{M}$-semirings:

Theorem 4.3.2. *For any $\mathcal{RA}^+$ query q we have*

$$q(R) \;=\; \textit{\textsf{Eval}}_{v,\mathcal{M}} \circ q(\bar{R})$$

4.4 Datalog on K-Relations

We now seek to give semantics on K-relations to datalog queries. It is more convenient to use the unnamed perspective [3] here, i.e., we ignore the names of the attributes of each relation and view its contents as ordered n-tuples, where n is the arity of the relation. We also consider only "pure" datalog rules in which all subgoals are relational atoms. First observe that for conjunctive queries over K-relations the semantics in Definition 4.2.2 simplifies to computing tags as sums of products, each product corresponding to a valuation of the query variables that makes the query body hold. For example, consider the conjunctive query and $\mathbb{N}$-relation shown in Figure 4.6(a) and (b), respectively.

There are two valuations that produce the answer $Q(a,b)$: $\{x = a, y = a, z = b\}$ yields the body $R(a,a), R(a,b)$ while $\{x = a, y = b, z = b\}$ yields the body $R(a,b), R(b,b)$. The sum of products of tags is $2 \cdot 3 + 3 \cdot 4$ which is exactly what the equivalent $\mathcal{RA}^+$ query yields according to Definition 4.2.2. If we think of this conjunctive query as a datalog program, the two valuations above correspond to the two *derivation trees* of the tuple $Q(a, b)$.

This suggests the following generalized semantics for datalog on K-relations: the tag of an answer tuple is the sum over all its derivation trees of the product of the tags of the leaves of each tree. Indeed, this generalizes the bag semantics of datalog considered in [85, 86] *when the number of derivation trees is finite.* In general, a tuple can have infinitely many derivation trees (an algorithm for detecting this appears in [87]) hence we need to work with semirings in which infinite sums are defined.

Closed semirings [96] have infinite sums but their "$+$" is idempotent which rules out the bag and provenance semantics. We will adopt the approach used in formal languages [75] and later show further connections with how semirings and formal power series are used for context-free languages. By assuming that $\mathbb{D}$ is countable, it will suffice to define countable sums.

Let $(K, +, \cdot, 0, 1)$ be a semiring and define $a \leq b \overset{\text{def}}{\Leftrightarrow} \exists x\ a + x = b$. When $\leq$ is a partial order we say that K is **naturally ordered**. $\mathbb{B}$, $\mathbb{N}$, $\mathbb{N}[X]$ and the other semiring examples we gave so far are all naturally ordered.

We say that K is an ω**-complete** semiring if it is naturally ordered and $\leq$ is such that ω-chains $x_0 \leq x_1 \leq \cdots \leq x_n \leq \cdots$ have least upper bounds. In such semirings we can define *countable sums*:

$$\sum_{n \in \mathbb{N}} a_n \ \triangleq\ \sup_{m \in \mathbb{N}} (\sum_{i=0}^{m} a_i)$$

Note that if $\exists N$ s.t. $\forall n > N, a_n = 0$ then $\sum_{n \in \mathbb{N}} a_n \ = \ \sum_{i=0}^{N} a_i$. All the semiring examples we gave so far are ω-complete with the exception of $\mathbb{N}$ and $\mathbb{N}[X]$.

An ω**-continuous** semiring is an ω-complete semiring in which the operations $+$ and $\cdot$ are ω-continuous in each argument. It follows that countable sums are associative and commutative, that $\cdot$ distributes over countable sums and that countable sums are monotone in each addend.

Semiring	Annotation	Fin.dist.lattice?
$(\mathbb{B}, \vee, \wedge, \mathsf{false}, \mathsf{true})$	Set semantics, boolean trust	Yes
$(\mathbb{N}^{\infty}, +, \cdot, 0, 1)$	Bag semantics	No
$(\mathsf{PosBool}(B), \vee, \wedge, \mathsf{false}, \mathsf{true})$	c-tables	Yes (with B finite)
$(\mathcal{P}(\Omega), \cup, \cap, \emptyset, \Omega)$	Probabilistic event tables	Yes (with Ω finite)
$(\mathcal{C}, \min, \max, 0, P)$	Confidentiality policies [50]	Yes
$(\mathcal{P}(X), \cup, \cup, \emptyset, \emptyset)$	Lineage	Yes (with X finite)
$(\mathbb{N}^{\infty}, \min, +, \infty, 0)$	tropical semiring [75]	No

Table 4.2: Examples of ω-continuous semirings

We show some interesting examples of ω-continuous semirings in Table 4.2. In this table, the first column shows the formal definition of the semiring, the second contains kinds of annotations it can be used to represent and the last column indicates whether the corresponding semiring is also a *finite distributive lattice* [34] and the natural order of the semiring is the lattice order. For bag semantics and the tropical semiring, we add ∞ to the natural numbers and define $\infty + n = n + \infty = \infty$ and $\infty \cdot n = n \cdot \infty = \infty$ except for $\infty \cdot 0 = 0 \cdot \infty = 0$. We can think of $\mathbb{N}^{\infty}$ as the ω-continuous "completion" of $\mathbb{N}$. For the semiring of confidentiality policies, the total order $\mathcal{C}$: $P < C < S < T < 0$ describes the following levels of "clearance": P = public, C = confidential, S = secret, and T = top-secret.

Definition 4.4.1. *Let $(K, +, \cdot, 0, 1)$ be a commutative ω-continuous semiring. To keep notation simple let q be a datalog query with one argument (it is easy to generalize to multiple arguments). For any K-relation R define*

$$q(R)(t) = \sum_{\tau \textit{ yields } t} \Big(\prod_{t' \in \textit{leaves}(\tau)} R(t') \Big)$$

where τ ranges over all q-derivation trees for t and t' ranges over all the leaves of τ.

The next result shows that Definition 4.4.1 does indeed give us a semantics for datalog queries on K-relations.

Proposition 4.4.1. *For any K-relation R, $q(R)$ has finite support and is therefore a K-relation.*

Proof. (sketch) Let S be the set of tuples t s.t. $R(t) \neq 0$ and let t' be a tuple s.t. $q(R)(t') \neq 0$. By Definition 4.4.1, this implies that there is a derivation tree for t (s.t. the tags of the tuples in its leaves are non-zero and correspond to this product) i.e., $t' \in q(S)$. Since $q(S)$ is finite, $q(R)$ has finite support. □

As an example, consider the datalog program q with output predicate Q defined by the rules shown in Figure 4.7(c), applied on the $\mathbb{N}$-relation R shown in Figure 4.7(a). Since any $\mathbb{N}$-relation is also a $\mathbb{N}^\infty$-relation and $\mathbb{N}^\infty$ is ω-continuous we can answer this query[5] and we obtain the table shown in Figure 4.7(b).

A couple of sanity checks follow.

Proposition 4.4.2. *Let q be an $\mathcal{RA}^+$ query in which the selection predicates only test for attribute equality and let q' be the (non-recursive) datalog query obtained by standard translation from q. Then q and q' produce the same answer when applied to the same instance of a database of K-relations.*

Proposition 4.4.3. *For any datalog query q and any $\mathbb{B}$-relation R, $\mathsf{supp}(q(R))$ is the same as the result of applying q to the standard relation $\mathsf{supp}(R)$.*

4.4.1 Provenance equations

The definition of datalog semantics given above is not so useful computationally. However, we can think of it as the proof-theoretic definition, and as with standard datalog, it turns out that there is an equivalent fixpoint-theoretic definition that is much more workable.

Intuitively, this involves representing the possibly infinite sum of products above as a system of fixpoint equations that reflect all the ways that a tuple can be

[5] This is transitive closure with bag semantics.

$$Q(x,y) \text{ :- } R(x,y)$$
$$Q(x,y) \text{ :- } Q(x,z), Q(z,y)$$

(a)

$a\ b$	2
$a\ c$	3
$c\ b$	2
$b\ d$	1
$d\ d$	1

(b)

$a\ b$	8
$a\ c$	3
$c\ b$	2
$b\ d$	∞
$d\ d$	∞
$a\ d$	∞
$c\ d$	∞

(c)

$a\ b$	m
$a\ c$	n
$c\ b$	p
$b\ d$	r
$d\ d$	s

(d)

$a\ b$	$\mathbf{x}$
$a\ c$	$\mathbf{y}$
$c\ b$	$\mathbf{z}$
$b\ d$	$\mathbf{u}$
$d\ d$	$\mathbf{v}$
$a\ d$	$\mathbf{w}$
$c\ d$	$\mathbf{t}$

(e)

$$\begin{aligned}
\mathbf{x} &= m + \mathbf{yz}\\
\mathbf{y} &= n\\
\mathbf{z} &= p\\
\mathbf{u} &= r + \mathbf{uv}\\
\mathbf{v} &= s + \mathbf{v}^2\\
\mathbf{w} &= \mathbf{xu} + \mathbf{wv}\\
\mathbf{t} &= \mathbf{zu} + \mathbf{tv}
\end{aligned}$$

(f)

$$\begin{aligned}
\mathbf{x} &= m_1(m) + m_2(\mathbf{yz})\\
\mathbf{y} &= m_1(n)\\
\mathbf{z} &= m_1(p)\\
\mathbf{u} &= m_1(r) + m_2(\mathbf{uv})\\
\mathbf{v} &= m_1(s) + m_2(\mathbf{v}^2)\\
\mathbf{w} &= m_2(\mathbf{xu}) + m_2(\mathbf{wv})\\
\mathbf{t} &= m_2(\mathbf{zu}) + m_2(\mathbf{tv})
\end{aligned}$$

(g)

Figure 4.7: Datalog example

produced as a result of applying the *immediate consequence operator* T_q (for a datalog query q) on other tuples. Since such an immediate consequence can involve other tuples in idb relations, that may themselves have infinitely many derivations, we introduce a new variable for each tuple in the idb relation and use that variable to refer to that tuple when calculating its immediate consequences. Thus, for every tuple there is an equation between the variable for that tuple and a polynomial over all the variables.

To make this precise we consider **polynomials** with coefficients in an arbitrary commutative semiring K. If the set of variables is X we denote the set of polynomials by $K[X]$. We have already used $\mathbb{N}[X]$ for provenance but $K[X]$ also forms a commutative semiring. We saw in Section 4.3 that because $\mathbb{N}$ can be embedded in any semiring K the polynomials in $\mathbb{N}[X]$ define polynomial functions over K. Similarly, if $X = \{x_1, \ldots, x_n\}$ then any polynomial $P \in K[X]$ defines a polynomial function $f_P : K^n \to K$. Most importantly, if K is ω-continuous then f_P is

ω-continuous in each argument.

Definition 4.4.2. *Let K be a commutative ω-continuous semiring. An* ***algebraic system*** *over K with variables $X = \{x_1, \ldots, x_n\}$ consists of a list of polynomials $P_1, \ldots, P_n \in K[X]$ and is written*

$$\begin{array}{rcl} x_1 & = & P_1(x_1, \ldots, x_n) \\ & \cdots & \\ x_n & = & P_n(x_1, \ldots, x_n) \end{array}$$

Together, $f_{P_1}, \ldots, f_{P_n}$ define a function $f_{\boldsymbol{P}} : K^n \to K^n$. K^n has a component-wise commutative ω-continuous semiring structure such that $f_{\mathbf{P}}$ is ω-continuous. Hence, the least fixed point

$$\mathrm{lfp}(f_{\boldsymbol{P}}) \;=\; \sup_{m \in \mathbb{N}} f_{\boldsymbol{P}}^m(0, \ldots, 0)$$

exists, and we call it the ***solution*** *of the algebraic system above.*

As an example, consider the one-variable equation $\mathbf{x} = a\mathbf{x} + b$ with $a, b \in K$. This is closely related to regular language theory and its solution is $\mathbf{x} = a^*b$ where

$$a^* \;\triangleq\; 1 + a + a^2 + a^3 + \cdots$$

For example, in $\mathbb{N}^\infty$ we have $1^* = \infty$ while in $\mathsf{PosBool}(B)$ we have $e^* = \mathsf{true}$ for any e.

Consider a datalog program q and to simplify notation assume just one edb predicate R and one idb-and-output predicate Q. Given an edb K-relation of finite support R we can effectively construct an algebraic system over K as follows. Denote by Q also the K-relation that is the output of the program and let $\bar{Q}$ be the abstractly-tagged (as in Theorem 4.3.1) version of Q where X is the set of ids of the tuples in $\mathsf{supp}(Q)$. Since $\bar{Q}$ is a $X \cup \{0\}$-relation and R is a K-relation both can be seen also as $K[X]$-relations. The immediate consequence operator T_q is in fact a union of conjunctive queries, hence Definition 4.2.2 shows how to calculate effectively $T_q(R, \bar{Q})$ as a $K[X]$-relation of finite support. By equating the tags of $\bar{Q}$ with those of $T_q(R, \bar{Q})$ we obtain the promised algebraic system. We will denote this system as $\bar{Q} = T_q(R, \bar{Q})$ (although it only involves the tags of these relations).

Theorem 4.4.1. *With the notation above, for any tuple t, the tag $Q(t)$ given by Definition 4.4.1, when not 0, equals the component of the solution (Definition 4.4.2) of the algebraic system $\bar{Q} = T_q(R, \bar{Q})$ corresponding to the id of t.*

To illustrate with an example, consider again the datalog program in Figure 4.7(a) applied to the same $\mathbb{N}$-relation, R shown in Figure 4.7(b). In Figure 4.7(e)we have the abstractly-tagged version of the output relation, $\bar{Q}$ in which the tuples are tagged with their own ids. The corresponding algebraic system is the one obtained from Figure 4.7(f)by replacing $m = 2, n = 3, p = 2, r = 1, s = 1$. (Note that $T_q(R, \bar{Q}) = R \cup \bar{Q} \bowtie \bar{Q}$.) Calculating its solution we get after two fixed point iterations $\mathbf{x} = 8, \mathbf{y} = 3, \mathbf{z} = 2, \mathbf{u} = 2, \mathbf{v} = 2, \mathbf{w} = 2$. In further iterations $\mathbf{x}, \mathbf{y}, \mathbf{z}$ remain the same while $\mathbf{u}, \mathbf{v}, \mathbf{w}$ grow unboundedly (in Section 4.7 we show how unbounded growth can be detected). Hence the solution is the one shown in Figure 4.7(c).

Note that semiring homomorphisms are monotone with respect to the natural order. However, to work well with the datalog semantics more is needed.

Proposition 4.4.4. *Let K, K' be commutative ω-continuous semirings and let $h : K \to K'$ be an ω-continuous semiring homomorphism. Then, the transformation given by h from K-relations to K'-relations commutes with any datalog query (for queries of one argument $q(h(R)) = h(q(R))$).*

4.5 Formal Power Series for Provenance

In Section 4.3 we showed how to use $\mathbb{N}[X]$-relations to capture an expressive notion of provenance for the tuples in the output of an $\mathcal{RA}^+$ query. However, polynomials will not suffice for the provenance of tuples in the output of datalog queries because the semiring $\mathbb{N}[X]$ does not define infinite sums. As with the transition from $\mathbb{N}$ to $\mathbb{N}^{\infty}$ we wish to "complete" $\mathbb{N}[X]$ to a commutative ω-continuous semiring. This problem has been tackled in formal language theory and it led to the study of *formal power series* [75].

Note that when we try to apply naively Definition 4.4.1 to datalog queries on $\mathbb{N}[X]$-relations we encounter two kinds of infinite summations. First, it is possible that we have to sum infinitely many *distinct* monomials. This leads directly to formal power series. Second, it is possible that we have to sum infinitely many copies of the *same* monomial. This means that we need coefficients from $\mathbb{N}^{\infty}$, not just $\mathbb{N}$.

Let X be a set of variables. Denote by $X^{\oplus}$ the set of all possible monomials over X. For example, if $X = \{x, y\}$ then $X^{\oplus} = \{x^m y^n \mid m, n \geq 0\} = \{\epsilon, x, y, x^2, xy, y^2, x^3, x^2y, \ldots\}$ where ϵ is the monomial in which both x and y have exponent 0.

Let K be a commutative semiring. A **formal power series** with variables from X and coefficients from K is a mapping that associates to each monomial in $X^{\oplus}$ a coefficient in K. A formal power series S is traditionally written as a possibly infinite sum

$$S \;=\; \sum_{\mu \in X^{\oplus}} S(\mu)\, \mu$$

and we denote the set of formal power series by $K[[X]]$. As with $K[X]$, there is a commutative semiring structure on $K[[X]]$ given by the usual way of adding and multiplying, for example

$$(S_1 \cdot S_2)(\mu) = \sum_{\mu_1 \mu_2 = \mu} S_1(\mu_1) \cdot S_2(\mu_2)$$

But the real reason we use formal power series is the fact that if K is ω-continuous then $K[[X]]$ *is also* ω-continuous (see [75], for example).

Definition 4.5.1. *Let X be the set of tuple ids of a database instance I. The* ***datalog provenance semiring*** *for I is the commutative ω-continuous semiring of formal power* *series* $\mathbb{N}^\infty[[X]]$.

Let us calculate, using the fixed point semantics, the provenances for the output of the datalog query in Figure 4.7(a). We now take as input the relation, call it $\bar{R}$, in Figure 4.7(d) which is the abstractly-tagged (tagged with tuple ids) version of the relation R in Figure 4.7(b). Note that we have *two sets of variables* here. The tuple ids of $\bar{R}$ form one set of variables and the provenance semiring in which we compute is $\mathbb{N}^\infty[[m, n, p, r, s]]$. At the same time, the ids of the tuples in $\bar{Q}$ in Figure 4.7(e) are used as variables in the algebraic system, whose right-hand sides belong to

$$\mathbb{N}^\infty[[m, n, p, r, s]][\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{u}, \mathbf{v}, \mathbf{w}]$$

i.e., they are polynomials in the variables $\{\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{u}, \mathbf{v}, \mathbf{w}\}$, with coefficients in the semiring of formal power series $\mathbb{N}^\infty[[m, n, p, r, s]]$. The **v** component of the solution can be calculated separately:[6]

$$\mathbf{v} = s + s^2 + 2s^3 + 5s^4 + 14s^5 + \cdots$$

Also, one can see that $\mathbf{x} = m + np$, $\mathbf{u} = r\mathbf{v}^*$, $\mathbf{w} = r(m + np)(\mathbf{v}^*)^2$. For example the coefficient of $rnps^3$ in the provenance **w** of $Q(a, d)$ is 5, which means this tuple can be obtained in 5 distinct ways using $R(a, c)$, $R(c, b)$ and $R(b, d)$ once and $R(d, d)$ three times.

Algebra provenance, $\mathbb{N}[X]$, is embedded in datalog provenance, $\mathbb{N}^\infty[X]$, by regarding polynomials as formal power series in which all but finitely many coefficients are 0. Here is the corresponding sanity check:

Proposition 4.5.1. *Let q be an $\mathcal{RA}^+$ query (of one argument, to simplify notation) in which the selection predicates only test for attribute equality, let q' be the (non-recursive) datalog query obtained by standard translation from q and let R be a $\mathbb{N}[X]$-relation. Modulo the embedding of $\mathbb{N}[X]$ in $\mathbb{N}^\infty[X]$ we have $q'(R) = q(R)$*

[6] [29] shows that the coefficient of s^{n+1} is $\frac{2n!}{n!(n+1)!}$.

Formal power series can be evaluated in commutative ω-continuous semirings:

Proposition 4.5.2. *Let K be a commutative ω-continuous semiring and X a set of variables. For any valuation $v : X \to K$ there exists a unique ω-continuous homomorphism of semirings*

$$\textsf{\textit{Eval}}_v : \mathbb{N}^\infty[[X]] \to K$$

such that for the one-variable monomials we have $\textsf{\textit{Eval}}_v(x) = v(x)$.

Therefore, just like polynomials, formal power series define **series functions** on any commutative ω-continuous semiring. Finally, we have the analog of Theorem 4.3.1.

Theorem 4.5.1. *The semantics of datalog on K-relations for any commutative ω-continuous semiring K **factors** through the semantics of the same in provenance semirings (of formal power series).*

Although the coefficients in the provenance series may be ∞, we can characterize exactly when this happens[7]:

Theorem 4.5.2. *A datalog query q has provenance series in $\mathbb{N}[[X]]$ for some tuple t if and only if the instantiation of q has no cycle of* unit rules *(rules whose body consists of a single idb) s.t. t is part of the cycle (i.e., appears on the head of one of those unit rules and the body of another) and t is in the result of q.*

We can extend these results to include mapping functions as follows: in the system of equations above, instead of polynomials in $K[X]$ the bodies of the equations are expressions over $(K, \mathcal{M})[X]$, corresponding to the immediate consequence operator "annotated" by the mapping used. Thus, if we call m_1 and m_2 the two

[7] In this theorem, the *instantiation* of a datalog query is the set of rules obtained by considering all satisfying valuations for the variables in rules of q.

rules of the datalog program in Figure 4.7(a), the corresponding system of equations with the mapping functions is shown in Figure 4.7(g). To ensure that the least fixed point of these equations exists, we need $\mathcal{M}$-semiring to also be ω-continuous:

Definition 4.5.2. *A $\mathcal{M}$-semiring $(K, +, \cdot, 0, 1, \mathcal{M})$ is ω-**continuous** if $(K, +, \cdot, 0, 1)$ is ω-continuous and*

$$\forall m \in \mathcal{M} \quad m(a_1 + a_2 + \dots) = m(a_1) + m(a_2) + \dots$$

In the following, abusing notation, we use $\mathcal{M}$ to refer both to the name of the set of mappings and the corresponding set of unary functions. The **mappings provenance $\mathcal{M}$-semiring** is the ($\mathcal{M}$-)semiring of formal power series with mapping functions $\mathcal{M}$. We can show that factorization also works for $\mathcal{M}$-semirings, as long as they are ω-continuous:

Theorem 4.5.3. *The semantics of applying a set $\mathcal{M}$ of mappings on K-relations for any commutative ω-continuous $\mathcal{M}'$-semiring K **factors** through the semantics of the same in the mappings provenance $\mathcal{M}$-semiring of $\mathcal{M}$.*

4.6 Provenance Graphs

In Section 4.4.1 we explained that, even if the provenances of some tuples are infinite, we can represent all provenance information through a finite system of equations. We can alternatively look at these equations as forming a graph. We will find graphs useful for encoding the provenance of data — as it is propagated during update exchange in a CDSS — in relations (Chapter 5) and, ultimately, as the data model for which we define our provenance query language (Chapter 7).

Definition 4.6.1 (Provenance Graph). *The graph has two types of nodes:* tuple nodes, *one for each tuple in the system, and* mapping nodes. *Each mapping node corresponds to*

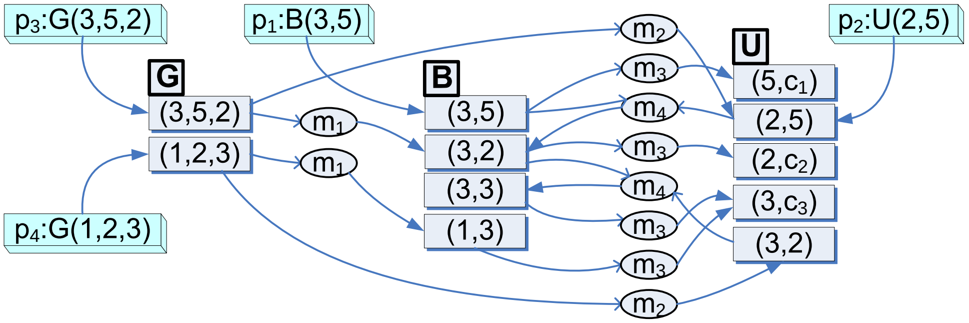


Figure 4.8: Provenance graph of update exchange in Example 5

an instantiation of a mapping, and is labeled by the name of the mapping. This instantiation defines some tuples that match the LHS of the mapping and we draw edges from the corresponding tuple nodes to the mapping node (this encodes conjunction among tuples). It also defines some tuples that match the RHS of the mapping and we draw edges from the mapping node to the tuple nodes that correspond to them. Multiple incoming edges to a tuple node encode the fact that a tuple may be derived multiple ways. Finally, some nodes in the graph have no incoming edges. Such base tuples are the result of direct user insertions. We call these leaf *nodes and in the provenance semiring they are annotated with globally unique* ***provenance tokens****, which correspond to the indeterminates of those semirings. In the graph representation we represent these tokens as extra labels on the tuple nodes.*

One can generate provenance expressions, that correspond to solutions of the system of equations, from the provenance graph, by traversing it recursively backwards along the arcs as in Example 8. We write $\mathsf{P}v(R(t))$ for the provenance expression of the tuple t in relation R. When the relation name is unimportant, we simply write $\mathsf{P}v(t)$. We sometimes omit $\cdot$ and use concatenation.

Example 8. *Consider the update exchange from Example 5. The provenance graph for all tuples in the canonical universal solution is shown in Figure 4.8, with tuple nodes shown as rectangles below, and mapping nodes shown as ellipses. Source tuples are annotated by unique provenance tokens $p_1, p_2, \ldots$. From this graph we can analyze the provenance* *of, say,* $B(3,2)$ *by tracing back paths to source data nodes — in this case through* (m_4) *to* p_1 *and* p_2 *and through* (m_1) *to* p_3*. In particular we can compute that* $\mathsf{P}v(U(2,5)) = p_2 + m_2(p_3)$ *and* $\mathsf{P}v(B(3,2)) = m_1(p_3) + m_4(\mathsf{P}v(B(3,5))\mathsf{P}v(U(2,5))) = m_1(p_3) + m_4(p_1p_2) + m_4(p_1m_2(p_3))$

Observe that, even for cases in which all provenance annotations are finite, the provenance graph provides a compact way of representing these annotations: even if a derived tuple is part of the provenance of multiple other tuples, it is represented by a single node in the graph. For example, $B(3,2)$ is involved in deriving both $U(2,c_2)$ and $B(3,3)$, so the provenance of both those tuples contains $\mathsf{P}v(B(3,2))$ or any of its subexpressions (e.g. $m_1(p_3)$). Such common provenance subexpressions are ubiquitous as updates are propagated through paths of mappings in CDSS update exchange, and as a result the overhead of storing them redundantly for each tuple can be prohibitive. Moreover, representing derived tuples in the provenance graph has an additional benefit: by traversing the graph, we can analyze the provenance of a derived tuple not only with respect to base tuples, but also with respect to other derived tuples and mappings between them. Indeed, many users may be interested in identifying such relationships between tuples or in focusing on parts of the graph that are of interest to them, while hiding the complexity of the complete provenance graph. For instance, some peer $\mathbf{P}_1$ may trust the curation methods of another peer $\mathbf{P}_2$ sufficiently that they want to consider all data that they receive from them as $\mathbf{P}_2$'s local data, regardless of where they were inserted locally at $\mathbf{P}_2$ or imported from other peers. In this case, they would like to project out the parts of derivations leading to some tuples being imported in $\mathbf{P}_2$ and only focus on the remaining parts between $\mathbf{P}_2$ and themselves. However, even for fairly small CDSS settings, with a few peers and mappings, the provenance graph can be very large and complex, tasks involving exploration of this graph can be very arduous. For this reason, in Chapter 7 we define these tasks more precisely and propose a query language that operates on provenance graphs and allows users to navigate and extract information from such graphs.

4.7 Computing Provenance Series

We show here that several natural questions that one can ask about the computability of formal power series in $\mathbb{N}^\infty[[X]]$ can in fact be decided and all finitely representable information can in fact be computed. To simplify presentation, in this section we consider semirings without mapping functions, but the results extend to the case of $\mathcal{M}$-semirings.

Given a datalog program q and a relational instance I, consider the formal power series provenance of some tuple t in the output $q(I)$, i.e., $q(I)(t)$ where the datalog semantics is taken in $\mathbb{N}^\infty[[X]]$ (X is the set of ids of the tuples in I). We show that it is decidable whether $q(I)(t)$ is in fact a polynomial in $N[X]$. The algorithm *All-Trees*, shown in Figure 4.9 (inspired by [87]) decides this for all output tuples and computes the polynomial when the answer is affirmative.

For an output tuple t for which the answer given by algorithm *All-Trees* is negative, we can use Theorem 4.5.2 to decide whether $q(I)(t)$ is in $\mathbb{N}[[X]]$. The remaining question is whether $q(I)(t)$ is in $\mathbb{N}^\infty[X]$, which we can decide by checking if there is any cycle in the instantiation of the query involving at least one non-unit rule, s.t. t is part of that cycle; otherwise, $q(I)(t)$ is in $\mathbb{N}^\infty[[X]]$. In algorithm *All-Trees* shown in Figure 4.9:

- **T** is the set of derivation trees computed thus far; T^∞ is the set of tuples with infinitely many derivations; fringe(τ) is the *bag* of labels of leaves of the tree τ.
- $T_q^\nu(\mathbf{T}, T^\infty) = \{\tau \mid \tau \notin \mathbf{T} \ \wedge \ \tau \in T_q(\mathbf{T}) \ \wedge \ \text{root}(\tau) \notin T^\infty\}$, where $T_q(\mathbf{T})$ is the set of trees produced by applying a rule on tuples in $\{\text{root}(\tau) \mid \tau \in \mathbf{T}\} \cup T^\infty$.

Algorithm *All-Trees*
Input: query q, instance I
Output: the power series $P(t)$ for every tuple $t \in q(I)$

```
Initialize $\mathbf{T} \leftarrow \{t() : t \in I\}$
Initialize $T^{\infty} \leftarrow \emptyset$
repeat
    $\mathbf{T}^{\nu} \leftarrow T^{\nu}_{q}(\mathbf{T}, T^{\infty})$
    for every tree $\tau \in \mathbf{T}^{\nu}$
        do if any child of root($\tau$) is in $T^{\infty}$ or any proper descendant of root($\tau$)
              to a node associated with the same tuple
            then $T^{\infty} \leftarrow T^{\infty} \cup \{\mathrm{root}(\tau)\}$
            else $\mathbf{T} \leftarrow \mathbf{T} \cup \{\tau\}$
until nothing added to either T or $T^{\infty}$ in last iteration
for every $t \in q(I)$
    do if $t \in T^{\infty}$
        then $P(t) \leftarrow \infty$
        else $P(t) \leftarrow \sum_{\substack{\tau \in \mathbf{T}: \\ \mathrm{root}(\tau)=t}} \big( \prod_{l \in \mathrm{fringe}(\tau)} l \big)$
return P
```

Figure 4.9: Algorithm *All-Trees*

Algorithm *All-Trees* terminates because at every iteration only trees which are not there already are produced and moreover, for every tuple that has infinitely many derivations, as soon as it is identified and inserted in T^{∞}, no more trees for it are produced. Note also that by Theorem 4.5.1 this algorithm will also give us, in particular, an algorithm for evaluating datalog queries on bag semantics, just like in [87].

