COLLABORATIVE DATA SHARING WITH MAPPINGS AND PROVENANCE

Todd J. Green

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, Associate Professor, Computer and Information Science
Graduate Group Chairperson

Dissertation Committee

Susan B. Davidson, Professor, Computer and Information Science
Sanjeev Khanna, Professor, Computer and Information Science
Jeffrey F. Naughton, Professor, Computer Science, University of Wisconsin-Madison
Benjamin C. Pierce, Professor, Computer and Information Science

COPYRIGHT
Todd J. Green
2009

To Elisabeth

ABSTRACT

COLLABORATIVE DATA SHARING WITH MAPPINGS AND PROVENANCE

Todd J. Green

Supervisors: Zachary G. Ives and Val Tannen

A key challenge in science today involves integrating data from databases managed by different collaborating scientists. In this dissertation, we develop the foundations and applications of collaborative data sharing systems (CDSSs), which address this challenge. A CDSS allows collaborators to define loose confederations of heterogeneous databases, relating them through schema mappings that establish how data should flow from one site to the next. In addition to simply propagating data along the mappings, it is critical to record data provenance (annotations describing where and how data originated) and to support policies allowing scientists to specify whose data they trust, and when. Since a large data sharing confederation is certain to evolve over time, the CDSS must also efficiently handle incremental changes to data, schemas, and mappings.

We focus in this dissertation on the formal foundations of CDSSs, as well as practical issues of its implementation in a prototype CDSS called Orchestra. We propose a novel model of data provenance appropriate for CDSSs, based on a framework of semiring-annotated relations. This framework elegantly generalizes a number of other important database semantics involving annotated relations, including ranked results, prior provenance models, and probabilistic databases. We describe the design and implementation of the Orchestra prototype, which supports update propagation across schema mappings while maintaining data provenance and filtering data according to trust policies. We investigate fundamental questions of query containment and equivalence in the context of provenance information. We use the results of these investigations to develop novel approaches to efficiently propagating changes to data and mappings in a CDSS. Our approaches highlight unexpected connections between the two problems and with the problem of optimizing queries using materialized views. Finally, we show that semiring annotations also make sense for XML and nested relational data, paving the way towards a future extension of CDSS to these richer data models.

Contents

1 Introduction **1**

1.1 Overview of CDSS and Orchestra . . . 3

1.2 Overview of Technical Contributions . . . 12

1.3 Roadmap . . . 25

2 Provenance Semirings **27**

2.1 Queries on Annotated Relations . . . 28

2.2 Positive Relational Algebra . . . 30

2.3 Polynomials for Provenance . . . 34

2.4 A Hierarchy of Provenance . . . 36

2.5 Datalog on *K*-Relations . . . 39

2.6 Formal Power Series for Provenance . . . 44

2.7 Computing Provenance Series . . . 46

2.8 Application to Incomplete/Probabilistic Databases . . . 49

2.9 Provenance-Annotated Queries . . . 49

3 Update Exchange in Orchestra **53**

3.1 CDSS Update Exchange . . . 55

3.2 Update Exchange Formalized . . . 60

3.3 Performing Update Exchange . . . 67

3.4 Implementation . . . 75

3.5 Experimental Evaluation . . . 77

4 Optimizing Queries on Annotated Relations **85**

4.1 Preliminaries . . . 87

4.2 Containment Mappings . . . 89

4.3 Bounds from Semiring Homomorphisms . . . 90

4.4 Main Results 91
4.5 Datalog 103

5 Ring-Annotated Relations and Differences **106**
5.1 Applications of Differences 109
5.2 Z-Relations 111
5.3 Reformulation Using Views 114
5.4 Finding Query Rewritings 123
5.5 Applications to Bag and Set Semantics 125
5.6 Built-in Predicates 128
5.7 Z[X]-relations 129

6 Semiring-Annotated XML **131**
6.1 Semiring Annotations 133
6.2 Annotated and Unordered XML 134
6.3 A Security Application 142
6.4 Incomplete and Probabilistic *K*-UXML 144
6.5 Semantics via Complex Values 146
6.6 Commutation with Homomorphisms 152
6.7 Semantics via Relations 154

7 Related Work **158**
7.1 Paradigms for Data Integration 158
7.2 Provenance and Annotated Data Models 164
7.3 Update Exchange 165
7.4 Query Containment and Equivalence 166
7.5 Ring-Annotated Relations and Updates 167
7.6 Semiring-Annotated XML 168
7.7 Further Work on Orchestra 168

8 Conclusions and Future Directions **170**
8.1 Immediate Next Steps 171
8.2 Longer-Term Directions 172

Bibliography **174**

List of Figures

1.1 Mappings among three bioinformatics databases 5
1.2 Bioinformatics database instances before and after data exchange 7
1.3 Two views of provenance for bioinformatics example 11
1.4 Tabular provenance representation using system of equations 16
1.5 Complexity of containment and equivalence with provenance annotations 21
1.6 Update propagation and query reformulation 22
1.7 Mapping evolution and query reformulation 22
1.8 Representing and computing updates with ring-annotated relations 23
1.9 Semiring-annotated XML example. 25

2.1 A maybe-table and a query result 29
2.2 Result of Imielinski-Lipski computation 29
2.3 Bag semantics example 30
2.4 Probabilistic example 30
2.5 Lineage and provenance polynomials 34
2.6 Comparison of provenance annotations 37
2.7 Provenance hierarchy 39
2.8 Datalog with bag semantics 39
2.9 Datalog example 42
2.10 Derivation trees for Datalog example 43
2.11 Probabilistic data integration example 52

3.1 Mappings among three bioinformatics databases 55
3.2 Dataflow at a single peer in a CDSS 57
3.3 Local edit logs and update translation 59
3.4 Provenance graph for bioinformatics example 60
3.5 Effects of a peer's edit log 61

3.6 Provenance graph for derivability testing example. 71

3.7 Example provenance graph showing relationships between tuples. 74

3.8 Relative performance of deletion propagation algorithms 80

3.9 Time to join and relation instance sizes 81

3.10 Scalability of incremental update propagation algorithms 82

3.11 Scalability for increasing base data sizes 83

3.12 Effects of mapping fan-in/fan-out on storage overhead and performance 84

4.1 Complexity of containment and equivalence 86

4.2 Logical implications of containment and equivalence 88

5.1 Algebraic identities for the difference operator under $\mathbb{Z}$-semantics 113

6.1 Simple `for` Example. 135

6.2 *K*-UXQuery Syntax. 135

6.3 *K*-UXQuery Typing Rules. 136

6.4 XPath Example. 137

6.5 Relational (encoded) example. 138

6.6 Extended Annotations Example. 139

6.7 Translation from positive relational algebra to *K*-UXQuery 140

6.8 Security Clearance Example. 143

6.9 Syntax and Typing rules for *NRC* 147

6.10 Semantic Equations for $NRC_K + srt$ 150

6.11 Compilation of UXQuery 151

6.12 Compilation of XPath to $NRC_K + srt$ 151

7.1 Data warehouse architecture 159

7.2 Feature comparison of data integration paradigms 160

7.3 Virtual data integration architecture 161

7.4 Peer-to-peer data integration architecture 162

7.5 Data exchange architecture 163

Acknowledgments

I would like to thank my advisers, Zack Ives and Val Tannen, for their generous support and guidance during my studies at Penn. It was a privilege to have been able to work with them.

Most of the work described in this dissertation was performed in collaboration with others, including my advisors and my fellow PhD students Grigoris Karvounarakis and Nate Foster. I would like to thank them here, and emphasize that many of the contributions presented in the present document were shaped by several of us working together. In particular, the material in Chapter 2 is based on a paper [62] by the author with Grigoris Karvounarakis and Val Tannen, as part of the larger Orchestra project headed by Zack Ives; Chapter 3 is based on a paper [60] by the author with Grigoris Karvounarakis, Zack Ives, and Val Tannen; Chapter 5 is based on a paper [59] by the author with Zack Ives and Val Tannen; and Chapter 6 is based on a paper [48] by the author with Nate Foster and Val Tannen.

The work described in this dissertation benefited at many points from useful feedback and suggestions from others. I would like to thank in particular Olivier Biton, James Cheney, Jan Chomicki, Sarah Cohen-Boulakia, Nilesh Dalvi, Floris Geerts, Giorgio Ghelli, Renée Miller, Tova Milo, Kristoffer Rose, Jérôme Siméon, Nicholas Taylor, Stijn Vansummeren, and the members of the Penn database group. I also would like to thank the members of my dissertation committee for their thoughtful feedback and comments that helped improve this document.

Finally, I would like to thank my wife, Elisabeth Dubin, for her love and support over the past six years.

Chapter 1

Introduction

Sharing *structured* data is one of the most basic yet persistently vexing problems of data management, and has been studied since the earliest days of database research. In recent years, with the growth of the Internet and explosion of data on the Web, there has been a renewed and sustained push to develop solutions for integrating data from heterogeneous sources, translating data between different schemas, and other forms of data sharing. Much progress has been made in both establishing solid theoretical foundations and developing practical systems. These efforts range from pre-loading all data into a single database instance after transforming it to a common format, then answering queries over this instance (as with data exchange [45]); to transforming data "on demand" or "on the fly" to answer specific queries (as with virtual data integration [135]). Commercially available tools using these techniques are available in several forms, including data warehousing systems and enterprise information integration systems.

Despite this progress, the practical impact of data sharing solutions proposed to date has been surprisingly modest, limited mainly to relatively small-scale efforts centralized inside a single corporation or controlling entity. At the same time, the need for sharing data across broader communities is increasing, especially in the sciences, which have become highly data-driven as they attempt to tackle larger questions. In bioinformatics, for example, there is a plethora of different databases, each providing a different perspective on a collection of organisms, genes, proteins, diseases, and so on. The data in these databases is highly interrelated — e.g., there are links between genes and proteins, and gene homologs between species — but the databases have proved very difficult to integrate using existing technologies.

One major reason for this is that conventional data integration tools require the development of a single global schema and assume that the data is globally consistent. In practice, both requirements are unrealistic. Designing a single schema for an entire community is arduous, involves many revisions, and requires a central administrator. Data from different sources can be often contradictory, reflecting different points of view, while current data integration tools assume there should be a consensus or “clean” version of the data. Moreover, users of integrated data often prefer to carefully **curate** it, manually or semi-automatically, rejecting some items and modifying others. The goals of curation are to eliminate inconsistencies, to fit data to user assumptions, as well as to enforce different degrees of **trust** in the sources. Tolerating inconsistency and supporting curation requires a measure of decentralization in data sharing.

Another key requirement in many data sharing scenarios, not supported by current tools, is to track and record where data has come from and how it has come to be where it is, in other words, its **provenance**. In scientific data sharing scenarios, this is not optional: data provenance is an essential part of the scientific record as well as crucial information used in curation. However, as data sharing typically involves complex transformations across the different schemas of participants, formulating the “right” notion of data provenance to capture these complexities is a challenging problem in itself.

Finally, most data sharing scenarios have important **dynamic** aspects. Data is constantly being **updated** – refined, corrected, expanded – and the participants naturally desire the most recent version of it. As has been observed since the earliest data integration investigations, this creates a tension between the efficiency of answering queries and the freshness of the obtained answers, with data warehousing favoring the former and virtual integration the latter. Moreover, it is not just the data that is dynamic, but also the set of participants, their schemas, and the relationships among them (as we shall see, the *mappings* which relate data sources). Especially in the “bootstrapping” phase of a collaboration — where the complex relationships among participants and data are only partially understood — such changes can occur as often as changes to the source data. We need a platform which copes with changes both to data and to the mappings among data sources.

To address these practical requirements, Ives, Khandelwal, and Kapur [82] described a vision for a **collaborative data sharing system** (**CDSS**), an approach that supports looser, decentralized confederations of participants (peers) with local curation abilities and tolerates inconsistencies in the data while still preserving a notion of semantic consistency. The goal of this dissertation is to realize this vision through a host of novel concepts and algorithms, developing both theoretical foundations and implementation techniques for supporting data provenance, dealing efficiently with the dynamic aspects of data and mappings through incrementality and optimization, and finally, evaluating these ideas in a prototype system called ORCHESTRA.

The main contributions of the dissertation are as follows:

- We develop a powerful formal model of provenance for relational data that fulfills CDSS requirements and captures as special cases many other provenance models proposed in earlier work.
- We implement these ideas in the Orchestra prototype CDSS, solving key engineering challenges such as the representation of provenance and efficient propagation of data updates in an off-the-shelf DBMS, and we also validate experimentally the feasibility of our approach.
- We make a systematic study of the fundamental questions of query containment and equivalence in the presence of provenance information, and obtain positive decidability results and complexity characterizations for five different models of provenance.
- We use these results to develop algorithms for efficient propagation of updates to mappings in a CDSS. The algorithms highlight unexpected connections with the problem of data update translation, and make fundamental use of provenance information. We also implement and evaluate these algorithms in the Orchestra prototype.
- We show that our provenance model extends to richer data models such as XML and nested relations, giving further evidence of the robustness of our approach and paving the way towards a future extension of CDSS to these data models.

The development of Orchestra and CDSS has itself been a collaborative effort, and we emphasize that most of the contributions described here were developed in collaboration with others, including Nate Foster, Zack Ives, Grigoris Karvounarakis, Val Tannen, and the members of the Penn database group. We provide a more detailed attribution of the shared contributions in Section 1.2.

In the remainder of this chapter, we will give an informal overview of the the main functionalities of the Orchestra CDSS, touching on the key issues mentioned above (Section 1.1), then elaborate on the technical contributions of the dissertation (Section 1.2). We finish with an outline of the rest of this dissertation (Section 1.3).

1.1 Overview of CDSS and Orchestra

The CDSS model builds upon the fundamentals of data integration, peer data management systems [73] (PDMS), and data exchange [45], while incorporating major new functionalities. 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, and the conditions under which the peer trusts data from other peers. In most of this dissertation we assume relational schemas, but our model extends naturally to other data models such as XML or nested relations (see Chapter 6). In this section, we give first an overview of how the peers' data is related by schema mappings, as used in PDMS and data exchange. Then we discuss the semantics of query answers where ORCHESTRA follows the principles of **data exchange** [45]. Most importantly, we describe ORCHESTRA's fundamentally novel features regarding **update exchange**, data provenance, and dynamic update strategies through incrementality and optimization.

The topology of data sharing: specifying how peers are related

Earlier work on PDMS pioneered the notion of supporting multiple mediated schemas, thereby relaxing some aspects of administration, and we adopt this perspective as a starting point in CDSS. As in PDMS, a CDSS collaboration begins with a set of *peer* databases, each with their own schema and local data instance.