If the answer of algorithm *All-Trees* for an output tuple t is negative, we can also use algorithm *Monomial-Coefficient*, shown in Figure 4.10, to compute the coefficient of a particular monomial μ in $q(I)(t)$, even when that coefficient is ∞. In this algorithm:

- M is the set of tuples whose ids appear in μ, a monomial represented as a bag of labels that appear in it; P^{∞} is a set of pairs (t, μ), representing tuples t

```
**Algorithm** *Monomial-Coefficient*
**Input:** query $q$, monomial $\mu$, tuple $t$
**Output:** $C$, the coefficient of $\mu$ in the power series $P(t)$
Initialize $\mathbf{T} \leftarrow \{t() : t \in M\}$
Initialize $P^\infty \leftarrow \emptyset$
**repeat**
    $\mathbf{T}^i \leftarrow T_q^i(\mathbf{T}, T^\infty)$
    **for** every tree $\tau \in \mathbf{T}^i$
        **do if** for any child tree $\tau'$ of root$(\tau)$, $(\text{root}(\tau'), \text{fringe}(\tau')) \in P^\infty$ **or** there
            is a chain from root$(\tau)$ to a node associated with the same tuple
              **then** $P^\infty \leftarrow P^\infty \cup \{(\text{root}(\tau), \text{fringe}(\tau))\}$
              **else** $\mathbf{T} \leftarrow \mathbf{T} \cup \{\tau\}$
**until** nothing added to either $\mathbf{T}$ or $T^\infty$ in last iteration
**if** $(t, \mu) \in P^\infty$
   **then** $C \leftarrow \infty$
   **else** **for** every $\tau \in \mathbf{T}$ s.t. root$(\tau) = t$ and fringe$(\tau) = \mu$
           **do** $C \leftarrow C + 1$
**return** $C$
```

Figure 4.10: Algorithm *Monomial-Coefficient*

for which infinite derivation trees whose leaves are equal to the monomial μ have been found.

- $T_q^i(\mathbf{T}, P^\infty, \mu) = \{\tau \mid \tau \notin \mathbf{T} \wedge \tau \in T_q(\mathbf{T}) \wedge \text{fringe}(\tau) \leq \mu \wedge (\text{root}(\tau), m) \notin P^\infty\}$, where $T_q(\mathbf{T})$ is the set of trees that can be produced by applying a rule on tuples in $\{\text{root}(\tau) \mid \tau \in \mathbf{T}\} \cup \{t \mid (t, m) \in P^\infty\}$ it, where the multiplicity of each is the corresponding exponent in the monomial.

If n is the length of the longest acyclic path of unit rules, then every $n+1$ iterations the algorithm *Monomial-Coefficient* either has produced a tree τ with larger fringe(τ) than the ones that were combined to produce it, or it has identified a pair (t', μ') whose derivation trees involve a cycle of unit rules. The algorithm then is guaranteed to terminate, because for all such tuples t' with infinitely many derivations, no trees for t' with fringe$(\tau) = \mu'$ are used in any subsequent derivations, and moreover, the set of trees τ s.t. τ does not involve nodes marked as infinite and fringe$(\tau) \leq \mu$ is finite.

4.8 Using Provenance to Compute Annotations

As we discussed earlier, our provenance model generalizes query answering on other forms of annotated relations and can be used to compute different kinds of annotations that can be expressed as semirings. Theorems 4.5.1, 4.5.3 suggest using algorithm *All-Trees* for computing annotations for answers of datalog queries. We already noted in Section 4.7 that this will work fine for $\mathbb{N}^\infty$. How about $\mathsf{PosBool}(B)$, $\mathcal{P}(\Omega)$, and, as a sanity check, $\mathbb{B}$? When algorithm *All-Trees* returns ∞, the evaluation on these semirings will return a normal value! We show that we can compute this value in the more general case when the semiring K is a *finite distributive lattice*. We can do so with some simple modifications to algorithm *All-Trees*:

- Redefine T_q^ν to take only $\mathbf{T}$ as a parameter, and return all τ in $T_q(\mathbf{T})$ such that for all τ' in $\mathbf{T}$, if root$(\tau) =$ root(τ'), then fringe$(\tau) <$ fringe(τ').

Thus a derivation tree for a tuple is considered "new" only when its associated monomial is smaller than any yet seen for that tuple. This modified algorithm always returns a polynomial for each tuple. Evaluating these polynomials in K gives the K-relation output.

The sanity check that for $K = \mathbb{B}$ the output tuples get the tag true is easy to check. For $K = \mathsf{PosBool}(B)$, after also checking that for any valuation $v : B \to \mathbb{B}$ we have $v(q(R)) = q(v(R))$, we get a **datalog on boolean c-tables semantics**. This is new for incomplete databases.

In probabilistic databases we restrict ourselves, as usual, to the case when the domain $\mathbb{D}$ is finite, hence the sample space Ω of all possible instances is finite.

$K = \mathcal{P}(\Omega)$ is a finite distributive lattice so we get an effective semantics for datalog on event tables. After checking that the resulting event tagging a tuple t does in fact say that t is in $q(R)$ for the random instance R, we conclude that our algorithm also generalizes that of [53].

Alternatively, for semirings that are *finite distributive lattices* we can use the provenance graph to compute different kinds of annotations, as suggested by Theorem 4.5.3. As we pointed out in Table 4.2, many interesting kinds of annotations can be represented by such semirings. Moreover, we can show that evaluation in the tropical semiring is also always computable and returns a normal value. The crux of the proof is that, even if there are infinitely many derivation trees for a tuple, the one with the $\min$ value is always among a (finite) set of *minimal* trees, since any tree whose set of leaves subsume the set of leaves of another tree also has value that is greater or equal to it.

In the next section we show an example of evaluation through provenance for a semiring for which this is always possible, namely one that represents boolean trust in a CDSS. In Chapter 7 we show that semiring evaluation can in fact be considered a core component of provenance querying, and we define clauses for specifying the semiring in which a provenance graph should be evaluated, as well as assignments of values for leaf nodes and mapping functions.

4.8.1 Semiring Evaluation Example: Assigning Trust to Provenance

In Section 3.1.2, we gave examples of *trust conditions* over provenance and data. Now that we have discussed our model for provenance as well as semiring evaluation through provenance, we can show how to determine trust assignments for tuples.

Consider a domain $\{\mathbf{T}, \mathbf{D}\}$ whose values represent **trust** and **distrust**. Consider also a second domain $\{\mathbf{Tm}, \mathbf{Dm}\}$ whose values represent **trust mapping, distrust mapping**. A participant will assign to each provenance token one of the values in $\{\mathbf{T}, \mathbf{D}\}$ and to each of the mappings one of the values in $\{\mathbf{Tm}, \mathbf{Dm}\}$. The intuition is that tuples produced by a mapping labeled **Dm** cannot be trusted, while **Tm** does not affect the trustworthiness of provenance tokens passing through it.

With such a trust assignment any finite provenance expression can be evaluated by interpreting multiplication akin to conjunction over booleans, i.e., $\mathbf{T} \cdot \mathbf{T} = \mathbf{T},\ \mathbf{T} \cdot \mathbf{D} = \mathbf{D}$ etc., and addition akin to disjunction over booleans, i.e., $\mathbf{T} + \mathbf{D} = \mathbf{T},\ \mathbf{D} + \mathbf{D} = \mathbf{D}$ etc. We also interpret the **Dm** mappings as the constantly **D** function and the **Tm** mappings as the identity on $\{\mathbf{T}, \mathbf{D}\}$.

In general the solutions of provenance equations may be infinite. Nonetheless the algorithm of Figure 4.11 evaluates them correctly with respect to trust assignments. We write $\mathsf{P}v(R(t))$ for the provenance expression of the tuple t in relation R. When the relation name is unimportant, we simply write $\mathsf{P}v(t)$. We describe how the algorithm works in terms of the provenance graph.

Trust assignments as in the previous algorithm are not the only ones possible. An equally useful assignment, involving the tropical semiring, associates "risk of accepting", to each provenance token, a positive real. Provenance multiplication is interpreted as risk addition and provenance addition as the $\min$ operation. The risk of tuple acceptance can then be calculated using a variant of Dijkstra's shortest paths algorithm, even when the provenance graph contains cycles. Peers can then compare this risk against a threshold in order to decide whether to accept the tuple.

The following example illustrates trust computations for the boolean and ranked trust models.

Example 9. *In Example 8 we calculated the provenances of some exchanged tuples, and found that the provenance of $B(3,2)$ is:*

$$\mathsf{P}v(B(3,2)) = m_1(p_3) + m_4(p_1 p_2) + m_4(p_1 m_2(p_3))$$

```
Algorithm ProvenanceTrustEval(trust assignment $\alpha$)
for every mapping node labeled $m$
    do if $\alpha(m) = \mathbf{Dm}$ replace the node label with $\mathbf{D}$
              and erase incoming edges
       if $\alpha(m) = \mathbf{Tm}$ erase the mapping node and
              connect incoming edges
              directly to outgoing edges
for every tuple node labeled with provenance token $a$
    do replace node label with $\alpha(a)$
Initialize OUTPUT $\leftarrow \emptyset$
repeat
    for every tuple node $N$
        do if $N$ has an incoming edge from a $\mathbf{T}$ label
              then replace $N$'s label with $\mathbf{T}$
                   erase incoming edges
                   and add $\mathsf{P}v(N) = \mathbf{T}$ to OUTPUT
until no more changes are made to OUTPUT
for every remaining tuple node $N$
    do add $\mathsf{P}v(N) = \mathbf{D}$ to OUTPUT
return OUTPUT
```

Figure 4.11: Evaluating trust based on provenance

*Suppose now that peer $\boldsymbol{P}_{BioSQL}$ trusts data contributed locally by $\boldsymbol{P}_{uBio}$ and itself, and hence assigns **T** to the provenance tokens p_2 and p_1, but does not trust $\boldsymbol{P}_{GUS}$'s tuple $(3, 5, 2)$ and so assigns **D** to p_3. Assuming that all mappings have the trivial trust conditions **Tm**, the provenance of $B(3, 2)$ evaluates as follows:*

$$\boldsymbol{Tm}(\boldsymbol{D}) + \boldsymbol{Tm}(\boldsymbol{T} \cdot \boldsymbol{T}) + \boldsymbol{Tm}(\boldsymbol{T} \cdot \boldsymbol{Tm}(\boldsymbol{D})) = \boldsymbol{D} + \boldsymbol{T} + \boldsymbol{D} = \boldsymbol{T}$$

therefore $\boldsymbol{P}_{BioSQL}$ should indeed have accepted $(3, 2)$.

For the ranked trust model, assume that peer $\boldsymbol{P}_{BioSQL}$ trusts completely all data contributed locally at $\boldsymbol{P}_{BioSQL}$ and hence assigns a cost of 0 to the provenance token p_1. Moreover, it trusts data from $\boldsymbol{P}_{uBio}$ more than those from $\boldsymbol{P}_{GUS}$, but not as much as local data, and thus assigns e.g., the costs of 1 to $p2$ and 5 to p_3. Suppose also that $\boldsymbol{P}_{BioSQL}$ does not trust equally $\boldsymbol{P}_{GUS}$'s tuples and thus assigns a cost of e.g., 5 to p_3. Finally, assume *mappings m_1 and m_2 are completely trusted, while m_4 is considered somewhat untrusted, and thus multiplies the cost of its input by 2. We write f_{m_4} to indicate this function below, and f_{id} for the identity function used for the other mappings that are completely trusted. Then, the cost of $B(3, 2)$ evaluates as follows:*

$$\boldsymbol{min}(f_{id}(5), f_{m_4}(0+1), f_{m_4}(0+f_{id}(5))) = \boldsymbol{min}(5, 2\cdot 1, 2\cdot 5) = \boldsymbol{min}(5, 2, 10) = 2$$

Note that, in the expression above, $\cdot$ is regular integer multiplication — introduced by the definition of the mapping function for m_4 — and not the multiplication operation of the semiring (which is $+$ for the case of the tropical semiring).

4.8.2 Update Exchange with Trust Conditions

We can now fully specify the mappings necessary to perform update exchange — which combines update translation with trust. For each relation R, the trust conditions are applied during update exchange to the input instance R^i of the peer, thus selecting the trusted tuples from among all the tuples derived from other peers. The result is an internal relation we can denote R^t. So instead of the internal mappings (i_R) described in Section 3.2 we actually have in $\mathcal{M}'$:

$$\begin{array}{ll} (i_R) & R^t(\bar{x}) = trusted(R^i(\bar{x})) \\ (t_R) & R^t(\bar{x}) \wedge R^\alpha(\bar{x}) \rightarrow R^o(\bar{x}) \end{array}$$

and the definition of consistent state remains the same.

4.9 Query Containment

Before proceeding to explain how we can perform update exchange while recording provenance that conforms to the model of this chapter, we present some results about query containment w.r.t. the general semantics in K-relations. Query containment checks can be used to determine query equivalence, and thus enable rewriting queries to equivalent more efficient ones, during query optimization.

Definition 4.9.1. *Let K be a naturally ordered commutative semiring and let q_1, q_2 be two queries defined on K-relations. We define containment with respect to K-relations semantics by*

$$q_1 \sqsubseteq_K q_2 \overset{def}{\Leftrightarrow} \forall R \, \forall t \; q_1(R)(t) \leq q_2(R)(t)$$

When K is $\mathbb{B}$ and $\mathbb{N}$ we get the usual notions of query containment with respect to set and bag semantics.

Some simple facts follow immediately. For example if $h : K \to K'$ is a semiring homomorphism such that $h(x) \leq h(y) \Rightarrow x \leq y$ and q_1, q_2 are $\mathcal{RA}^+$ queries it follows from Prop. 4.2.3 that $q_1 \sqsubseteq_{K'} q_2 \Rightarrow q_1 \sqsubseteq_K q_2$. If instead h is a surjective homomorphism then $q_1 \sqsubseteq_K q_2 \Rightarrow q_1 \sqsubseteq_{K'} q_2$. Similarly when K, K' and h are also ω-continuous and q_1, q_2 are datalog queries (via Prop. 4.4.4).

The following result allows us to use the decidability of containment of unions of conjunctive queries [25, 92].

Theorem 4.9.1. *If K is a* distributive lattice *then for any q_1, q_2 unions of conjunctive queries we have*

$$q_1 \sqsubseteq_K q_2 \;\; \textit{iff} \;\; q_1 \sqsubseteq_{\mathbb{B}} q_2$$

Proof. (sketch) One direction follows because $\mathbb{B}$ can be homomorphically embedded in K. For the other direction we use the existence of query body homomorphisms to establish mappings between monomials of provenance polynomials. Then we apply the factorization theorem (4.3.1) and the idempotence and absorption laws of K. □

Therefore, if K is a distributive lattice for (unions of) conjunctive queries containment with respect to K-relation semantics is decidable by the same procedure as for standard set semantics. $\mathsf{PosBool}(B)$, $\mathcal{P}(\Omega)$ and the fuzzy semiring are all distributive lattices. A theorem similar to the one above is shown in [67] but the class of algebraic structures used there does not include $\mathsf{PosBool}(B)$ or $\mathcal{P}(\Omega)$ (although it does include the fuzzy semiring).

In this chapter we presented a novel model of provenance that generalizes query answering over different models of annotated relations and past models of provenance. In the next chapter we show how such provenance information can be stored in a relational database and maintained during update exchange. Moreover, we explain how the existence of such provenance information enables algorithms for internal CDSS operations, for different kinds of updates and mappings, that lie at the core of update exchange.

Chapter 5

Performing Update Exchange

We now discuss how to actually compute peer data instances in accordance with the model of Chapter 3, while maintaining a provenance graph as described in Chapter 4. We begin with some preliminaries, describing how we express the computations as queries and how we model provenance using relations. Then we describe how incremental update exchange can be attained.

In order to meet participants' needs for anonymity (they want all data and metadata to be local in order to prevent others from snooping on their queries), our model performs all update exchange computation locally, in auxiliary storage alongside the original DBMS (see Chapter 6). It imports any updates made directly by others and incrementally recomputes its own copy of all peers' relation instances and provenance — also filtering the data with its own trust conditions as it does so. Between update exchange operations, it maintains copies of all relations, enabling future operations to be incremental. In ongoing work, we are considering more relaxed models in which portions of the computation may be distributed.

5.1 Computing Instances with Provenance

In the literature [45, 80, 65], chase-based techniques have been used for computing candidate universal solutions. However, these are primarily of theoretical interest and cannot be directly executed on a conventional query processor. In constructing the ORCHESTRA system, we implement update exchange using **relational query processing** techniques, in order to take advantage of robust existing DBMS engines, as well as to ultimately leverage multi-query optimization and distributed query execution. We encode the provenance in relations alongside the data, making the computation of the data instances and their provenance a seamless operation. The Clio system [89] used similar techniques to implement data exchange but did not consider updates or provenance.

5.1.1 Datalog for Computing Peer Instances

Work in data integration has implemented certain types of chase-like reasoning with relational query processors for some time [44, 89]: datalog queries are used to compute the certain answers [23] to queries posed over integrated schemas. Our goal is to do something similar. However, when computing canonical universal solutions for the local peer instances, we face a challenge because the instance may contain *incomplete* information, e.g., because not all attributes may be provided by the source. In some cases, it may be known that two (or more) values are actually **the same**, despite being of unknown value[1]. Such cases are generally represented by using existential variables in the target of a mapping tgd (e.g., (m_3) in Example 5). This requires *placeholder values* in the canonical universal solution. Chase-style procedures [45, 80, 65] use *labeled nulls* to encode such values. In our case, since we wish to use datalog-like queries, we rely on *Skolem functions* to specify the placeholders, similarly to [89]. Each such function provides a unique

[1] This enables joins on tuples on unknown values that may result in additional answers.

placeholder value for each combination of inputs; hence two placeholder values will be the same if and only if they were generated with the same Skolem function with the same arguments.

Normal datalog does not have the ability to compute Skolem functions; hence, rather than converting our mapping tgds into standard datalog, we instead use a version of datalog extended with Skolem functions. (Chapter 6 discusses how these queries can in turn be executed on an SQL DBMS.)

Notation. Throughout this section, we will use the syntax of datalog to represent queries. Datalog rules greatly resemble tgds, except that the output of a datalog rule (the "head") is a single relation and the head occurs to the **left** of the body. The process of transforming tgds into datalog$^{\text{sk}}$ rules is the same as that of the *inverse rules* of [44]:

$$\phi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z}\ \psi(\bar{x}, \bar{z}) \text{ becomes } \psi(\bar{x}, \bar{f}(\bar{x})) \text{ :- } \phi(\bar{x}, \bar{y})$$

We convert the RHS of the tgd into the head of a datalog rule and the LHS to the body. Each existential variable in the RHS is replaced by a Skolem function over the variables in common between LHS and RHS: $\bar{z}$ is replaced by $\bar{f}(\bar{x})$.[2]

Our scheme for parameterizing Skolem functions has the benefit of producing universal solutions (as opposed, e.g., to using a subset of the common variables) while guaranteeing termination for weakly acyclic mappings (which is not the case with using all variables in the LHS, as in [65]). If ψ contains multiple atoms in its RHS, we will get multiple datalog rules, with each such atom in the head. Moreover, it is essential to use a **separate** Skolem function for each existentially quantified variable in each tgd.

Example 10. *Recall schema mapping* (m_3) *in Example 5. Its corresponding internal*

[2] Although the term has been occasionally abused by computer scientists, our use of *Skolemization* follows the standard definition from mathematical logic.

schema mapping:

$$B^o(i,n) \rightarrow \exists c\, U^i(n,c) \;\; \textit{becomes} \;\; U^i(n,f(n)) \text{ :- } B^o(i,n)$$

We call the resulting datalog rules **mapping rules** and we use $P_\mathcal{M}$ to denote the datalog$^{\text{sk}}$ program consisting of the mapping rules derived from the set of schema mappings $\mathcal{M}$.

Proposition 1. *If $\mathcal{M}$ is a weakly acyclic set of tgds, then $P_\mathcal{M}$ terminates for every edb instance I and its result, $P_\mathcal{M}(I)$, is a universal solution.*

Proof. (Sketch) Observe that, whenever the chase is applicable, the rules of $P_\mathcal{M}$ are also applicable, and moreover the tuples produced by $P_\mathcal{M}$ in this case are isomorphic to those produced by the chase. On the other hand, there are cases when the chase is not applicable with some homomorphism, because the homomorphism can be extended to the conclusion of the tgd (but not necessarily mapping a variable z to $f_{m,z}(\bar{x})$), but some rule of $P_\mathcal{M}$ still is. However, there is still a homomorphism from the extra tuples to the chase result, namely mapping the extra tuples image of the head of the tgds under the extension of the homomorphism. Finally, since the active domain and the set of names of Skolem functions in $P_\mathcal{M}$ are finite, for $P_\mathcal{M}(I)$ to have an infinite result, there must be tuples containing Skolem terms of the form $f(...f(...)...)$. However, it is straightforward to verify that such Skolem terms cannot be produced for weakly acyclic sets of tgds.

□

This basic methodology produces a program for recomputing CDSS instances, given a datalog engine with fixpoint capabilities.

5.1.2 Incorporating Provenance

We now show how to encode the provenance graph *together* with the data instances, using additional relations and datalog rules. This allows for seamless recomputation of both data and provenance and allows us to better exploit conventional relational processing.

We observe that in a set-based relational model, there exists a simple means of generating a unique ID for each base tuple in the system: within any relation, a tuple is uniquely identified by its values. We exploit this in lieu of generating provenance IDs: for the provenance of $G(3, 5, 2)$, instead of inventing a new value p_3 we can use $G(3, 5, 2)$ itself.

Then, in order to represent a product in a provenance expression, which appears as a result of a join in the body of a mapping rule, we can record all the tuples that appear in an instantiation of the body of the mapping rule. However, since some of the attributes in those tuples are always equal in all instantiations of that rule (i.e., the same variable appears in the corresponding columns of the atoms in the body of the rule), it suffices to just store the value of each unique variable in a rule instantiation. To achieve this, for each mapping rule

$$(m_i) \quad R(\bar{x}, \bar{f}(\bar{x})) \text{ :- } \phi(\bar{x}, \bar{y})$$

we introduce a new relation $P_{Ri}(\bar{x}, \bar{y})$ and we replace (m_i) with the mapping rules

$$\begin{aligned} (m_i') &\qquad P_{Ri}(\bar{x}, \bar{y}) \text{ :- } \phi(\bar{x}, \bar{y}) \\ (m_i'') &\qquad R(\bar{x}, \bar{f}(\bar{x})) \text{ :- } P_{Ri}(\bar{x}, \bar{y}) \end{aligned}$$

Note that (m_i') mirrors m_i but *does not project any attributes*. Note also that (m_i'') derives the actual data instance from the provenance encoding.

Example 11. *Since tuples in B (as shown in the graph of Example 8) can be derived through mappings m_1 and m_4, we can represent their provenance using two relations P_1*

and P_4*, using the mapping rules:*

$$\begin{aligned}
&P_1(i,c,n) \text{ :- } G(i,c,n)\\
&B(i,n) \text{ :- } P_1(i,c,n)\\
&P_4(i,n,c) \text{ :- } B(i,c), U(n,c)\\
&B(i,n) \text{ :- } P_4(i,n,c)
\end{aligned}$$

Recall from example 8 that $\textsf{\textbf{P}}v(B(3,2)) = m_1(\textsf{\textbf{P}}v(G(3,5,2)))+m_4(\textsf{\textbf{P}}v(B(3,5))\textsf{\textbf{P}}v(U(2,5)))$. *The first part of this provenance expression is represented by the tuple* $P_1(3,5,2)$ *and the second by the tuple* $P_4(3,2,5)$*. (Both can be obtained by assigning* $i = 3, n = 2, c = 5$ *in each of the rules above.) In general, after applying mapping rules such as the ones shown above, for every node labeled by a mapping* m *in a provenance graph there is a tuple representing it in the provenance relation* P*. In the case of the graph of Figure 4.8 from example 8, the contents of* P_1 *and* P_4 *are as follows:*

P_1

3	5	2
1	2	3

P_4

3	2	5
3	3	2

As we detail in Chapter 6, we can further optimize this representation in order to efficiently implement it in an RDBMS.

5.1.3 Derivation Testing

In addition to computing instances and their provenance, we can use datalog rules to determine *how a tuple was derived.* This is useful in two contexts: first, to assess whether a tuple should be trusted, which requires that it be derivable only from trusted sources and along trusted mappings; and second, to see if it is derivable even if we remove certain tuples, which is necessary for performing incremental maintenance of CDSS instances.

The challenge is to compute the set of base insertions (i.e., the union of their lineages) from which a tuple (or set of tuples) was derived in a *goal-directed* way. In essence, this requires reversing or inverting the mappings among the provenance relations. We do this by first creating a new relation R' for each R, and populating each R' with the tuples whose derivation we wish to check. Then, for each mapping rule specifying how to derive R from P_i:

$$(m_i'') \quad R(\bar{x}, \bar{f}(\bar{x})) \text{ :- } P_i(\bar{x}, \bar{y})$$

we define an inverse rule that uses the existing provenance table to fill in the possible values for $\bar{f}(\bar{x})$, namely the $\bar{y}$ attributes that were projected away during the mapping. This results in a new relation P_i' with exactly those tuples from which R can be derived using mapping P_i:

$$P_i'(\bar{x}, \bar{y}) \text{ :- } R'(\bar{x}), P_i(\bar{x}, \bar{y})$$

If we run the program starting with the tuples whose derivability we want to check in the corresponding R' relations, its fixpoint will contain the set of tuples involved in some derivation of some of those tuples. If we filter the R' relations to only include values from local contributions tables, the result would be the union of all lineages of all tuples whose derivability we wanted to check. In general, we will perform one final step to identify these base tuples, as well as to ensure we only include trusted tuples. Finally, we essentially run a goal-directed variation[3] of the original datalog program over the filtered R' instances to identify the tuples that can indeed be re-derived.

Example 12. *Consider a CDSS with the following mappings from Example 5:*

$$(m_1) \qquad G(i, c, n) \to B(i, n)$$

$$(m_2) \qquad G(i, c, n) \to U(n, c)$$

$$(m_3) \qquad B(i, n) \to \exists c\, U(n, c)$$

[3] to check only for tuples of interest, instead of producing all possible tuples

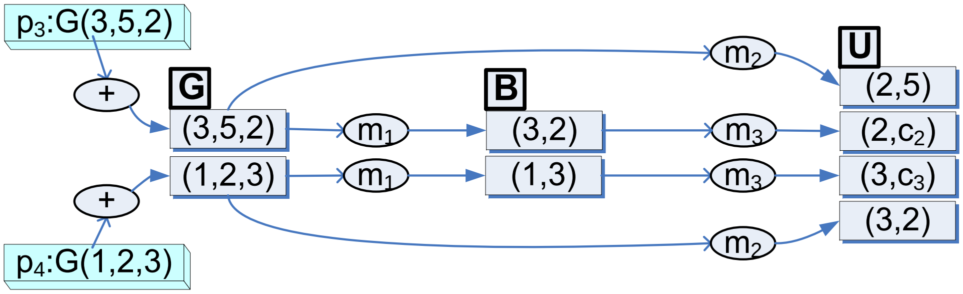


Figure 5.1: Provenance graph for derivability testing example

and the provenance graph shown in Figure 5.1. The corresponding mapping rules are:

$$(m_1') \qquad P_1(i,c,n) \text{ :- } G(i,c,n)$$

$$(m_1'') \qquad B(i,n) \text{ :- } P_1(i,c,n)$$

$$(m_2') \qquad P_2(i,c,n) \text{ :- } G(i,c,n)$$

$$(m_2'') \qquad U(n,c) \text{ :- } P_2(i,c,n)$$

$$(m_3') \qquad P_3(i,n) \text{ :- } B(i,n)$$

$$(m_3'') \qquad U(n,f(n)) \text{ :- } P_3(i,n)$$

From these we compute the following inverse rules:

$$(r_1) \qquad P_1'(i,c,n) \text{ :- } B'(i,n), P_1(i,c,n)$$

$$(r_2) \qquad P_2'(i,c,n) \text{ :- } U'(n,c), P_2(i,c,n)$$

$$(r_3) \qquad P_3'(i,n) \text{ :- } U'(n,c), P_3(i,n)$$

$$(r_4) \qquad G'(i,c,n) \text{ :- } P_1'(i,c,n)$$

$$(r_5) \qquad G'(i,c,n) \text{ :- } P_2'(i,c,n)$$

$$(r_6) \qquad B'(i,n) \text{ :- } P_3'(i,c,n)$$

Since B is an idb relation, we replace the atom in the head of (r_6) using rules with B in the target, namely (m_1''), to get:

$$(r_6^1) \quad P_1'(i,c,n) \text{ :- } P_3'(i,n), P_1(i,c,n)$$

Rules (r_1)-(r_5), (r_6^1) *form the derivation testing program. In particular, if we set* $U' = \{(2, c_2), (3, 2)\}$ *and run this program, its fixpoint will also contain:* $P_2'(1, 2, 3)$, $P_3'(3, 2)$, $B'(3, 2)$, $P_1'(3, 5, 2)$, $G'(3, 5, 2)$, $G'(1, 2, 3)$. *Of these, only* $G'(3, 5, 2)$ *and* $G'(1, 2, 3)$ *are base tuples. We use those to check which of the tuples in* U' *are still derivable. For example, if* $G'(1, 2, 3)$ *has been deleted or is untrusted, we can run a goal-directed variation of the original program on the only remaining base tuple* $G'(3, 5, 2)$ *to infer that* $U'(2, c_2)$ *is still derivable but* $U'(3, 2)$ *is not.*

5.2 Incremental Update Exchange

One of the major motivating factors in our choice of provenance formalisms has been the ability to *incrementally maintain* the provenance associated with each tuple, and also the related data instances. We now discuss how this can be achieved using the relational encoding of provenance of Section 5.1.2.