Example 1.1. Consider (see Figure 1.1) a bioinformatics collaboration scenario based on databases of interest to affiliates of the Penn Center for Bioinformatics. In general, GUS, the Genomics Unified Schema[1] covers gene expression, protein, and taxon (organism) information; BioSQL, affiliated with the BioPerl project[2], covers very similar concepts; and a third schema, uBio[3], establishes synonyms among taxa. Instances of these databases contain taxon information that is autonomously maintained but of mutual interest to the others. For the purposes of our example we show only one relational table in each of the schemas, as follows. Peer GUS associates taxon identifiers, scientific names, and what it considers *canonical* scientific names via relation $G(\textsc{gid}, \textsc{nam}, \textsc{can})$; peer BioSQL associates its own taxon identifiers with scientific names via relation $B(\textsc{bid}, \textsc{nam})$; and peer uBio records synonyms of scientific names via relation $U(\textsc{nam}_1, \textsc{nam}_2)$. □

Again as in PDMS, the participants of the CDSS collaboration specify the relationships among their databases using **schema mappings**.

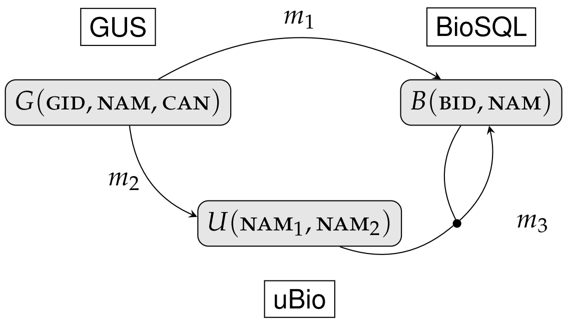
Example 1.2. Continuing with Example 1.1, suppose it is agreed in this collaboration that certain data in GUS should also be in BioSQL. This represented in Figure 1.1 by the arc labeled m_1. The specification $G(g, n, c) \rightarrow \exists b\ B(b, n)$ associated with m_1 is read as follows: if (g, n, c) in table G, the value n must also be in some tuple (b, n) of table B, although the value b in a such a tuple is not determined. The specification just says that there must be such a b and this is represented by the existential quantification $\exists b$. Here m_1 is an example of **schema mapping**. Two other mappings appear in Figure 1.1. Peer uBio should also have some of GUS's data, as specified by m_2. Mapping

---

[1] `http://www.gusdb.org`

[2] `http://bioperl.org`

[3] `http://www.ubio.org`


GUS
m_1
BioSQL
$G(\textsc{gid}, \textsc{nam}, \textsc{can})$
$B(\textsc{bid}, \textsc{nam})$
m_2
$U(\textsc{nam}_1, \textsc{nam}_2)$
m_3
uBio

$$
\begin{array}{ll}
(m_1) & G(g,n,c) \to \exists b\ B(b,n) \\
(m_2) & G(g,n,c) \to U(n,c) \\
(m_3) & B(b,n_1) \wedge U(n_1,n_2) \to B(b,n_2)
\end{array}
$$

Figure 1.1: Mappings among three bioinformatics databases

m_3 is quite interesting: it stipulates data in BioSQL based on data in uBio but also on data *already* in BioSQL. As seen in m_3, relations from multiple peers may occur on either side. We also see that individual mappings can be “recursive” and that cycles are allowed in the graph of mappings. $\square$

Schema mappings are logical assertions that the data instances at various peers are expected to jointly *satisfy*. We shall see (Section 1.2) that they correspond to the well-known formalism of *tuple-generating dependencies* (tgds) [7]. We shall also see (Chapter 3) that, as in data exchange [45], a large class of mapping graph cycles can be handled safely, while certain complex examples cause problems.

Thus, every CDSS specification begins with a collection of peers/participants, each with its own relational schema, and a collection of schema mappings between some of these peers. Like the schemas, the mappings are designed by the participants’ administrators. By joining Orchestra, the participants agree to share the data from their local databases. The sharing can be further modulated through the mappings, which should therefore be subject to agreement between participants.

The semantics of query answers

Given a CDSS configuration of peers and schema mappings, the question arises of how data should be propagated using the mappings, and what should be the answer to a query asked by one of the peers. The whole point of data integration is for such an answer to use data from *all* the peers. In CDSS, we wish to accomplish this by materializing at every peer an instance containing not only the peer’s locally contributed data, but also additional facts that *must* be true, given the data at the other peers along with the constraints specified by the mappings. Queries at a peer will be answering using this local materialized instance. However, while the mappings relating the peers tell us which peer instances are together considered “acceptable,” they do not fully specify the complete peer instances to materialize.

In CDSS we follow established practice in data integration, data exchange and incomplete information databases [1, 45] and use **certain answers** semantics: a tuple is "certain" if it appears in the query answer no matter what data instances (satisfying the mappings) we apply the query to. In virtual data integration, the certain answers to a query are computed by reformulating the query across all peer instances using the mappings, and combining the answers together from the local results computed at each peer. In CDSS, as in data exchange, we materialize special local instances that can be used to compute the certain answers. This makes query evaluation a fast, local process.

We illustrate this with our running example.

Example 1.3. Suppose the contents of G, U, and B are as shown in Figure 1.2(a). Note that the mappings of Figure 1.1 are not satisfied: for example, G contains a tuple

$$(828917, \text{“}\textit{Oscinella frit}\text{”}, \text{“}\textit{Drosophila melanogaster}\text{”})$$

but B does not contain any tuple with "*Oscinella frit*" which is a violation of m_1. Let us patch this by adding to B a tuple $(\perp_1, \text{“}\textit{Oscinella frit}\text{”})$ where $\perp_1$ represents the unknown value specified by $\exists b$ in the mapping m_1. We call $\perp_1$ a **labeled null**. Adding just enough patches to eliminate all violations results in the data in Figure 1.2(b). (Note that sometimes patching one violation may introduce a new violation, which in turn must be patched; hence this process is generally an iterative one, however under certain constraints it always terminates.) Observe that we can replace $\perp_1$ and the other labeled nulls with combinations of arbitrary values and we get an entire class of data instances that satisfy the mappings.

Now, consider two relational algebra queries:

$$\begin{aligned} Q_1 &\overset{\text{def}}{=} \sigma_{\text{NAM}=\text{“}\textit{Oscinella frit}\text{”}}(B) \\ Q_2 &\overset{\text{def}}{=} \pi_{\text{NAM}}(\sigma_{\text{NAM}=\text{“}\textit{Oscinella frit}\text{”}}(B)) \end{aligned}$$

and let us see which answers are *certain* when we apply these queries to the data instances obtained by replacing the labeled nulls with various values. There is no tuple in common among the various answers Q_1 produces, hence its certain answer semantics is empty. However, all answers produced by Q_2, have the tuple $(\text{“}\textit{Oscinella frit}\text{”})$ in common. This is a certain answer for Q_2. $\Box$

The procedure we used in the example to resolve mapping violations by patching instances is intuitive but it is not clear that (1) it always works, and (2) the data instances obtained by replacing labeled nulls with arbitrary values are representative of *all* instances satisfying the mappings and therefore give us the certain answers. In fact, the theory of **data exchange** [45] has resolved both these problems (see Section 1.2 and Chapter 3). Moreover, it has established the following

G :

GID	NAM	CAN
828917	*Oscinella frit*	*Drosophila melanogaster*
2616529	*Musca domestica*	*Musca domestica*

B :

BID	NAM
4472	*Periplaneta americana*

U :

NAM$_1$	NAM$_2$

(a) Tables from GUS, uBio, and BioSQL before data exchange (U is empty)

U :

NAM$_1$	NAM$_2$
Oscinella frit	*Drosophila melanogaster*
Musca domestica	*Musca domestica*

B :

BID	NAM
4472	*Periplaneta americana*
$\perp_1$	*Oscinella frit*
$\perp_1$	*Drosophila melanogaster*
$\perp_2$	*Musca domestica*

(b) Updated tables after data exchange. Shaded rows indicate newly-inserted tuples. (G is unchanged.)

Figure 1.2: Bioinformatics database instances before and after data exchange

convenient query answering algorithm: to obtain the certain answers to a query it suffices to evaluate the query over a specifically computed data instance with labeled nulls (as if the labeled nulls were ordinary values) and then to discard any tuples in the result containing labeled nulls. As a consequence, ORCHESTRA works with peer instances in which tuples may contain labeled nulls (actually, in the slightly more complicated form of **Skolem terms**; see Section 1.2).

Dynamic operation and update exchange

We have seen that ORCHESTRA performs query answering on locally stored instances rather than on-the-fly. This raises the question of data freshness in the peer instances built via data exchange. Data exchange is essentially a static procedure: it focuses on the one-time computation of entire peer instances satisfying the mappings. Across an entire CDSS this can be disruptive, as not all peers may wish to refresh their data at the same time.

In contrast, the CDSS design sees data sharing as a fundamentally *dynamic* process, with frequent "refreshing" *data updates* that need to be propagated efficiently. This process of dynamic update propagation is called *update exchange*. It is closely related to the classical *view maintenance problem*, and we shall discuss these connections at length in later chapters.

Operationally, CDSS functions in a manner reminiscent of *revision control systems*, but with peer-centric conflict resolution strategies. The users located at a peer 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 [131]. 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 incremental *update translation* using the schema mappings to compute corresponding updates over P's local instance: the translation finds matches of incoming tuples to the LHS of the mapping, and applies these matchings to the RHS to produce outgoing tuples (recall the "patching" process in Example 1.3).

Doing this only on updates means performing data exchange incrementally, with the goal of maintaining peer instances satisfying the mappings. Therefore, in CDSS the mappings are more than just static specifications. They can be seen as underlying the dynamic process of *propagation* of updates.

Example 1.4. Refer again to Figure 1.2(b), and suppose now that the curator of uBio updates her database by adding another synonym for the fruit fly: $U(\text{“}Oscinella\ frit\text{”}, \text{“}Oscinella\ frit\ Linnaeus\text{”})$. When this update is published, it introduces a violation of mapping m_3 that must again be "patched." The update translation process therefore inserts a corresponding tuple into BioSQL: $B(\bot_1, \text{“}Oscinella\ frit\ Linnaeus\text{”})$. ☐

Local curation and trust policies

The CDSS paradigm also differs from traditional data exchange in its incorporation of mechanisms allowing *local curation* of database instances. This allows CDSS users to "override" the system and modify or delete any data in their local database, even data that has been imported from elsewhere via schema mappings. In this way, CDSS users retain full control over the contents of their local database. The technical obstacle in supporting such a feature is how to preserve a notion of semantic consistency with respect to the mappings, when such modifications may introduce violations of the mappings.

Example 1.5. Refer again to Figure 1.2(b), and suppose that the curator of BioSQL decides that she is not interested in house flies, and wishes to delete tuple $B(\bot_2, \text{“}Musca\ domestica\text{”})$ from her local instance. Since as we have seen, the presence of this tuple was *forced* by the mappings, the deletion of the tuple leads to a violation of the mappings. ☐

To allow such local curation, CDSS stores deleted tuples in *rejection tables*, and converts user-specified mappings into internal mappings that take the rejection tables explicitly into account. The notion of "solution" adopted in CDSS is with respect to these converted mappings. Details are discussed in Chapter 3. Note that, sometimes, local curation corrects *mistakes* in the imported data, and it is desirable for the corrections to be propagated back to the sources from which the data came. To support this, the ORCHESTRA prototype incorporates facilities for *bidirectional* *mappings*. We do not discuss bidirectional mappings in this dissertation, but the interested reader can refer to [87, 61, 86] for details.

CDSS incorporates a second mechanism for local control of database instances in the form of *provenance-based trust policies*. These are essentially selection predicates, but of a special kind that operates on *data provenance*. They allow CDSS administrators to specify which data is "trusted," depending on its provenance (ORCHESTRA's mechanism for managing data provenance is discussed in Section 1.2).

Example 1.6. Some possible trust policies in our bioinformatics example:

- Peer BioSQL distrusts any tuple $B(b, n)$ if the data came from GUS and $n =$ "*Musca domestica*", and trusts any tuple from uBio.
- Peer BioSQL distrusts any tuple $B(b, n)$ that came from mapping m_2. □

Once specified, the trust policies are incorporated in the update exchange process: when the updates are being translated into a peer P's schema they are accepted or rejected based on P's trust conditions.

The example above illustrates *Boolean trust policies*, which specify a black-or-white classification of tuples as either (completely) trusted or (completely) untrusted, depending on their provenance and contents. In fact, CDSS also allows richer forms of *ranked trust policies* in which *trust scores* are computed for tuples indicating various "degrees of trust." When a conflict is is detected among data from multiple sources (for example, by a primary key violation), these scores can be used to resolve the conflict by selecting the tuple with the highest trust score and discarding those with which it conflicts. We focus mainly on Boolean trust policies in this dissertation, but briefly discuss ranked trust policies in Chapter 3.

Provenance

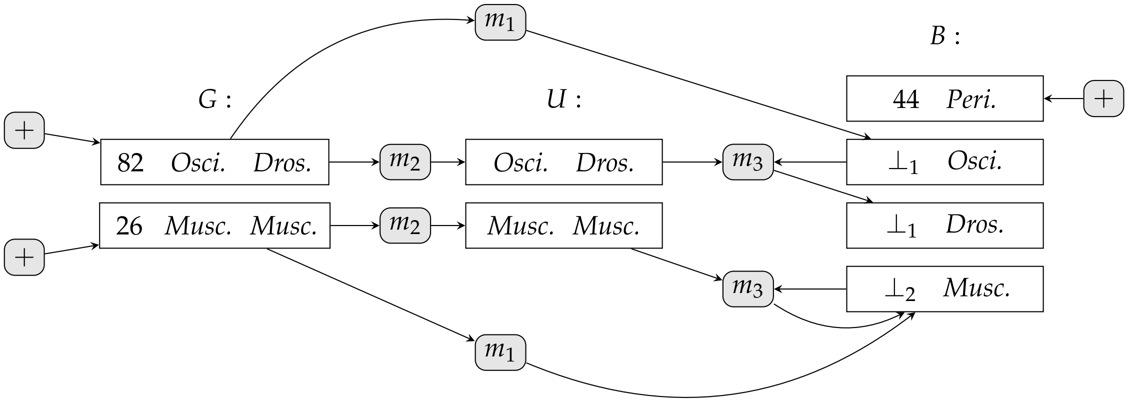
A major novel feature of CDSS, and a central focus of this dissertation, is the tracking of *data provenance*. This is the information which records "how data was propagated" during the update exchange process. Data provenance is needed for several purposes in CDSS: as the foundation of provenance-based *trust policies*, for *update exchange* itself, as a means of guiding efficient incremental algorithms, and as additional information that can be displayed and searched by CDSS users as they perform data curation and other manual tasks. Chapter 2 develops the formal machinery of the provenance model used in CDSS; here we just give an informal overview highlighting the main features.