Following [61] we convert each mapping rule (after the relational encoding of provenance) into a series of *delta rules*. For the insertion delta rules we use new relation names of the form R^+, ${P_i}^+$, etc. while for the deletion delta rules we use new relation names of the form R^-, ${P_i}^-$, etc.

For the case of **incremental insertion** in the absence of peer-specific trust conditions, the algorithm is simple, and analogous to the **counting** and DRed incremental view maintenance algorithms of [61]: we can directly evaluate the insertion delta rules until reaching a fixpoint and then add R^+ to R, ${P_i}^+$ to P_i, etc. Trust combines naturally with the incremental insertion algorithm: the starting point for the algorithm is already-trusted data (from the prior instance), plus new "base" insertions which can be directly tested for trust (since their provenance is simply their source). Then, as we derive tuples via mapping rules from trusted tuples, we simply apply the associated trust conditions to ensure that we only derive new trusted tuples.

```
Algorithm PropagateDelete
for every $P_{Ri}$, let $R^0 \leftarrow R$
Initialize $c \leftarrow 0$
repeat
    Compute all $P_i^-$ based on their delta rules
    ($*$ Propagate effects of deletions $*$)
    for each idb $R$
        do update each associated $P_i$, by applying $P_i^-$ to it
            Define new relation $R^{c+1}$ to be the union of all $P_i$, projected to the
                $\bar{x}$ attributes.
    ($*$ Check tuples whose provenance was affected $*$)
    for each idb $R$
        do Let  $R'$ be an empty temporary relation with $R$'s schema
            for each tuple $R^c(\bar{a})$ not in $R^{c+1}(\bar{a})$
                do if there exists, in any provenance relation $P_i$ associated with
                    $R$, a tuple $(\bar{a}, \bar{y}_i)$
                    then add tuple $(\bar{a})$ to $R'$
                    else  add tuple $(\bar{a})$ to $R^-$
            Test each tuple in $R_{chk}$ for derivability from edbs; add it to $R^-$ if it
            fails
    Increment $c$
until no changes are made to any $R^-$
return the set of all $P_i^-$ and $R^-$
```

Figure 5.2: Deletion propagation algorithm

Incremental deletion (also called *decremental* maintenance) is significantly more complex. When a tuple is deleted, it is possible to remove any provenance expressions and tuples that are its immediate consequents and are no longer directly derivable. However, the provenance graph may include cycles: it is possible to have a "loop" in the provenance graph such that several tuples are mutually derivable from one another, yet none are derivable from edbs, i.e., local contributions from some peer in the CDSS. Hence, in order to "garbage collect" these no-longer-derivable tuples, we must test whether they are derivable from trusted base data in local contributions tables; those tuples that are not must be recursively deleted following the same procedure.

Suppose that we are given each list of initial updates $\bar{R}^-$ from all of the peers. Our goal is now to produce a set of R^i update relations for the peer relations and a corresponding set $P_i{}^-$ to apply to each provenance relation. Figure 5.2 shows pseudocode for such an algorithm. First, the algorithm derives the deletions to apply to the provenance mapping relations; based on these, it computes a new version of the peer schema relations and their associated provenance relations (Lines 4–8). Next, it must determine whether a tuple in the instance is no longer derivable (Lines 10-16): such tuples must also be deleted. The algorithm first handles the case where the tuple is not directly derivable (Line 13), and then it performs a more extensive test for derivability from edbs (Line 16). The "existence test" is based on the derivation program described in Section 5.1.3, which determines the set of edb tuples that were part of the original derivation. Given that set, we must actually validate that each of our R_{chk} tuples are *still* derivable from these edbs, by re-running the original set of schema mappings on the edb tuples.

These two steps may introduce further deletions into the system; hence it is important to continue looping until no more deletions are derived.

Example 13. *Revisiting Example 8 and the provenance graph there, suppose that we wish to propagate the deletion of the tuple $T(3, 2)$. This leads to the invalidation of mapping node labeled m_4 and then the algorithm checks if the tuple $S(1, 2)$ is still derivable. The check succeeds because of the inverse path through (m_1) to $U(1, 2, 3)$.*

We note that a prior approach to incremental view maintenance, the DRed algorithm [61], has a similar "flavor" but takes a more pessimistic approach. (DRed was formulated for view maintenance without considering provenance, but it can be adapted to our setting.) Upon the deletion of a set of tuples, DRed will pessimistically remove all tuples that can be transitively derived from the initially deleted tuples. Then it will attempt to re-derive the tuples it had deleted. Intuitively, we should be able to be more efficient than DRed on average, because we

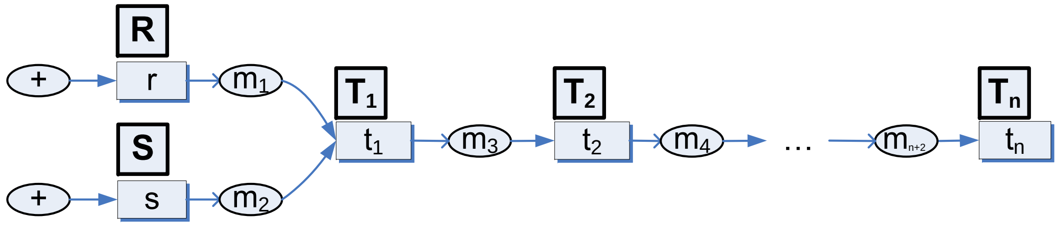


Figure 5.3: Example provenance graph showing relationships between tuples

can exploit the provenance trace to test derivability in a goal-directed way. Moreover, since our algorithm proceeds one step at a time, if it finds that a tuple is still derivable and does not need to be deleted, it will not need to test for derivability any other tuples transitively derived from it. Finally, DRed's re-derivation should typically be more expensive than our test for derivability, because insertion is more expensive than querying. In Section 6.4 we validate this hypothesis. However, to provide an intuition of the differences, we give a brief example.

Example 14. *Consider Figure 5.3, where nodes represent tuples and arrows represent the immediate-consequence relationship. The diagram corresponds to the mappings:*

$$R \rightarrow T_1, \quad S \rightarrow T_1,$$

$$\begin{array}{cc} T_1 \rightarrow T_2, & T_2 \rightarrow T_1, \\ T_2 \rightarrow T_3, & T_3 \rightarrow T_2, \\ \ldots & \\ T_{n-1} \rightarrow T_n, & T_n \rightarrow T_{n-1}, \end{array}$$

and an instance where R and S contain one tuple each.

Now suppose r is deleted. This is a bad case for DRed, because it initially takes n iterations to erroneously delete all t_i-tuples, and subsequently takes another n iterations to re-derive them. On the other hand, our algorithm would find that t_1 is still reachable *after deleting r and stop immediately, without affecting any other t_i-tuples. The behavior is similar if only s is deleted.*

On the other hand, suppose both r and s are deleted. Then this is a good case for DRed *since all t_i-tuples in fact need to be deleted, so no re-derivation is required. On the other hand, our algorithm would require a derivation test before deleting each t_i-tuple, although each of these tests would only take one step to find that the corresponding tuple is not derivable.*

5.3 Extensions for Bidirectional Update Exchange

The CDSS model presented in the previous chapters employs mappings from source to target peers, similar to those used in data integration and exchange. An update made to a peer's instance is applied to the peer's local database instance. Upon request, the CDSS propagates this update to all downstream instances using update exchange. This matches situations where one database is more authoritative than another: updates from a curated database like SWISS-PROT should propagate to individual biologists' databases, but not the converse.

However, in some cases two peers, even with different schemas, want to mirror data: either peer may update data from itself or its neighbor, and the effects should propagate to the other peer. We are aware of no existing solution to this problem in a setting with schema mappings. In this dissertation, we consider the problems of specifying **bidirectional** mappings between instances, and propagating updates along these mappings. We briefly illustrate with an example.

Example 15. *Figure 5.4 shows a variant of the example bioinformatics CDSS setting of Figure 1.1. $\boldsymbol{P}_{uBio}$ and $\boldsymbol{P}_{GUS}$ want to propagate updates to each other via mapping m_1, and so do $\boldsymbol{P}_{uBio}$ and $\boldsymbol{P}_{BioSQL}$ via m_2. Note that update propagation composes: an update to $\boldsymbol{P}_{BioSQL}$ will result in a update of $\boldsymbol{P}_{uBio}$, which in turn induces an update over $\boldsymbol{P}_{GUS}$.*

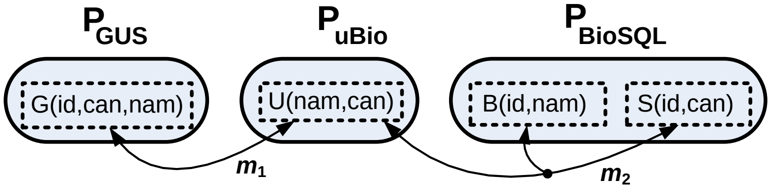


Figure 5.4: Collaborative data sharing system with bidirectional mappings among 3 peers: $\mathbf{P}_{uBio}$, $\mathbf{P}_{GUS}$, and $\mathbf{P}_{BioSQL}$.

Our problem generalizes two separately studied topics in the traditional relational database realm: a materialized view may be simultaneously *maintainable* (i.e., updates made to the base instance are propagated to the view instance) and *updatable* (i.e., updates made to the materialized view are propagated to the base relations). However, a view is a *function* between source instance and materialized view instance; whereas a schema mapping represents a *containment constraint* among instances. Moreover, we consider settings with multiple mappings and peers, some of which can interact with one another or share target relations, whereas the view update literature typically focuses on a single view, whose definition is a single (typically conjunctive) query. Another important difference is that in a view definition, one side only contains base tuples, while the other only consists of data derived from those base tuples. In a CDSS with bidirectional mappings between peers, each peer typically contributes its own data as well as imports data from the other peers through mappings. These differences have significant consequences, which we consider in this section.

5.3.1 Preliminaries: Bidirectional Data Exchange

Before considering bidirectional update exchange, we first consider an extension to the traditional data exchange setting to support multiple peers, each with its own data instance, and bidirectional schema mappings. Our setting looks like:

- Peer schemas $P_1, \ldots, P_n$.

- Instances $I_1, \ldots, I_n$ of $P_1, \ldots, P_n$, respectively.
- A set of mappings $\mathcal{M}$ among the *peer relations* of $P_1, \ldots, P_n$, specified as logical expressions of the form:

  $$(m) \quad \forall \bar{x}(\exists \bar{y}\ \phi(\bar{x}\bar{y}) \leftrightarrow \exists \bar{z}\ \psi(\bar{x}\bar{z}))$$

  where the formula in each side of the mappings is a conjunction of atoms over one of the schemas (e.g., ϕ is a conjunction of atoms over P_1 and ψ is a conjunction of atoms over P_2).

Every bidirectional mapping m of the form shown above is logically equivalent to a pair of tgds:

$$\begin{array}{ll}(m^{\rightarrow}) & \forall \bar{x}\bar{y}\ \phi(\bar{x}\bar{y}) \rightarrow \exists \bar{z}\ \psi(\bar{x}\bar{z}) \\ (m^{\leftarrow}) & \forall \bar{x}\bar{z}\ \psi(\bar{x}\bar{z}) \rightarrow \exists \bar{y}\ \phi(\bar{x}\bar{y})\end{array}$$

Example 16. *The mappings for Figure 5.4 are:*

$$\begin{array}{ll}(m_1) & \forall cn\ (\exists i\ G(i, c, n) \leftrightarrow U(n, c)) \\ (m_2) & \forall nc\ (\exists i\ B(i, n) \wedge S(i, c) \leftrightarrow U(n, c))\end{array}$$

These mappings are equivalent to the following tgds:

$$\begin{array}{ll}(m_1^{\rightarrow}) & \forall icn\ G(i, c, n) \rightarrow U(n, c) \\ (m_1^{\leftarrow}) & \forall nc\ U(n, c) \rightarrow \exists i\ G(i, c, n) \\ (m_2^{\rightarrow}) & \forall inc\ B(i, n) \wedge S(i, c){\rightarrow}U(n, c) \\ (m_2^{\leftarrow}) & \forall nc\ U(n, c){\rightarrow}\exists i\ B(i, n) \wedge S(i, c)\end{array}$$

For readability, as in Chapter 3, in the rest of this Chapter we will omit the universal quantifiers for variables that appear in the left-hand side (LHS) of mappings.

Bidirectional Data Exchange Semantics

Any set of bidirectional mappings can be converted to a standard data exchange setting $(\mathcal{S}, \mathcal{T}, \Sigma_{st}, \Sigma_t)$ as follows: Let $P_1^\ell, \ldots, P_n^\ell$ be the schemas obtained by replacing each relation R of $P_1, \ldots, P_n$, respectively, by R^ℓ (the *local contribution* relations). In the data exchange setting, let:

- Source schema $\mathcal{S} = P_1^\ell \cup \cdots \cup P_n^\ell$,
- Target schema $\mathcal{T} = P_1 \cup \cdots \cup P_n$
- Source instance $I = I_1 \cup \cdots \cup I_n$
- Source-target mappings $\Sigma_{st} = \{R^\ell(\bar{x}_R) \rightarrow R(\bar{x}_R) \mid R \in P_1 \cup \cdots \cup P_n\}$
- Target mappings $\Sigma_t = \mathcal{M}$ (i.e., the set of tgds that the bidirectional mappings are equivalent to)

We define the canonical universal solution for our bidirectional data exchange setting to be the one for this translated data exchange setting.

Example 17. *For the mappings in Example 16, assume local contribution relations:*

G^ℓ

1	a	b
2	d	e

B^ℓ

3	b
3	c
4	h

S^ℓ

3	a
5	k

U^ℓ

f	g

The canonical universal solution (according to [65]) of the corresponding data exchange *setting is:*

G		
1	a	b
2	d	e
x_5	g	f
x_6	a	b
x_7	a	c
x_8	d	e

B	
3	b
3	c
4	h
x_1	b
x_2	e
x_3	c
x_4	f

S	
3	a
5	k
x_1	a
x_2	d
x_3	a
x_4	g

U	
b	a
e	d
c	a
f	g

Values $x_1, ..., x_8$ are labeled nulls*: placeholder values for unknown values that are generated by mappings with existential variables in the right-hand side (RHS) ($m_1^{\leftarrow}, m_2^{\leftarrow}$ here).*

Existential variables must be used with care in mappings, since bidirectional mappings introduce cycles. The canonical universal solution is guaranteed to exist, and the algorithms in [45, 65] compute it, if the set of target dependencies is *weakly acyclic* [45, 42]. For a single bidirectional mapping, we can show [69] that if there are no self-joins on either side of the mapping, the resulting pair of mappings is always weakly acyclic, even if there are existential variables on both sides.

Theorem 5.3.1. *If a bidirectional mapping $\mathcal{M}$ between two peers with disjoint schemas has no self-joins on either side, then the pair of tgds that is equivalent to $\mathcal{M}$ is weakly-acyclic.*

Proof. Let $\forall \mathbf{x}(\exists \mathbf{y}\ \phi(\mathbf{xy}) \leftrightarrow \exists \mathbf{z}\ \psi(\mathbf{xz}))$ be a bidirectional mapping. As explained above, it is equivalent to the pair of tgds:

$$\forall \mathbf{xy}\ \phi(\mathbf{xy}) \rightarrow \exists \mathbf{z}\ \psi(\mathbf{xz})$$

$$\forall \mathbf{xz}\ \psi(\mathbf{xz}) \rightarrow \exists \mathbf{y}\ \psi(\mathbf{xy})$$

Consider the dependency graph for this pair of tgds, according to the definition in [45]. For the pair of tgds above, this graph has a node for each attribute in each relation appearing in ϕ and ψ. Moreover, there is an edge $A \rightarrow B$ in this graph if:

- the same variable $x_i \in \mathbf{x}$ appears at the attributes for A and B on opposite sides of the tgds. We call these *common* variables. Since bidirectional mappings are translated to a pair of tgds as above, for each such pair of attributes there is in fact a pair of edges, $A \rightarrow B$ and $B \rightarrow A$.
- an existentially quantified variable (e.g., $z_j \in \mathbf{z}$ in the first of the pair of tgds, $y_k \in \mathbf{y}$ in the second) appears at attribute B and any of the common variables $x_i \in \mathbf{x}$ appears at A at the source of the tgd. In this case, the edge is called "starred".

According to [45], a set of dependencies is weakly acyclic if this graph doesn't contain any cycle going through a starred edge.

In the case of a bidirectional mapping, since peer schemas are disjoint and ϕ and ψ are conjunctions of atoms over different peer schemas, the corresponding graph is bipartite. Moreover, if there are not multiple atoms of the same relation in each side (i.e., self-joins) in either side of the tgd, each attribute appears once on each side of the tgd (and only on one side of the bipartite graph). Then, for all attributes at which an existentially quantified variable appears in either tgd, there is a starred edge going in it, according to the definition of edges above, but there are no outgoing edges (starred or not) in the opposite direction, since they are not among the common variables of the other tgd.[4] As a result, there cannot be any cycle containing a starred edge, and thus the graph is weakly acyclic. $\square$

[4] Note that this would not necessarily be true if we could have atoms of the same relation in the same side, because then there could exist an attribute in which both a common and an existentially quantified variable appear (in different atoms in the body of the same tgd), and thus there could be outgoing edges from this node because of the common variables.

If there are multiple mappings with the same target, the situation is more complex and we must apply the weak acyclicity test given in [45].

As explained in [57], one way to compute the canonical universal solution is to translate mappings into a datalog program, whose least fixpoint is the canonical universal solution. For every atom in the RHS of the mapping, we create a rule with that atom as its head and the LHS of the mapping as its body. To deal with mappings with existential variables in the RHS, we again use datalog$^{\text{sk}}$, which uses Skolem functions to create unique placeholder values for each combination of relevant values on which the mapping is applied. The mappings above are translated to the rules:

$$\begin{array}{ll} 1 & U(n,c) \text{ :- } G(i,c,n) \\ 2 & G(f(n,c),c,n) \text{ :- } U(n,c) \\ 3 & B(g(n,c),n) \text{ :- } U(n,c) \\ 4 & S(g(n,c),c) \text{ :- } U(n,c) \end{array}$$

where f (see Rule 2) is the Skolem function for the existential variable i in mapping $m_1^{\leftarrow}$, and g is the one for variable i in mapping $m_2^{\leftarrow}$. Note that the same Skolem term appears in Rules 3 and 4, since the corresponding mapping atoms share the same variable i.

5.3.2 Performing Bidirectional Update Exchange Incrementally

We now consider updates in the form of insertions and deletions. The previous section identified a means of generating a (recursive) datalog program for computing instances for the peers, given instances of locally introduced data (local contributions). Now our goal is to take as input updates made by users over the computed instances, translate these updates into modifications over local contribution relations (i.e., base data) as appropriate, and then achieve the update over a recomputed version of the canonical universal solution. In essence, this is a version of the view update problem, over the datalog program for generating the canonical universal solution. However, in contrast to a view setting, here tuples may be introduced locally by *any* peer, and deletion of a tuple must remove data from *every* peer from which that tuple can be derived.

We first consider insertions and deletions that affect only the local contributions tables at the same peer, before considering how to propagate deletions to local contribution relations at other peers. (Insertions will always be made locally, in accordance with the existing CDSS model.)

Insertions and Deletions at the Same Peer

For insertions, we start with a previous instance of the CDSS, which is a solution $\langle I, J \rangle$, and we take a set of insertions Δ^+ that we apply directly over the local contribution relations at the peers that originated the updates. Then we compute a new canonical universal solution $\langle I + \Delta^+, J + Y^+ \rangle$. We can directly recompute the instance using the datalog program of the previous section, adding new tuples to the peer relations until the mappings are satisfied. Even better, since bidirectional mappings are equivalent to a pair of unidirectional mappings, we can derive an *incremental maintenance program* using the *delta rules* [62] extension that was presented in [57], and perform the recomputation more efficiently.

If a tuple is deleted from relation R at the peer where it originated, we can simply remove the tuple from the local contribution relation R^ℓ, and then propagate the effects of the deletion "forward" in incremental fashion, quite similar to the program described for insertions, but with a caveat. As in *decremental view maintenance* [62], there are subtleties in determining whether to remove a derived tuple, since that tuple could be derived in an alternative way. Two general schemes exist for performing decremental maintenance (when recursion is present, as with our data exchange program of the previous section). The first is the DRed (Delete and Rederive) algorithm of [62], which removes derived tuples, then tries to see if there is an alternate derivation. A more efficient alternative, presented in [57], makes use of *data provenance* [21, 35, 58], encoded as edge relations in a graph describing which tuples are directly derived from one another, to determine when a tuple is no longer derivable from local contributions.

Deleting from a Different Peer

When a tuple is deleted from a peer other than its origin, we must propagate the effects to the local contribution relation(s) of the tuple's originating peers, in a manner analogous to view update. More precisely, we want to derive a set of updates over local contribution relations that *perform* the update requested on the target peer:

Definition 5.3.1 (Performs). *Let $\langle I, J \rangle$ be the canonical universal solution and Y^- be a set of tuples of J (i.e., peer relations) to be deleted. Let Δ^- be a set of deletions over I (i.e., local contribution relations) and let $\langle I - \Delta^-, J' \rangle$ be the canonical universal solution. We say that Δ^-* performs *Y^- iff $J' \cap Y^- = \emptyset$.*

This generalizes a definition by Dayal and Bernstein [38] to canonical universal solutions in data exchange. As with view update, there may be multiple ways to perform a target deletion. For example, if the LHS of the mapping involves a join, the desired effect may be achieved by deleting tuples from either (or both) of the relations in the join.[5] We now discuss how an administrator may specify *policies* for performing the updates. We assume that an administrator may wish to manage, or even override, default behaviors. In the next section we consider side effects and how to ensure updates do not produce them. However, we note that in certain settings with many interacting mappings, the administrator may be willing to allow side effects.

[5] We only consider options where deletions are accomplished by removing tuples, as in [38] and unlike [71].

Update Policies. We specify update policies as annotations on mappings: if an atom for relation R on one side of a mapping is annotated with *, this means that if a tuple in the opposite side of the mapping is deleted, then any tuples from R, as well as its corresponding local contribution relation R^l, should be deleted. An annotated version of m_2 from Example 16 is:

$$(m_2) \qquad \exists i \; B(i,n) \wedge {}^\star S(i,c) \leftrightarrow {}^\star U(n,c)$$

If a tuple is deleted from U, we delete any tuples of S from which it can be derived. Similarly, deleting B and/or S tuples results in a deletion of U tuples, thanks to the update policy in the opposite direction. In some cases, the composition of update policies may cause cascading deletions: e.g., deleting from U as above may trigger further deletions from S. We can show [69] that any update policy of a bidirectional mapping for which there is at least one atom in each side that is annotated with *, is guaranteed to perform any given set of updates.

We generate delta rules for deletion propagation **only** for the marked relations and their corresponding local contribution relations; the set of such rules for all mappings form the *update policy program*. The rules for the m_2 update policy shown above would be:

$$
\begin{array}{ll}
1 & S^-(i,c) \text{ :- } U^-(n,c), B(i,n), S(i,c) \\
2 & U^-(n,c) \text{ :- } B^-(i,n), U(n,c) \\
3 & U^-(n,c) \text{ :- } S^-(i,c), U(n,c) \\
4 & S^{\ell-}(i,c) \text{ :- } S^-(i,c), S^\ell(i,c) \\
5 & U^{\ell-}(n,c) \text{ :- } U^-(n,c), U^\ell(n,c) \\
6 & B^{\ell-}(i,n) \text{ :- } B^-(i,n), B^\ell(i,n)
\end{array}
$$

Rules 1-3 (and the delta tables U^-, B^-, S^- involved in them) are used to propagate deletions "backwards" along bidirectional mappings, specifying deletions over peer relations U, B, S, respectively. Rules 4-6 "collect" in the delta tables $U^{\ell-}, B^{\ell-}, S^{\ell-}$ the actual local contribution tuples to delete from U^ℓ, B^ℓ, S^ℓ, if such tuples exist.

Interactions among Mappings. With bidirectional mappings, a deletion over a peer relation may propagate to deletions over *multiple* local contribution relations, from both sides of the bidirectional mapping. Moreover, in certain cases tuples can be transitively derived by going back and forth through the two directions of the bidirectional mapping more than once. For instance, in Example 17, $B(x_3, c)$ and $S(x_3, a)$ were produced by applying $m_2^{\leftarrow}$ to $U(c, a)$, which in turn was derived by applying $m_2^{\rightarrow}$ to $B(3, c), S(3, a)$. The situation gets even more complex when there are multiple bidirectional mappings with relations in common: their update policies can interact. In general, computing the set of local deletions (Δ^-) necessary to perform the deletions in Y^- requires us to compute the *fixpoint of the update policy program*. The computation of the local updates using this update policy program also deletes tuples from peer relations derived "on the path" from the user deletions to the base data in local contribution relations. The update policy program helps us compute two sets of updates, given a set of user updates Y^-: Δ^- over local contribution relations and another set $Y'^- \supseteq Y^-$ over peer relations. We can compute these sets using the following algorithm:

Algorithm *PropagatePeerDeletions*

1. Run the **update policy program** (Sect. 5.3.2) on Y^- to compute $R^{\ell-}$ for each local contribution relation R^ℓ
2. For each local contribution relation R^ℓ, remove tuples in $R^{\ell-}$ from R^ℓ
3. Run the **decremental maintenance program** (Sect. 5.3.2) on the local deletions $R^{\ell-}$ computed in the previous step. For each peer relation P, this computes a set of deletions P^-; the set of all P^- is Y'^- above
4. For each peer relation P, remove tuples in P^- from P

The following example illustrates a run of this algorithm:

Example 18. *Suppose the mappings are:*

$$(m_1) \qquad {}^{\star}R(xy) \wedge S(xzw) \leftrightarrow T(xyz) \wedge {}^{\star}V(wx)$$

$$(m_2) \qquad {}^{\star}T(xyz) \leftrightarrow {}^{\star}U(xyz)$$

Then, given the local insertions indicated by $+$ below, the result of update exchange is:

R

$+$	1	1
$+$	3	2

S

$+$	1	1	4
$+$	1	2	4
	3	3	5

T

	1	1	1
	1	1	2
	3	2	3

V

	4	1
$+$	5	3

U

	1	1	1
	1	1	2
$+$	3	2	3

Suppose now that the user deletes $T(1,1,1)$ and $T(3,2,3)$. In Step 1 of the algorithm, according to the right-to-left update policy of m_1, $R(1,1)$ and $R(3,2)$ should be deleted, while according to the left-to-right update policy of m_2 $U(1,1,1)$ and $U(3,2,3)$ should be deleted. Moreover, according to the left-to-right update policy of m_1 on $R(1,1)$ and $R(3,2)$, $V(4,1)$ and $V(5,3)$ should also be deleted. After these deletions, no more tuples need to be deleted, i.e., the deletion program has reached a fixpoint. Among those deletions, in Step 2 we "apply" the ones corresponding to local insertions, namely $R(1,1), R(3,2), V(5,3), U(3,2,3)$ and propagate their effects forward, by running the maintenance program (Steps 3-4). As a result, we get the following peer relation instances.

R

S

$+$	1	1	4
$+$	1	2	4

T

V

U

Note that forward propagation of the deletions identified by the update policy resulted in the unintentional deletion of some additional tuples (namely $S(3,3,5), T(1,1,2), U(1,1,2)$). These deletions are called side effects, *and we present an algorithm that avoids them in the next section.*

5.3.3 Avoiding Side Effects at Run Time

The term *side effect* was invented in the view update literature to refer to a propagation of an update to a source, which in turn causes other, undesired effects when the contents of a view are recomputed (e.g., because multiple view tuples were derived from the same source tuple). In other words, we propagate an update *backwards* via a policy, and then its *forward* effects (via maintenance or recomputation) change tuples that were not part of the original modification. (We do not consider cascading deletions caused by multiple update policies to be side effects.)

Definition 5.3.2 (Side effects). *Let $\langle I, J \rangle$ be the canonical universal solution, where I is an instance of local contribution relations and J is an instance of peer relations. Let Y^- be a set of updates over J, and Δ^-, Y'^- be the output of the update policy program on Y^-. Let $\langle I - \Delta^-, J' \rangle$ be the canonical universal solution, then the translation that produced Δ^- is side-effect-free iff $J' = J - Y'^-$, while it has side effects iff $J' \subset J - Y'^-$.*

An administrator may wish to propagate updates only if they avoid side effects on a given instance. Previous work typically considers *static* checking, based on functional dependencies and other constraints, on whether a view can be updated without introducing side effects. We believe such checking is inappropriate for large-scale data sharing: in databases produced by non-expert users, constraints are often *under-specified*, making static checking overly pessimistic and checking statically may prevent *any* update to a view, even when some tuples may be updatable without causing side effects. Thus we allow the administrator to request detection and elimination of side effects at update-time, based on the actual contents of the database instances.

The following algorithm identifies which of the local deletions returned by the update policy cause side effects, and only applies to local contribution relations those that do not, before computing the new canonical solution.

Algorithm *PropagatePeerDeletionsWithoutSideEffects*

1. Run the **update policy program** on Y^- to compute $R^{\ell-}$ for each local contribution relation R^ℓ and P^- for each peer relation P (but do not modify R, P)
2. Run the **decremental maintenance program** on the local deletions $R^{\ell-}$, to get sets of peer deletions P^d for every peer relation P (do not apply updates to the peer relations)
3. For each peer relation P, set $P^{se} := P^d - P^-$ and $P^- := \emptyset$. These are the **side effects** on P
4. For each tuple $t \in P^{se}$, compute its **lineage**, i.e., the set of all tuples in local contribution relations involved in some derivation of t.[6] For each local contribution relation R^ℓ, collect all such sources of side effects in a relation R^ℓ_{inv}
5. For each local contribution relation R^ℓ, set $R^{\ell-} := R^{\ell-} - R^\ell_{inv}$. These are the side effect-free source updates
6. For each local contribution relation R^ℓ, remove tuples in $R^{\ell-}$ from R^ℓ
7. Run the **decremental maintenance program** on the local deletions $R^{\ell-}$ computed in the previous step. For each peer relation P, this computes deletions P^-
8. For each peer relation P, remove tuples in P^- from P

The algorithm applies deletions of local tuples identified by the update policy program, when these do not cause side effects (tested in Line 3); it can additionally be relaxed to consider cases where some peers tolerate side effects and others do not. Importantly, each of the steps of the algorithm above (as well as the one in the previous section) can be expressed as a datalog-like program, which can be translated to SQL queries that can be evaluated over an RDBMS.

The following example illustrates an application of this algorithm.

[6] An algorithm for this was sketched in Chapter 5, as part of **decremental maintenance**. The main idea is to traverse mappings backwards, starting from each side-effecting tuple. In Chapter 7 we will see that this can also be expressed as a provenance query.

Example 19. *Consider again the mappings and instances in Example 18, and suppose we want to avoid side effects on T, but don't mind side effects on other peer relations. As in Example 18, suppose also that the user wants to delete $T(1, 1, 1)$ and $T(3, 2, 3)$. In Step 1, according to the right-to-left update policy of m_1, $R(1, 1)$ and $R(3, 2)$ need to be deleted, while according to the left-to-right update policy of m_2 $U(1, 1, 1)$ and $U(3, 2, 3)$ should be deleted. Moreover, according to the left-to-right update policy of m_1 on $R(1, 1)$ and $R(3, 2)$, $V(4, 1)$ and $V(5, 3)$ should also be deleted. After these deletions, no more tuples need to be deleted, i.e., the deletion program has reached a fixpoint. Let D be the set of the tuples in local contribution relations among these, i.e., $D = \{R(1, 1), R(3, 2), V(5, 3), U(3, 2, 3)\}$. At this point, instead of actually deleting all these tuples, we use the decremental maintenance algorithm in Step 2 to identify which of them cause side effects to the relations for which we want to avoid them in Step 3 - $T(1, 1, 2)$, in this case. Then, in Step 4 we trace the tuples in local contribution relations from which $T(1, 1, 2)$ was derived, namely $D' = \{R(1, 1), S(1, 2, 4)\}$. Consequently, in order to ensure that there are no side effects on T, we only delete (Steps 5-6) tuples in $D - D' = \{R(3, 2), V(5, 3), U(3, 2, 3)\}$. Finally, in Steps 7-8 we propagate the effects of these source deletions forward, as in Example 18, to obtain the following peer relation instances:*

R

+	1	1

S

+	1	1	4
+	1	2	4

T

	1	1	1
	1	1	2

V

	4	1

U

+	1	1	1
+	1	1	2

Observe that the deletion of $T(1, 1, 1)$ was not performed, as a result of not deleting $R(1, 1)$ to avoid side effects on T, while the deletion of $T(3, 2, 3)$ was performed, since that was possible without causing side effects on T.

We note that our treatment of side effects is different from that in the *deletion minimization problem* [22], where the authors show that finding a set of source deletions that perform a view deletion and cause the *minimal* of side effects is NP-hard. In our case we are only interested in propagations with no side effects at all and, as we showed in the example above, it is possible that some deletions are not performed, if this we cannot be done without side effects.

In the next chapter we describe the implementation of our ORCHESTRA prototype, which includes the algorithms presented above, as well as the ones for unidirectional update exchange from Chapter 5. We also compare the performance of unidirectional vs. bidirectional update exchange, as well as the overhead of avoiding side effects during deletion propagation.

Chapter 6

The ORCHESTRA Prototype Implementation

We have implemented unidirectional and bidirectional update exchange, as defined in Chapters 5 and 5.3, in the ORCHESTRA system, the first real-world implementation of a CDSS. In particular, we have developed the following components in order to perform update exchange *incrementally*:

- *Wrappers* connect to RDBMS data sources, obtain logs of their updates, and apply updates to them.
- A global *update store*, where all peers publish their updates, so that other peers can obtain them during update exchange.
- *Auxiliary storage* holds and indexes provenance tables for peer instances.
- The *update exchange engine* performs the actual update exchange operation, given schemas, mappings, and trust conditions.

ORCHESTRA performs two different but closely related tasks. When a peer first joins the system, the system *imports* its existing RDBMS instance and logs, and creates the necessary relations and indices to maintain provenance. Later, when the peer actually wishes to share data, we (1) obtain its recent updates via a wrapper and publish these to the CDSS, by inserting them to the update store, and (2) perform the remaining steps of update exchange in an incremental way. To review, these are update translation, provenance recomputation, and application of trust conditions. The final resulting instance is written to ORCHESTRA's auxiliary storage, and a derived version of it is recorded in the peer's local RDBMS.

The majority of our technical contributions were in the update exchange engine, which comprises around 50,000 lines of Java code, in about 500 classes. A demonstration of ORCHESTRA has been presented in [60] and we plan to make it available under an open-source license. ORCHESTRA has also been a valuable research platform, for developing techniques to learn trust levels from user feedback [94] as well as for exploring and querying provenance, as explained in Chapter 7. Last but not least, ORCHESTRA has been deployed in pPOD [90], a bioinformatics system for sharing of phylogenetic data intended to become a production system as a part of NSF's AToL (Assembling the Tree of Life) program, that aims to reconstruct the evolutionary origins of all living things.

Figure 6.1 illustrates the main steps involved in performing update exchange at each peer in ORCHESTRA. An RDBMS is used as the central component, to store data and updates, as well as the provenance of their propagation. The engine parses tgds and creates rule-based (datalog^{sk}) programs to perform update exchange operations, as described in Chapter 5. Individual rules are then translated into SQL queries that can be evaluated over the underlying RDBMS, using data and provenance stored in it, as well as updates retrieved from the global update store. The control-logic of the datalog^{sk} programs is handled by the fixpoint engine that we implemented in Java, and together with the RDBMS comprises a pluggable back-end for ORCHESTRA.

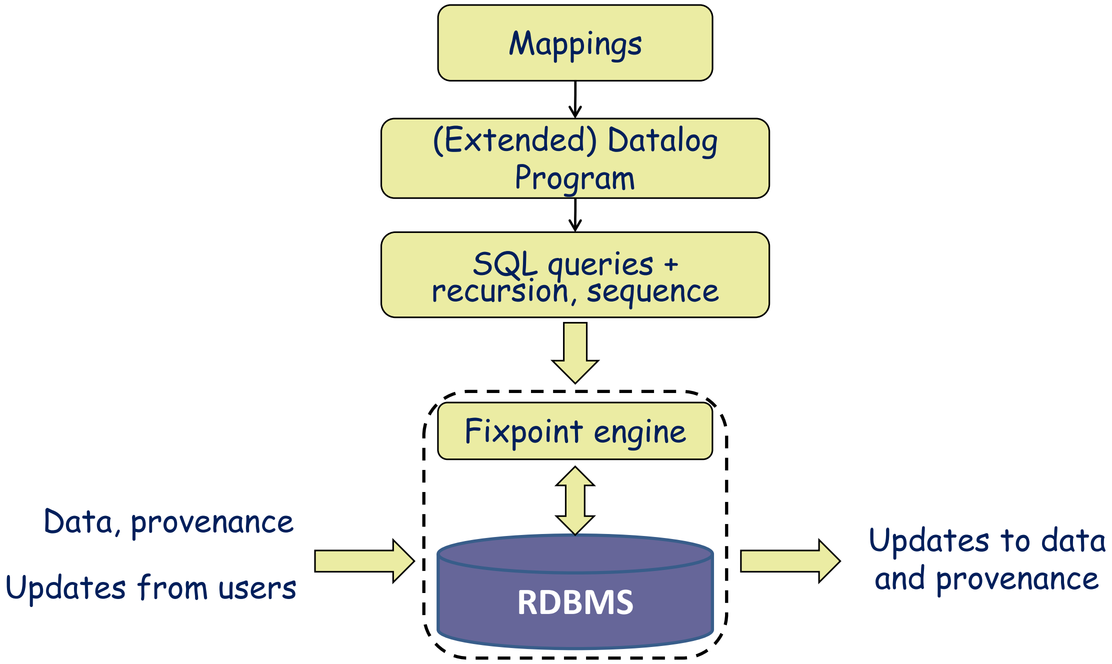

Mappings
(Extended) Datalog Program
SQL queries + recursion, sequence
Fixpoint engine
RDBMS
Data, provenance
Updates from users
Updates to data and provenance

Figure 6.1: Steps in performing update exchange over RDBMS

6.1 Data and Provenance Storage

Our system receives as its input a "real" scientific database. In order to be able to handle labeled nulls, introduced during update exchange through mappings with existential variables, we extend the relations in the schema of the input database. In particular, for every attribute A that can "receive" a labeled null, we introduce an additional integer attribute A_LN to the schema of its relation. Moreover, we associate each Skolem term (i.e., combination of a Skolem function name and a set of parameter values) with a unique integer. Then, if a tuple has a regular value for A, we enter that value in A and fill A_LN with a special integer value. However, if the tuple has a labeled null in the corresponding position, we put a special value – which depends on the data type of A – in A and the appropriate integer value in the Skolem term. By using special values instead of regular NULLs we can transparently compare tuples with and/or without labeled null values, by converting each comparison of the form `(X.A = Y.A)` to a pair of comparisons `(X.A = Y.A AND X.A_LN = Y.A_LN)`.

We also extract the updates from the user database by scanning the DBs edit logs (offline), while what the users see in the end is in terms of their original schema. This way, the (non-ORCHESTRA-admin) user still only needs to interact with their favourite RDBMS and can completely ignore the existence of ORCHESTRA in the middle, and only the institute's ORCHESTRA admin needs to know more.

For the storage of provenance relations we considered several alternatives along the lines of 5.1.2. Our first approach was to convert mapping tgds into rules with a single atom in the head, and create one provenance relation for each such rule (i.e., essentially one relation for each pair of mapping tgd and relation in its RHS). However, we found that we needed to pursue strategies for reducing the number of relations (and thus the number of operations performed by the query engine). For this reason, we experimented with the so-called "outer union" approach of [24] — which allows us to union together the output of multiple rules even if they have different arity.

However, we found that in practice an alternate approach, which we term the *composite mapping* table, performed better. Rather than creating a separate provenance table for each source relation, we instead create a single provenance table per mapping tgd, even if the tgd has multiple atoms on its RHS. This approach essentially takes advantage of the fact that tuples produced by different atoms in the head of the same mapping tgd have common provenance derivations. Thus, instead of redundantly storing such common provenance information in separate relations for each source relation, it stores the shared provenance once.

In order to make processing of provenance tables more efficient, we also needed to define appropriate indexes on them. For this purpose, we define indexes/keys for these tables, as the union of all attributes in the keys of the relations in the body of the mapping corresponding to these provenance relations.

6.2 Creating datalog$^{\text{sk}}$ Programs for Update Exchange

The update exchange engine is essentially a middleware layer. It is configured with the peer schema mappings specified as tgds, which it converts into datalog$^{\text{sk}}$ programs (cf. Chapter 5) over the stored and temporary source and provenance relations. In particular, we have implemented translation modules that:

- Introduce provenance relations in mappings, by converting each mapping tgd into a pair of mappings, a) from the LHS of the original mapping to the provenance relation and b) from that to the RHS of the original mapping, and convert the resulting mappings to inverse rules, where the existential variables in the RHS are replaced by appropriate Skolem functions.
- Based on these inverse rules, create a datalog$^{\text{sk}}$ program consisting of delta rules, as explained in Section 5.2, that can be used to perform incremental insertion propagation. These delta rules employ some of the temporary relations, which represent insertions in the corresponding relations, to compute some intermediate results of the propagation, before finally adding their contents to the original stored relations. The resulting program is generally recursive but its fixpoint is the result of performing the incremental insertion propagation.
- Moreover, create a datalog$^{\text{sk}}$ program to perform incremental deletion propagation, according to the algorithm described in Section 5.2. Part of this program involves delta rules that involve temporary deletion relations, for the original and the provenance relations. It also involves the derivability testing

algorithm of Section 5.1.3, that uses some more temporary tables to explore the provenance of tuples that may need to be deleted, in order to determine if some alternative derivations still exist for them. The derivation program requires a stratified form of recursion, where a part of it iterates until reaching a fixpoint, before another part is evaluated (also until reaching a fixpoint). This derivability program is nested inside another recursive program, which also involves this form of stratification between its components. Finally, in contrast to common datalog evaluation, only some of the rules of the deletion propagation program should not be considered when checking whether a fixpoint has been reached. This is necessary to achieve termination, e.g., if in one iteration all of the tuples that were checked for derivability were determined to still be derivable; in this case the deletion propagation is complete, but the rules of the derivability testing program have returned non-zero answers.

- For settings with bidirectional mappings, create datalog^{sk} programs to perform bidirectional propagation algorithms *PropagatePeerDeletions* and *PropagatePeerDeletionsWithoutSideEffects*. The rules of these programs involve some more temporary relations, e.g., to store tuples that we want to determine whether they can be deleted without causing side effects, or to identify side effects, by comparing the effects of deleting a source tuple with the set of deletions specified by the the user and the update policies. These programs also require the complex control-flow capabilities explained for the deletion propagation program above.

The programs produced by these translation modules can be used to perform the corresponding update exchange operations, if executed over an query processing engine with the required capabilities. In the next section we illustrate how an RDBMS can be used as the central component of such an engine, combined with a Java layer that provides the advanced control flow capabilities required for the processing of datalog^{sk} programs.

6.3 RDBMS-based Implementation

Using an off-the-shelf RDBMS as a basis for the update exchange component is attractive for several reasons: (1) each peer is already running an RDBMS, and hence there is a resource that might be tapped; (2) much of the data required to perform maintenance is already located at the peer (e.g., its existing instance), and the total number of recent updates is likely to be relatively small; (3) existing relational engines are highly optimized and tuned.

However, there are several ways in which our requirements go beyond the capabilities of a typical RDBMS. The first is that the datalog^{sk} rules are often mutually recursive, whereas commercial engines such as DB2 and Oracle only support *linearly* recursive queries. The second is that our incremental deletion algorithm involves a series of datalog^{sk} computations and updates, which must themselves be stratified and repeated in sequence. Finally, RDBMSs do not directly support Skolem functions or labeled nulls. For this reason, we implemented the control logic of a recursive datalog engine in Java and evaluate individual "rules" as SQL queries through JDBC, over several different commercial RDBMSs.

Hence, we take a datalog^{sk} program and compile it to a combination of Java objects and SQL code. The control flow and logic for computing fixpoints is in Java,[1] using JDBC to execute SQL queries and updates on the RDBMS (which is typically on the same machine). To achieve good performance, we make heavy use of prepared statements and keep the data entirely in RDBMS tables. Moreover, we use semi-naive datalog evaluation, so that in every iteration we only use data from the latest one (instead of using all previous ones, as in naive datalog evaluation).

[1] We chose to use Java over SQL rather than user-defined functions for portability across different RDBMSs.

To avoid the cost of packaging and returning query results through JDBC to the Java layer, we store results into temporary tables, and the Java code only receives the number of tuples returned, which it uses to detect fixpoint. Moreover, we exploit key information to optimize the translation of rules into SQL queries. For example, even if a rule joins two atoms on all attributes, if e.g., the keys of the two corresponding relations are among the equated attributes we suppress from the SQL query the redundant equality conditions between non-key attributes.

For Skolem functions we create one DB2 user-defined function for each possible arity (i.e., number of parameters). Because of RDBMS limitations we create these functions statically, without looking at a particular set of mappings. In order to avoid creating too many functions, we cast all input parameters to strings and create all functions up to a reasonable maximum arity (e.g., if we know what the maximum possible number of attributes in each relation and relations in the body of a mapping are). These user-defined functions use a persistent store in order to "remember" Skolem terms that have been created before, and avoid creating new values for such terms, as well as to guarantee that unique values are given to new Skolem terms.

As part of the process of building the system, we experimented with several different DBMSs; the one with the best combination of performance and consistency was DB2, so we report those numbers in Section 6.4. Even for DB2, we found that achieving satisfactory performance was challenging for several reasons.

First, update translation requires many round-trips between the Java and SQL layers. While this might be reduced by using the stored procedure capabilities available in one of the DBMSs, there is still a fundamental impedance mismatch. Moreover, even for settings of tens of peers, update exchange ends up producing hundreds of tables and the update exchange programs comprise thousands of rules. Combined with the need to iterate until a fixpoint is reached, executing these programs essentially involves evaluating a very large numbers of queries over relatively few tuples, while, unfortunately, conventional DBMSs are optimized for few queries over large numbers of tuples. As a result, getting this implementation functional and usable for such settings involved a great deal of experimentation with numerous parameters of the RDBMS configuration. More challenging was the fact the general-purpose query optimizer would occasionally chose poor plans in executing the rules, partly because of the large number of queries and the fact that the cardinality of many relations varies largely during update exchange, as updates are propagated through mappings. In fact, since these rules are produced by the automatic translation, it would have been fairly straightforward to determine efficient evaluation strategies and heuristics, that are common for all mappings. Unfortunately, there was no way to influence the choice of plans by the optimizer directly, e.g., by pre-selecting appropriate query plans to be used. However, through extensive tuning and experimentation with the DB2 configuration we were able to achieve good and consistent performance, that satisfies the requirements of our target usage model, as we discuss in Section 6.4.

6.4 Experimental Evaluation

In this section, we investigate the performance of our incremental update exchange strategies, answering several questions. First, we consider the impact of different strategies for computing the effects of updates: complete recomputation from an updated set of base tuples, the DRed strategy of [61], and our new provenance-based propagation strategy. (The latter two only differ with respect to propagation of *deletions* and use the same insertion propagation techniques.) Next we consider scalability with respect to several factors: number of peers, number of base tuples, and relative size of attributes in the tuples. Finally, we study the effects of mappings: what happens as we vary the number of mappings among a set of peers, change from unidirectional to bidirectional mappings, and change our deletion policies.

6.4.1 Experimental Setup

We conducted all experiments on our full ORCHESTRA implementation, which consists of the previously-discussed 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 6GB of RAM.

Experimental CDSS Configurations

To test the system at scale, we developed a synthetic workload generator based on bioinformatics data and schemas to evaluate performance, by creating different configurations of peer schemas, mappings, and updates. The workload generator takes as input a single universal relation based on the SWISS-PROT protein database [7], which has 25 attributes.

For each peer, it first chooses a random number i of relations to generate at the peer, where i is chosen with Zipfian skew from an input parameter representing the maximum number of schemas. It then similarly chooses j attributes from SWISS-PROT's schema, partitions these attributes across the i relations, and adds a shared key attribute to preserve losslessness. Next, full mappings[2] are created among the relations via their shared attributes: a mapping source is the join of all relations at a peer, and the target is the join of all relations with these attributes in the target peer. Typically for a set of n nodes, we generate a spanning tree containing $n - 1$ edges. For experiments where we increase the fan-in or fan-out (Section 6.4.6) we start with this spanning tree and add edges until we satisfy the fan-in and fan-out constraints. We emphasize that this was a convenient way to

[2] A full mapping is one that preserves all source attributes in the target.

synthesize mappings; no aspect of our architecture or algorithms depends on this structure.

Finally, we generate fresh insertions by sampling from the SWISS-PROT database and generating a new key by which the partitions may be rejoined. We generate deletions similarly by sampling among our insertions. The SWISS-PROT database, like many bioinformatics databases, has many large strings, meaning that each tuple is quite large. Clearly, the size of tuples (and attributes) makes a significant difference in performance, as does, e.g., whether non-key attributes are stored as CLOBs or as strings.

To study a range of different possible input workloads, we conduct most of our experiments with two different sizes for the initial database instance (2,000 and 10,000 original insertions over the base tables). We also use two different sizes for the tuples: "integer," where we substituted integer hash values for each string, providing the properties of CLOBs or other small attribute types; and "string", where we use 1KB VARCHAR strings to directly encode the data from SWISS-PROT. We vary the number of peers and mappings, as well as their properties. Apart from our first experiment in the next section (where we consider the effects of tuples with multiple derivations), we keep a disjoint set of tuples in the base instances of our peers.

Terminology

We refer to the *base size* of a workload to mean the number of SWISS-PROT entries inserted initially into each peer's local tables and propagated to the other peers before the experiment is run. Thus, in a setting of 10 peers, a base size 2000 begins with 20000 SWISS-PROT entries, but as these are normalized into each of the peers' schemas, this results in 176,000 total tuples in peer and mapping relations, for a setting with one incoming and one outgoing mapping per peer. When we discuss *update sizes*, we mean the number of SWISS-PROT entries per peer to be updated (e.g., 200 deletions in the setting above translates to 2000 SWISS-PROT entries, or about 15000 tuples total).

Experimental Methodology

Each individual experiment was repeated seven times, with the final number obtained by discarding the best and worst results — to ensure that the final result is not affected by cold caches — and computing the average of the remaining five numbers.

6.4.2 Incremental vs. Complete Recomputation

Our first experiment investigates where our incremental maintenance strategy provides benefits, when compared with simply recomputing all of the peers' instances from the base data. The interesting case here is deletion (since incremental insertion obviously requires a subset of the work of total recomputation). Moreover, our rationale for developing a new incremental deletion algorithm, as opposed to simply using the DRed algorithm, was that our algorithm should provide superior performance to DRed in these settings.

Figure 6.2 shows the relative performance of recomputing from the base data after it is updated ("Non-incremental"), our incremental deletion algorithm, and the DRed algorithm, for a setting of 10 peers, full mappings, and 10,000 base tuples in each peer. One important differentiation between DRed and our algorithm is how they perform when a tuple is derivable multiple ways. Hence in this set of experiments we allow each tuple to exist in multiple base instances (as opposed to all remaining experiments). As a result, in this case some of the derived tuples have *multiple alternative derivations*, and even though some of the source tuples get deleted, the output tuples may still be derivable from other sources. We note several key facts: first, our deletion algorithm is faster than a full recomputation even

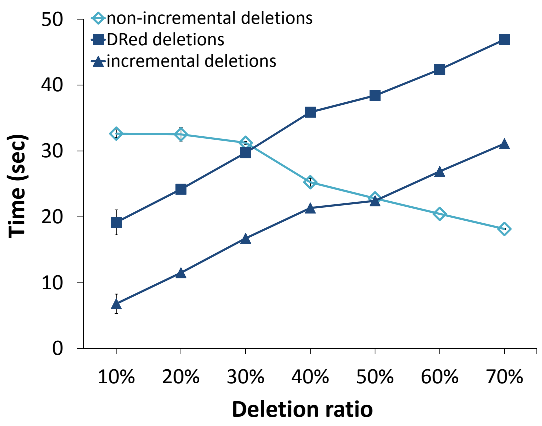


Figure 6.2: Deletion alternatives

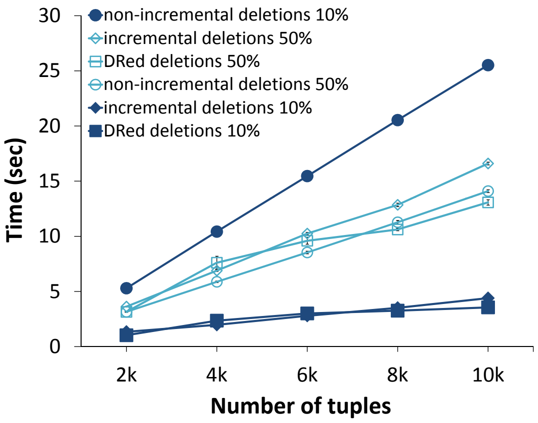


Figure 6.3: Scalability of deletion alternatives

when deleting up to approximately 50% of the instance. Second, in comparison DRed performs worse in all measured settings — in fact, only outperforming a recomputation for delete settings of under 30%. One reason for these results is that, when a tuple t has alternative derivations and some other tuples are derived from it, DRed transitively deletes all of them and then rederives them. In contrast, our algorithm identifies the existence of alternative derivations for t and thus avoids deleting t, as well as other tuples transitively derived from it. Moreover, our algorithm does the majority of its computation while *only* using the keys of tuples (to trace derivations), whereas DRed (which does reinsertion) needs to use the complete tuples.

Figure 6.3 shows the relative performance of these algorithms, for a setting of 10 peers with *disjoint* data at each peer, for different base sizes when deleting either 10% or 50% of the base data at each peer. We note that in this case our algorithm and DRed perform and scale very similarly, even though this case is ideal for DRed, because it never needs to rederive any tuples. Moreover, both algorithms vastly outperform a full recomputation when deleting 10% of the base data at each peer, and perform similarly to it when deleting 50% of those data.

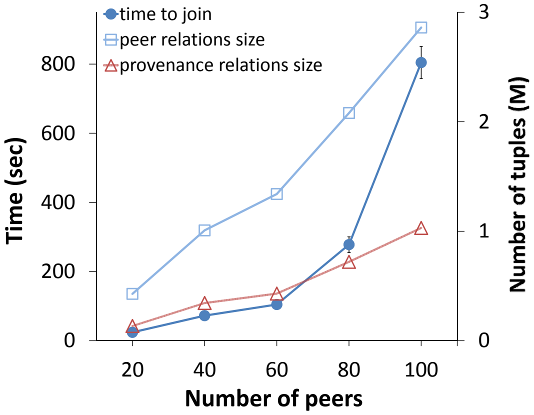


Figure 6.4: Time to join and relation instance sizes (integer dataset - 2k base size)

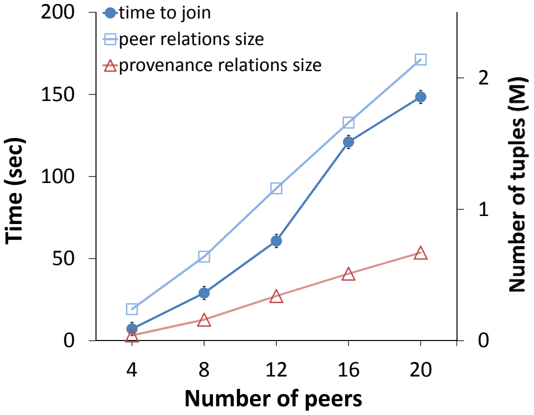


Figure 6.5: Time to join and relation instance sizes (integer dataset - 10k base size)

6.4.3 Cost and Overhead of Computing Instances

In our second experiment, we look at how the algorithms scale with respect to the number of peers, the complexity of the tuples, and the base size. Our parameters of interest include both running times and storage overhead.

In this set of experiments (and all subsequent ones), we assume a setting with $n - 1$ mappings among n peers (in the form of a spanning tree, generated as described above). In general, the spanning tree is somewhat irregular, but we can expect the size of the joint instances across the peers will grow at an approximately quadratic rate.

We compare the instance sizes, provenance overhead, and running for our different configurations: Figure 6.4 shows the results for 2,000 tuples of integer data; Figure 6.5 for a *larger* set of base instances with 10,000 tuples of integer data; and Figure 6.6 shows what happens if we increase the *tuple size* (using 2,000 tuples of large strings) instead of the number of tuples. The left y-axis shows time and is used for the time-to-join plot. The right y-axis indicates the number of tuples and is used for the other line plots.

We see that the instance size indeed grows roughly quadratically in all cases (with some minor variance). The cost of computing these instances grows roughly correspondingly with the number of tuples that must be created. We observe that the size of the provenance relations (in terms of tuples) is relatively small compared with the total instance size, averaging around 25% of the total number of tuples in the relations. In Section 6.4.6 we examine the overhead for storing provenance in settings with more mappings/peer. The size of a tuple in a provenance relation is roughly the same as that of a typical integer-dataset relation tuple, and significantly smaller than an average string-dataset tuple: provenance relations do not contain copies of attributes from both the source and target tuples — but only of the *keys* of both relations.

Overall, the 2,000-tuple relation cases show running times on the order of 13-15 minutes to materialize all instances in a 100-peer configuration. We feel this is a reasonable startup cost for a system that will be initialized once and incrementally maintained during off-hours. For the 10,000-tuple instance, as one would expect, running times increase by approximately by a factor of 5 from the equivalent 2,000-tuple case. Here we only show scalability to 20 peers because our original SWISS-PROT dataset did not contain enough unique tuples for us to create non-overlapping base instances for more than 20 peers. In any case, it is evident that, even with these large sizes, we can scale well beyond 20 peers (since running times are in the 3 minute range). We note that most real bioinformatics data sharing confederations are much smaller than 20 peers today.

6.4.4 Incremental Update Exchange vs. Number of Peers

Next, we look at how incremental insertion and deletion scale with the number of peers. Figure 6.7 begins with the integer/2,000-tuple base instances. We see that insertion and deletion in these cases (with single derivations for every tuple) is often significantly faster than the time necessary to recompute from scratch

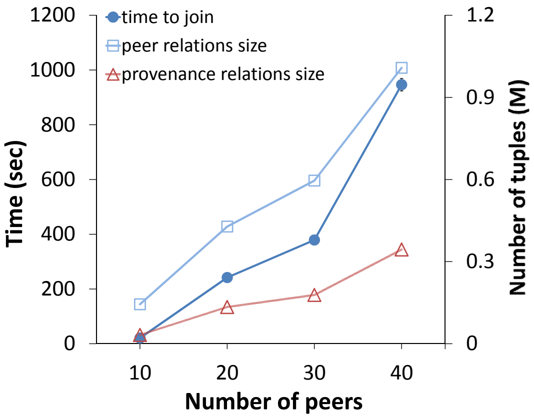


Figure 6.6: Time to join and relation instance sizes (string dataset - 2k base size)

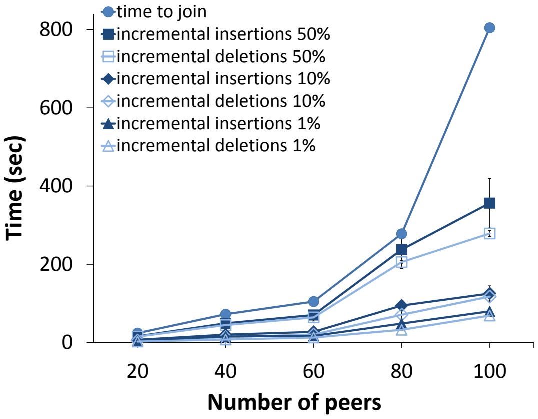


Figure 6.7: Scalability of incremental algorithms (integer dataset - 2k base size)

(equivalent, for a small update ratio, to the time to join the system, shown as the top-most line plot). The performance of insertions and deletions is quite comparable, with deletions running slightly faster because they reduce the table sizes used in the computation as they are applied (whereas insertions increase the size of the tables). Given that our target application domain is one where updates are only occasionally propagated (likely during off-hours), performance seems entirely acceptable: we can propagate updates to 50% of the tuples in 100 peers with running times in the range of 6 minutes.