The most intuitive way to picture CDSS provenance is as a *graph* having two kinds of nodes: *tuple nodes*, one for each tuple in the system, and *mapping nodes*, where several such nodes can be labeled by the same mapping name. Edges in the graph represent *derivations* of tuples from other tuples using mappings. Special mapping nodes labeled "+" are used to identify original source tuples.

Example 1.7. Refer to Figure 1.3, which depicts the provenance graph corresponding to the bioinformatics example from Figure 1.2 (we have abbreviated the data values to save space). Observe there is a "+" mapping node pointing to $G(26, \mathit{Musc.}, \mathit{Musc.})$; this indicates that it is one of the source tuples from Figure 1.2(a), present before data exchange was performed. Next, observe that $G(26, \mathit{Musc.}, \mathit{Musc.})$ is connected to $U(\mathit{Musc.}, \mathit{Musc.})$ via a mapping node labeled m_2. This indicates that $U(\mathit{Musc.}, \mathit{Musc.})$ was derived using $G(26, \mathit{Musc.}, \mathit{Musc.})$ with m_2. Also, notice that $B(\bot_2, \mathit{Musc.})$ has a derivation from $G(26, \mathit{Musc.}, \mathit{Musc.})$ via mapping m_1. Finally, note that $B(\bot_2, \mathit{Musc.})$ also has a second derivation, from itself and $U(\mathit{Musc.}, \mathit{Musc.})$ via mapping m_3. The graph is thus cyclic, with tuples involved in their own derivations. In general, when mappings are recursive, the provenance graph may have cycles. $\square$

The graphical model of provenance is convenient for several purposes in CDSS, including the system implementation that we shall describe in Chapter 3. However, most of the theoretical development in this dissertation will focus on another, equivalent perspective on CDSS provenance based on a powerful framework of **semiring-annotated relations**. As we will explain in Section 1.2 (and Chapter 2), **semirings** are algebraic structures that arise naturally with database provenance. This framework has the virtue of uniformly capturing as special cases many other provenance models that have been proposed in the literature (such as the *why-provenance* of [17], the *data warehousing lineage* of [35], the *Trio lineage* of [8], and the *event tables* used in probabilistic databases [50, 138]), as well as the traditional set and bag semantics. By putting all these models on equal footing, we are able to make precise comparisons of various kinds — we can compare their relative *informativeness*, for example, or study their various interactions with the *query optimization* process. In particular, it allows us to explain precisely why the ORCHESTRA provenance is strictly more informative than any of these. Provenance annotations also turn out to be useful in formulating efficient algorithms for update exchange and mapping evolution (as overviewed in Section 1.2).

Briefly, the main idea in the semiring-annotations framework is as follows. Tuples in source relations are annotated with elements from some domain of annotations K. During query processing, when tuples are joined together to produce another tuple, their annotations are combined using an abstract $\otimes$ operation. When output tuples are produced by alternative combinations (such



(a) Graphical provenance representation

$G:$

GID	NAM	CAN	
82	*Osci.*	*Dros.*	g_1
26	*Musc.*	*Musc.*	g_2

$U:$

NAM$_1$	NAM$_2$	
Osci.	*Dros.*	$m_2 \otimes g_1$
Musc.	*Musc.*	$m_2 \otimes g_2$

$B:$

BID	NAM	
44	*Peri.*	b_1
$\bot_1$	*Osci.*	$m_1 \otimes g_1$
$\bot_1$	*Dros.*	$m_3 \otimes (m_1 \otimes g_1) \otimes (m_2 \otimes g_1)$
$\bot_2$	*Musc.*	$(m_1 \otimes g_2) \oplus (m_3 \otimes (m_1 \otimes g_2) \otimes (m_2 \otimes g_2)) \oplus \cdots$

(b) Tabular provenance representation

Figure 1.3: Two views of provenance for bioinformatics example

as via union or projection operators), the alternatives are recorded using an abstract $\oplus$ operation.

Example 1.8. Refer to Figure 1.3(b), which shows the tables corresponding to Figure 1.3(a) with their semiring provenance annotations. Observe that the annotation of $G(26, \textit{Musc.}, \textit{Musc.})$ is g_2, indicating that this is a source tuple (cf. the "+" in Figure 1.3(a)). Next observe that the annotation of $U(\textit{Musc.}, \textit{Musc.})$ is $m_2 \otimes g_2$, indicating that the tuple was derived from $G(26, \textit{Musc.}, \textit{Musc.})$ using mapping m_2. Finally, observe that the annotation of $B(\bot_2, \textit{Musc.})$ is an *infinite* sum (using $\oplus$), one term for each of the (infinitely many) derivations of that tuple (cf. the cycle in the provenance graph for that tuple). The first term in the sum, $m_1 \otimes g_2$, indicates its derivation from $G(26, \textit{Musc.}, \textit{Musc.})$ using mapping m_1. The rest of the terms in the sum correspond to the derivations using the tuple itself and $U(\textit{Musc.}, \textit{Musc.})$ repeatedly with mapping m_3. □

Mapping evolution

One of the most difficult tasks for users of a CDSS is to formulate the mappings that relate the peers. In practice, mappings are not static, but are subject to frequent revisions, corrections, additions, etc. which occur as peers join and leave the system, as the schemas of peers evolve over time, and as the understanding of relationships among peers becomes clarified over time. This phenomenon has been referred to as *mapping evolution* in [136], and handling mapping evolution efficiently (i.e., updating instances in an incremental fashion as mappings change) is another key requirement in CDSS. In early phases of a collaboration's development, changes to mappings can occur at least as often as changes to data in the peers, while in later phases, changes are less frequent but potentially even more disruptive (we would prefer to avoid recomputing instances from scratch every time, e.g., a new peer joins a collaboration).

Example 1.9. Consider the bioinformatics example from Figure 1.1, and suppose that the administrator of BioSQL notices that mapping m_1 is incorrect, because the "id" field of GUS needs to be translated into the style used in BioSQL. To accomplish this, she introduces a *correspondence table*[4] $T(\text{gid}, \text{bid})$ and modifies mapping m_1 to translate ids using T:

$$(m_1') \quad G(i,n,c) \land T(i,b) \rightarrow B(b,n)$$

In order to process this update, note that some – but not all! – of the data in B will need to be updated (only those tuples which came from G), and since the mapping from B to U drops the id field, the data in U should be wholly unaffected. □

Handling mapping evolution efficiently in a CDSS is closely related to the so-called *view adaptation* problem [65], in which a materialized view must be updated after the view definition changes.

1.2 Overview of Technical Contributions

Having surveyed the main functions of ORCHESTRA and CDSS in the previous section, let us now examine some of the details of how these functionalities are accomplished. These details constitute the novel technical contributions of this dissertation. For the contributions resulting from joint work, we give paper references in each section below.

Before covering the technical contributions, we first need to recall some necessary background concepts from previous work on data exchange.

Background: data exchange, chase, and universal solutions

We saw in Example 1.3 that a basic task in data exchange and CDSS involves "patching" database instances related by schema mappings, in order to satisfy the mappings. We begin this section by explaining more about the kind of schema mappings used in CDSS, and how this fundamental task of "patching" is accomplished.

---

[4] In practice, correspondence tables are often implemented using user-defined functions; but conceptually a user-defined function may be viewed as a relation.

Tuple-generating dependencies (tgds) are a widely-used means of specifying constraints and mappings [45, 38] in data integration and data exchange.[5] A tgd is a first-order logical assertion of the form

$$\forall \bar{x}\ \forall \bar{y}\ (\varphi(\bar{x},\bar{y}) \rightarrow \exists \bar{z}\ \psi(\bar{x},\bar{z})),$$

where the left hand side (LHS) of the implication, φ, is a conjunction of relational atoms over variables $\bar{x}$ and $\bar{y}$, and the right hand side (RHS) of the implication, ψ, is a conjunction of relational atoms over variables $\bar{x}$ and $\bar{z}$. For readability, we will generally omit the universal quantifiers and simply write

$$\varphi(\bar{x},\bar{y}) \rightarrow \exists \bar{z}\ \psi(\bar{x},\bar{z}),$$

as in the mappings in Figure 1.1. The tgd expresses a constraint about the existence of tuples in the relations on the RHS, given a particular combination of tuples satisfying the constraint of the LHS.

Tgds by themselves merely describe the database instances which together are considered "acceptable." To put the mappings to work in data sharing, CDSS builds upon earlier work in data exchange where the well-known *chase* procedure can be used to *propagate* data among databases, in accordance with the schema mappings, until the mappings are satisfied. Essentially, this procedure involves repeatedly looking for *violations* of mappings, and adding new tuples to the database instances to repair the violations. If mappings contain existentially quantified variables, the new tuples will contain special placeholder *labeled null* values. The addition of new tuples may in turn introduce new violations, so the procedure must be repeated until all the mappings are satisfied. This process is guaranteed to terminate in time polynomial in the size of the input database instances, so long as the topology of the mappings is *weakly acyclic* (explained in Chapter 3).

The peer database instances computed by the chase together constitute a *solution*: a (joint) database instance satisfying the mappings which is obtained from the starting (joint) database instance by adding tuples to it.[6] In general, for a given starting joint instance and set of mappings, there are infinitely many possible solutions. The one that is computed by the chase, however, has a special property: it is a *universal solution*, essentially, a solution that can be homomorphically embedded in any other solution. Because of this, a universal solution can be used to obtain the certain answers to positive relational algebra queries (Section 1.1).

---

[5] Tgds are related to so-called *global-local-as-view* or *GLAV* mappings [49, 73], which in turn generalize the earlier *global-as-view* (*GAV*) and *local-as-view* (*LAV*) mapping formulations [100].

[6] CDSS does not draw the sharp distinction between *source* and *target* instances that is usually made in data exchange. We justify this perspective in Chapter 3.

Contribution 1: mappings as Datalog programs

(with Karvounarakis, Ives, and Tannen [60])

Although so far we have focused on mappings as logical constraints, another point of view proves useful in developing much of the theoretical machinery for CDSS, as well as its practical implementation. This is the perspective where we interpret a set of tgds as a *Datalog program*. Recall the tgds of Figure 1.1(b):

$$
\begin{aligned}
&(m_1) \qquad G(g,n,c) \rightarrow \exists b\ B(b,n) \\
&(m_2) \qquad G(g,n,c) \rightarrow U(n,c) \\
&(m_3) \qquad B(b,n_1) \wedge U(n_1,n_2) \rightarrow B(b,n_2)
\end{aligned}
$$

We can interpret these tgds as Datalog rules (possibly with *Skolem functions*), where the RHS of the implication becomes the *head* of the rule, and the LHS becomes the *body*. Thus the set of mappings above corresponds to the Datalog program:

$$
\begin{aligned}
&(r_1) \qquad B(f(n),n) \text{ :- } G(g,n,c) \\
&(r_2) \qquad U(n,c) \text{ :- } G(g,n,c) \\
&(r_3) \qquad B(b,n_2) \text{ :- } B(b,n_1), U(n_1,n_2)
\end{aligned}
$$

Note that rule r_1 contains a Skolem function f. This corresponds to the existential quantifier in m_1. Intuitively, the Skolem function introduces a unique labeled null $f(n)$ whose value depends on f and n.

Although it was not illustrated in the above example, some tgds may have multiple relational atoms in the RHS. For example, we could have written a mapping like

$$
(m_4) \qquad G(g,n,c) \rightarrow \exists b\ B(b,n), C(b,c)
$$

where C is also a relation of peer BioSQL, used to store canonical names. In this case, m_5 would be interpreted as a *pair* of Datalog rules

$$
\begin{aligned}
&(r_4) \qquad B(g(n,c),n) \text{ :- } G(g,n,c) \\
&(r_5) \qquad C(g(n,c),c) \text{ :- } G(g,n,c)
\end{aligned}
$$

where g is a Skolem function. Note that both rules use the **same** Skolem function g, with the same parameters; this ensures that the association between scientific names and canonical names in G and C is recorded and can be recovered later by a query joining G with C.

The key fact supporting this Datalog-based perspective on mappings is that, given a set of tgds, **evaluating the corresponding Datalog program also produces a universal solution**. This enables us to move freely between the two perspectives. Most of this dissertation will adopt the Datalog-based perspective.

Contribution 2: semiring-annotated relations and provenance

(with Karvounarakis and Tannen [62])

Provenance information plays a central role in many of the basic tasks of collaborative data sharing, including the specification of trust policies and algorithms for efficient propagation of updates to data and mappings. Here we outline some more of the details of the novel semiring-based model used in ORCHESTRA.

As already mentioned in Section 1.1, the main idea in the semiring-annotations framework is to decorate tuples in source relations with annotations from a domain K and combine these annotations during relational algebra query processing using abstract $\otimes$ (joint use) and $\oplus$ (alternative use) operators. Additionally, to model selection predicates, the $\otimes$ operation is used together with distinguished elements 0 and 1 from K. Finally, we observe that certain identities on relational algebra queries (needed for compatibility with standard DBMS query optimizers) hold exactly when the structure $(K, \oplus, \otimes, 0, 1)$ is a *commutative semiring*.

By plugging in the Boolean semiring $(\mathbb{B}, \vee, \wedge, \mathsf{false}, \mathsf{true})$ we obtain the usual set semantics (where tuples "in" a relation are annotated true, and tuples "not in" a relation are annotated false). By plugging in the semiring of natural numbers $(\mathbb{N}, +, \cdot, 0, 1)$, we obtain the usual bag semantics (where a relation with duplicate entries is modeled as a set relation with annotations representing tuple multiplicities). Still other semirings can be used to capture the various provenance models listed earlier. The "most informative" kind of provenance annotations in this framework is the semiring of *provenance polynomials*: this is the commutative semiring $(\mathbb{N}[X], +, \cdot, 0, 1)$ of polynomials with natural number coefficients over a set of uninterpreted variables X.

For recursive queries, such as Datalog programs corresponding to cyclic sets of tgds, a technical difficulty arises because provenance annotations may become *infinite*. Indeed, we have already seen this phenomenon illustrated in Example 1.8. The annotations in that example (cf. Figure 1.3) are not polynomials; rather they are *formal power series*.

As a practical matter, working directly with infinite formal power series is infeasible. However, as we shall see in Chapter 2, it turns out that the formal power series which arise in provenance computations have a natural, finite representation as a *system of algebraic equations*. The main idea is fairly intuitive. For each tuple t in the output of a recursive Datalog query, we associate a variable $\mathbf{x}$ and write an equation $\mathbf{x} = \ldots$ whose RHS represents the alternative ways in which t can be produced as an immediate consequence of other tuples. By doing this for all tuples in

G :

GID	NAM	CAN	
82	*Osci.*	*Dros.*	$\mathbf{g}_1 = g_1$
26	*Musc.*	*Musc.*	$\mathbf{g}_2 = g_2$

U :

NAM_1	NAM_2	
Osci.	*Dros.*	$\mathbf{u}_1 = m_2 \otimes \mathbf{g}_1$
Musc.	*Musc.*	$\mathbf{u}_2 = m_2 \otimes \mathbf{g}_2$

B :

BID	NAM	
44	*Peri.*	$\mathbf{b}_1 = b_1$
$\perp_1$	*Osci.*	$\mathbf{b}_2 = m_1 \otimes \mathbf{g}_1$
$\perp_1$	*Dros.*	$\mathbf{b}_3 = m_3 \otimes \mathbf{b}_2 \otimes \mathbf{u}_1$
$\perp_2$	*Musc.*	$\mathbf{b}_4 = (m_1 \otimes \mathbf{g}_2) \oplus (m_3 \otimes \mathbf{b}_4 \otimes \mathbf{g}_2)$

Figure 1.4: Tabular provenance representation using system of equations

the output, we obtain a system of algebraic equations whose least fixpoint is exactly the (possibly infinite) formal power series annotations.

Example 1.10. Refer to Figure 1.4, which shows the same provenance-annotated tables of Figure 1.3(b), but this time represented using a system of equations. Note that the annotation for the source tuple $G(82, \mathit{Osci.}, \mathit{Dros.})$ is now $\mathbf{g}_1$, which is trivially equal to g_1. Tuple $U(\mathit{Osci.}, \mathit{Dros.})$ was obtained as an immediate consequence of $G(82, \mathit{Osci.}, \mathit{Dros.})$ using mapping m_2, so the equation for its annotation is $\mathbf{u}_1 = m_2 \otimes \mathbf{g}_1$; by plugging in $\mathbf{g}_1 = g_1$ we obtain the annotation $m_2 \otimes g_1$, as in Figure 1.3(b). The interesting case in the example is the annotation $\mathbf{b}_4$ for $B(\perp_2, \mathit{Musc.})$, whose equation involves the tuple itself: $\mathbf{b}_4 = (m_1 \otimes \mathbf{g}_2) \oplus (m_3 \otimes \mathbf{b}_4 \otimes \mathbf{g}_2)$ (representing the two alternative ways of deriving the tuple as an immediate consequence). The solution for $\mathbf{b}_4$ is the infinite formal power series

$$\mathbf{b}_4 = (m_1 \otimes g_2) \oplus (m_3 \otimes (m_1 \otimes g_2) \otimes (m_2 \otimes g_2)) \oplus \cdots$$

shown in Figure 1.3(b). ☐

The representation of provenance annotations as a system of algebraic equations is equivalent to the graphical representation we saw earlier in this section.

Note that in the examples, variables representing mapping names themselves participate in the provenance calculations. To support this, the semiring framework includes a *scalar multiplication* construct that can be used freely in mappings and queries. Thus, for the tgd

$$(m_2) \qquad G(g, n, c) \rightarrow U(n, n)$$

from the example above, we would employ the corresponding Datalog rule

$$(r_2) \qquad m_2 \otimes U(n, n) \text{ :- } G(g, n, c)$$

which multiplies its output in the head by the uninterpreted variable m_2. This effectively "tags" the output by the mapping name.

Contribution 3: Orchestra system implementation

(with Karvounarakis, Ives, and Tannen [60])

The Orchestra prototype described in this dissertation consists of two main components: a client layer, installed at each peer, which performs functions including update translation, provenance recordkeeping, and transaction log crawling (in order to extract edit logs); and a distributed storage component, used to keep catalog-level information such as mapping definitions, as well as archives of published updates.

Orchestra uses a modified distributed hash table implementation for its distributed storage component [131, 132], hence most of the complexity of the implementation is in the client component. This is a middleware layer implemented in around 100,000 lines of Java code that sits atop each peer's local DBMS. Chapter 3 describes this layer in detail, but we overview here how one of the key implementation issues is addressed, namely the encoding and storage of provenance information.

A basic philosophy in the Orchestra implementation is to push as much work as possible down into the underlying DBMS, with the Java layer acting mainly as a coordinator for its activities. This allows the system to benefit from the existing DBMS facilities for storage and scalable query processing over large data sets. At the same time, Orchestra attempts to remain relatively agnostic as to the particular DBMS implementation underneath — in particular, it avoids as much as possible the use of user-defined functions, which are generally not portable across implementations.

In keeping with this philosophy, Orchestra uses an relational encoding schema for provenance that allows the information to be maintained *using only SQL queries* (without user-defined functions). This is done by adopting the graph-based perspective on provenance (cf. Figure 1.3(a)), and encoding the graph using *provenance tables*. For each mapping m_i, we use a separate provenance table P_i recording the edges through nodes labeled m_i.

For example, to store the graph in Figure 1.3(a), Orchestra uses three edge tables P_1, P_2, and P_3. P_3 contains one tuple for each occurrence of m_3 in the graph:

P_3 :

B1.BID	B1.NAM	U.NAM$_1$	U.NAM$_2$	B2.BID	B2.NAM
$\bot_1$	*Osci.*	*Osci.*	*Dros.*	$\bot_1$	*Dros.*
$\bot_2$	*Musc.*	*Musc.*	*Musc.*	$\bot_2$	*Musc.*

Note that the table above contains a number of redundancies, due to the fact that the mapping forces equivalences among some of the columns. In particular, the values in columns B1.BID and B2.BID are always the same, as are those in B1.NAM and U.NAM$_1$, and those in U.NAM$_2$ and B2.NAM. Orchestra actually stores a compressed version of the edge table with such redundancies eliminated, producing (in the case of P_3 above):

B1.BID	B1.NAM	U.NAM$_2$
$\bot_1$	*Osci.*	*Dros.*
$\bot_2$	*Musc.*	*Musc.*

Additional information such as key constraints can be used to compress these tables even further.

A major benefit of this encoding scheme is that the mapping tables can be constructed and maintained using ordinary queries which are derived from the mapping definitions. Continuing with the example, recall that mapping m_3 corresponds to the Datalog rule:

$$(r_3) \qquad B(b, n_2) \text{ :- } B(b, n_1), U(n_1, n_2)$$

From this, we derive two rules:

$$\begin{aligned} &(r_3') \qquad P_3(b, n_1, n_2) \text{ :- } B(b, n_1), U(n_1, n_2) \\ &(r_3'') \qquad B(b, n_2) \text{ :- } P_3(b, n_1, n_2) \end{aligned}$$

The first rule joins tuples from B and U, putting the result in P_3 (the compressed version); the second takes tuples from P_3 and projects the relevant columns, putting the result in B.

The translated rules for all the mappings together form a Datalog program that can be executed to populate the peer instances and their mapping tables. Since mainstream commercial database systems support only limited forms of recursive queries, the Orchestra client layer also includes a Datalog engine, which uses the ordinary DBMS query and update facilities in combination with Java control logic for iteration to fixpoint.

Contribution 4: algorithms for incremental update exchange

(with Karvounarakis, Ives, and Tannen [60])

A major technical challenge in CDSS is how to implement update exchange *incrementally*. Intuitively, the CDSS operating mode resembles deferred view maintenance across a set of views; and indeed, as we shall see, classical techniques from view maintenance can be applied usefully in update exchange. However, the presence of provenance information also provides a kind of "index" that can be exploited to improve on the classical techniques.

Classical view maintenance techniques are essentially based on *query reformulation*. The premier examples of this approach in view maintenance are the count and DReD algorithms of [66], each of which is based the so-called *delta rules* for a Datalog program. These are another Datalog program, obtained from the original, that can be used used to compute sets of insertions or deletions to a materialized view, given sets of insertions or deletions to base relations. We shall discuss delta rules further in the next section (and later in Chapters 3 and 5); for now, we illustrate with a simple example that gives the flavor of the technique.

Example 1.11. Let Q be the conjunctive query

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

which returns paths of length 2 through R and S. Suppose that R and S are modified by inserting some tuples, stored in relations R^+ and S^+. The *delta rules query* for Q is the following union of conjunctive queries, which computes the set Q^+ of corresponding insertions in Q:

$$\begin{array}{lcl} Q^+(x,y) & \text{:-} & R^+(x,z), S(z,y) \\ Q^+(x,y) & \text{:-} & R(x,z), S^+(z,y) \\ Q^+(x,y) & \text{:-} & R^+(x,z), S^+(z,y) \end{array}$$

These rules compute all combinations of joins of tuples in R and S with newly-inserted tuples from R^+ and S^+, and also joins of newly-inserted tuples from R^+ and S^+. □

As we show in Chapter 3, count and DReD can be adapted to perform CDSS update exchange as well. (*A priori*, this fact is not obvious, because in CDSS we are not just maintaining materialized views, but also their accompanying provenance information.)

Classical techniques such as DReD do not perform very well, however, in one important case: incremental propagation of tuple deletions with recursive mappings. For DReD, this is because the approach involves computing a loose over-estimate of the corresponding deletions in materialized views (using delta rules), then attempting to re-derive those tuples from other existing tuples (hence its name: DReD stands for "delete and re-derive"). This over-estimate can be large, leading to poor performance as many tuples may need to be re-derived.

To overcome this limitation, we observe that provenance annotations can be exploited as the basis of a much more efficient algorithm for propagating tuple deletions in the presence of recursive mappings. Intuitively, the idea is quite simple: the provenance annotations explain how output tuples were derived from source tuples; hence the effects of a source tuple deletion can be computed by examining the provenance annotations of output tuples. Roughly speaking, given a deletion of source tuple with provenance token p, we replace all occurrences of p by 0 in provenance expressions of derived tuples, and then simplify the expressions. Derived tuples whose provenance expressions simplify to 0 are then discarded.

Example 1.12. Suppose $R(a,b)$ is a derived output tuple with provenance $u \otimes v$, and we delete the source tuple corresponding to u. To propagate the deletion, we evaluate $0 \otimes v = 0$ and find that $R(a,b)$ should be deleted as well. On the other hand, suppose $R(a,c)$ is a derived tuple with provenance $(u \otimes v) \oplus r$. After the deletion, its provenance expression becomes $(0 \otimes v) \oplus r = r$, hence we see that the tuple should not be deleted because it has another derivation. $\square$

For recursive views, the procedure also involves an extra step needed to detect and delete tuples whose provenance expressions do not seem to simplify to 0 only because of cycles in the provenance graph. This extra step carries some cost; still, the provenance-based algorithm turns out to be the method of choice for propagating tuple deletions in CDSS, as validated by experimental results in Chapter 3.

Contribution 5: query containment and equivalence

Semiring annotated relations are compatible with the standard optimizations used in DBMS query optimizers, as we show in Chapter 2. However, for more advanced optimizations based on eliminating *redundant joins* from conjunctive queries [22] (or *redundant conjunctive queries* from unions of conjunctive queries [123]), this is not necessarily the case. Indeed, this phenomenon was already observed for the case of bag semantics, where eliminating a "redundant" self-join from a query actually changes the query's meaning [24]. As a simple example, consider the following two conjunctive queries:

$$Q_1(x,y) \text{ :- } R(x,y), R(x,z) \qquad\qquad Q_2(x,y) \text{ :- } R(x,y)$$

Under set semantics, these two queries are equivalent; hence Q_1 can be optimized by eliminating the redundant join predicate, producing Q_2. However, under bag semantics, the queries are not equivalent: the "redundant" join predicate in Q_1 has the effect of increasing the multiplicities of output tuples.

Motivated by these observations, we make a thorough study of the interplay between provenance annotations and the fundamental problems of checking query *containment* and *equivalence*, which underly advanced query optimizations (Chapter 4). We study these problems for conjunctive queries and unions of conjunctive queries, for five different kinds of provenance information that can be captured using semiring annotations (including the provenance polynomials used in CDSS), leading to four different hierarchies among these semantics. We show that, for each form of provenance we consider, all four problems turn out to be decidable, and we identify the complexity in each case (see Figure 1.5). Several of these results are surprising because, for example, provenance polynomials are closely related to bag semantics, and containment of unions of conjunctive queries under bag semantics is known to be undecidable [81]. We also identify a fundamental tradeoff between informativeness of provenance information and query optimization: the richer the provenance information, the fewer optimizations possible. However, even for the

		$\mathbb{B}$	PosBoolX	Lin(X)	Why(X)	Trio(X)	$\mathbb{B}[X]$	$\mathbb{N}[X]$	$\mathbb{N}$
CQs	cont	NP-C	NP-C	NP-C	NP-C	NP-C	NP-C	NP-C	? (Π_2^p-hard)
	equiv	NP-C	NP-C	NP-C	GI-C	GI-C	GI-C	GI-C	GI-C
UCQs	cont	NP-C	NP-C	NP-C	NP-C	in PSPACE	NP-C	in PSPACE	undec
	equiv	NP-C	NP-C	NP-C	NP-C	GI-C	NP-C	GI-C	GI-C

In the table, non-shaded boxes indicate contributions of this dissertation. NP-C is short for NP-complete. GI-C is short for GI-complete (i.e., complete for the class of problems polynomial time reducible to graph isomorphism).

Figure 1.5: Complexity of containment and equivalence with provenance annotations

most informative model possible in the framework, provenance polynomials, we show that equivalence of unions of conjunctive queries with provenance polynomial annotations holds exactly when bag equivalence holds, hence any query reformulation which is sound for bag semantics (and therefore might be employed by a commercial DBMS optimizer) is sound with respect to all forms of semiring annotations. As a corollary, we also obtain a new proof of the decidability of bag equivalence of unions of conjunctive queries.

The theoretical results on the decidability of query equivalence with provenance annotations play a crucial role in the subsequent development of techniques for handling CDSS evolution in Chapter 5.

Contribution 6: ring-annotated relations for updates

(with Ives and Tannen [59])

We have already seen that a major focus of CDSS is on the efficient handling of the dynamic aspects of data sharing, in particular, handling updates to data (*update exchange*) and to mappings (*mapping evolution*). And we have already mentioned that for update exchange, ORCHESTRA adapts techniques based on the so-called *delta rules* technique [66], and improves upon them by exploiting the presence of provenance annotations.

Mapping evolution seems on the face of it a very different problem than update exchange. Somewhat surprisingly, then, it turns out that both problems (along with their classical cousins, view maintenance and view adaptation) can be attacked uniformly by the same approach, where we view each as a special case of a more general problem of *optimizing queries using materialized views* [101, 23]. This is the problem which asks, given a query Q and a set $\mathcal{V}$ of materialized views, to find a *reformulation* of Q using the materialized views which can be executed more efficiently.