Increasing the size of the base data to 10,000 tuples (Figure 6.8) increases the running times by approximately a factor of 5 — as with our previous time-to-join experiment. Again, we only show scalability to 20 peers because our original SWISS-PROT dataset did not contain enough unique tuples for us to create non-overlapping base instances for more than 20 peers.

Our experiments to this point considered data with “small” attributes — integers or CLOBs. If we consider large attributes (long VARCHARs), the cost of insertions goes dramatically up (since insertions must contain *all attributes*), whereas the cost of deletions can remain similar (since deletions can be done purely with

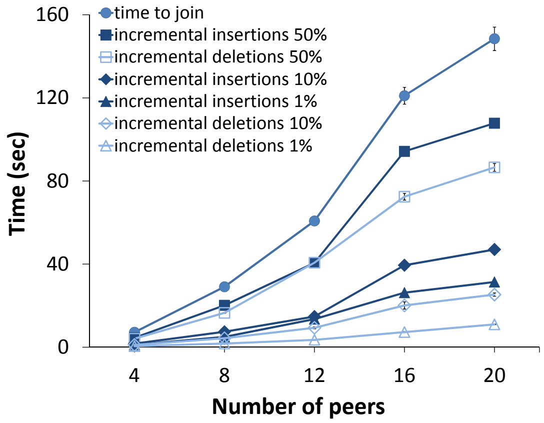


Figure 6.8: Scalability of incremental algorithms (integer dataset - 10k base size)

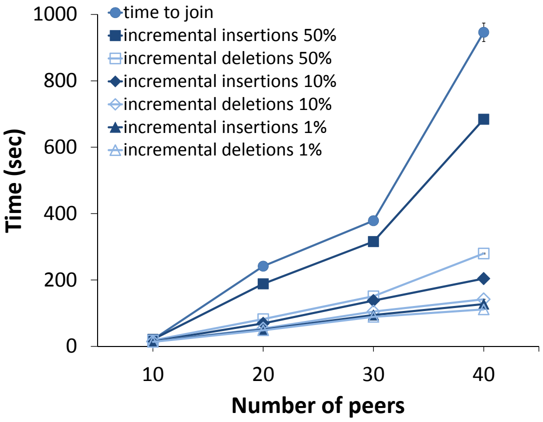


Figure 6.9: Scalability of incremental algorithms (string dataset - 2k base size)

key attributes). We see this validated in Figure 6.9, where the gap between deletion and insertion cost widens.

6.4.5 Performance vs. Base Data Size

The previous experiments studied the effects of increasing the number of peers. We now see what happens if we fix the number of peers (10 peers with 9 mappings among them) and vary the base size. Integer data is shown in Figure 6.10, and string data in Figure 6.11. We observe that in both cases the running times of the incremental algorithms scale at a slightly-more-than-linear rate with the number of tuples. Moreover, propagation of incremental deletions is generally faster than that of equal insertion loads. The difference is more extreme for the string dataset, due to the use of keys by the deletion algorithm, as explained above.

6.4.6 Performance vs. Mappings

Our next experiment focuses on the impact of adding more mappings among a fixed number of peers. More mappings result in more individual datalog rules in the update exchange process, more alternative derivations and hence provenance

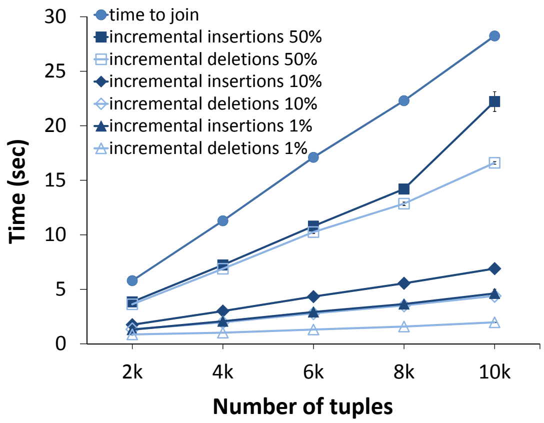


Figure 6.10: Scalability for increasing base sizes (integer dataset)

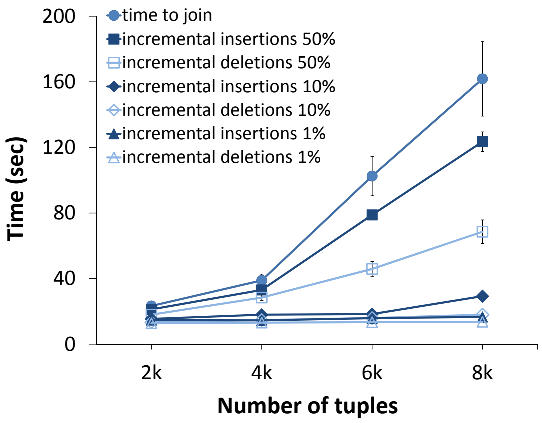


Figure 6.11: Scalability for increasing base sizes (string dataset)

values, and often more overall tuples in the resulting instance. Hence we expect the cost of computation to go up significantly.

Here we focused simply on the cost of joining the system, computing peer instances from scratch, for a case where each base instance contains 2,000 tuples with integer data. As Figure 6.12 shows, we varied the fan-in and fan-out values from the default of 1 all the way up to 3 — this results in 9, 18, and 27 mappings, respectively, with the mappings distributed uniformly among the 10 peers. We see that, especially for more than 10 peers, increasing the fan-in dramatically increases the rate of growth of the computation times. However, for up to 20 peers and a fan-in/fan-out of 2 mappings the running times are quite reasonable.

Moreover, we measured the impact of adding more mappings to the size of provenance and peer relations, with a particular interest in the relative overhead incurred by storing provenance. As shown in Figure 6.13, the number of tuples in provenance relations is around 25% of that in peer relations for a fan-in/fan-out of 1, but increases to more than 50% for fan-in/fan-out of 2, while for a fan-in/fan-out of 3 the number of tuples in peer relations and provenance relations is about equal. This is expected, since the addition of more mappings results in more alternative derivations for each tuples that need to be recorded in provenance relations.

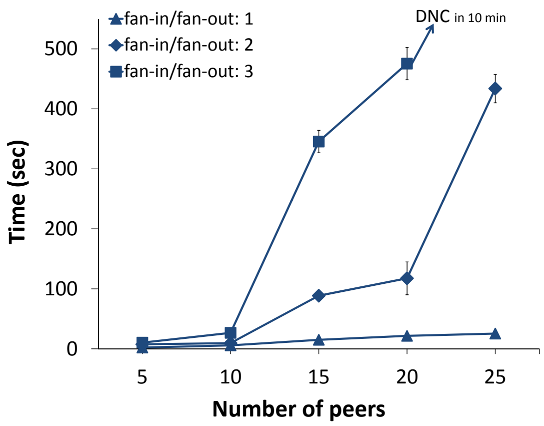


Figure 6.12: Effect of fan-in/fan-out of mapping graph in instance size and performance

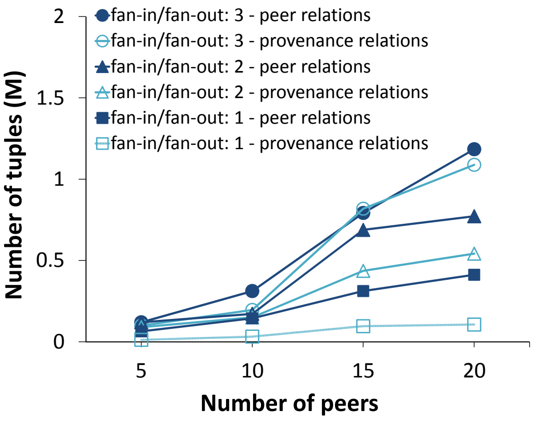


Figure 6.13: Effect of fan-in/fan-out in storage overhead of provenance relations

6.4.7 Bidirectional Mappings

We now investigate the performance of bidirectional update exchange in a CDSS. First, we compare bidirectional and unidirectional update exchange properties, for the same number of peers. Then we compare preliminary implementations of our deletion propagation algorithms, with and without detection of side effects. For both experiments, each peer had a base size of 2,000 tuples from the integer dataset — different for each peer — in its local contributions tables. We randomly generated graphs of unidirectional mappings among the peers with fan-in/fan-out 1, and converted those mappings to bidirectional, to test the bidirectional update exchange algorithms (essentially resulting to a fan-in/fan-out of 2 in this case).

Cost of Unidirectional vs. Bidirectional Mappings

We first consider the effects of unidirectional mappings vs. bidirectional ones: in general, bidirectional mappings should result in larger data instances (since all data will propagate to all peers) and longer computation times. Figure 6.14 shows the total size of peer instances after propagating 2,000 base tuples (from the integer dataset) inserted at each peer, measured in millions of tuples (scale on the

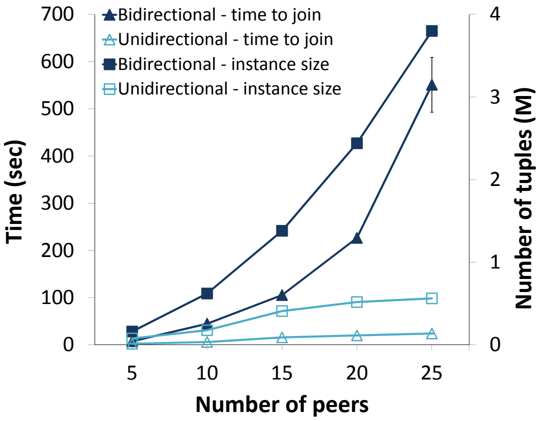


Figure 6.14: Solution size and computation time for unidirectional and bidirectional mappings

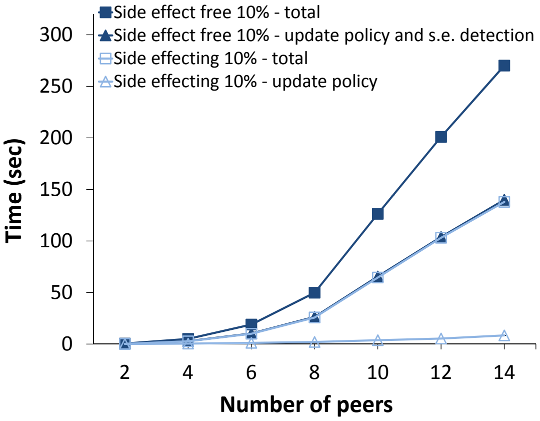


Figure 6.15: Propagation time for bidirectional deletion policies

right y-axis), as well as the total time for propagating these insertions (scale on the left y-axis). As we scale the CDSS to increasingly larger sizes, we see that for unidirectional mappings, the total instance sizes and running times grow at an approximately linear rate; whereas for bidirectional mappings, the number of tuples and the computation time grow quadratically. This mirrors our expectations, given the topologies and the amount of data exchanged. Moreover, the setting with bidirectional mappings essentially corresponds to one with unidirectional mappings with a fan-in/fan-out of 2. Similar performance characteristics are indeed seen between Figure 6.14 and the plot in Figure 6.12 for the fan-in/fan-out of 2 . However, bidirectional mappings incur an additional overhead, since they essentially propagate all base data to all other peers. We note that running times of several minutes are tolerable for offline batch operations, which are the emphasis in the CDSS. However, we also observe that the slower running times for bidirectional mappings suggest opportunities for optimization and indexing.

Deletion Policies and Side Effect Detection

Our first experiment analyzed how bidirectional mappings increase the complexity of computation and the size of instances. If we consider deletion in a bidirectional context, in fact the cost of deletion can be divided into the time for applying the update policy and removing side effects followed by *forward propagation* of source tuples deleted by the policy.

For this experiment we start by deleting 10% of the SWISS-PROT entries at every peer (i.e., 200 entries per peer). In Figure 6.15 we show the total time for incremental maintenance, and separately plot the time for applying the update policy (the difference between the two is the forward propagation cost). We consider settings where we do not attempt to avoid side effects, versus those where we remove side effecting deletions.

Clearly, it is more expensive to use the side effect-free strategy, since it must do additional work. We observe that the update policy and side effect detection phase is the dominant cost in side effect-free maintenance, whereas forwards propagation represents almost the entire cost in the side effecting mode. For side effecting propagation, the costs are still acceptable for 14 peers, although they grow quite fast over 10 peers. For the side effect-free mode, the amount of data and the mapping complexity results in a more expensive operation for settings with more than 10 peers. Again, we believe this suggests opportunities for future research on optimization.

6.4.8 Overall Conclusions

Our final conclusion from these experiments is that the CDSS approach is amenable to incremental maintenance of both data and provenance. Our algorithms scale to tens and possibly low 100s of peers, depending on the data set sizes and the complexity of the mappings. This performance is easily adequate for relatively small data sharing confederations, as we are targeting for bioinformatics domains. We also believe there is opportunity to further improve these running times by developing custom optimization techniques, which we hope to do as future work.

Chapter 7

ProQL: a Query Language for Provenance

As we explained in previous chapters, we maintain *data provenance* in a CDSS with several objectives in mind. First, CDSS administrators want to be able to express provenance-based trust policies. Second, in many settings scientific users actually find it useful to *visualize* provenance (as a graph showing the derivations) in order to understand how it came to be in their local instance. Third, internally the ORCHESTRA incremental update propagation algorithms make use of provenance to boost performance, as it enables them to test when a tuple is no longer derivable and should be removed from a data instance. Fourth, in the case of bidirectional update exchange, determining the lineage of a tuple is an important part of detecting and avoiding side effects at run time. Finally, [94] have developed techniques to *learn* relative authorities of data sources based on *user feedback on query answers*.

In developing the primitives for these operations, we have found a significant amount of overlap in the implementation strategies. From a theoretical perspective, we have in fact shown that all of these operations involve identifying the part of the provenance graph that is relevant for certain tuples, traversing it to compute provenance expressions and evaluating those expressions in a particular semiring. As a result, we have come to the question of what the core principles, operations, and computation schemes should be for a general *provenance query language* that supports such operations. This chapter describes our preliminary ideas, framed within the ORCHESTRA architecture but generalizable to a wide variety of other applications. First, we identify important provenance querying use cases in Section 7.1. We proceed with an informal description of the semantics of ProQL queries in Section 7.2. Finally, in Section 7.3 we present the syntax of each ProQL clause in detail and illustrate their functionality by using them to express queries for the use cases of Section 7.1.

7.1 Interesting types of provenance queries

To give a better sense of the provenance querying problem, we first describe a number of common use cases. We model provenance as a *graph* specifying tuples, mapping or view operations performed on these tuples, and results produced as *immediate consequents*, as explained in Section 4.6. Provenance queries traverse this graph, returning either a projection of it or a mapping from nodes of the graph to values in a particular semiring (trust value, Boolean, score, etc.). We will later show a formal specification of each of these queries.

Q1. The ways a tuple is derivable. A scientist, intelligence analyst, or author of mappings may want to visualize the ways a tuple can be derived — including the source tuple values and the combination of mappings used. This is intuitively 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 Alternatively, one may not be interested in visualizing all derivations from base tuples, but only in derivations involving tuples from a certain relation.

Q3. Results derivable from a given mapping or view. The above examples started with a tuple and considered its provenance. Conversely, we can query the provenance for tuples that were derived using a particular mapping 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*. In that case, it is important to be able to identify peers that essentially received such information from the same other peer or source.

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 this test [57].

Q6. The lineage of a tuple. The algorithm of Section 5.3.3 needs to know the *lineage* of a tuple — i.e., the set of all base tuples it can be derived from, without distinguishing between different derivations — in order to determine whether update propagation can be performed without side effects.

Q7. Whether to trust a tuple. Given a set of *trust policies* assigning trust/distrust and authority levels to different data sources, views, and mappings, it is possible to determine a *trust level* for each derived tuple based on its provenance.

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 [8], or by machine learning based on user feedback about query answers [94]. 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 [94].

Q9. A tuple's associated probability. In ULDBs [11], query results are first computed with their provenance (called lineage in Trio), and then probabilities are assigned based on this lineage. While to the best of our knowledge the Trio system has not considered materialized views, one would need to materialize and query provenance in order to support such views in a ULDB.

Q10. Computing confidentiality/access control levels for data. Recent work [50] has shown how provenance can be used to assigning 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.

Provenance query capabilities are additionally useful in a variety of other problem settings. For instance, they can be used to facilitate the debugging of schema mappings (as in [27]).

7.2 Core ProQL Semantics

In this section, we propose a language, ProQL (for Provenance Query Language), which can query provenance in support of these types of queries. We can summarize the use cases **Q1-Q10** by noting that provenance is primarily useful in two ways: (1) it helps a user or application determine the relationship between sets of tuples, or between mappings and tuples; (2) it can be used to provide a *score/rank*, *access control level* or *assessment of derivability* or *trust* for a tuple or set of tuples. Consequently, there are two core operations that must be performed over provenance. The first is computing *projections of the provenance graph*, typically with respect to some tuple or tuples of interest: we would like to extract the derivations of a certain tuple, or the parts of derivations "connecting" certain tuples. The second is *evaluating a subgraph as an expression under a specific semiring*, e.g., in order to obtain a score, as explained in Section 4.8. Our language, ProQL, is oriented around these core operations.

A ProQL query takes as its input a provenance graph G, as described in Section 4.6. The *graph projection* part of the query:

- matches parts of the input graph according to *path expressions* (also filtering them based on predicates specified by the query author)
- binds variables on tuple and mapping nodes of the 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. Apart from the nodes on the paths, G' also contains tuple nodes required to maintain the correct arity of mapping inputs. For example, if there is a mapping node along a path and the corresponding mapping has a join in the LHS, the returned path will include both input nodes for that mapping node.

- also returns a set of tuples of bindings from distinguished variables in the query to nodes in G', henceforth called distinguished nodes.

Note that, unlike other graph query languages, such as GraphLog [31], UnQL [20], Lorel [4] or StruQL [47], and similarly to XPath [30], ProQL cannot create new nodes or graphs, but always returns a subgraph of the original graph. Moreover, provenance graphs are different from the underlying graph models of those languages, in that they contain two kinds of nodes, i.e., tuple and mapping nodes, with different properties and returned paths over such graphs have the unusual "shape" explained above.

If the ProQL query only consists of a graph projection part, it can return the subgraph described above directly. However, ProQL queries can also contain a *semiring evaluation* part, to compute the annotations for the distinguished nodes of the returned subgraph in a particular semiring. This is a unique feature for 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. The semiring evaluation 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 of a semiring evaluation query is a mapping from a set of tuples of bindings to distinguished nodes, as returned by the graph projection part of the query, to values in the specified semiring. Intuitively, this mapping associates each distinguished tuple node with its annotation, as computed according to the provenance graph.

7.3 ProQL Syntax

As we explained above, ProQL queries can have two main components, graph projection and semiring evaluation. The graph projection part can be used independently, if one only needs to compute a projection of a provenance graph. The

```
query ::= 'EVALUATE ' semiring ' OF {' graphProjection '}' assignment?
query ::= graphProjection
graphProjection ::= forClause whereClause? inclPathClause returnClause
forClause ::= 'FOR ' pathExpression ( ',' pathExpression )*
whereClause ::= 'WHERE ' conditions
inclPathClause ::= 'INCLUDE PATH ' retPathExpression ( ',' pathExpression )*
returnClause ::= 'RETURN ' var ( ',' var )*
pathExpression ::= node? ( derivation node )*
retPathExpression ::= retNode? ( derivation node )*
node ::= '[' relation? var? ']'
retNode ::= '[' var? ']'
derivation ::= '<-+' | '<-' | '<' identifier | '<$' identifier
assignment ::= 'ASSIGNING EACH ' leaf_node var listAssgn
assignment ::= 'ASSIGNING EACH ' mapping var listAssgn
assignment ::= 'ASSIGNING EACH ' leaf_node var listAssgn mapping var '(' var ')' listAssgn
listAssgn ::= ' {' caseClause* defaultClause? '}'
caseClause ::= 'CASE ' conditions : ' SET ' semiringValue
defaultClause ::= 'DEFAULT : SET ' semiringValue
semiring ::= DERIVABILITY | LINEAGE | TRUST | CONFIDENTIALITY | PROBABILITY | WEIGHT
var ::= '$' identifier
relation ::= identifier
conditions elided for brevity.
semiringValue elided for brevity.
```

Figure 7.1: Excerpt of EBNF form of grammar

semiring evaluation part can apply an assignment to a provenance graph and compute values for its distinguished nodes in the corresponding semiring. Figure 7.1 shows the key portions of the EBNF grammar for our language, and we discuss the various clauses below. To simplify the presentation of the language, we first explain the main two operations of ProQL separately.

7.3.1 Graph Projection

First, we present the clauses of the graph projection part of ProQL.

FOR: This clause is used to bind variables to sets of tuple or mapping nodes in the graph, through appropriate path expressions. In particular, the `FOR` clause contains a list of path expressions of the form shown in Figure 7.1. The following examples illustrate the semantics of these path expressions:

- `[ R $x ]` binds the variable `$x` to all tuple nodes corresponding to tuples in the relation `R` (or a subset of it, as specified by conditions in the `WHERE` clause, explained below).
- `[ R $x ] <- [ $y ]` binds `$x` as above and `$y` to all tuple nodes from which nodes in `R` can be derived in one step, through any mapping. Note that this means that `y` can range over tuples of more than one relation, i.e., a heterogeneous collection.
- `[ R $x ] <$p [ $y ]` binds `$x` and `$y` as above, and `p` to the mapping nodes, from which tuples in R can be derived in one step.
- `[ R $x ] <`$\mathtt{m}_1$` [ $y ]` binds `$x` as above and `$y` to all tuple nodes in `S` from which nodes in `R` can be derived in one step, through mapping $\mathtt{m}_1$. Note that if, in fact, `R` does not appear in the target of mapping $\mathtt{m}_1$, the path expression above should cause a type error.
- `[ R $x ] <-+ [ $y ]` binds `$x` as above and `$y` to all tuple nodes from which nodes in `R` can be derived in one or more steps.
- `[ R $x ] <-+ [ T $y ]` binds `$x` as above and `$y` to all tuple nodes in T from which nodes in `R` can be derived in one or more steps.

Note that specifying a relation name in the square brackets is essentially a syntactic shortcut for a filtering condition, restricting the range of the variable in the same square brackets to tuple nodes from that relation, and the path expression to paths involving such tuple nodes. Alternatively, one can specify such a condition explicitly in the `WHERE` clause, as explained below.

As shown in the general form of the expressions above, one can also create longer path expressions, to match derivations involving several steps, e.g., the expression
`[ R $x ] <m`$_1$` [ ] <-+ [ S $y ]` only matches derivations of tuples in `R` from tuples in `S` whose last step involves mapping m_1.

WHERE: This clause is used to specify filtering conditions on the variables bound in the `FOR` clause. Some examples for the variables from the `FOR` clauses above:

- `$x.a > 5` selects the tuple nodes for which the corresponding tuple's value for attribute `a` has a value greater than 5. In this case, since `$x` is bound to tuples from `R` if `R` does not have an `a` attribute, this condition should cause a type error.[1] Similarly, one can compare values of attributes of tuple nodes, e.g., `$x.a = $y.b`, if they have compatible data types.

- `$y in S` restricts the values of `$y` to those corresponding to tuple nodes in `S` (e.g., if `$y` is bound to a heterogeneous collection in the `FOR` clause).

- `$p = m`$_1$ restricts `$p` to mapping nodes of type `m`$_1$.

- Path expressions are interpreted as existential conditions, i.e., they are true if the specified path exists between the nodes to which variables are bound (in the `FOR` clause). If the path expression also involves variables that do not appear in the `FOR` clause, they are also interpreted as existential, i.e., the path condition is satisfied if there exists some assignment for the corresponding variable for which the path exists. For example, the path expression `[ R $x ] <-+ [ T $y ]` appearing in the `WHERE` clause — if `$x` has been defined

[1] If `$x` is bound to a heterogeneous collection, this expression does not cause a type error, but will return false for tuples that don't have an `a` attribute

in the `FOR` clause but `$y` has not — is satisfied if there is a path of any length from any tuple in `T` to the tuple to which `$x` is bound.

`INCLUDE PATH`: This clause defines the output graph, by specifying the nodes and paths to be included in it. In particular, it specifies a list of paths involving (tuple or mapping node) variables defined in the `FOR` clause or constant mapping names or wildcards (i.e., `<-+`), but not relation names. Then, for each valuation of these variables that satisfies the conditions in the `WHERE` clause, the corresponding path or set of paths involving them is included in the output graph. Some examples follow:

- `[ $x ] <m`$_1$ `[ $y ]`. If `$x`=t_1, `$y`=t_2 is a valuation satisfying the `FOR` and `WHERE` clauses of the query (henceforth called satisfying valuation), this expression includes in the result all one-step paths from t_2 to t_1 using mapping `m`$_1$ (if `$x`, `$y` are bound to single relations that do not appear on the source or target of m$_1$, respectively, this query should return an error). Note that, if there is e.g., a 3-way join in the LHS of `m`$_1$, this path will include the nodes corresponding to the other two tuples that join with t_2 using `m`$_1$, but not their provenance.
- `[ $x ] <-+ [ $y ]`. If `$x`=t_1, `$y`=t_2 is a satisfying valuation, this expression includes in the result all paths (of any length) from t_2 to t_1.

`RETURN`: Finally, this clause specifies the distinguished variables of the graph projection part of a ProQL query. The query returns a set of tuples of distinguished tuple nodes, corresponding to satisfying valuations of those variables.

Query Examples

Using these clauses we can express the following ProQL queries, e.g., over the provenance graph of Figure 4.8, that match the use cases **Q1-Q4** of Section 7.1.

Q1. Return the subgraph containing all derivations of tuples in U from base tuples:

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

A couple of similar queries, where we use a path expression or a relational join to identify the tuples whose derivations we want to return:

Q1a. Find all tuples from which tuples in U can be derived in one step, and return the subgraph containing all of their derivations from base tuples:

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

Q1b. Return the graph containing derivations of tuples that are selected through a relational join:

```
FOR [U $x], [B $y], [G $z]
WHERE $x.nam = $y.nam AND $x.can = $z.can
INCLUDE PATH [$x] <-+ []
RETURN $x
```

Q2. Return the part of derivations of tuples in U that involve tuples in relation G.

```
FOR [U $x] <-+ [G $y]
INCLUDE PATH [$x] <-+ [$y]
RETURN $x
```

Q3. Find tuples that can be derived through mappings $\mathtt{m}_1$ or $\mathtt{m}_2$ and return all one-step derivations from those tuples.

```
FOR  [$x] <$p [] <-+ [], [$y] <- [$x]
WHERE  $p = $m_1$ OR $p = $m_2$
INCLUDE PATH  [$y] <- [$x]
RETURN  $y
```

Note that the second path expression in the `FOR` clause of the query above refers to `$x`, which is defined in the first path expression. This is essentially a syntactic shortcut for:

```
FOR  [$x] <$m_1$ [] <-+ [], [$y] <- [$z]
WHERE  $x = $z
INCLUDE PATH  [$y] <- [$y]
RETURN  $y
```

In the `WHERE` clause above, `$x`= `$z` iff they refer to the same tuple node, i.e., `$x`and `$z`belong to the same relation and have the same values on all attributes.