Example 1.13. To illustrate update exchange / view maintenance, consider a source relation R

$$\begin{array}{rcl} R'(x,y) & \text{:-} & R(x,y) \\ R'(x,y) & \text{:-} & R^{\Delta}(x,y) \\ \\ V(x,y) & \text{:-} & R(x,y), R(y,z) \end{array}$$

(a) Materialized view definitions

$$\begin{array}{rcl} V'(x,y) & \text{:-} & R'(x,y), R'(y,z) \\ \\ V^{\Delta}(x,y) & \text{:-} & V'(x,y) \\ -V^{\Delta}(x,y) & \text{:-} & V(x,y) \end{array}$$

(b) New view definition, and difference with old

$$\begin{array}{rcl} V^{\Delta}(x,y) & \text{:-} & R^{\Delta}(x,z), R(z,y) \\ V^{\Delta}(x,y) & \text{:-} & R(x,z), R^{\Delta}(z,y) \\ V^{\Delta}(x,y) & \text{:-} & R^{\Delta}(x,z), R^{\Delta}(z,y) \end{array}$$

(c) A delta-rules style reformulation

Figure 1.6: Update propagation and query reformulation

$$\begin{array}{rcl} Q(x,y) & \text{:-} & R(x,u), R(u,v), R(v,y) \\ Q(x,y) & \text{:-} & R(x,z), R(z,y) \end{array}$$

(a) Old mappings

$$\begin{array}{rcl} Q'(x,y) & \text{:-} & R(x,u), R(u,v), R(v,y) \\ Q'(x,y) & \text{:-} & R(x,y) \end{array}$$

(b) New mappings

$$\begin{array}{rcl} Q^{\Delta}(x,y) & \text{:-} & Q(x,y) \\ -Q^{\Delta}(x,y) & \text{:-} & Q'(x,y) \end{array}$$

(c) Their difference

$$\begin{array}{rcl} Q^{\Delta}(x,y) & \text{:-} & R(x,y) \\ -Q^{\Delta}(x,y) & \text{:-} & R(x,z), R(z,y) \end{array}$$

(d) A reformulated plan

Figure 1.7: Mapping evolution and query reformulation

and a collection R^{Δ} of updates to R (consisting of tuple insertions and deletions), which when applied to R yields the relation R'. R' can be thought of as a *materialized view* over R and R^{Δ} whose definition is given in Figure 1.6(a). Now, consider the materialized view V over R whose definition is also given in Figure 1.6(a). To reflect the updates made to R, we need to compute the new version V' of V, shown in Figure 1.6(b). Moreover, to perform this computation *incrementally*, we would like to compute just the *difference* V^{Δ} between V' and V, also defined in Figure 1.6(b), where the leading "$-$" indicates a difference operation. This could then be applied to V to produce V'. Note that the definition of V^{Δ} involves first computing V', then subtracting V from it, a rather inefficient approach! A better plan is the *delta rules reformulation* of V^{Δ} shown in Figure 1.6(b), which we can imagine as having been produced via a general process of reformulating V^{Δ} using the materialized views V and R'. ☐

Example 1.14. To illustrate mapping evolution / view adaptation, Figure 1.7(a) shows a materialized view definition Q which is modified (by changing its second rule) to produce the view definition Q' shown in Figure 1.7(b). To compute Q' incrementally, we might prefer to compute the difference Q^{Δ} between Q and Q', shown in Figure 1.7(c). As with the previous example, the definition does not produce a very efficient plan! Instead, we can reformulate Q^{Δ} using materialized view Q, producing the plan shown in Figure 1.7(d). ☐

These two examples suggest certain requirements that need to be addressed. First, they both

R :

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

R^{Δ} :

$a\ b$	-2
$b\ c$	-1
$e\ f$	1

R' :

$a\ d$	1
$d\ c$	1
$d\ e$	3
$e\ f$	1

V :

$a\ c$	3
$a\ e$	3

V^{Δ} :

$a\ c$	-2
$d\ f$	3

V' :

$a\ c$	1
$a\ e$	3
$d\ f$	3

(a) Delta rules example with $\mathbb{Z}$-relations

R :

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

Q :

$a\ d$	$p \cdot r \cdot s$
$a\ c$	$p \cdot r$
$b\ d$	$r \cdot s$

Q' :

$a\ d$	$p \cdot r \cdot s$
$a\ b$	p
$b\ c$	r
$c\ d$	s

Q^{Δ} :

$a\ c$	$-p \cdot r$
$b\ d$	$-r \cdot s$
$a\ b$	p
$b\ c$	r
$c\ d$	s

(b) Mapping evolution example with $\mathbb{Z}[X]$-relations

Figure 1.8: Representing and computing updates with ring-annotated relations

make essential use of the *difference* operator, not present in the framework of semiring-annotated relations, and not usually considered in previous work on optimizing queries using materialized views. Second, both involve *updates* (tuple insertions and deletions) being represented and manipulated as ordinary relations.

To handle these requirements, we propose the use of *ring-annotated relations* to represent data and updates uniformly, and with the *difference* operator defined in the natural way using inverses in the ring. We study two concrete instantiations of this framework. The first is $\mathbb{Z}$*-relations*, where tuples are annotated with (positive or negative) integers. This is the ring-annotated analog of bag semantics, with positive multiplicities representing insertions and negative ones representing deletions. The second is $\mathbb{Z}[X]$*-relations*, the ring-annotated analog of the provenance polynomials, where we allow positive or negative integer coefficients.

Example 1.15. Figure 1.8(a) shows an example of $\mathbb{Z}$-annotated relations corresponding to Example 1.13. Note that R^{Δ} has tuples with positive and negative multiplicities, e.g., the annotation of (a, b) is -2 indicating that a deletion of two copies of that tuple. V^{Δ} corresponds to the result of evaluating the query from Figure 1.6(c) over R, R', and V under the $\mathbb{Z}$-semantics, and note that V^{Δ} also has tuples with positive and negative multiplicities. By applying V^{Δ} to V via a union operation (which sums tuple multiplicities), we obtain V'. ☐

Example 1.16. Figure 1.8(b) shows an example of $\mathbb{Z}[X]$-annotated relations corresponding to Example 1.14. Note that Q^{Δ} shows the result of evaluating the plan in Figure 1.7(d) over the $\mathbb{Z}[X]$-relations Q and R, and by applying Q^{Δ} to Q (by union, which again sums tuple annotations), we obtain the table Q' shown in the figure. ☐

The extension to ring-annotated relations and relational queries using difference gives us a rich framework for representing and computing updates, but we still need a general procedure for optimizing queries using materialized views in this new context. Herein lies the big surprise:

we are able to give a *sound and complete* procedure for enumerating query reformulations under the $\mathbb{Z}$ and $\mathbb{Z}[X]$ annotated semantics, for relational algebra queries and views. This is in sharp contrast to traditional set and bag semantics, where it can be shown that no such procedure exists. The enumeration procedure is based on a simple term rewrite system, the details of which are described in Chapter 5. We also show that equivalence of relational algebra queries under these new semantics is decidable (in PSPACE). This also stands in contrast to the situation for set and bag semantics, where allowing the difference operator in queries leads to undecidability of equivalence. Thus, the ring-annotated relations framework gives us benefits both in expressiveness (for representing and computing updates), but is also surprisingly tractable.

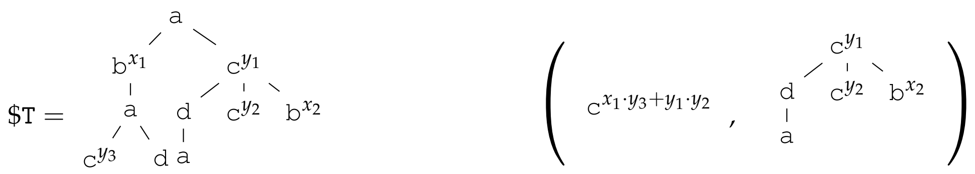
Contribution 7: provenance for XML and nested relations

(with Foster and Tannen [48])

The ORCHESTRA prototype in its current incarnation handles only relational data. However, other kinds of data, such as XML and nested relational data, are also used commonly in the sciences and other collaborative domains. Thus an important longer-term goal of CDSS involves extending the paradigm to work with these richer data models as well. As a key first step towards this goal, we develop an extension of the theoretical framework of semiring annotations to unordered XML (UXML) and nested relational data. For UXML, this involves decorating XML elements in a tree with semiring annotations, and extending the semantics of a positive fragment of XML's premier query language, XQuery, to operate on semiring-annotated, unordered trees. For nested relations, we decorate both tuples and elements of nested collections with semiring annotations, and extend the semantics of the positive nested relational calculus to work with these annotated nested relations.

Example 1.17. To illustrate the flavor of this model for XML, Figure 1.9(a) shows an example of an XML tree, some of whose elements are annotated with provenance polynomials. Now consider the XPath query $Q \overset{\text{def}}{=}$ `$T//c` which picks out the set of subtrees of elements of `$T` whose label is `c`. The result of evaluating Q on the source tree in Figure 1.9(a) is shown in Figure 1.9(b). Note that the annotation of the root element of the first tree in the output is $x_1 \cdot y_3 + y_1 \cdot y_2$. This indicates that the tree was produced in two different ways: one by traversing a path through elements in the input tree with annotations x_1 and y_3, and the other via a path through elements annotated y_1 and y_2. ☐

We show that this semantics is conservative with respect to the semantics of positive relational algebra on semiring-annotated relations, and that key properties underlying the relational version of the framework continue to hold for annotated XML and nested relations. Hence, we give strong



(a) A provenance-annotated XML tree

(b) XPath query result

Figure 1.9: Semiring-annotated XML example.

evidence that provenance annotations for XML and nested relations fundamentally "make sense," paving the way towards a future extension of CDSS implementations to these data models. We also propose novel applications of independent interest, including *incomplete XML* and *XML with access control policies*, which are enabled by the framework.

1.3 Roadmap

The remainder of the thesis is structured as follows:

- Chapter 2 introduces semiring-annotated relations and develops the formal foundations of provenance needed for CDSS. We also discuss connections with bag (multiset) semantics, incomplete and probabilistic databases, and other models of provenance such as lineage [34] and why-provenance [17] which can be captured using semiring annotations.
- Chapter 3 describes in more detail the formal model and prototype implementation of the ORCHESTRA CDSS, including the implementation of provenance storage, incremental update exchange, and trust policies.
- Chapter 4 examines the fundamental issues of query containment and equivalence for provenance-annotated relations. These problems turn out to be highly sensitive to the presence of provenance information. We give positive decidability results for containment/equivalence of conjunctive queries and unions of conjunctive queries for several forms of provenance information. We also highlight connections with the same problems for set and bag semantics.
- Chapter 5 explains how incremental view maintenance, view adaptation, and mapping evolution can be seen as special cases of a more general problem of answering queries using materialized views. We argue for signed ring-annotated relations as a natural representation of both data and updates to data, and for the incorporation of the difference operator to enable more rewritings. We show that, surprisingly, equivalence of relational algebra queries is

decidable for two instantiations of this framework relevant to CDSS and traditional database systems. We also present a sound and complete procedure for reformulating queries with materialized views under these semantics.

- Chapter 6 extends the semiring provenance model of Chapter 2 to work with unordered XML and nested relational data. We show that the key properties of Datalog queries on semiring-annotated relations carry over to (positive) XQuery queries on semiring-annotated unordered XML, and to nested relational calculus queries on semiring-annotated complex values. This paves the way towards a future extension of CDSS to XML and nested relational data.
- We discuss related work in Chapter 7, and we conclude in Chapter 8.

Chapter 2

Provenance Semirings

A major component of the vision for CDSS [82] is the recording of *data provenance* as data and updates are transformed according to schema mappings. We begin, therefore, by developing a suitable formal model of data provenance for CDSS. The particular provenance model we use emerges as a by-product of the development of a more general theory of queries on *annotated relations*.

Several forms of annotated relations have appeared in various contexts in the literature. Query answering in these settings involves generalizing the relational algebra ($\mathcal{RA}$) to perform corresponding operations on the annotations. The seminal paper in incomplete databases [79], for example, generalized $\mathcal{RA}$ to *conditional tables* (*c*-tables), where relations are annotated with Boolean formulas. In probabilistic databases, [50] and [138] generalized $\mathcal{RA}$ to event tables, also a form of annotated relations. In data warehousing, [34] and [35] 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. We shall examples of all of these kinds of annotated relations in Section 2.1.

We observe that in all of these cases, the calculations with annotations are strikingly similar. 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 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 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 such as CDSS 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, just as for rings, the symbolic representation is that of polynomials. We therefore propose to use polynomials to capture provenance. Next we look to extend our approach to recursive Datalog queries (needed in CDSS when mappings have cycles). To achieve this we combine semirings with fixed point theory.

The outline of this chapter is as follows:

- We introduce K**-relations**, in which tuples are annotated (tagged) with elements from K. We define a generalized positive algebra on K-relations and argue that K must be a **commutative semiring** (Section 2.2).
- For recording data provenance, we propose polynomials with integer coefficients, and we show that positive algebra semantics for any commutative semirings **factors** through the provenance semantics (Section 2.3).
- We discuss other forms of provenance information, previously proposed in the literature, which can be captured using semirings, and show that they can be organized in a **provenance hierarchy** (Section 2.4).
- We extend these results to **Datalog** queries by considering semirings with **fixed points** (Section 2.5).
- For the (possibly infinite) provenance in Datalog query answers we propose semirings of **formal power series** that are shown to be generated by finite **algebraic systems** of fixed point equations (Section 2.6).
- We give **algorithms** for deciding the finiteness of these formal power series, for computing them when finite, and for computing the coefficient of an arbitrary monomial otherwise (Section 2.7).
- We show how to specialize our algorithms for computing full Datalog answers when K is a **finite distributive lattice**, in particular for **incomplete** and **probabilistic** databases (Section 2.8).

2.1 Queries on Annotated Relations

We motivate our study by considering three important examples of query answering on annotated 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* [124, 63], in which tuples annotated with a '?' may or may not be in

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)

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

(c)

Figure 2.1: 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 2.2: Result of Imielinski-Lipski computation

the instance, independently from each other. An example is given in Figure 2.1(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

$$Q \overset{\text{def}}{=} \pi_{\mathrm{AC}}\big(\pi_{\mathrm{AB}}R \bowtie \pi_{\mathrm{BC}}R \cup \pi_{\mathrm{AC}}R \bowtie \pi_{\mathrm{BC}}R\big)$$

the query result is the set of possible worlds shown in Figure 2.1(c). The second table in the query result, for example, is obtained by evaluating Q over R with optional tuples (d, b, e) and (f, g, e) omitted.

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 [79] 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 2.1(b). In contrast to weaker representation systems, *c*-tables are expressive enough to be *closed* under $\mathcal{RA}$ queries, and the main result of [79] 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 2.2(a), which can be simplified to the *c*-table shown in Figure 2.2(b); this *c*-table represents exactly the set of possible worlds shown in Figure 2.1(c).

Another kind of table with annotations is a *multiset* or *bag*. In this case, the annotations are

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 2.3: Bag semantics example

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 2.4: Probabilistic example

natural numbers which represent the multiplicity of the tuple in 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 R shown in Figure 2.3(a). Then Q applied to R, where Q is the same query from before, produces the multiset shown in Figure 2.3(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 [50] and Zimányi [138] 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 2.4(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-Rölleke-Zimányi query answering algorithm produces the event table shown in Figure 2.4(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.

2.2 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 2.1 will turn out to be particular cases.

---

[1] The Fuhr-Röllecke-Zimányi algorithm is a general-purpose *intensional* algorithm. Dalvi and Suciu [36] 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. For instance, our example does not admit a safe plan.

We use here the named perspective [2] of the relational model in which tuples are functions $t : U \rightarrow \mathbb{D}$ 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 $U\text{-}\mathsf{Tup}$. (Usual) relations over U are finite subsets of $U\text{-}\mathsf{Tup}$.

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 $U\text{-}\mathsf{Tup}$ 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 2.1. Let K be a set of annotations (or tags) containing a distinguished element 0. A K**-relation** over a finite set of attributes U is a function $R : U\text{-}\mathsf{Tup} \rightarrow K$ such that its **support** defined by $\mathsf{supp}(R) \overset{\text{def}}{=} \{t \mid R(t) \neq 0\}$ is finite. A K**-instance** I is a mapping which associates each predicate symbol R with a K-relation R^I.

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 "$\oplus$." 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 "$\otimes$."

Definition 2.2. Let $(K, \oplus, \otimes, 0, 1)$ be an algebraic structure with two binary operations and two distinguished elements. The **denotation** $[\![Q]\!]^I$ of a **positive relational algebra query** Q on a K-instance I, a K-relation of finite support, is defined inductively as follows:

empty relation For any set of attributes U, there is $\emptyset : U\text{-}\mathsf{Tup} \rightarrow K$ such that $[\![\emptyset]\!]^I(t) \overset{\text{def}}{=} 0$.

predicate If R is a predicate symbol with attributes U then $[\![R]\!]^I : U\text{-}\mathsf{Tup} \rightarrow K$ is simply R^I.

union If $[\![Q_1]\!]^I, [\![Q_2]\!]^I : U\text{-}\mathsf{Tup} \rightarrow K$ then $[\![Q_1 \cup Q_2]\!]^I : U\text{-}\mathsf{Tup} \rightarrow K$ is defined by

$$[\![Q_1 \cup Q_2]\!]^I(t) \overset{\text{def}}{=} [\![Q_1]\!]^I(t) \oplus [\![Q_2]\!]^I(t)$$

Notice that if $[\![Q_1]\!]^I$ and $[\![Q_1]\!]^I$ have finite support then so has $[\![Q_1 \cup Q_2]\!]^I$, provided we have $0 \oplus 0 = 0$.

projection If $[\![Q_1]\!]^I : U\text{-}\mathsf{Tup} \to K$ and $V \subseteq U$ then $[\![\pi_V Q_1]\!]^I : V\text{-}\mathsf{Tup} \to K$ is defined by

$$[\![\pi_V Q_1]\!]^I(t) \overset{\mathtt{def}}{=} \bigoplus_{\substack{t=t' \text{ on } V \text{ and} \\ [\![Q_1]\!]^I(t') \neq 0}} [\![Q_1]\!]^I(t')$$

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, assuming $[\![Q_1]\!]^I$ has finite support; and $[\![\pi_V Q_1]\!]^I$ also has finite support, again provided $0 \oplus 0 = 0$.

selection If $[\![Q_1]\!]^I : U\text{-}\mathsf{Tup} \to K$ and the selection predicate $\mathbf{P}$ maps each U-tuple to either 0 or 1 then $[\![\sigma_{\mathbf{P}} Q_1]\!]^I : U\text{-}\mathsf{Tup} \to K$ is defined by

$$[\![\sigma_{\mathbf{P}} Q_1]\!]^I(t) \overset{\mathtt{def}}{=} [\![Q_1]\!]^I(t) \otimes \mathbf{P}(t)$$

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. Note that if $[\![Q_1]\!]^I$ has finite support, then so does $[\![\sigma_{\mathbf{P}} Q_1]\!]^I$, provided that $0 \otimes 0 = 0$ and $0 \otimes 1 = 0$.

natural join If $[\![Q_i]\!]^I : U_i\text{-}\mathsf{Tup} \to K$ for $i = 1, 2$ then $[\![Q_1 \bowtie Q_2]\!]^I$ is the K-relation over $U_1 \cup U_2$ defined by

$$[\![Q_1 \bowtie Q_2]\!]^I(t) \overset{\mathtt{def}}{=} [\![Q_1]\!]^I(t_1) \otimes [\![Q_2]\!]^I(t_2)$$

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). Notice that if $[\![Q_1]\!]^I$ and $[\![Q_2]\!]^I$ have finite support, then so does $[\![Q_1 \bowtie Q_2]\!]^I$, provided $a \otimes 0 = 0 \otimes a = 0$, for all $a \in K$.

renaming If $[\![Q_1]\!]^I : U\text{-}\mathsf{Tup} \to K$ and $\beta : U \to U'$ is a bijection then $[\![\rho_\beta Q_1]\!]^I$ is a K-relation over U' defined by

$$[\![\rho_\beta Q_1]\!]^I(t) \overset{\mathtt{def}}{=} [\![Q_1]\!]^I(t \circ \beta)$$

and notice that if $[\![Q_1]\!]^I$ has finite support then so does $[\![\rho_\beta Q_1]\!]^I$.

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

---

[2] in order to permit simplifications; it turns out that this is the same as transforming using the axioms of *distributive lattices* [33]

Definition 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*.

Definition 2.3. An algebraic structure $(K, \oplus, \otimes, 0, 1)$ is called a *commutative semiring* if $(K, \oplus, 0)$ and $(K, \otimes, 1)$ are commutative monoids, $\otimes$ is distributive over $\oplus$, and $\forall a \in K$, $0 \otimes a = a \otimes 0 = 0$.

Further evidence for requiring K to form such a semiring is given by

Proposition 2.4. *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);*
- $\sigma_{\mathsf{false}}(Q_1) = \emptyset$ *and* $\sigma_{\mathsf{true}}(Q_1) = Q_1$.

hold for the positive algebra on K-relations if and only if $(K, \oplus, \otimes, 0, 1)$ 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 $h : K_1 \to K_2$ which maps 0_{K_1} to 0_{K_2} can be used to transform K_1-relations to K_2-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 transformations from K_1-relations to K_2-relations and from K_1-instances to K_2-instances also by h. The $\mathcal{RA}^+$ operations we have defined work nicely with semiring structures:

Proposition 2.5. *Let $h : K_1 \to K_2$ and assume that K_1, K_2 are commutative semirings. The transformation given by h from K_1-relations to K_2-relations commutes with any $\mathcal{RA}^+$ query Q and K-instance I, $[\![Q]\!]^{h(I)} = h([\![Q]\!]^I)$, if and only if h is a semiring homomorphism.*

Example 2.6. If B is a set of Boolean variables, any Boolean valuation $\nu : B \to \mathbb{B}$ extends uniquely to a semiring homomorphism $\mathsf{Eval}_\nu : \mathsf{PosBool}(B) \to \mathbb{B}$ such that $\mathsf{Eval}_\nu(b) = \nu(b)$. The *possible worlds* of a Boolean c-table T (a $\mathsf{PosBool}(B)$-annotated relation) can be formally defined as follows:

$$\mathrm{Rep}(T) \overset{\mathrm{def}}{=} \{\mathsf{Eval}_\nu(T) \mid \nu \text{ is a Boolean valuation } \nu : B \to \mathbb{B}\}$$

Using Proposition 2.5, we see that for any positive relational query $Q \in \mathcal{RA}^+$, Boolean c-table T, and Boolean valuation ν, we have

$$[\![Q]\!]^{\mathsf{Eval}_\nu(T)} = \mathsf{Eval}_\nu([\![Q]\!]^T)$$

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 2.5: Lineage and provenance polynomials

The correctness of the Imielinski-Lipski algebra on c-tables with respect to the possible worlds semantics follows as a corollary. □

Example 2.7. Commercial relational database systems typically implement bag semantics, but include a `DISTINCT` operator which may be used to eliminate duplicates from a query result. This operator can be viewed as the semiring homomorphism $\delta : \mathbb{N} \rightarrow \mathbb{B}$ which maps 0 to false and everything else to true. Using Proposition 2.5, we see that for any bag instance I and positive relational query $Q \in \mathcal{RA}^+$, we have

$$[\![Q]\!]^{\delta(I)} = \delta([\![Q]\!]^I)$$

Hence eliminating duplicates as a last step yields (not surprisingly) the same answer that would have been obtained by first eliminating duplicates from source tables, then evaluating the query under set semantics. □

2.3 Polynomials for Provenance

Lineage was defined in [34, 35] as a way 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.

Computing the lineage for queries in $\mathcal{RA}^+$ turns out to be exactly Definition 2.2 for the semiring [13] $(\mathsf{Lin}(X), +, \cdot, \perp, \emptyset)$ where X consists of the ids of the tuples in the input instance, where $\mathsf{Lin}(X) = \mathcal{P}(X) \cup \perp$, $\perp + S = S + \perp = S$, $\perp \cdot S = S \cdot \perp = \perp$, and $S + T = S \cdot T = S \cup T$ if $S, T \neq \perp$. For example, we consider the same tuples as in relation R used in the examples of Section 2.1 but now we tag them with their own ids p,r,s, as shown in Figure 2.5(a). The resulting R can be seen as a $\mathsf{Lin}(p,r,s)$-relation by replacing p with $\{p\}$, etc. Applying the query Q from Section 2.1 to R we obtain according to Definition 2.2 the $\mathsf{Lin}(p,r,s)$-relation shown in Figure 2.5(b).

For example, in the query result in Figure 2.5(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. This example illustrates the limitations of lineage (also recognized in [25]). It seems that we need to know not just *which* input tuples contribute but also *how* they contribute.

On the other hand, by using the different operations of the semiring, Definition 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, as in ring theory, these are the *polynomials*.

Definition 2.8. Let X be a set of indeterminates, which can be thought of as tuple identifiers. The **positive algebra provenance semiring** for X is the semiring of polynomials with variables (a.k.a. indeterminates) from X and coefficients from $\mathbb{N}$, with the operations defined as usual[3]: $(\mathbb{N}[X], +, \cdot, 0, 1)$. We refer to the elements of $\mathbb{N}[X]$ has **provenance polynomials**.

Example 2.9. Start again from the relation R in Figure 2.5(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 2.1 and doing the calculations in the provenance semiring we obtain the $\mathbb{N}[p, r, s]$-relation shown in Figure 2.5(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 just the input tuple s, but *twice*; the third uses input tuples r and s. We also see that the provenance of (d, e) is different from that of (f, e) 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 2.10. *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 $\mathsf{Eval}_v : \mathbb{N}[X] \to K$ such that for the one-variable monomials we have $\mathsf{Eval}_v(x) = v(x)$.*

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

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

---

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

Putting together Propositions 2.5 and 2.10 we obtain Theorem 2.11 below, a conceptually important fact that says, informally, that the semantics of $\mathcal{RA}^+$ on K-instances for any semiring K **factors** through the semantics of the same in provenance semirings.

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

We associate to I an "abstractly tagged" version, denoted $\mathsf{ab}(I)$, which is an $X \cup \{0\}$-relation. $\mathsf{ab}(I)$ is such that $\mathsf{supp}(\mathsf{ab}(I)) = \mathsf{supp}(I)$ and the tuples in $\mathsf{supp}(\mathsf{ab}(I))$ are tagged by their own tuple id. For example, in Figure 2.9(d) we show an abstractly-tagged version of the relation in Figure 2.9(b). Note that as an $X \cup \{0\}$-relation, $\mathsf{ab}(I)$ is a particular kind of $\mathbb{N}[X]$-relation.

Theorem 2.11 (factoring)**.** *For any $\mathcal{RA}^+$ query Q and K-instance I we have $[\![Q]\!]^I = \mathsf{Eval}_v \circ [\![Q]\!]^{\mathsf{ab}(I)}$.*

To illustrate an instance of this theorem, consider the provenance polynomial $2r^2 + rs$ of the tuple (d, e) in Figure 2.5(c). Evaluating it in $\mathbb{N}$ for $p = 2, r = 5, s = 1$ we get 55 which is indeed the multiplicity of (d, e) in Figure 2.3(a).

The factoring theorem thus allows pre-computing of provenance polynomial annotations, then evaluation of them under a particular semantics in a specific K for various applications. As we shall see in Chapter 3, this is precisely the strategy employed by ORCHESTRA.

2.4 A Hierarchy of Provenance

Besides the provenance models discussed so far, several other forms of provenance information proposed in the literature can also be captured using semiring annotations. As we shall see, the various models also turn out to be related in a precise manner, namely by *surjective semiring homomorphisms*, as summarized in Figure 2.7.

One such provenance model is obtained from the provenance polynomials by replacing natural number coefficients with Boolean coefficients:

Definition 2.12 (Boolean Provenance Polynomials)**.** The *Boolean provenance polynomials semiring* for X is the semiring of polynomials over variables X with Boolean coefficients: $(\mathbb{B}[X], +, \cdot, 0, 1)$.

Considering the same positive relational algebra query Q as in Section 2.1, Figure 2.6(c) shows the result of applying Q to R, where R is interpreted as a $\mathbb{B}[X]$-relation. Note that the annotations in Figure 2.6(c) can be obtained from those in Figure 2.6(b) by simply dropping the numeric coefficients. In fact, one can check that the operation $f : \mathbb{N}[X] \to \mathbb{B}[X]$ which "drops coefficients" (i.e., by replacing non-zero coefficients with true) is a *surjective semiring homomorphism*.