Q4. Select some tuples from `U` and `B` that have common provenance ("provenance join"), and return their derivations:

```
FOR  [U $x] <-+ [$z], [B $y] <-+ [$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 tuple nodes that have common provenance.

7.3.2 Annotation computation

Next, we present the clauses of the annotation computation part of ProQL, which specify the semiring in which these annotations belong and the assignment of values to nodes in the projected provenance graph that are used as the provenance tokens for semiring evaluation.

ProQL name	**Semiring**	**Annotation**
DERIVABILITY/TRUST	$(\mathbb{B}, \vee, \wedge, \textsf{false}, \textsf{true})$	Derivability, trust
PROBABILITY	$(\mathcal{P}(\Omega), \cup, \cap, \emptyset, \Omega)$ (with Ω finite)	Probabilistic event tables
CONFIDENTIALITY	$(\mathcal{C}, \min, \max, 0, P)$	Confidentiality policies [50]
LINEAGE	$(\mathcal{P}(X), \cup, \cup, \emptyset, \emptyset)$ (with X finite)	Lineage
WEIGHT	$(\mathbb{N}^{\infty}, \min, +, \infty, 0)$	Ranks, scores, weights

Table 7.1: Examples of semirings in 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. Recall that in Chapter 4 we identified semirings for which evaluation is always computable and returns non-infinite values. We summarize these semirings in Table 7.1, and explain their use in ProQLbelow.

- DERIVABILITY corresponds to the boolean semiring $(\mathbb{B}, \vee, \wedge, \textsf{false}, \textsf{true})$, with the default assignment of true to all leaf nodes of the graph, and determines whether a tuple is derivable from those nodes (although more complicated assignments can also be specified).

- TRUST also refers to the boolean semiring, but we can specify an assignment of true or false to leaf nodes, and **Tm**, **Dm** for mapping functions, as explained in Section 4.8.1, according to whether specific base tuples and mappings are trusted or not.

- PROBABILITY computes probabilistic event expressions in the semiring $(\mathcal{P}(\Omega), \cup, \cap, \emptyset, \Omega)$. Such expressions can be subsequently used to compute tuple probabilities (as in Trio [11]).

- CONFIDENTIALITY evaluates provenance in the semiring of confidentiality policies [50] $(\mathcal{C}, \min, \max, 0, P)$, where the total order $\mathcal{C}$: $P < C < S < T < 0$ describes the following levels of "clearance": P = public, C = confidential, S = secret, and T = top-secret. It is useful in assigning a access control 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.

- LINEAGE corresponds to the lineage semiring $(\mathcal{P}(X), \cup, \cup, \emptyset, \emptyset)$ where X consists of the ids of the tuples in the input instance. This is useful e.g., for detection of side effects in bidirectional update exchange.
- WEIGHT corresponds to the tropical semiring [75] $(\mathbb{N}^{\infty}, \min, +, \infty, 0)$ and 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 compute costs according to the common TF/IDF document/phrase similarity metric, by ObjectRank and similar authority-based schemes [8], or by machine learning based on user feedback about query answers [94].

Using these clauses we can express the following ProQL queries, match the use cases **Q5-Q6** of Section 7.1.

Q5. Determine derivability of the tuples in U from base tuples.

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

Q5. Compute the lineage of the tuples in U.

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

ASSIGNING EACH: For the provenance semiring(s), each leaf tuple is associated with a unique provenance token and the provenance of each tuple is an expression over these provenance tokens and the mapping functions. To evaluate queries in other semirings, one needs to assign values from that semiring to leaf nodes, as well as definitions to the mapping functions. The `ASSIGNING EACH` clause can be used to specify such assignments for leaf (tuple) nodes similarly to a *switch* statement in C or Java: first, we define a variable that iterates over the set of all leaf nodes of the provenance graph returned by the nested graph projection query, and then a list of cases specifies what the value of a node should be if the condition of that `CASE` is met.[2] 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 value 1 of the semiring, i.e., the identity element for the $\cdot$ operation of the semiring.

Similarly, a second `ASSIGNING EACH` clause can be used to specify an assignment of values in the semiring for the mapping functions. In this case, one can specify conditions over the name of the 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. Note that function definitions need to satisfy the conditions of Section 4.3.1, i.e. one cannot specify an assignment that returns a non-zero value when the input of the function is 0 and mapping application must commute with (finite and infinite) sums.

One or both of these clauses can be used in a semiring evaluation query, depending on whether a user wants to "customize" their value assignment for both leaf nodes and mappings or they are satisfied with default values. We illustrate the usage of the `ASSIGNING EACH` clause(s) its usage through the following queries,

[2] if multiple `CASE` statements match, the first one is followed

that can be used to express use cases **Q7-Q10**.

Q7. Assuming peer B distrusts any tuple $\texttt{B}(i, n)$ if the data came from G and $n \geq 3$, and trusts any tuple from U, as in the first trust condition in Section 6, and distrusts $\mathtt{m}_2$ while trusting all other mappings if their input is trusted, determine what set of tuples in B is trusted:

```
EVALUATE TRUST OF {
   FOR [B $x]
   INCLUDE PATH [$x] <-+ []
   RETURN $x
} ASSIGNING EACH leaf_node $y {
   CASE $y in U : SET true
   CASE $y in G and $y.nam >= 3:  SET false
   DEFAULT : SET true
} ASSIGNING EACH mapping $p($z) {
   CASE $p = m2 :  SET false
   DEFAULT : SET $z
}
```

Q8. Find scores for the tuples in U according to the cost function described below:

```
EVALUATE WEIGHT OF {
   FOR [U $x]
   INCLUDE PATH [$x] <-+ []
   RETURN $x
} ASSIGNING EACH leaf_node $y {
   CASE $y in B AND $y.id < 5 :  SET 0
   CASE $y in G AND $y.id >= 7 :  SET 5
```

```
    DEFAULT  :  SET edit_distance($y.nam, "GENE")
} ASSIGNING EACH mapping $p($z) {
    CASE  $p = m2 :  SET 0
}
```

Q9. Assuming the correlation between tuples in B and G described in the assignment below, compute the probabilistic events associated with tuples in U:

```
EVALUATE PROBABILITY OF {
    FOR  [U $x]
    INCLUDE PATH  [$x] <-+ []
    RETURN  $x
} ASSIGNING EACH leaf_node $y {
    CASE  $y in B AND $y.id < 5 :  SET X1
    CASE  $y in B AND $y.id >= 5 :  SET X2
    CASE  $y in G : SET X1
}
```

Q10. Find access control levels of tuples in U, assuming tuples in B are confidential, those in G are secret, mapping m_3 is top-secret and everything else is public:

```
EVALUATE CONFIDENTIALITY OF {
    FOR  [U $x]
    INCLUDE PATH  [$x] <-+ []
    RETURN  $x
} ASSIGNING EACH leaf_node $y {
    CASE  $y in B : SET C
    CASE  $y in G : SET S
    DEFAULT  :  SET P
} ASSIGNING EACH mapping $p($z) {
   CASE  $p = m3 and $z != 0 :  SET T
}
```

In the next chapter we focus on developing a prototype implementation for the core semantics of the language, as presented in the previous sections. To this end, in the next chapter we outline our main strategies for implementing ProQL over an RDBMS, using the relational encoding for provenance graph that we presented in Chapter 5 to store provenance and translating ProQL queries to SQL queries that can be executed by the RDBMS. We also suggest structures for indexing paths in provenance graphs that can be maintained, along with the relations in which the provenance graph is stored, in an RDBMS.

Chapter 8

ProQL Query Processing and Optimization

In this chapter we describe our implementation of ProQL for acyclic provenance graphs. In the first section we describe the core aspects of our execution strategy that exploits a relational DBMS engine. In the next section we discuss how we enhanced this basic engine with indexing techniques, that we believe will speed up query answering significantly. Finally, we demonstrate experimentally the performance of provenance querying and illustrate the benefits of our indexing techniques.

8.1 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 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 Section 5. Thus, our approach relies on converting ProQL queries into SQL queries (or sets of SQL queries) that can ultimately be executed over an underlying RDBMS. More precisely, we break the query answering process into several stages:

- Converting the schema mappings into a *provenance schema graph* (this is common for all queries).
- Matching the ProQL query against the provenance schema graph to identify nodes that match patterns, as described by path expressions.
- Creating a datalog program based on the set of schema mappings and mapping tables that correspond to the schema graph nodes, as well as the source relations whose EDB data is to be included.
- Executing 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.

8.1.1 Provenance Schema Graph

While paths in the provenance graph exist in the instance (tuple) level, in fact these tuples belong to specific relations that are related 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

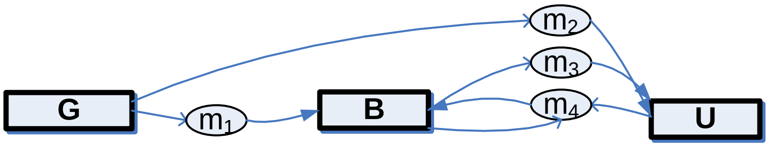


Figure 8.1: Provenance schema graph for running example

a Dataguide [56] 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*. We construct it as follows. We begin by creating one node for each relation (a *relation node*, which we label with the name of the relation) and one *mapping node* for each mapping table (where, again, we label the node with the table's name). 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. We direct 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 8.1, where we show relation nodes with rectangles and mapping nodes with ellipses.

8.1.2 Matching ProQL Path Expressions

The next step is to determine which subgraphs of the provenance schema graph match the ProQL patterns. We start with the distinguished reference nodes specified to be returned by the ProQL query: these nodes can range over all relations, or if specified by the query, may be restricted to a single relation. 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.

8.1.3 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.[1] This process is fairly straightforward. For each pair of relation nodes R, S returned from the matching step, we determine if any schema mappings define R in terms of S, or vice versa: if so, we add these mappings as rules to the program. For every relation node matched in the schema graph, we also add any rules necessary to test when we have reached an EDB relation (containing leaf nodes of the provenance graph).[2]

8.1.4 Executing the Program

In this section we discuss how to execute the datalog program over a provenance graph stored in an RDBMS. Recall from 5.1.2 that our storage scheme uses one provenance relation for all one-step derivations involving a particular mapping. On our running example, for the provenance graph of Figure 4.8, the one-step derivations involving the mappings

$$(m_1) \qquad G(i,c,n) \to B(i,n)$$

$$(m_4) \qquad B(i,c) \wedge U(n,c) \to B(i,n)$$

are stored in the relations P_1 and P_4 shown below:

P_1

i	c	n
3	5	2
1	2	3

P_4

i	n	c
3	2	5
3	3	2

[1] This program can be recursive for cyclic provenance graphs. However, in this dissertation we focused on ProQL evaluation over acyclic provenance graphs, for which this program is not recursive.

[2] This may be a join with a separate "hidden" EDB table, or a selection against the provenance relation in which the EDB bit is set to **true**.

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, in order to execute ProQL queries with an annotation computation component, we need to identify complete derivations of certain tuples from base data, for which an assignment of semiring values is given in the ASSIGNING EACH clause. For example, to execute the ProQL query

```
EVALUATE DERIVABILITY OF {
   FOR [U $x]
   INCLUDE PATH [$x] <-+ []
   RETURN $x
} ASSIGNING EACH leaf_node $y {
   CASE $y in B : SET false
   DEFAULT : SET true
}
```

we need to reconstruct complete derivations of tuples in U from base tuples, and use those derivations — together with the assignment specified in the ASSIGNING EACH clause — to determine which tuples in U are trusted.

For acyclic provenance graphs, each tuple can only have a finite number of *distinct derivation tree shapes*. For each one 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 8.1.3. The result of this unfolding is a union of conjunctive rules over provenance relations and the base data "reachable" from those tables. Moreover, during this unfolding we can create a semiring expression corresponding to this derivation tree shape. This expression can then 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. Finally, we can evaluate the unfolded rules and annotation computation expressions by translating them to SQL queries that can be evaluated over the underlying RDBMS.

In the rest of this section, we describe in more detail the unfolding process for graph projection queries, as well as the creation of semiring expressions that can be used for annotation computation.[3] We also explain how unfolded rules and semiring expressions can be translated to SQL queries that can be evaluated directly over an RDBMS.

Graph projection. The basic strategy is to take the datalog program as defined above, and to recursively unfold it, creating a union of conjunctive queries over provenance relations and base data. To perform every step in this unfolding, we need to find a homomorphism from the head of the rule we use at each unfolding step to the body of the current result of the unfolding.

However, simple unfolding will result in a query that recomputes *all of the tuples* derivable in the relation of interest. In many cases we only seek the provenance of a few target tuples, and here we would like to make our execution *goal-directed*. We achieve this by modifying each of the conjunctive queries as follows. Suppose we are computing the derivations for tuples in relation T, and we are given an initial set of target tuples T^{trg}. If our query is of the form $T(\bar{x})$:- $\Psi(\bar{x}, \bar{y})$ (where Ψ represents a series of query atoms), then we rewrite it as $T(\bar{x})$:- $T^{trg}(\bar{x}), \Psi(\bar{x}, \bar{y})$. A similar technique was used in [61].

We illustrate this unfolding in the following example.

[3] We note that the approach described in this section would not work for cyclic provenance graphs, for which the recursive unfolding process may not terminate. In Chapter 10 we outline an alternative strategy, based on bottom-up execution of the datalog program and materialization of intermediate results, that can also be applied in cyclic settings.

Example 20. *Suppose we have the mappings:*

$$(m_1) \qquad R_2(x,z) \rightarrow R_1(x,z)$$

$$(m_2) \qquad R_3(x,y) \wedge R_4(y,z) \rightarrow R_2(x,z)$$

$$(m_3) \qquad R_5(x,y') \wedge R_6(y',z) \rightarrow R_3(x,y) \wedge R_4(y,z)$$

Suppose also that all relations have local data. Following the notation of Section 3.2, we write R_i^ℓ to indicate the data inserted locally at R_i. Now, consider the ProQL query:

```
FOR [R₁ $x]
WHERE $x.att1 < 5
INCLUDE PATH [$x] <-+ []
RETURN $x
```

Each tuple in R_1 above can be derived in one of the following ways:

- *Locally, from R_1^ℓ*
- *Through m_1, from R_2s local data (i.e., R_2^ℓ)*
- *Through m_1 and m_2. In this case, since there is a join in the body of m_2, we need to consider all combinations of alternative derivations for tuples in R_3 and R_4. Since both may include both local and derived data (through m_3) we need to consider all four combinations (local data from both relations, derived data from both relations and 2 combinations with local data from one relation and derived from the other).*
- *For data derived through m_3 above, the only way to derive them is by joining local data from R_5 and R_6.*

For simplicity, in this example we use mappings for which the homomorphisms involved in all unfolding steps are trivial identity functions (i.e., the same atom appears in the body of the partially unfolded rule and in the head of the rule used at that step). Finally, suppose that R_1^{trg} contains the tuples from R_1 that satisfy the conditions in the

`WHERE` clause of the query. Then, this unfolding would produce a union of the following conjunctive queries:

$$R_1^{trg}(x,z), R_1^{\ell}(x,z)$$
$$R_1^{trg}(x,z), P_1(x,z), R_2^{\ell}(x,z)$$
$$R_1^{trg}(x,z), P_1(x,z), P_2(x,y,z), R_3^{\ell}(x,y), R_4^{\ell}(y,z)$$
$$R_1^{trg}(x,z), P_1(x,z), P_2(x,y,z), R_3^{\ell}(x,y), P_3(x,y',z), R_5^{\ell}(x,y'), R_6^{\ell}(y',z)$$
$$R_1^{trg}(x,z), P_1(x,z), P_2(x,y,z), R_4^{\ell}(y,z), P_3(x,y',z), R_5^{\ell}(x,y'), R_6^{\ell}(y',z)$$
$$R_1^{trg}(x,z), P_1(x,z), P_2(x,y,z), P_3(x,y',z), R_5^{\ell}(x,y'), R_6^{\ell}(y',z)$$

Observe that, apart from the atoms specifying sets of tuples whose provenance we want to explore (such as R_1^{trg}),[4] the unfolded rules consist only of atoms of provenance (such as P_1) and EDB (such as R_1^{ℓ}) relations. The former are the relations where the provenance of the corresponding mapping is stored, according to the provenance storage scheme presented in Chapter 5. Including the provenance relations in these rules is necessary, in order to be able to apply filtering conditions at intermediate nodes of a derivation tree. Moreover, provenance relations incorporate possible overrides by CDSS users that cannot be determined by only looking at the sources of a mapping, e.g., when users delete imported tuples that they consider untrusted.

Since the result of the unfolding is a non-recursive datalog program, we can execute each rule in that program by translating it to an SQL query. In our implementation, we compute such SQL queries using the same translation layer as in the case of the RDBMS implementation of update exchange (cf. Section 6.3). The resulting SQL queries can then be executed directly over the underlying RDBMS.[5]

Annotation computation. Additionally, for ProQL queries with an annotation computation component, we need to translate unfolded rules to SQL queries that

[4] If this set of tuples can be expressed through some filtering conditions, e.g., as specified in the `WHERE` clause of a ProQL query, we can avoid introducing an R_1^{trg} atom and instead push these conditions to the appropriate provenance or EDB relations.

[5] For cyclic provenance graphs, ProQL query evaluation could involve recursive datalog programs, whose evaluation would need to use the datalog$^{\text{sk}}$ fixpoint layer described in Section 6.3.

also compute such annotations based on an assignment of values to base tuples and the stored provenance information. To achieve this, as a part of the unfolding process we generate a *semiring evaluation expression* for each unfolded rule as follows:

- For each unfolding step, we insert a *semiring product* operation, whose operands are filled by later unfolding steps.
- When an unfolding step expands to a *leaf relation* — i.e., a relation with no incoming mappings in the provenance graph returned by the graph projection part of the query — we insert the appropriate assignment of semiring values from the corresponding ProQL `ASSIGNING EACH` clause.

Of course, each conjunctive query only computes a subset of the tuples and their provenance — specifically the tuples and provenance values for one potential derivation tree shape. We combine these to compute an annotation for each tuple as follows:

- We convert each unfolded rule into an SQL query, that also computes an annotation for each output tuple, according to the semiring evaluation expression for this rule, as computed during the unfolding. This annotation is stored with each tuple as an additional attribute.
- We introduce another additional attribute for the provenance expression evaluation. In particular, this attribute has a unique value for each different SQL query generated in the first step above.
- We combine all SQL queries into a single query using UNION ALL; the additional attribute introduced in the second step ensures that different derivations that result in the same value will not be eliminated by set semantics.

- We GROUP BY the values of the attributes of the tuples (not including any of the special attributes above).
- We aggregate the annotations — as found in the first provenance attribute — within each group, using an appropriate aggregation function depending on the kind of annotations we are computing. Referring to the possible semirings of the `EVALUATE` clause, for `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 annotation. `CONFIDENTIALITY` and `WEIGHT` can be evaluated using the MIN function. For `LINEAGE` and `PROBABILITY` we need SQL user-defined table functions to union the annotations together.
- Finally, we threshold the results with a HAVING expression if, e.g., we only want to return tuples with non-zero annotations, or those whose rank is under a certain value.

If the query does not contain an annotation computation component, we create a string representation of each derivation, which is also stored in the additional attribute, using appropriate string concatenations for each of the semiring operations.

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.

8.2 Indexing for Provenance

The main challenge in answering ProQL queries is essentially in navigating through graph-structured data, according to path expressions. As we explained in Section 8.1, such path traversals are translated into joins among provenance relations, representing one-step derivations. 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 often correspond to derivation tree shapes with overlapping paths. As a result, some joins between provenance relations may be common among several unfolded rules.

Based on these observations, a natural question to ask is whether one could optimize ProQL queries by *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, in a way that our SQL-based strategy could benefit from it.

Among a variety of path indices that have been studied in the literature [84, 56, 73, 70, 33], the most natural indexing technique to adapt for our provenance query scheme is the *access support relation* [73, 74] from object-oriented databases. An access support relation (ASR) is an n-ary relation among sets of objects connected through paths. For example, suppose there is a class A that has a field b, and b is an object of class B. Moreover, suppose class B has a field c that is an object of class C. Then, one may create an ASR whose entries correspond to triplets of objects from classes A, B and C that appear along such a path. For instance, if there is an object a_1 of class A such that $a_1.b = b_1$ and $b_1.c = c_1$, the ASR would contain an entry for the path $a_1.b_1.c_1$. Then, this ASR can be used to speed up queries involving the path expression $X.b.c$, where X ranges over a set of objects of class A. Importantly, the same ASR can be used to answer queries beginning from different classes of objects, as long as some path expression in those queries overlaps with the paths stored in the ASR.

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 relations 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 simple case of answering queries using materialized views [63]. 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).

We illustrate such an ASR definition in the following example.

Example 21. *Continuing from Example 20, we saw that path traversals are translated into joins among provenance relations. For example, creating an ASR that corresponds to the path that involves m_2 followed by m_1 amounts to joining P_2 with P_1.*

In the sequel, we write $P_{(2,1)}$ to refer to the relation used to store the ASR corresponding to the path(s) involving m_2 followed by m_1.

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.

8.2.1 ASR Design Choices

As we explained in the example above, to index paths in a provenance graph we need to store joins between provenance relations. In general, however, there may be tuples in each relation that do not join with a tuple in the other relation, e.g., because they originated from the propagation of disjoint data coming from different mapping paths. Moreover, some queries may only require looking at one mapping in such a path, or a subpath.

As a result, we have several options with respect to what to store in ASR relations in order to optimize provenance queries. In this section, we present these options and discuss their theoretical advantages and disadvantages. In the sequel we describe in more detail the implementation of several of these options over ORCHESTRA and evaluate their performance experimentally.

For an ASR indexing a path of n steps, one of the choices involves whether to materialize only the complete path or (some or all) of its subpaths. This choice is related to the type of join among provenance relations that we materialize in the ASR. The type of join that we materialize also affects whether we need to store separately individual provenance relations that appear in some ASR. More precisely, for the case of an ASR for a path containing mapping m_2 followed by m_1, we have the following options:

1. Store the full outer join $P_2 ⟗ P_1$ in $P_{(2,1)}$. This includes all tuples from both relations, whether they join or not, and amounts to storing the path and all its subpaths in the ASR.

2. Variations of (1) involving the left or right outer join, i.e., all subpaths starting (or ending) with the first (last) mapping in the path.

3. Store just the inner join $P_2 \bowtie P_1$ in $P_{(2,1)}$ (i.e., only the complete path). This only includes tuples from both relations that join with each other. For tuples

that don't join, we still need to maintain P_2 and P_1.

4. Store $P_2 \bowtie P_1$ in $P_{(2,1)}$, and all tuples (whether they join or not) in P_2 and P_1.

For longer paths we only allow combinations that follow the same option for each step of the path.

Another design choice is related to the ability to incrementally maintain these ASR relations, when updates occur. Some of the join types presented above are more amenable to incremental maintenance than others. In particular, in cases (1), (2) and (3), if we insert a tuple t_1 in P_1 that joins with t_2 in P_2, incremental maintenance involves:

- checking whether t_2 used to join with any tuples before the insertion of t_1, and if not *removing* it from the corresponding table (e.g., removing t_3 from P_2 in case (3)).
- inserting a "join" tuple involving t_1 and t_2 in $P_{(2,1)}$

Similarly, in cases (1), (2) and (3), deleting a tuple t_1 from P_1 that used to join with t_3 in P_3 requires:

- deleting the "join" tuple involving t_1 and t_2 from $P_{(2,1)}$
- checking whether t_2 still joins with some other tuple, an if not "reinserting" it in the appropriate table (e.g., inserting t_2 in P_2 in case (3), or t_2 padded with nulls for P_1s distinct attributes in $P_{(2,1)}$ in case (1)).

Note that, apart from their conceptual complexity, the above operations involve an additional (anti)semi-join with the index relations for each insertion and deletion, e.g., to check whether a tuple that used to join with some deleted tuple also joins with some other tuple. This can be costly and the effect on overall incremental maintenance performance should be taken into account. On the other hand, case (4) is simpler, since it does not require these existential checks.

In previous chapters we have presented a framework in which such updates can be propagated, but this is only possible for mappings that can be expressed as tgds (or, equivalently, as datalog^{sk} rules). As a result, if we can express the ASR definition as a set of such rules, we can use this framework for its incremental maintenance. On the other hand, for cases where a more expressive language is needed for their definition, it is not possible to maintain them incrementally using existing ORCHESTRA facilities.[6]

Finally, another choice involves whether to allow a mapping to appear in more than one ASR relation or not. Overlapping ASRs allow more flexibility for covering a broader range of queries, possibly traversing different but overlapping paths. However, their existence also make it more difficult to rewrite the original queries in order to use those ASRs.

8.2.2 ASR Implementation over ORCHESTRA

In this dissertation, we chose to implement those of the options above that fit within the framework of CDSS, as outlined by the functionality supported by ORCHESTRA. As a result, we have implemented the different types of joins described above in ways that their definition can be expressed using datalog^{sk} rules (essentially, unions of conjunctive queries), similar to the ones we use in Chapter 5 to maintain individual provenance relations. Moreover, to keep unfolding simple and efficient, in the current implementation we assume non-overlapping ASR definitions for the case of outer joins, i.e., no mapping can appear in more than one outer join ASR. Non-overlapping outer join definitions allow for a greedy unfolding algorithm, which is not possible if the same mapping or path appears in more than one ASR definition. This greedy unfolding algorithm works for overlapping inner join definitions, so long as no ASR is contained in another.

[6] In future work we intend to explore extensions to ORCHESTRA to allow outer joins in mappings.

For the case of inner joins, this can be done as follows:

- Store the mapping in individual provenance relations (e.g., m_1 in P_1)
- Add a rule of the form:

$$P_{(2,1)}(x, y, z) \text{ :- } P_2(x, y), P_1(x, z)$$

In the rule above, the variables have been renamed according to the homomorphism from the body of m_1 to the head of m_2, as described in the previous section. The ones that are common (i.e., **x**) correspond to the join conditions and only need to be stored once, since $P_{(2,1)}$ contains the inner join between the two relations, and thus the values of those variables for both relations P_2, P_1 are equal. Finally, we index the ASR relation on the attributes that belong to the key of at least one of the individual provenance relations.

On the other hand, the outer join cases cannot be directly expressed using datalog$^{\text{sk}}$ rules. However, it is possible to use datalog$^{\text{sk}}$ rules to define null-padded relations that resemble outer joins as follows:

- Store the provenance of each mapping in an individual provenance relation (e.g., m_1 in P_1)
- For the case of full outer join, add rules of the form:

$$\begin{aligned}
&P_{(2,1)}(1, 1, x, y, x, z) \text{ :- } P_2(x, y), P_1(x, z)\\
&P_{(2,1)}(1, 0, x, y, -, -) \text{ :- } P_2(x, y)\\
&P_{(2,1)}(0, 1, -, -, x, z) \text{ :- } P_1(x, z)
\end{aligned}$$

In the rules above, the symbol '$-$' indicates that we pad the tuples with NULLs for all attributes that are irrelevant in the corresponding case. Observe that, in the case of outer joins the schema of the ASR relation needs to have a distinct attribute for every attribute in each of the individual provenance relations. This is necessary for tuples of one relation that don't join with any tuples from the other relation. The first rule above inserts to $P_{(2,1)}$ the inner join between the two relations. The other two rules also insert tuples from one relation that don't join with a tuple from the other relation, padding the remaining attributes in the join relation with NULLs. Note that, for each individual relation, the tuples that join with tuples in other relations will be inserted twice in the ASR relations, using the rules above (and multiple times, for longer paths). This could be avoided using some datalog$^{\text{sk}}$ rules with negation in their body, i.e., to ensure that tuples that appear in the join are not also inserted by the second or third rule, padded with nulls. Incremental insertion propagation for such rules would require that the rules in the program above are executed in the order presented above, using the explicit stratification capabilities of datalog$^{\text{sk}}$. However, incremental deletion propagation would not be possible. For this reason, in this dissertation we chose to implement outer joins using the rules shown above.

For left (right) outer joins, we can use the first and third (resp. first and second) rules above.

Rewriting unfolded rules to use existing ASRs

In order to take advantage of existing ASRs, we need to rewrite the rules produced by the unfolding described earlier, to replace provenance relation atoms with ASRs that contain those provenance relations. We have developed a greedy algorithm that performs this rewriting as follows:

1. For every ASR, consider the paths contained in it in inverse order of length, and if there is a homomorphism from such a path into the body of the rule, replace the image of the homomorphism with an appropriate ASR atom.

2. Once such a homomorphism has been found for an ASR, don't consider subpaths of shorter length, but consider the next ASR on the rule produced as a

result of the rewriting in step 1.

In the algorithm above, the ASRs are considered in some random order. For the case of ASRs containing inner joins, the first step essentially simplifies into one homomorphism check, since no subpaths are stored in the ASR. Then, if we have two overlapping inner join ASRs, only one of which can be used in some unfolded rules, the correct rewriting employing this ASR will be produced, no matter which ASR we use first, as long as there is no ASR that is completely contained in another ASR. On the other hand, with overlapping outer joins, even if the path indexed by an ASRs A is completely contained in an unfolded rule, if we first consider some other overlapping ASR B — that is not completely contained in the rule — we may unfold some subpath of B, thus making it impossible to take full advantage of A in the rewriting, since some of the atoms in the path indexed by A have been replaced by a B atom. In this case, we would need to consider a dynamic programming approach, considering the ASRs in all possible orders and assigning some value on the resulting reordering, in order to finally pick the one considered optimal. In our implementation, we selected to only allow non-overlapping outer join ASR definitions, for which our greedy algorithm above is sufficient.

8.3 Experimental Evaluation

In this section, 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. For this setting, we investigate the effect of the number of peers with local data to the number of unfolded rules, and the resulting performance in terms of unfolding and query evaluation time. Next, we consider more complex topologies for moderate numbers of peers with local data, and investigate the performance and scalability of both the unfolding algorithm and evaluation time for the unfolded rules on these topologies, for different numbers of peers and amounts of data at each peer. Finally, we consider grouping mappings along paths in ASRs of different types and lengths, and we investigate the effects to unfolding time (including the time for rewriting rules to use these ASRs), evaluation time and maintenance time.

8.3.1 Experimental Setup

We conducted all experiments on our ProQL implementation on top of ORCHESTRA. As with the rest of ORCHESTRA, the ProQL implementation, 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.

Experimental CDSS Configurations and Terminology

For CDSS settings, we extended the workload generator used in the ORCHESTRA experiments of Section 6.4, in order to generate topologies to test provenance querying at scale.[7] The resulting schemas are still based on the schema of the SWISS-PROT protein database [7], from which we also extract data to use as local insertions for the peers that have local data. For the provenance querying experiments, we partition the 25 attributes in the SWISS-PROT universal relation into two relations, adding a shared key to preserve losslessness. Then, each mapping has a join between two such relations in the body and another join between the two relations of the peer in the target of the mapping. We give more details about specific mapping topologies in the discussion of experiments below.

[7] for randomly generated topologies we typically got very short paths with mappings in the same direction, and thus we have omitted them from this experimental evaluation.

As in the case of the experiments of Section 6.4, we generate fresh insertions 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, under the assumption that such data would be stored as CLOBs in a real bioinformatics system. We refer to the *base size* of a workload to mean the number of SWISS-PROT entries inserted initially into each peer's local contribution relations and propagated to the other peers before provenance queries were executed. Unless otherwise specified, in our experiments we insert 10,000 tuples in each relation of each peer that has local data, and propagate these tuples through mappings, before evaluating provenance queries. In some experiments, we insert data at every peer.

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 from them 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 where all peers have local data, as a stress test. However, in most of the remaining experiments we consider settings where local data is only contributed by *leaf* peers, i.e., peers with no incoming mappings, or leaf and *middle* peers, i.e., peers that appear roughly halfway along some path from a leaf peer to the *root* peer, i.e. the peer with no outgoing edges (which is unique in all of our topologies).

The particular topologies we present in the experiments below should not be interpreted as a complete CDSS setting, but instead as patterns that could appear within a CDSS setting. In particular, they focus on a projection of the complete mapping graph that only contains peers from which our root peer of interest is reachable. 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) and such other mapping paths do not affect the evaluation or the result of the provenance queries whose performance we measure.

Provenance Queries

The main goal of these experiments is to evaluate the performance of the path traversal component of ProQL, with or without the use of ASRs. As a result, for our experiments, we used queries of the form

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

where in most cases R is a relation at the peer at the root of the corresponding topology, unless otherwise specified. Such queries traverse all the paths in the mappings graphs up to their end, and thus are ideal in order to evaluate path traversal, both in terms of unfolding cost and evaluation time of the unfolded rules. We also experimented with similar queries involving annotation computation, such as:

```
EVALUATE DERIVABILITY OF {
  FOR [R $x]
  INCLUDE PATH [$x] <-+ []
  RETURN $x
} ASSIGNING EACH leaf_node $y {
  CASE $y in T : SET false
  CASE ...
  DEFAULT : SET true
}
```

Perhaps surprisingly, we found that the execution time for queries involving such annotation computation was very similar to that the execution time for their graph projection component, i.e., the graph projection component dominates execution time. Thus, for simplicity, in the experiments below we will focus on graph projection queries that don't involve annotation computation.

Experimental Methodology

Each individual experiment was repeated seven times, with the final number obtained by discarding the best and worst results and computing the average of the remaining five numbers, to ensure warm caches. We note that the first run was usually a lot slower than all subsequent ones, likely because of the time the DB2 optimizer took to find a good query plan for each of the unfolded rules (which, for some of the topologies considered could have up to around 40 atoms, resulting in 40-way joins with numerous join conditions). In subsequent runs, this plan selection was likely cached, and evaluating the selected plan turned out to be pretty efficient, even for such large queries, as we show in our experiments below.

8.3.2 Effect of Number of Peers with Local Data

In the experiments of this section we use a simple topology, where mappings form a *chain*, as shown in Figure 8.2. For the first experiment, we perform a "stress-test" by assuming that all peers have local data and investigate the performance of the following query (hereby called the "root" query) for different numbers of peers:

```
FOR [R_0 $x]
INCLUDE PATH [$x] <-+ []
RETURN $x
```

Figure 8.3 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

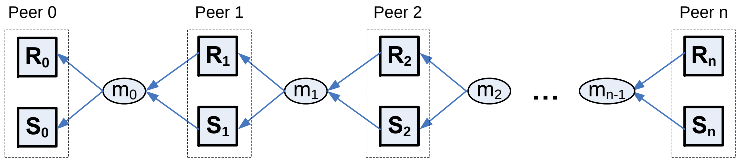


Figure 8.2: Chain topology with n+1 peers

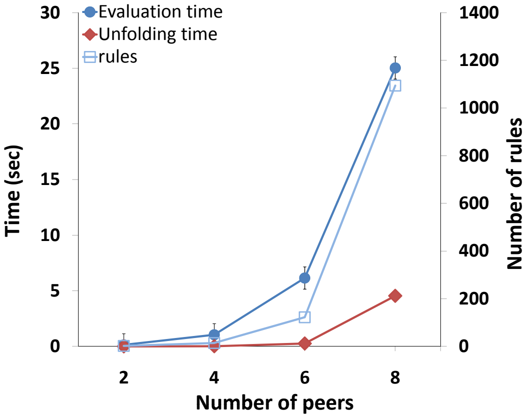


Figure 8.3: Performance of provenance querying and number of unfolded rules for chain topology of varying length with data at every peer

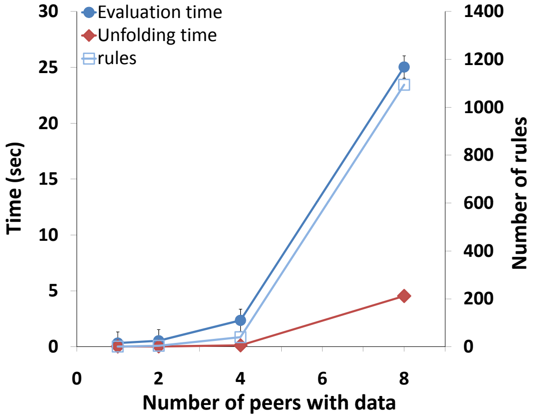


Figure 8.4: Performance of provenance querying and number of unfolded rules for chain topology of 8 peers with varying number of peers with data

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 8.3, unfolding time and evaluation time for the unfolded rules also grow exponentially and is efficient for up to 10 peers. In a related experiment, we varied the number of peers with local data, out of a total of 8 peers. Figure 8.4 shows that the number of unfolded rules, as well as unfolding and evaluation times also grow exponentially when the ratio of peers with local data increases.

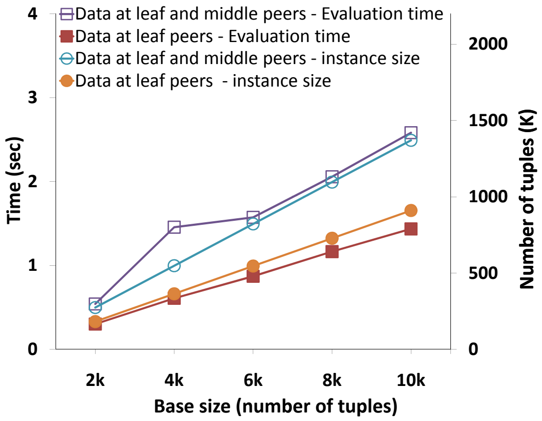


Figure 8.5: Performance of provenance querying and instance size for a chain topology of 30 peers and varying base sizes

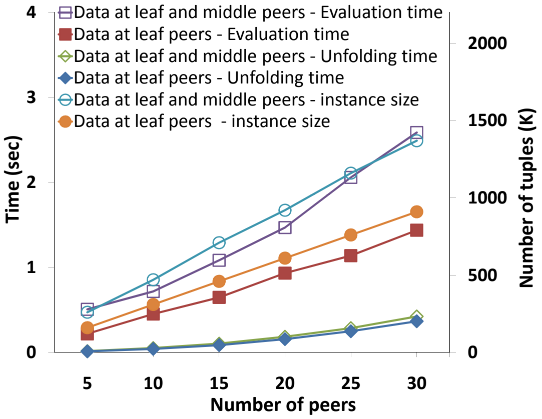


Figure 8.6: Performance of provenance querying and instance size for a chain topology of varying numbers of peers and a base size of 10k tuples

8.3.3 Scalability of Provenance Querying vs Number of Peers and Base Size

The experiments above show that the number of unfolded rules as well as the time required for their evaluation grows very fast with the number of peers, for settings where most peers have local data. As explained earlier, these experiments were meant mostly for stress-testing, but more realistic bioinformatics settings are unlikely to contain long chains where each of the peers along the way contributes more data. It is more likely that mappings in a chain would propagate data between user databases from some authoritative source at the source of the chain, without introducing new data, or that a peer imports data from multiple authoritative sources along different (and typically shorter) mapping paths.

For this reason, in the next experiments, we consider CDSS settings with the chain topology of Figure 8.2 that only have data either at the last peer (hereby called "leaf" peer), or at the leaf and middle peer (i.e., the one half-way between the leaf and root peer, along the path of mappings).

Figure 8.5 shows that the size of the instance produced as a result of the propagation of local data at the leaf or leaf and middle peers grows linearly with the base size, for a settings of 30 peers with mappings forming a chain. Unfolding time and evaluation time for the unfolded rules also grow linearly up to a few seconds, even for a base size of 10k tuples per peer relation. For this reason, in subsequent experiments we use 10k tuples per peer relation as the base size.

In Figure 8.6 we show that the size of the instance, that results from the propagation of 10k tuples at the leaf or leaf and middle peers also grows linearly with the total number of peers. Unfolding time and evaluation time for the unfolded rules also grow linearly, and is within few seconds, even for a chain of 30 peers. However, we were unable to run experiments for settings with more than 30 peers because the resulting SQL queries are too large for DB2 (recall from the translation explained in the previous section that the unfolded rules would contain up to n-way joins, where n is roughly equal to the number of peers along a path from the root peer to a peer with local data).

We also study scalability of provenance querying for other topologies. In particular, we considered the *branched* topology of mappings shown in Figure 8.7, where half of the mappings form a chain, while the remaining ones form two shorter branches, connected at different points with the chain. For this topology, we assigned local data to the peers at the end of each mapping path, as well as possibly in the middle of each such path.

Figure 8.8 shows that unfolding and evaluation time grow fairly slowly for branched settings up to 30 peers, with data at the leaf or leaf and middle peers. The “jump” that occurs between 5 and 10 peers — in the case where both leaf and middle nodes have local data — is caused by the fact that the number of unfolded rules jumps between the corresponding cases (essentially because a setting of 5 peers is too small to have “middle” peers). Moreover, even for 30 peers in that case, the total provenance query processing time (i.e., the sum of unfolding and

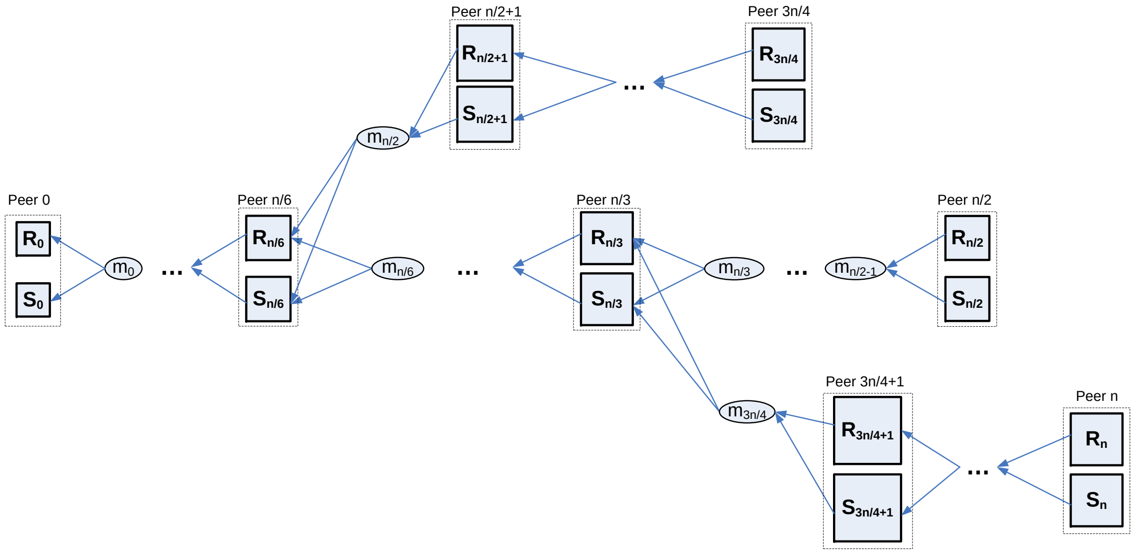
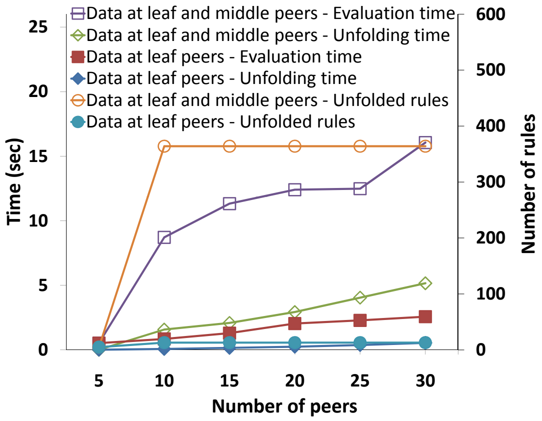


Figure 8.7: Branched topology with n+1 peers



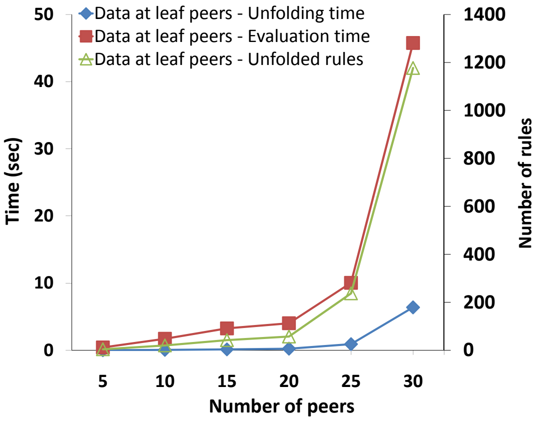


Figure 8.8: Performance of provenance querying and number of unfolded rules for a branched topology of varying numbers of peers and a base size of 10k tuples

Figure 8.9: Performance of provenance querying and number of unfolded rules for a 4-ary tree topology of varying numbers of peers and a base size of 10k tuples

evaluation times) is within 20 sec, which is well within the requirements of our target applications.

Finally, we also considered a topology that corresponds to a situation where some peer imports data from many different authoritative sources. In this topol-

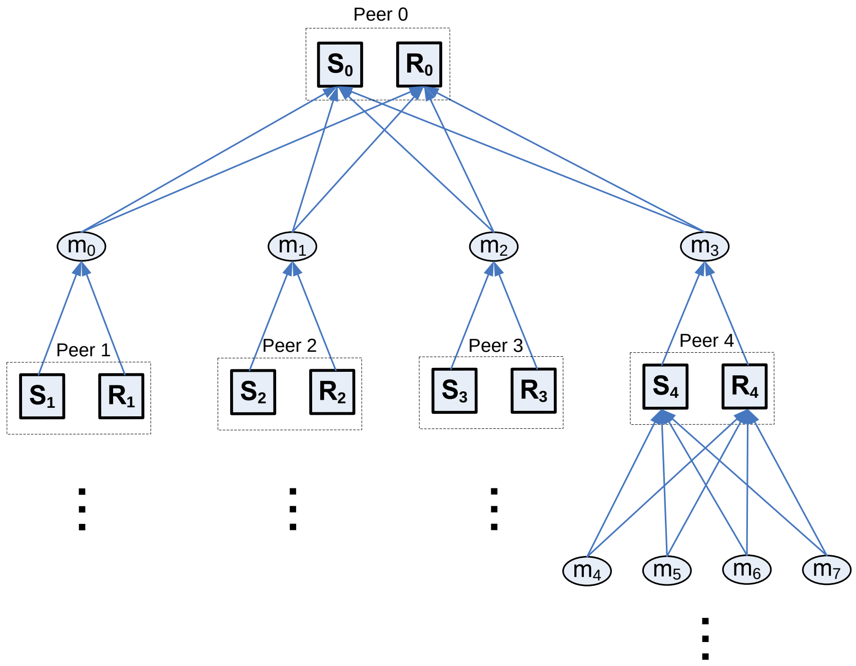


Figure 8.10: 4-ary tree topology

ogy, the mappings form a *4-ary tree*, as the one shown in Figure 8.10. As in the previous topologies, we assign local data to all peers with no incoming mappings. However, in this topology the number of such peers (e.g., for the same total number of peers) is considerably larger. For instance, in the case of 30 peers, 24 of them contain local data. This, combined with the existence of many alternative derivations due to the large number of branches at every level of the tree, results in a large number of unfolded rules, as shown in Figure 8.9. As a result, unfolding and evaluation times for the root query scale linearly for small number of peers and exponentially, for over 20 peers — although at a slower rate than in the case of a chain topology with data at every peer. Even so, total query processing time is under a minute, even for a 4-ary tree of 30 peers, which is within the requirements of our target applications.

8.3.4 Comparison of Different Join Types for ASRs

Even though we showed that query performance 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 essentially materialize joins that appear in many of these unfolded rules. As we discussed earlier, there are several options about what kind of join to store in an ASR. In this section, we investigate — for the topologies presented in the previous section — the effect of these different options to query processing as well as maintenance times, i.e., the time to propagate local data to peer, provenance and ASR relations.

First, we consider a CDSS setting with 30 peers, with mappings forming a chain topology, such as the one of Figure 8.2, with 10k tuples of base data only at the leaf peer. When defining ASRs of different types, we varied the length of paths that we store in them. In the following graphs we call this length the *maximum join width*, since the length of the paths we store is reflected in the width of the corresponding join between provenance relations (i.e., a path of length n is translated to an n-way join). For the chain topology, we essentially "split" the chain into paths up to this maximum width, and possibly store the remaining mappings in a shorter ASR, if the number of mappings is not a multiple of this join width. We note that, in this topology, the root query is essentially translated into a single unfolded rule, and the join in the body of the rule contains all ASRs we define (and in that sense it is an ideal case for optimization using ASRs).

Figure 8.11 shows the total query processing time (i.e., the sum of evaluation and unfolding times) for the root query in the cases of inner, full outer, left outer and right outer join being stored in the ASR. The dotted line indicates the processing time for this query without using any ASRs. First, we observe that, in this case, all kinds of ASRs improve performance. Among different join types, inner joins provide the best performance, left and outer joins are identical and full outer join

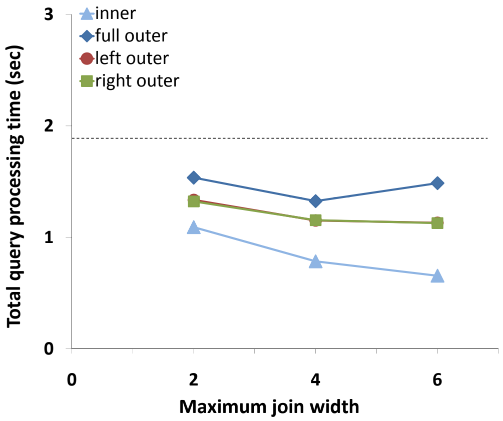


Figure 8.11: Total query processing time for different kinds and widths of ASRs, for chain topology of 30 peers with data at the leaf peer

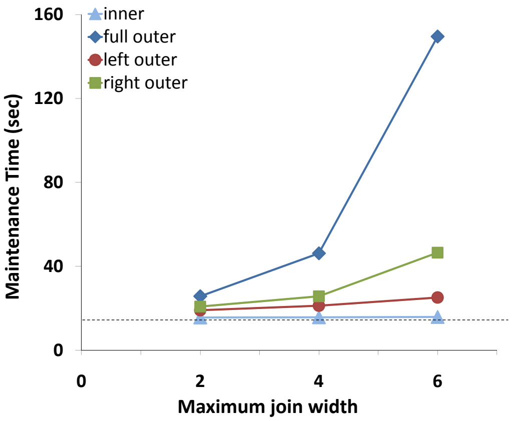


Figure 8.12: Maintenance time for different kinds and widths of ASRs, for chain topology of 30 peers with data at the leaf peer

comes in last. We also note that inner join performance improves, as the joins get eider, while for outer joins, performance gets worse beyond a certain length. One reason for the latter is that, for wider outer joins, the resulting ASR relation is significantly wider (e.g., in our experimental workload we have to add 26 attributes to it for every additional “unit” of width). Moreover, since in the outer join cases we have to consider subpaths of the complete path, the number of these subpaths grows roughly quadratically with the length of the complete path. Finally, recall that our “simulation” of outer joins using conjunctive queries repeats tuples that appear in a path for each subpath of it. Thus, this redundancy is more pronounced for wider outer joins, because they contain more subpaths. On the other hand, for inner joins the width of ASR relations is the same for paths of different length, since we have the same attributes at all peers (and would grow more slowly for any CDSS, since we only need to store joined attributes once). Moreover, in this case, longer paths can only contain the same or fewer tuples than shorter paths.

This redundancy affects the size of the instance produced from the propagation of local data, including peer, provenance and ASR relations. Thus, it also affects the maintenance time for different kinds of joins, as illustrated in Figure 8.12. In this graph, the dotted line indicates the maintenance time for the case where no ASRs are defined. The overhead of inner joins in both cases is minimal, and decreases for wider joins, due to the fact that splitting the chain topology into wider joins produces fewer ASR relations. At the other extreme, for full outer joins, the overhead for both maintenance time and instance size is significant, and grows fast with join width, due to the large number of subpaths stored in the ASR, and the redundancy between contained subpaths, as explained above. For left and right outer joins the number of subpaths is smaller — and grows more slowly with join width — and so does maintenance time. We note that in these cases the instance size actually gets smaller in terms of the total number of tuples in the instance beyond a certain join width, again because splitting the chain topology into wider joins produces fewer ASR relations. However, even in this case the produced ASR relations are wider, resulting in slower maintenance times.

We also performed the same experiment on a CDSS setting with 30 peers where mappings form a branched topology, such as the one shown in Figure 8.7, with 10k tuples of base data at the leaf peers (i.e., the three peers with no incoming mappings at the end of each branch). In this topology, the root query is translated to 13 unfolded rules, each containing paths along combinations of these branches. The total query processing times for this query, for different kinds and widths of joins, are illustrated in Figure 8.13, where the dotted line indicates the query processing time when no ASRs are defined. First, we observe that inner joins provide a significant performance benefit. Note that, going from a join width of 5 to 7, this speedup decreases. This is because, due to the branched topology and the large width of the joins, some of the unfolded queries do not contain completely some of the ASRs, and thus cannot take advantage of them. On the other hand, outer joins contain all subpaths and thus there is always a way to use them with any rule. However, the very large number of attributes they contain and their large size, partly due to

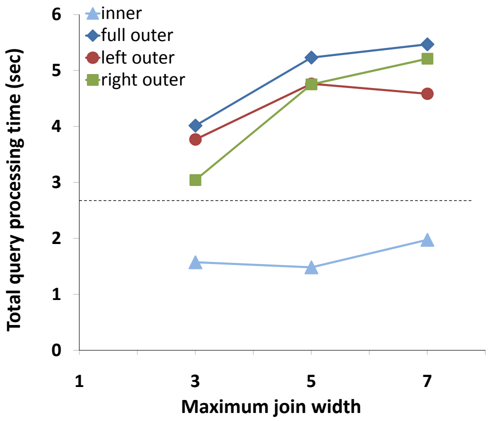


Figure 8.13: Total query processing time for different kinds and widths of ASRs, for branch topology of 30 peers with data at the leaf peer

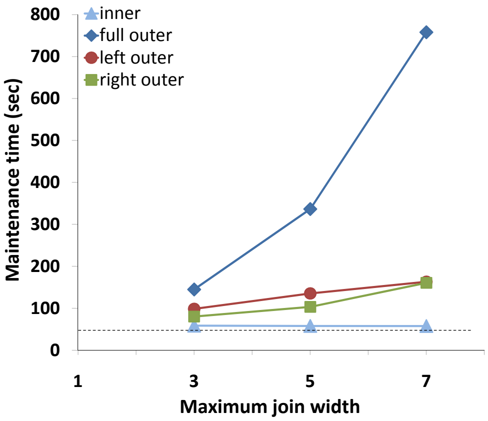


Figure 8.14: Maintenance time for different kinds and widths of ASRs, for branch topology of 30 peers with data at the leaf peer

redundancy, as explained above, end up having a negative impact in performance. Thus, as shown in the figure, query processing time for all kinds of outer joins is worse than in the case when no ASRs have been defined. Figure 8.14 shows that the large number of tuples and attributes in the outer join relations (more so in the case of full outer joins) also incurs a larger overhead in maintenance time for the different kinds of joins, while the overhead for the case of inner joins is minimal.

Finally, we investigate the performance of provenance querying on the 4-ary tree topology, which contains multiple overlapping short paths, as shown in Figure 8.10. In particular, we consider a setting with 30 peers, where each of the 24 leaf peers is initialized with 10k tuples of local insertions. For 30 peers, the longest path ending at the root peer only contains three mappings. For outer joins, since no overlapping between ASRs is allowed, we defined as many non-overlapping paths as possible, of length 3 or 2. Thus, only a small part of the paths in the mapping graph is covered by ASRs. For inner joins, we considered two cases: one where we defined the ASRs for the same paths as in the case of the outer joins, and another where we defined overlapping ASRs of length 3, for each path from a leaf

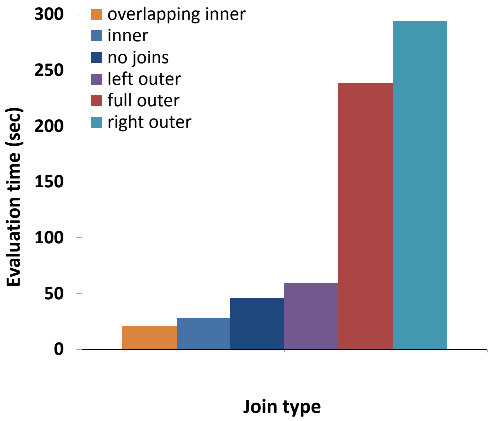


Figure 8.15: Total query processing time for different kinds and widths of ASRs, for 4-ary tree topology of 30 peers with data at the leaf peer

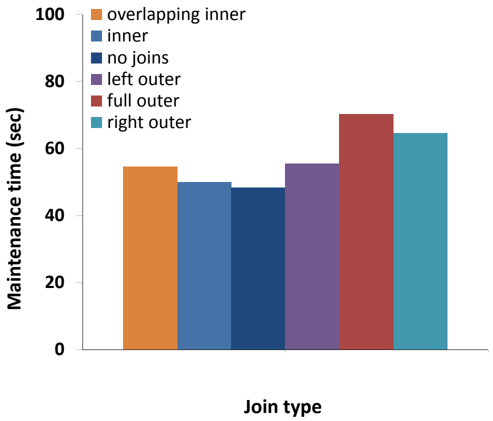


Figure 8.16: Maintenance time for different kinds and widths of ASRs, for 4-ary tree topology of 30 peers with data at the leaf peer

peer to the root.

Figure 8.15 shows the total query processing time for each of these cases. We observe that, as in the case of the branched topology, outer joins perform worse than the case where no ASRs have been defined (slightly worse for left outer joins, much worse for the case of full and right outer join). On the other hand, inner joins yield a significant performance benefit, especially when overlapping ASR definitions are allowed. Figure 8.16 shows the corresponding maintenance times for the different join kinds, showing a moderate overhead in the case of outer joins and a small overhead for inner joins. We note that the overhead for outer joins is relatively smaller for the 4-ary tree topology than for the chain and branched topologies. This is due to the fact that in the 4-ary topology the ASRs have relatively small width, while there is also a relatively small number of them, due to the requirement that they do not overlap.

Inner Join Width

In the experiments above, inner join ASRs always yielded a performance benefit, and in most cases this benefit increased for wider inner joins. In this section we investigate the effect of inner join width in more detail, and identify cases where wider inner joins may provide smaller performance benefits, as hinted before in the case of Figure 8.13 in the previous section.

We first examine the case of a chain topology of 30 peers, with local data at the leaf and middle peers, and the provenance query (hereby called "middle" query):