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

(a) Source R

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

(b) $\mathbb{N}[X]$

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

(c) $\mathbb{B}[X]$

A	B	
a	c	$2p$
a	e	pr
d	c	pr
d	e	$2r + rs$
f	e	$2s + rs$

(d) $\mathsf{Trio}(X)$

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

(e) $\mathsf{Why}(X)$

A	B	
a	c	p
a	e	$p \wedge r$
d	c	$p \wedge r$
d	e	r
f	e	s

(f) $\mathsf{PosBool}(X)$

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

(g) $\mathsf{Lin}(X)$

Figure 2.6: Comparison of provenance annotations

The Trio system [8] has recently proposed a form of annotated relation suitable as a representation system for incomplete *bag* databases (i.e., where the possible worlds are bag instances rather than set instances). Trio-style tables can also be captured using annotations from a semiring we shall denote $\mathsf{Trio}(X)$. Like $\mathbb{B}[X]$, this semiring can be viewed as being obtained from $\mathbb{N}[X]$, but instead of "dropping coefficients," this time we "drop exponents." This is most conveniently formalized using the notion of *quotient semirings* (see Appendix for definition). Let $f : \mathbb{N}[X] \to \mathbb{N}[X]$ be the mapping that "drops exponents," e.g., f maps $2x^2y + 3xy + 2z^3 + 1$ to $5xy + 2z + 1$. Denote by $\approx_f$ the equivalence relation on $\mathbb{N}[X]$ defined by $a \approx_f b \overset{\text{def}}{\iff} f(a) = f(b)$. One can check that $\approx_f$ is a *congruence relation* (see Appendix for definition). This justifies the following:

Definition 2.13 (Trio Semiring)**.** The *Trio semiring* for X is the *quotient semiring* of $\mathbb{N}[X]$ by $\approx_f$, denoted $\mathsf{Trio}(X)$.

As an example, considering again the same query Q, Figure 2.6(d) shows the result of applying Q to R, where R is interpreted as a $\mathsf{Trio}(X)$-relation, and an annotation a is understood to represent its equivalence class $a/{\approx_f}$ in $\approx_f$. Note that the mapping $h : \mathbb{N}[X] \to \mathsf{Trio}(X)$ defined by $h(a) \mapsto a/{\approx_f}$ is a surjective semiring homomorphism.

Still another provenance model that can be captured using semirings is the *why-provenance* of [17]. The why-provenance of a tuple is the *set of sets* of "contributing" source tuples, which is called the *proof witness basis* in [17]. This can be captured using a semiring [13] (called the *proof why-provenance* semiring in [13]):

Definition 2.14 (Why-Provenance)**.** The *why-provenance semiring* for X is

$(\mathsf{Why}(X), \cup, \Cup, \emptyset, \{\emptyset\})$ where $\mathsf{Why}(X) \overset{\text{def}}{=} \mathcal{P}(\mathcal{P}(X))$ and $\Cup$ denotes *pairwise union*:

$$A \Cup B \overset{\text{def}}{=} \{a \cup b : a \in A, b \in B\}$$

Considering again the same query Q, we can interpret the source relation in Figure 2.6(a) as a why-provenance relation by doubly-nesting the variables (e.g., p becomes $\{\{p\}\}$). Figure 2.6(e) shows the query output and the resulting why-provenance annotations. Note that these annotations can be obtained from the $\mathbb{B}[X]$-annotations by dropping exponents (and writing the result as a set of sets rather than sum of monomials). One can check that the corresponding operation $g : \mathbb{B}[X] \to \mathsf{Why}(X)$ which "drops exponents" is in fact a surjective semiring homomorphism. Note also that the annotations can be obtained from the $\mathsf{Trio}(X)$-annotations by dropping coefficients, and it is easy to verify that the corresponding operation $h : \mathsf{Trio}(X) \to \mathsf{Why}(X)$ which does this is also a surjective semiring homomorphism.

Note that the lineage for an output tuple can be obtained from the why-provenance of the tuple by *flattening* the set of sets, i.e., applying the function $h : \mathsf{Why}(X) \to \mathsf{Lin}(X)$ defined by $h(I) = \bigcup_{S \in I} S$. Once again, we can show that h is a surjective semiring homomorphism.

Finally, an interesting variation on the why-provenance semiring is obtained by requiring that the witness basis for an output tuple be *minimal*. Here the domain is $\mathsf{irr}(\mathcal{P}(X))$ the set of *irredundant* subsets of $\mathcal{P}(X)$, i.e., W is in $\mathsf{irr}(\mathcal{P}(X))$ if for any A, B in W neither is a subset of the other. We can associate with any $W \subseteq \mathcal{P}(X)$ a unique irredundant subset $\mathsf{irr}(W)$ by repeatedly looking for elements A, B such that $A \subseteq B$ and deleting B from W. Then we define a semiring $(\mathsf{irr}(\mathcal{P}(X)), +, \cdot, 0, 1)$ as follows:

$$\begin{array}{rcl@{\qquad}rcl} I + J & \overset{\text{def}}{=} & \mathsf{irr}(I \cup J) & I \cdot J & \overset{\text{def}}{=} & \mathsf{irr}(I \Cup J) \\ 0 & \overset{\text{def}}{=} & \emptyset & 1 & \overset{\text{def}}{=} & \{\emptyset\} \end{array}$$

This is the semiring in which we compute the *minimal witness basis* [17]. It is a well-known semiring: the construction above is the construction for the free distributive lattice generated by the set X. Moreover, it is isomorphic to a semiring we have already seen, $\mathsf{PosBool}(X)$.

These various models of provenance can be neatly arranged in the diagram of Figure 2.7, where a path downwards from K_1 to K_2 indicates that there exists a *surjective semiring homomorphism* $h : K_1 \to K_2$. Coupled with Proposition 2.5, this gives a clear picture of the relative "*informativeness*" of the various provenance models, since provenance computations for models lower in the hierarchy can always be factored through computations involving models above them in the hierarchy. The existence of such homomorphisms also turns out to be useful in Chapter 4 where we shall use them to derive some easy bounds on the relative behavior of the provenance models with respect to query containment.

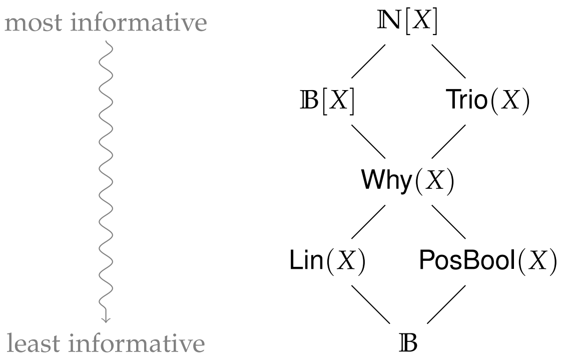


Figure 2.7: Provenance hierarchy

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

(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 2.8: Datalog with bag semantics

2.5 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 here. We also consider only "pure" Datalog rules in which all subgoals are relational atoms. Also, to simplify the presentation, we shall focus on Datalog programs with a single idb-and-output predicate, and we shall identify a Datalog program by its output predicate name.

First observe that for conjunctive queries over K-instances the semantics in Definition 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 2.8(a) and (b), respectively.

There are two valuations that produce the answer $Q(a,b)$: $\{x \mapsto a, y \mapsto a, z \mapsto b\}$ yields the body $R(a,a), R(a,b)$ while $\{x \mapsto a, y \mapsto b, z \mapsto b\}$ 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 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 proof-theoretic 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, stated formally in Definition 2.15. Indeed, this generalizes the bag semantics of Datalog considered in [113, 114] *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 [115]) hence we need to work with semirings in which infinite sums are defined.

Closed semirings [133] have infinite sums but their "$\oplus$" is idempotent which rules out the bag and provenance semantics. We will adopt the approach used in formal languages [95] 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 $(K, \oplus, \otimes, 0, 1)$ be a semiring and define $a \leq b \overset{\text{def}}{\Leftrightarrow} \exists x\ a \oplus x = b$. 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 have least upper bounds. In such semirings we can define *countable sums*:

$$\bigoplus_{n \in \mathbb{N}} a_n \overset{\text{def}}{=} \sup_{m \in \mathbb{N}} (\oplus_{i=0}^{m} a_i)$$

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

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

Examples of commutative ω-continuous semirings:

- $(\mathbb{B}, \vee, \wedge, \mathsf{false}, \mathsf{true})$
- $(\mathbb{N}^\infty, +, \cdot, 0, 1)$ where 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 $\mathbb{N}$.
- $(\mathsf{PosBool}(B), \vee, \wedge, \mathsf{false}, \mathsf{true})$ with B finite. This commutative semiring is in fact a *distributive lattice* [33] and the natural order is the lattice order. Since we identify those expressions that yield the same truth-value for all boolean assignments for the variables, B finite makes $\mathsf{PosBool}(B)$ finite, hence ω-continuous.
- $(\mathcal{P}(\Omega), \cup, \cap, \emptyset, \Omega)$, used for event tables which is also an example of distributive lattice.
- $(\mathbb{N}^\infty, \min, +, \infty, 0)$, the *tropical semiring* [95].
- $([0, 1], \max, \min, 0, 1)$ is related to *fuzzy sets* [137] so we will call it the *fuzzy semiring*.

Definition 2.15. Let $(K, \oplus, \otimes, 0, 1)$ be a commutative ω-continuous semiring, and let Q be a Datalog query. For any K-instance I, define

$$[\![Q]\!]^I(t) = \bigoplus_{\tau \text{ yields } t} \Big(\bigotimes_{R(t') \in \text{leaves}(\tau)} R^I(t') \Big)$$

where τ ranges over all Q-derivation trees for t and $R(t')$ ranges over all the leaves of τ.

Proposition 2.16. *For any K-instance I, $[\![Q]\!]^I$ has finite support and is therefore a K-relation. Hence, Definition 2.15 gives us a semantics for Datalog queries on K-instances.*

Proof. (sketch) Let J be the set instance corresponding to the support of I, and let t' be an output tuple such that $[\![Q]\!]^I(t') \neq 0$. By Definition 2.15, this implies that there is a derivation tree τ for t' (such that the tags of the tuples in its leaves are non-zero and correspond to this product). But then $t' \in [\![Q]\!]^J$. Since $[\![Q]\!]^J$ is finite, $[\![Q]\!]^I$ has finite support. ☐

Example 2.17. As an example, consider the Datalog program Q defined by the rules shown in Figure 2.9(c), applied on $\mathbb{N}$-relation R shown in Figure 2.9(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[4] and we obtain the table sohown in Figure 2.9(b). Observe that the annotation of (a, b) in the output is 8. Figure 2.10(a) shows how this value was computed as the sum of products of the annotations of the leaves of the two derivation trees for $Q(a, b)$ On the other hand, observe that (b, d) has annotation ∞ in the output. Figure 2.10(b) illustrates the computation of this value, which involves infinitely many derivation trees. ☐

A couple of sanity checks follow.

Proposition 2.18. *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 K-instance.*

Proposition 2.19. *For any Datalog query Q and any $\mathbb{B}$-instance I, $\mathsf{supp}([\![Q]\!]^I)$ is the same as the result of applying Q to the standard set instance $\mathsf{supp}(I)$.*

The definition of Datalog semantics given above is not so useful computationally. However, we can think of it as a 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 produced as a result of applying the

---

[4] This particular query computes 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)

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

(d)

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

(e)

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

(f)

Figure 2.9: Datalog example

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.

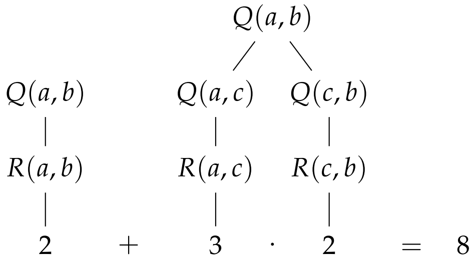
To make this precise we consider **polynomials** with coefficients in an arbitrary commutative semiring K. If the set of variables is X we denote the set of polynomials by $K[X]$. We have already used $\mathbb{N}[X]$ for provenance but $K[X]$ also forms a commutative semiring. We saw in Section 2.3 that because $\mathbb{N}$ can be embedded in any semiring K the polynomials P in $\mathbb{N}[X]$ define polynomial functions f_P 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 2.20. 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

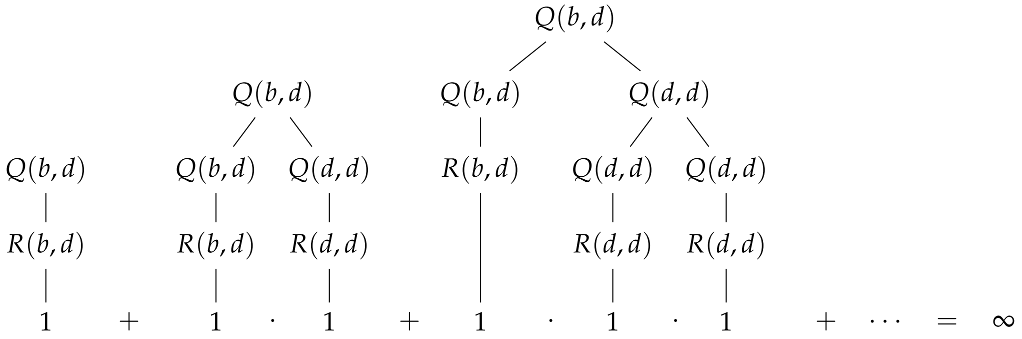
$$\begin{aligned}
x_1 &= P_1(x_1, \ldots, x_n) \\
&\cdots \\
x_n &= P_n(x_1, \ldots, x_n)
\end{aligned}$$

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

$$\operatorname{lfp}(f_{\mathbf{P}}) = \sup_{m \in \mathbb{N}} f_{\mathbf{P}}^m(0, \ldots, 0)$$



(a) Derivation trees for $Q(a,b)$



(b) Derivation trees for $Q(b,d)$

Figure 2.10: Derivation trees for Datalog example

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

$$a^* \stackrel{\text{def}}{=} 1 + a + a^2 + a^3 + \cdots$$

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 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 $\mathsf{ab}(Q)$ be the abstractly-tagged (as in Theorem 2.11) version of Q where X is the set of ids of the tuples in $\mathsf{supp}(Q)$. Since $\mathsf{ab}(Q)$ is a $X \cup \{0\}$-relation and R is a K-relation both can be seen also as $K[X]$-instances. The immediate consequence operator T_Q is in fact a union of conjunctive queries, hence Definition 2.2 shows how to calculate effectively $T_Q(R, \mathsf{ab}(Q))$ as a $K[X]$-relation of finite support. By equating the tags of $\mathsf{ab}(Q)$ with those of $T_Q(R, \mathsf{ab}(Q))$ we obtain the promised algebraic system. We will denote this system as $\mathsf{ab}(Q) = T_Q(R, \mathsf{ab}(Q))$ (although it only involves the tags of these relations).

Theorem 2.21. *With the notation above, for any tuple t, the tag $Q(t)$ given by Definition 2.15, when not*

0, *equals the component of the solution (Definition 2.20) of the algebraic system* $\mathsf{ab}(Q) = T_Q(R, \mathsf{ab}(Q))$ *corresponding to the id of* t.