```
FOR [$R_{15}$ $x]
INCLUDE PATH [$x] <-+ []
RETURN $x
```

Figure 8.17 shows the effect of different maximum width of inner join ASRs on unfolding and evaluation times for the middle query, as well as maintenance time, i.e., the time for propagating 10k base tuples at each peer with local data. In this graph, width of 1 indicates the case where no ASRs have been defined. We observe that wider ASRs generally result in faster evaluation times. However, when the width reaches 15 the inner join is too wide to be used by the single unfolded rule, and the evaluation time is essentially the same as in the case where no ASRs have been defined. We also observe that unfolding time stays approximately constant for wider inner joins. This is because wider ASRs are more complex to unfold, but there are fewer of them. For the same reason, inner joins incur a small overhead to maintenance time, but this overhead stays approximately constant for different inner join widths.

Figure 8.18 shows a similar phenomenon for 30 peers in a branched topology with 10k base tuples at the leaf peers. Evaluation time for the root query improves for wider inner joins, but the benefit is smaller when the width reaches 7. This

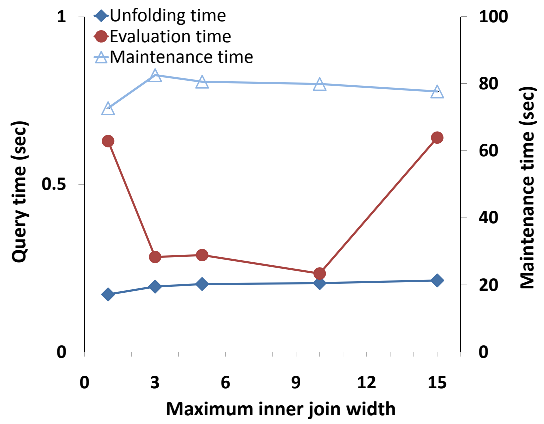


Figure 8.17: Performance of middle query for different inner join widths, in a chain topology of 30 peers with 10k data at the leaf peer

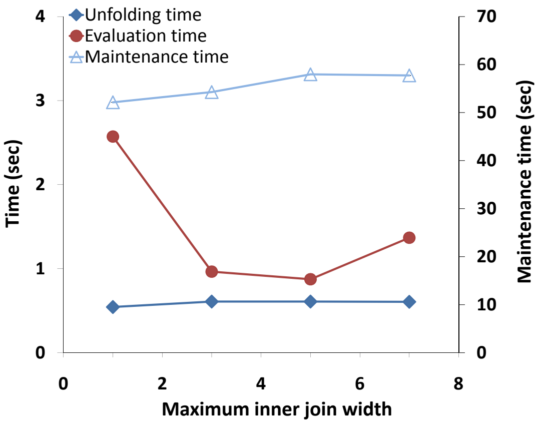


Figure 8.18: Performance of root query for different inner join widths, in a branched topology of 30 peers with 10k data at the leaf peers

is because these inner joins cross the boundaries where the branches "connect" with the chain in the topology of Figure 8.7, and the utilization of the ASRs in the unfolded rules is lower than in the case of narrower joins.

8.3.5 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. Moreover, inner join ASRs can be used to improve this performance significantly. 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 limit for scaling to larger settings comes from limitations of the underlying DBMS, regarding the size and complexity of the generated SQL queries. Inner join ASRs improve the performance of path traversal significantly, and the benefit increases for wider ASRs, as long as they do not exceed some width that depends on the particular query and topology. On the other hand, outer join ASRs often performed worse even than the case when no

ASRs have been defined. We believe that this is due to the large number of tuples in the outer join relations, that is partly due to the redundancy caused by our implementation of inner joins, using unions of conjunctive queries. In future work, we intend to explore the performance of ASRs using real outer joins, avoiding this redundancy, as well as techniques for their incremental maintenance. We expect that such outer join ASRs will provide a speedup that is closer to that of inner joins, since they will avoid some of the redundancy in our current outer join implementation. Moreover, for large outer join widths, we plan to consider ASRs that only contain a subset of all possible subpaths, depending on the particular mapping topology, to further decrease the number of tuples and attributes in outer join ASRs, which we believe to be the cause of the unsatisfactory performance of outer join ASRs in many of our experiments above.

Finally, the conclusions drawn from this experimental evaluation provide some hints for heuristics that can used for determining which ASRs to create and maintain, given a CDSS setting and a workload of provenance queries. One obvious such heuristic could be that, for any chain of mappings with no branches and no local data along the way, creating an as wide as possible inner join ASR is guaranteed to provide a significant performance benefit. Such heuristics, combined with cost-based techniques, as we discuss in Chapter 10, can form the basis a framework for automated index seletion for provenance queries.

Chapter 9

Related Work

Our work on CDSS update exchange takes advantage of previous work on PDMS (e.g., [64]) and on data exchange [45, 80, 65, 89]. With our encoding in datalog^{sk}, we reduce the problem of incremental updates in CDSS to that of recursive view maintenance where we contribute an improvement on the classic algorithm of [61]. Incremental maintenance of recursive views is also considered in [81] in the context of databases with constraints using the Gabrielle-Levi fixpoint operator; we plan to investigate the use of this technique for CDSS. The AutoMed system [83] implements data transformations between pairs of peers (or via a *public schema*) using a language called BAV, which is bidirectional but less expressive than tgds; the authors consider incremental maintenance and lineage [46] under this model. In [54], the authors use target-to-source tgds to express trust. Our approach to trust conditions has several benefits: (1) trust conditions can be specified between target peers or on mappings themselves; (2) each peer may express different levels of trust for other peers, i.e., trust conditions are not always "global"; (3) our trust conditions compose along paths of mappings. Finally, our approach does not increase the complexity of computing a solution.

Bidirectional update exchange is closely related to the *view update* problem, in which tuples in the base instance are to be changed in order to accomplish updates over the view, is more subtle because each source tuple may produce **several** tuples in the view. A given view update may thus introduce *side effects*: in order to modify one tuple in the view, we must modify a tuple in the base, which in turn causes other tuples in the same view to be inadvertently changed (a "side effect"). Dayal and Bernstein identified constraints under which an update does not introduce side effects within the same view [38]; other work has explored a variety of other, generally stricter, restrictions over what data is allowed to be affected [9, 38, 71]. Recent work [17] has considered restricted view definition languages in which view update is side effect-free. Generally the view update literature considers only a single view, which is typically a conjunctive query. Finally, [22] considered the related problem of *deletion minimization*, i.e., finding a set of source deletions that perform a view deletion and cause the *minimal* of side effects. This is different from our approach, where we want to detect and avoid all side effects, even at the cost of not performing the requested update.

We propose a novel provenance model that is useful for a variety of applications and generalizes previous models of provenance (lineage, why-provenance) and query answering on annotated relations. Lineage was first introduced in [35, 36] and why-provenance in [21] but the relationship with [66] was not noticed. The papers on probabilistic databases [52, 98, 76] note the similarities with [66] but do not attempt a generalization. Datalog with bag semantics in which derivation trees are counted was considered in several papers, among them [82, 85, 86]. The evaluation algorithms presented in these papers do not terminate if some output tuple has infinite multiplicity. Datalog on incomplete and on probabilistic databases is considered in [43, 77], again with non-terminating algorithms. Later [87] gave an algorithm for detecting infinite multiplicities in datalog with bag semantics and [53] gave a terminating algorithm for datalog on probabilistic databases.

Two recent papers develop among other things provenance models that bear a relationship to our approach. Like us, [27] identifies the limitations of lineage and why-provenance and proposes *route-provenance*, which is also related to derivation trees, but uses it for a different purpose—debugging schema mappings. Our model maintains a graph from which provenance can be incrementally recomputed or explored, whereas their model is based on recomputing the shortest routes on demand. [11] proposes a notion of lineage of tuples which is a combination of sets of relevant tuple ids and bag semantics. This is more detailed than the older notion of lineage [35] but we can also describe it by means of a special commutative semiring, so our approach is more general. The paper also does not mention recursive queries, which are critical for our work. Moreover, [11] does not support any notion of incremental translation of updates over mappings or incompleteness in the form of tuples with labeled nulls. Our provenance model also generalizes the duplicate (bag) semantics for datalog [86] and supports generalizations of the results in [87].

The first attempt at a general theory of relations with annotations appears to be [67] where axiomatized *label systems* are introduced in order to study containment. Our provenance model borrows the machinery of semirings and formal power series from the theory of formal languages (see [75] and references in there). For example, (non-commutative) algebraic systems of equations can be associated to context-free grammars and the integer coefficients in the formal power series solutions count the "degree of ambiguity" of a string in the language [29] (their restriction to grammars without unit rules inspired our Theorem 4.5.2). Context-free grammars have been used in the study of datalog but mainly *chain* datalog programs were considered (e.g., [5]) in order to capture the inherent order in strings. Closed semirings are used in [96, 32] but only in order to use Kleene's regular expression algorithm to optimize special classes of datalog programs.

Semirings have also been used in AI, in a line of work on *constraint satisfaction problems* (CSP) [14, 15]. Their constraints over semirings are in fact the same as our K-relations and the two operations on constraints correspond indeed to relational join and projection. CSP *solutions* are expressed as projection-join queries in [14] and as Prolog programs in [15]. Computing solutions is the same as the evaluation of join and projection in Section 4.2 and [15] also uses fixed points on semirings. There are some important differences though. The semirings used in [14, 15] are such that $+$ is idempotent and 1 is a top element in the resulting order. This rules out our semirings $\mathbb{N}, \mathbb{N}^\infty, \mathbb{N}[X], \mathbb{N}^\infty[[X]]$ hence the bag and provenance semantics.[1] More importantly, much of the focus in CSP is in choosing optimal solutions rather than how these solutions depend on the constraints.

The design of ProQL has been influenced by graph query languages, such as GraphLog [31], UnQL [20], Lorel [4], StruQL [47]. However, provenance graphs are different from the graph models of those languages, in that they have two kinds of nodes, for tuples and mappings, and paths of this graph need to maintain all inputs of each mapping node along them. Moreover, semiring evaluation capabilities as well as the fact that ProQL queries only compute projections of a provenance graph, without the ability to create new nodes or graphs, are unique in our language.[2] [55] propose a query language for data annotations consisting of blocks and colors, which can considered as a form of data provenance, that is however less suitable than our graph model for applications such as the ones discussed in Chapter 7.

A related line of work involves languages that manipulate annotations that are stored together with the data. Bhagwat et al. [13] propose an annotation mechanism for relational databases where annotations are stored in extra attributes, and extend the Select-Project-Join-Union fragment of SQL with a clause which allows the user to specify explicitly how annotations should propagate. The focus is thus on the propagation of the annotations through queries, and the issue of how to

[1] Another difference is that for datalog semantics we require our semirings to be ω-continuous while [15] uses the less well-behaved fixed points given by Tarski's theorem for monotone operators on complete lattices. However, the semiring examples [15] appear to be in fact ω-continuous.

[2] with the possible exception of XPath [30] for the latter

query the annotations themselves is not addressed. The DBNotes system [28] extends this framework and offers limited support of querying stored annotations. Finally, [19] studied the expressive power of languages that manipulate annotations explicitly, and compared them with the implicit provenance associated with a query or update. However, none of these approaches provides the flexibility of annotations from different semirings provided through a query and combined using appropriate operators for that semiring. Moreover, they refer to propagation through a single query, and do not deal with propagation through complex graphs of schema mappings, as in the case of CDSS.

Although we focus on data provenance, some of our provenance querying use cases have been influenced by work on workflow provenance querying [91, 26, 16]. In particular, the provenance queries proposed as part of the Second Provenance Challenge [91] includes identifying derived data that has been produced from specific source data. The use cases suggested by the authors of [26] include retrieving the provenance of a subset of all data or using provenance to filter data, either through a selection condition or "joining" data with common provenance. Finally, in [16] the authors propose the use of views to present (parts of) the graph to users at different levels of abstraction, in order to deal with the complexity of workflow provenance graphs and allow users to only focus on what is interesting to them.

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 with operations such as union and join, for which particular identities hold, and relationships between data come from the properties of these operations. Some of the unique features of our query language, such as the ability to use provenance to compute annotations, are only possible because the operators we consider form commutative semirings. On the other hand, workflow provenance models, such as [88], typically describe procedural workflows and involve operations that are treated as black boxes, because of their complexity, while connections between different operations are explicitly prescribed in the workflow specification. Identifying workflows whose runs can be described declaratively or whose operations satisfy similar identities, as in the case for relational operators, could be an interesting direction of future research.

Business process querying can also be considered a form of workflow provenance querying. In [10], the authors introduce BP-QL, a *visual* language for querying business processes, that have been produced by runs of BPEL specifications. This is especially convenient for BPEL users, because the graphical user interface used to formulate BP-QL queries resembles that of graphical BPEL specification editors. An analogous graphical interface could possibly be built on top of the graph projection component of ProQL, although in the case of schema mappings and/or datalog queries there is no "standard" graphical user interface for their specification that would be familiar to users. Moreover, it is not clear how annotation computation could be expressed in such a graphical interface.

Our encoding of the provenance graph in relations resembles the approach of [48, 49], 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 semistructured data [84, 56] to objects [73, 74] to XML [70, 33]. However, virtually all XML index techniques are based on the notion of a distinguished document root, and that they also do not tolerate cycles. Our queries can have multiple relation nodes of interest, and the provenance graph can indeed have cycles. It may be possible to leverage the index techniques for semistructured data in a custom provenance storage system, but in general their representations (state machines over the graph, with *extent* records pointing to data) do not fit well into the relational DBMS model we currently use. As explained in Section 8.2, ASRs [73, 74] were the most natural index technique to adapt for our provenance query scheme, and thus we chose to use them as the basis for our provenance indexing techniques, with encouraging perforance results, as we showed in Section 8.3. In future work, we intend to explore automated techniques, such as [6], to select the set of indexes to create, given a provenance graph and ProQL query workload.

Chapter 10

Conclusions and Future Research Directions

In this dissertation we have described the fundamental role of provenance in Collaborative Data Sharing Systems, both as a first-class artifact, that users can explore and query, and as an enabler for complex CDSS operations involved in update exchange. In order to realize the vision of such systems described in [68], we had to address several challenges.

First, we had to define *formal semantics* for *update exchange*, that tolerate disagreement between participants and allow them to filter data according to their own provenance-based trust policies, and develop algorithms to *perform* it. To address this challenge, in this dissertation we built upon techniques for exchanging *data* using networks of schema mappings to define the semantics of update exchange over unidirectional and bidirectional mappings.

Second, we needed a rich model of provenance that captures enough information as updates are propagated through mappings both to answer CDSS users queries, e.g., regarding trust, and to enable and optimize internal CDSS operations. For this reason, we defined a rich model of provenance for relational and datalog queries, based on the mathematical framework of semirings, that is sufficiently informative for the needs of a CDSS. In particular, we showed that this form of provenance can be used to compute annotations for propagated data, such as whether they should be trusted. Moreover, we described how this form of provenance can be stored in relations and maintained together with update exchange, using datalog programs generated from the schema mappings. We also presented algorithms exploiting this provenance information in order to enable or optimize update exchange operations, such as detecting which tuples are no longer derivable and should be deleted, when a deletion is propagated over unidirectional mappings, or avoiding updates that cause *side effects* at *run time*, in CDSS with bidirectional mappings. Finally, we developed a complete implementation of unidirectional and bidirectional update exchange in our ORCHESTRA CDSS prototype, with novel algorithms and encoding schemes to translate updates, maintain provenance, and apply trust conditions and provided a detailed experimental study of the scalability and performance of our implementation of CDSS operations, illustrating the feasibility of our approach

Last but not least, we discussed that CDSS users need tools that allow them to *exploit* the provenance of data, after updates have been exchanged, e.g., in order to include *provenance testing* in their data querying or *compute annotations* for their data that are useful for a variety of applications. For this reason, we first provided an alternative equivalent graph representation of our provenance semirings, called the *provenance graph model*, that is more suitable for visualization and querying by CDSS users, while also capturing relationships between the provenance of derived tuples. We then defined a query language for provenance graphs, ProQL which is useful in supporting a wide variety of applications with derived data. This language can be used to assess trust and derivability or detect side effects, as required for CDSS operations, as well as to express more complicated provenance queries and, optionally, compute data annotations in particular semirings. Finally, 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 of the benefit yielded by different indexing techniques.

10.1 Future Research Directions

The work in this dissertation laid the foundations for the realization of Collaborative Data Sharing Systems, and illustrated the importance of provenance both as enabler for update exchange operations and for provenance querying. However, there are several ways to extend the applicability, functionality and performance of update exchange and provenance querying.

10.1.1 Exploiting Provenance to Support More Flexible Update Exchange

In this dissertation, we have developed algorithms for propagating updates in settings with either only unidirectional or only bidirectional mappings. In practical data sharing scenarios it is more likely that a combination of them could be required, e.g., bidirectional mappings between "authoritative" sources and unidirectional ones from those sources to the databases of individual researchers who want to import data from them. However, this raises interesting research questions, such as identifying which updates to propagate to which sources while preventing side effects, i.e., causing unintended deletions of tuples with shared sources with those that the user deleted. Moreover, techniques from propagation along unidirectional mappings can be used in order to perform deletions that cannot be propagated to the sources without causing side effects. Furthermore, in our work on bidirectional mappings we took a conservative approach regarding the treatment of side effects:

we only delete source tuples that are not in the provenance of any side effect. However, in some cases it may be possible to delete some of those source tuples while not causing a side effect, e.g., if an alternative derivation exists. An interesting research direction would involve examining if provenance can be used in order to efficiently compute a *maximal* set of source tuples that can be deleted without causing side effects. These issues are closely related to the *view update* problem, which has been an ongoing research topic in the database community for the last 25+ years, and we believe that using rich provenance information could lead to a better solution.

10.1.2 ProQL User Evaluation and Possible Extensions

In Chapter 7 we proposed core semantics and a syntax for ProQL and showed how it can be used to express the use cases we presented in Section 7.1. In our design of ProQL we tried to envision the provenance querying requirements of CDSS users, as well as provenance querying use cases that have appeared in related work on data and workflow provenance querying. In the future, it would be important to deploy ProQL together with ORCHESTRA in a real-world system, such as pPOD [90], in order to assess its impact in practice, and possibly identify new use cases to be handled by future ProQL extensions.

One possible such extension involves the ability to assign values to mapping nodes individually, e.g., according to values of the tuples on which the mapping is applied, instead of in terms of a definition of a function in the corresponding semiring, as in the current language specification. However, to provide this functionality we would also need to extend our provenance model, intuitively to treat mapping nodes as “first-class” citizens, similar to the tuple nodes. Another possible extension involves an “advanced” form of projection over provenance graphs, that involves replacing paths of mappings with a single mapping representing their composition.

10.1.3 Evaluating ProQL Queries Over Cyclic Provenance Graphs

In this dissertation we presented a scheme for evaluating provenance queries over acyclic graphs, based on unfolding the queries according to the provenance schema graph. An alternative scheme — that may also provide better performance if there are large numbers of possible derivations for each tuple, as e.g., in the case when most peers have local data, and which also handles recursion — is to execute the set of rules in bottom-up fashion, materializing the results. We can convert each rule to an SQL query that creates or adds tuples to an intermediate relation. This query also adds two attributes, one to store the provenance value of the tuple, and another encoding the sequence of views by which the tuple was derived, up to but not including cycles. As in the per-derivation-tree method, we must ensure that derivations of the same tuple value with the same provenance value are not removed by SQL set semantics.

If there are no cycles present, we can perform a topological sort on the rule dependency graph, and execute the SQL queries according to that ordering. In the presence of recursion, we may need to iterate through this process multiple times, adding new tuples to intermediate relations in each step, until the computation reaches fixpoint.

Using this scheme, each view to be created may be defined as a union of conjunctive queries. For each conjunctive query in the union, our SQL statement will evaluate its output tuple's provenance attribute by applying the product operation to the values of its input tuples' provenance attributes. The creation of the sequence-of-views attribute is done using SQL string manipulation (the current view name is appended to the input sequence, unless the view name already appears within the sequence). Finally, as with the unfolding scheme presented in this dissertation, once the set of tuples is computed to fixpoint, we can take the resulting tuples and apply a GROUP BY and aggregation operation over them to produce our final result.

10.1.4 Cost-based Index Selection for Provenance Querying

In the experimental evaluation of our ProQL implementation we manually selected ASR definitions to index paths in the provenance graph, according to the mapping topology in each case. In general, one would like these definitions to be generated automatically, for a given workload of ProQL queries over a stored provenance graph. It would be interesting to investigate whether automated index selection techniques, such as [6] can be applied in our case, directly or with some extensions and combined with cost estimates from the optimizer of the underlying RDBMS. Moreover, it would be important to evaluate experimentally whether the indexing techniques presented in this dissertation can yield similar benefits for ProQL queries involving more complicated tree patterns.

Bibliography

[1] Karl Aberer, Philippe Cudré-Mauroux, and Manfred Hauswirth. The chatty web: Emergent semantics through gossiping. In *Proceedings of the Twelfth International World Wide Web Conference, Budapest, Hungary, May 20-24 2003*, 2003.

[2] Serge Abiteboul and Oliver Duschka. Complexity of answering queries using materialized views. In *Proceedings of the Seventeenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 254–263, Seattle, WA, 1998.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Winer. The Lorel query language for semistructured data. In *Proceedings of International Journal on Digital Libraries*, volume 1(1), pages 68–88, April 1997.

[5] Foto N. Afrati and Christos H. Papadimitriou. The parallel complexity of simple logic programs. *J. ACM*, 40(4):891–916, 1993.

[6] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB 2000,*

Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt, pages 496–505, 2000.

[7] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL. *Nucleic Acids Research*, 28:45–48, 2000.

[8] Andrey Balmin, Vagelis Hristidis, and Yannis Papakonstantinou. ObjectRank: Authority-based keyword search in databases. In *VLDB 2004, Proceedings of 30th International Conference on Very Large Data Bases, August 29-September 3, 2004, Toronto, Canada*, pages 564–575, 2004.

[9] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

[10] Catriel Beeri, Anat Eyal, Simon Kamenkovich, and Tova Milo. Querying business processes. In *VLDB 2006, Proceedings of 32nd International Conference on Very Large Data Bases, September 12-15, 2006, Seoul, Korea*, pages 343–354, 2006.

[11] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, and Jennifer Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB 2006, Proceedings of 31st International Conference on Very Large Data Bases, September 12-15, 2006, Seoul, Korea*, pages 953–964, 2006.

[12] Philip A. Bernstein, Fausto Giunchiglia, Anastasios Kementsietsidis, John Mylopoulos, Luciano Serafini, and Ilya Zaihrayeu. Data management for peer-to-peer computing: A vision. In *ACM SIGMOD Workshop on the Web (WebDB) 2002, Madison, WI*, June 2002.

[13] Deepavali Bhagwat, Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. In *VLDB 2004, Proceedings of 30th International Conference on Very Large Data Bases, August 29-September 3, 2004, Toronto, Canada*, pages 900–911, 2004.

[14] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[15] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint logic programming: syntax and semantics. *ACM TOPLAS*, 23(1):1–29, 2001.

[16] Olivier Biton, Sarah Cohen Boulakia, Susan B. Davidson, and Carmem S. Hara. Querying and managing provenance through user views in scientific workflows. In *Proceedings of the 24th International Conference on Data Engineering, April 7-12, 2008, Cancn, Mxico*, pages 1072–1081, 2008.

[17] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: A language for updateable views. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, IL*, pages 338–347, 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.

[18] Peter Buneman, James Cheney, Wang Chiew Tan, and Stijn Vansummeren. Curated databases. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 9-11, 2006, Vancouver, Canada*, pages 1–12, 2008.

[19] Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. In *Database Theory — ICDT 2003, 11th International Conference, Barcelona, Spain, January 10-12, 2007, Proceedings*, pages 209–223, 2007.

[20] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. *SIGMOD Rec.*, 25(2):505–516, 1996.

[21] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *Database Theory — ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*, 2001.

[22] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. On propagation of deletions and annotations through views. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 3-5, 2002, Madison, Wisconsin USA*, pages 150–158, 2002.

[23] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Logical foundations of peer-to-peer data integration. In *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 241–251, 2004.

[24] Michael J. Carey, Daniela Florescu, Zachary G. Ives, Ying Lu, Jayavel Shanmugasundaram, Eugene Shekita, and Subbu Subramanian. XPERANTO: Publishing object-relational data as XML. In *ACM SIGMOD Workshop on the Web (WebDB) 2000, Dallas, TX*, pages 105–110, 2000.

[25] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[26] Adriane Chapman, H. V. Jagadish, and Prakash Ramanan. Efficient provenance storage. In *SIGMOD 2008, Proceedings of the ACM International Conference on Management of Data, June 10-12, 2007, Vancouver, Canada*, pages 993–1006, 2008.

[27] Laura Chiticariu and Wang-Chiew Tan. Debugging schema mappings with routes. In *VLDB 2006, Proceedings of 31st International Conference on Very Large Data Bases, September 12-15, 2006, Seoul, Korea*. ACM Press, 2006.

[28] Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In *SIGMOD 2005, Proceedings of the-fourth ACM International Conference on Management of Data, June 14-16, 2005, Baltimore, MD*, pages 942–944, 2005.

[29] Noam Chomsky and Marcel-Paul Schützenberger. The algebraic theory of context-free languages. *Computer Programming and Formal Systems*, pages 118–161, 1963.

[30] James Clark and Steve DeRose. XML path language (XPath) recommendation. Available from `http://www.w3.org/TR/1999/REC-xpath-19991116`, November 1999.

[31] M. P. Consens and A. O. Mendelzon. Expressing structural hypertext queries in GraphLog. In *HYPERTEXT '89*, pages 269–292, New York, NY, USA, 1989.

[32] Mariano P. Consens and Alberto O. Mendelzon. Low complexity aggregation in graphlog and datalog. In *Proceedings of the International Conference on Database Theory*, 1990.

[33] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 341–350, 2001.

[34] Peter Crawley and Robert P. Dilworth. *Algebraic Theory of Lattices*. Prentice Hall, 1973.

[35] Yingwei Cui. *Lineage Tracing in Data Warehouses*. PhD thesis, Stanford University, 2001.

[36] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2), 2000.

[37] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *VLDB 2004, Proceedings of 30th International Conference on Very Large Data Bases, August 29-September 3, 2004, Toronto, Canada*, 2004.

[38] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, 1982.

[39] A. Deutsch, L.Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.

[40] Alin Deutsch, Bertram Ludäscher, and Alan Nash. Rewriting queries using views with access patterns under integrity constraints. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 352–367, 2005.

[41] Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. The chase revisited. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 9-11, 2006, Vancouver, Canada*, pages 149–158, 2008.

[42] Alin Deutsch and Val Tannen. Reformulation of XML queries and constraints. In *Database Theory — ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, pages 225–241, 2003.

[43] F. Dong and L. V. S. Lakshmanan. Deductive databases with incomplete information. In *Symposium on Logic Programming*, 1992.

[44] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proceedings of the Sixteenth ACM SIGMOD-SIGACT-SIGART*

Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona, USA, pages 109–116, 1997.

[45] Ronald Fagin, Phokion Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. *Theoretical Computer Science*, 336:89–124, 2005.

[46] Hao Fan and Alexandra Poulovassilis. Using schema transformation pathways for data lineage tracing. In *BNCOD*, volume 1, pages 133–144, 2005.

[47] Mary F. Fernandez, Daniela Florescu, Jaewoo Kang, Alon Y. Levy, and Dan Suciu. Catching the boat with strudel: Experiences with a web-site management system. In *SIGMOD 1998, Proceedings of the ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 414–425. ACM Press, 1998.

[48] Daniela Florescu and Donald Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report 3684, INRIA, March 1999.

[49] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, September 1999.

[50] J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated xml: queries and provenance. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 9-11, 2008, Vancouver, Canada*, pages 271–280, 2008.

[51] Marc Friedman, Alon Y. Levy, and Todd D. Millstein. Navigational plans for data integration. In *Proceedings of the AAAI Sixteenth National Conference on Artificial Intelligence, Orlando, FL USA*, pages 67–73, 1999.

[52] N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *TOIS*, 14(1), 1997.

[53] Norbert Fuhr. Probabilistic datalog — a logic for powerful retrieval methods. In *SIGIR*, 1995.

[54] Ariel Fuxman, Phokion G. Kolaitis, Renée J. Miller, and Wang-Chiew Tan. Peer data exchange. In *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, MD*, pages 160–171, 2005.

[55] Floris Geerts, Anastasios Kementsietsidis, and Diego Milano. Mondrian: Annotating and querying databases through colors and blocks. In *Proceedings of the 22nd International Conference on Data Engineering, April 3-8, 2006, Atlanta, GA, USA*, page 82, 2006.

[56] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, 1997.

[57] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *VLDB 2007, Proceedings of 32nd International Conference on Very Large Data Bases, September 25-27, 2007, Vienna, Austria*, 2007. Amended version available as Univ. of Pennsylvania report MS-CIS-07-26.

[58] 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, June 11-13, 2007, Beijing, China*, 2007.

[59] Todd J. Green and Val Tannen. Models for incomplete and probabilistic information. In *EDBT Workshops*, 2006.

[60] Todd J. Green, Nicholas Taylor, Grigoris Karvounarakis, Olivier Biton, Zachary Ives, and Val Tannen. ORCHESTRA: Facilitating collaborative data sharing. In *SIGMOD 2007, Proceedings of the ACM International Conference on Management of Data, June 11-14, 2007, Beijing, China*, 2007. Demonstration description.

[61] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 157–166. ACM Press, 1993.

[62] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 157–166. ACM Press, 1993.

[63] Alon Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[64] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema mediation in peer data management systems. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 505–516. IEEE Computer Society, March 2003.

[65] André Hernich and Nicole Schweikardt. CWA-solutions for data exchange settings with target dependencies. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, 2007.

[66] Tomasz Imielinski and Witold Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, 1984.

[67] Yannis E. Ioannidis and Raghu Ramakrishnan. Containment of conjunctive queries: beyond relations as sets. *ACM Transactions on Database Systems*, 20(3), 1995.

[68] Zachary Ives, Nitin Khandelwal, Aneesh Kapur, and Murat Cakir. ORCHESTRA: Rapid, collaborative sharing of dynamic data. In *CIDR 2005: Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA*, pages 107–118, January 2005.

[69] Grigoris Karvounarakis and Zachary G. Ives. Bidirectional mappings for data and update exchange. In *WebDB*, 2008. Extended version available as Univ. of Pennsylvania report MS-CIS-08-17.

[70] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings of the 18th International Conference on Data Engineering, February 26-March 1, 2002, San Jose, CA USA*, pages 129–140, 2002.

[71] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, TX, May 28-31, 1985*, pages 154–163. ACM, 1985.

[72] Anastasios Kementsietsidis, Marcelo Arenas, and Renée J. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *SIGMOD 2003, Proceedings of the ACM SIGMOD International Conference on Management of Data, June 9-12, 2003, San Diego, California, USA*. ACM, June 2003.

[73] Alfons Kemper and Guido Moerkotte. Access support in object bases. In *SIGMOD 1990, Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 364–374. ACM Press, 1990.

[74] Alfons Kemper and Guido Moerkotte. Advanced query processing in object bases using access support relations. In *Proceedings of 17th International Conference on Very Large Data Bases*, pages 290–301, San Francisco, CA, USA, 1990.

[75] W. Kuich. Semirings and formal power series. In *Handbook of formal languages*, volume 1, pages 609–677. Springer, 1997.

[76] Laks V. S. Lakshmanan, Nicola Leone, Robert Ross, and V. S. Subrahmanian. Probview: a flexible probabilistic database system. *ACM Trans. Database Syst.*, 22(3):419–469, 1997.

[77] Laks V. S. Lakshmanan and Fereidoon Sadri. Probabilistic deductive databases. In *Symposium on Logic Programming*, 1994.

[78] Maurizio Lenzerini. Tutorial - data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 3-5, 2002, Madison, Wisconsin USA*, 2002.

[79] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 251–262. Morgan Kaufman, 1996.

[80] Leonid Libkin. Data exchange and incomplete information. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, IL*, pages 60–69, 2006.

[81] James J. Lu, Guido Moerkotte, Joachim Schue, and V.S. Subrahmanian. Efficient maintenance of materialized mediated views. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, CA, May 26-26, 1995*, pages 340–351. ACM Press, 1995.

[82] M. Maher and R. Ramakrishnan. Déjà vu in fixpoints of logic programs. In *NACLP*, 1989.

[83] Peter J. McBrien and Alexandra Poulovassilis. P2P query reformulation over both-as-view data transformation rules. In *DBISP2P*, 2006.

[84] Tova Milo and Dan Suciu. Index structures for path expressions. In *Database Theory — ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings*, pages 277–295, 1999.

[85] Inderpal Singh Mumick. *Query Optimization in Deductive and Relational Databases*. PhD thesis, Stanford University, 1991.

[86] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *VLDB J.*, pages 264–277, 1990.

[87] Inderpal Singh Mumick and Oded Shmueli. Finiteness properties of database queries. In *Fourth Australian Database Conference*, February 1993.

[88] Open provenance model. http://twiki.ipaw.info/bin/view/Challenge/OPM, 2008.

[89] Lucian Popa, Yannis Velegrakis, Renée J. Miller, Mauricio A. Hernández, and Ronald Fagin. Translating web data. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China*, 2002.

[90] Processing phylOData (pPOD). http://phylodata.seas.upenn.edu/cgi-bin/wiki/pmwiki.php.

[91] Provenance challenge. http://twiki.ipaw.info/bin/view/Challenge/WebHome, 2007.

[92] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4), 1980.

[93] Anish Das Sarma, Omar Benjelloun, Alon Halevy, and Jennifer Widom. Working models for uncertain data. In *ICDE*, 2006.

[94] Partha Pratim Talukdar, Marie Jacob, Muhammad Salman Mehmood, Koby Crammer, Zachary G. Ives, Fernando Pereira, and Sudipto Guha. Learning to create data-integrating queries. In *VLDB 2008, Proceedings of 33nd International Conference on Very Large Data Bases, August 26-28, 2008, Auckland, New Zealand*, 2008.

[95] Nicholas E. Taylor and Zachary G. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD 2006, Proceedings of the ACM International Conference on Management of Data, June 27-29, 2006, Chicago, IL*. ACM, 2006.

[96] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, 1989.

[97] Ron van der Meyden. Recursively indefinite databases. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 364–378, 1990.

[98] Esteban Zimányi. Query evaluation in probabilistic relational databases. *Theoretical Computer Science*, 171(1-2), 1997.