To illustrate with an example, consider again the Datalog program Q in Figure 2.9(a) applied to the same $\mathbb{N}$-relation R shown in Figure 2.9(b). In Figure 2.9(e) we have the abstractly-tagged version of the output relation, $\mathsf{ab}(Q)$ in which the tuples are tagged with their own ids. The corresponding algebraic system is the one obtained from Figure 2.9(f) by replacing $m = 2, n = 3, p = 2, r = 1, s = 1$. (Note that $T_Q(R, \mathsf{ab}(Q)) = R \cup \mathsf{ab}(Q) \bowtie \mathsf{ab}(Q)$.) 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 2.7 we show how unbounded growth can be detected). Hence the solution is the one shown in Figure 2.9(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 2.22. *Let* K_1, K_2 *be commutative* ω*-continuous semirings and let* $h : K_1 \to K_2$ *be an* ω*-continuous semiring homomorphism. Then, the transformation given by* h *from* K_1*-instances to* K_2*-instances commutes with Datalog query evaluation: for any Datalog query* Q *and* K_1*-instance* I,

$$[\![Q]\!]^{h(I)} = h([\![Q]\!]^I).$$

2.6 Formal Power Series for Provenance

In Section 2.3 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* [95].

Note that when we try to apply naively Definition 2.15 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 formal power series (which generalizes the same operations for polynomials):

$$\begin{aligned} (S_1 \otimes S_2)(\mu) &= \sum_{\mu_1\mu_2=\mu} S_1(\mu_1) \otimes S_2(\mu_2) \\ (S_1 \oplus S_2)(\mu) &= \sum_{\mu_1\mu_2=\mu} S_1(\mu_1) \otimes S_2(\mu_2) \end{aligned}$$

But the real reason we use formal power series is

Theorem 2.23. *(e.g., [95]) If K is ω-continuous then $K[[X]]$ is also ω-continuous.*

Definition 2.24. 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 provenance for the output of the Datalog query in Figure 2.9(a). The input, with tuple ids as tags, is R in Figure 2.9(d). Note that we have *two sets of variables* here. The tuple ids of R forms 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 $\mathsf{ab}(Q)$ 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 $\mathbf{v}$ component of the solution can be calculated separately:[5]

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

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 $\mathbf{w}$ 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:

---

[5] [26] shows that the coefficient of s^{n+1} is $\frac{2n!}{n!(n+1)!}$.

Proposition 2.25. *Let Q be an $\mathcal{RA}^+$ query 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 I be an $\mathbb{N}[X]$-instance. Modulo the embedding of $\mathbb{N}[X]$ in $\mathbb{N}^\infty[[X]]$ we have $[\![Q]\!]^I = [\![Q']\!]^I$.*

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

Proposition 2.26. *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*

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

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

Therefore, just like polynomials, formal power series define **series functions** on any commutative ω-continuous semiring. Finally, we have the analog of Theorem 2.11.

Theorem 2.27. *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. 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.

Theorem 2.28. *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) such that 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.*

2.7 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.

Given a Datalog program Q and an abstractly-tagged $\mathbb{N}[X]$-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 $N[X]$. Algorithm 1 (inspired by [115]) 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 1 is ∞, we can use Theorem 2.28 to decide whether $[\![Q]\!]^I(t)$ is in $\mathbb{N}[[X]]$. The remaining question is whether $[\![Q]\!]^I(t)$ is in $\mathbb{N}^\infty[X]$,

Algorithm 1 All-Trees(Datalog query Q, $\mathbb{N}[X]$-instance I) returns the power series $P(t)$ for every tuple $t \in [\![Q]\!]^I$

```
All-Trees
Initialize $F \leftarrow \{t : t \in \mathsf{supp}(I)\}$
Initialize $\Omega \leftarrow \emptyset$
repeat
    $F^\nu \leftarrow T^\nu_Q(F, \Omega)$
    for every tree $\tau \in F^\nu$ do
        if any child of root($\tau$) is in $\Omega$ or any proper descendant of root($\tau$) to a node associated with the same tuple then
            $\Omega \leftarrow \Omega \cup \{\mathrm{root}(\tau)\}$
        else
            $F \leftarrow F \cup \{\tau\}$
        end if
    end for
until nothing added to either $F$ or $\Omega$ in last iteration
for every $t \in [\![Q]\!]^{\mathsf{supp}(I)}$ do
    if $t \in \Omega$ then
        $P(t) \leftarrow \infty$
    else
        $P(t) \leftarrow \sum_{\substack{\tau \in F \text{ s.t.} \\ \mathrm{root}(\tau) = t}} \prod_{R(t') \in \mathrm{fringe}(\tau)} R^I(t')$
    end if
end for
return $P$
```

i.e., it is a polynomial instead of a formal power series. We can decide this 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 1:

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

Algorithm 1 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 Ω, no more trees for it are produced. Note also that by Theorem 2.27 this algorithm will also give us, in particular, an algorithm for evaluating Datalog queries on bag semantics, similar to the one in [115].

If the answer of Algorithm 1 for an output tuple t is ∞, we can also use Algorithm 2 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 for which infinite derivation trees whose leaves are equal to the monomial μ have been found.
- $T_Q^i(F, P^\infty, \mu) = \{\tau \mid \tau \notin F \wedge \tau \in T_Q(F) \wedge \text{fringe}(\tau) \leq \mu \wedge (\text{root}(\tau), m) \notin P^\infty\}$, where $T_Q(F)$ is the set of trees that can be produced by applying a rule on tuples in $\{\text{root}(\tau) \mid \tau \in F\} \cup \{t \mid (t, m) \in P^\infty\}$ it, where the multiplicity of each is the corresponding exponent in the monomial.

Algorithm 2 Monomial-Coefficient(Datalog query Q, monomial μ, tuple t)
returns C, the coefficient of μ in the power series $P(t)$

```
Monomial-Coefficient
  $F \leftarrow \{t() : t \in M\}$
  $P^\infty \leftarrow \emptyset$
  repeat
    $F^i \leftarrow T_Q^i(F, \Omega)$
    for every tree $\tau \in F^i$ 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
        $F \leftarrow F \cup \{\tau\}$
      end if
    end for
  until nothing added to either $F$ or $\Omega$ in last iteration
  if $(t, \mu) \in P^\infty$ then
    $C \leftarrow \infty$
  else
    for every $\tau \in F$ s.t. $\text{root}(\tau) = t$ and $\text{fringe}(\tau) = \mu$ do
      $C \leftarrow C + 1$
    end for
  end if
  return $C$
```

If n is the length of the longest acyclic path of unit rules, then every $n + 1$ iterations Algorithm 2 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 $\text{fringe}(\tau) = \mu'$ are used in any subsequent derivations, and moreover, the set of trees τ s.t. τ does not involve nodes marked as infinite and $\text{fringe}(\tau) \leq \mu$ is finite.

2.8 Application to Incomplete/Probabilistic Databases

Theorem 2.27 suggests using algorithm 1 for evaluating Datalog queries in various semiring. We already noted in Section 2.7 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 1 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 do so with some simple modifications to algorithm 1:

- Redefine T_Q^ν to take only F as a parameter, and return all τ in $T_Q(F)$ such that for all τ' in F, if $\text{root}(\tau) = \text{root}(\tau')$, then $\text{fringe}(\tau) < \text{fringe}(\tau')$.

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 verify. For $K = \mathsf{PosBool}(B)$, after also checking that for any valuation $\nu : B \to \mathbb{B}$ we have $\mathsf{Eval}_\nu([\![Q]\!]^I) = [\![Q]\!]^{\mathsf{Eval}_\nu(I)}$, 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 $[\![Q]\!]^I$ for the random instance I, we conclude that our algorithm also generalizes that of [51].

2.9 Provenance-Annotated Queries

Sometimes, one would like to know not just which source tuples were involved in producing an output tuple, but also which part of the query. In CDSS, for example, we need a mechanism that can be used to "tag" output tuples with information which mapping rules were used to produce those output tuples (in addition to the information about the source tuples which were used). We close this chapter with an extension to the framework that can accomplish this by means of a *scalar multiplication* construct which may be used freely in relational algebra or Datalog queries. The idea is to decorate portions of a relational algebra or Datalog query with *query provenance* annotations, and combine these annotations with the source level provenance in the query output.

While motivated by our immediate concerns in CDSS, query provenance has some wider applications that we shall discuss at the end of the section.

Definition 2.29. For a semiring K, the language K-$\mathcal{RA}^+$ extends the positive relational algebra

(Definition 2.2) with a *scalar multiplication* operation $a \otimes Q_1$, where $a \in K$, whose denotation on a K-instance I is defined as follows:

scalar If $[\![Q_1]\!]^I : U\text{-}\mathsf{Tup} \to K$ and $a \in K$ then $[\![a \otimes Q_1]\!]^I$ is defined

$$[\![a \otimes Q_1]\!]^I \overset{\mathtt{def}}{=} a \otimes [\![Q_1]\!]^I$$

(where $\otimes$ is multiplication in K) and notice that if $[\![Q_1]\!]^I$ has finite support, then so does $[\![a \otimes Q_1]\!]^I$, provided $a \otimes 0 = 0$.

It is straightforward to verify that the key properties of $\mathcal{RA}^+$ on K-instances — namely, commutation with semiring homomorphisms (Proposition 2.5), and factoring of computations through $\mathbb{N}[X]$ (Theorem 2.11) — continue to hold for K-$\mathcal{RA}^+$. The statement of the commutation with homomorphisms changes slightly as now we lift a homomorphism $h : K_1 \to K_2$ to also map K_1-$\mathcal{RA}^+$ queries to K_2-$\mathcal{RA}^+$ queries, by replacing every scalar a occurring in the K_1-$\mathcal{RA}^+$ query with $h(a)$, and show that for any semiring homomorphism $h : K_1 \to K_2$, query Q in K_1-$\mathcal{RA}^+$, and K_1-instance I, we have

$$[\![h(Q)]\!]^{h(I)} = h([\![Q]\!]^I)$$

Similarly, to obtain *K-Datalog*, we extend the syntax of Datalog to allow the heads of Datalog rules to be decorated with scalar multiplications. For example, the transitive closure query Q from Figure 2.9(a) might be decorated with fresh variables m_1 and m_2 from X to obtain the $\mathbb{N}[X]$-Datalog query

$$\begin{aligned} m_1 \cdot Q(x,y) &\quad \text{:-} \quad R(x,y) \\ m_2 \cdot Q(x,y) &\quad \text{:-} \quad Q(x,z), Q(z,y) \end{aligned}$$

We then generalize the proof-theoretic semantics on Datalog on K-instances (Definition 2.15) to take these into account. We omit the details which are straightforward. Correspondingly, we extend the fixpoint-theoretic semantics (Section 2.5) to include these scalar multiplications in the provenance annotations produced by the immediate consequence operator. For example, the result of applying the $\mathbb{N}[X]$-Datalog query Q above to the $\mathbb{N}[X]$-instance containing the single table R from Figure 2.9(d) using the fixpoint-theoretic semantics:

a	b	$\mathbf{x} = m_1 m + m_2 \mathbf{yz}$
a	b	$\mathbf{y} = m_1 n$
c	b	$\mathbf{z} = m_1 p$
b	d	$\mathbf{u} = m_1 r + m_2 \mathbf{uv}$
d	d	$\mathbf{v} = m_1 s + m_2 \mathbf{v}^2$
a	d	$\mathbf{w} = m_2 \mathbf{xu} + m_2 \mathbf{wv}$

As with K-$\mathcal{RA}^+$, we can show that key properties of Datalog on K-instances (commutation with homomorphisms, factoring, conservativeness with respect to $\mathcal{RA}^+$ on K-relations) are preserved under the extension to K-Datalog.

Wider applications of provenance-annotated queries

We close the section by mentioning some further applications of query provenance outside of CDSS worth exploring in future work. One interesting application involves "debugging" a query by annotating portions of a query in order to determine which "parts" of the query were involved in producing a tuple of interest in the output.

Example 2.30. Consider again the $\mathbb{N}[X]$-instance containing a single table R shown in Figure 2.5, and suppose now that we wish to determine which "side" of the union of the query Q from Section 2.1 was involved in producing the various output tuples. To accomplish this, we annotate Q itself with fresh tags u_1 and u_2 from X, producing the following $\mathbb{N}[X]$-$\mathcal{RA}^+$query Q':

$$Q' \overset{\text{def}}{=} \pi_{\text{AC}}\big(u \otimes (\pi_{\text{AB}}R \bowtie \pi_{\text{BC}}R) \cup v \otimes (\pi_{\text{AC}}R \bowtie \pi_{\text{BC}}R)\big)$$

The annotation of (a, e) in $[\![Q']\!]^R$, for example, becomes pru (rather than pr as in Figure 2.5(c)); hence we see that this tuple was produced using the *left* side of the union (and source tuples p and r). $\square$

Indeed, data provenance has been proposed before for *debugging schema mappings* in the Spider system [25], which uses a kind of provenance called *route-provenance*. Route provenance is related to our semiring-based Datalog provenance (see further discussion in Chapter 7). Query provenance seems to nicely complement data provenance for such applications.

Another promising application of query provenance is in *probabilistic data integration* [41]. Here, the queries (or schema mappings) used for data integration are themselves uncertain, for example, because the relationships between databases are not yet well understood. Such uncertainties may be modeled in a number of ways, for example by assigning probabilities to Datalog rules corresponding to the likelihood that the rule is correct. The goal, as with probabilistic databases, is to compute the corresponding probabilities of output tuples.

Example 2.31. Consider a source relation R and a target relation S, and suppose that there are two possibilities for a mapping relating them, shown in Figure 2.11(a), but we are not sure which is the correct mapping. Instead, we suspect that the first rule is the correct one with probability 0.4, say, and the other with probability 0.6. We can model this situation using query provenance annotations from the event tables semiring $(\mathcal{P}(\Omega), \cup, \cap, \emptyset, \Omega)$, as shown in Figure 2.11(b), where the head of the first rule is annotated with event $\textsc{r}$, and the other with its complementary event

$$\begin{array}{lcl} S(x,y) & \text{:-} & R(x,y,z) \\ S(x,y) & \text{:-} & R(x,z,y) \end{array}$$

(a)

$$\begin{array}{lcl} \textsc{r} \cap S(x,y) & \text{:-} & R(x,y,z) \\ \overline{\textsc{r}} \cap S(x,y) & \text{:-} & R(x,z,y) \end{array}$$

(b)

E	Pr
$\textsc{r}$	0.6
$\textsc{x}$	0.6
$\textsc{y}$	0.5
$\textsc{z}$	0.1

(c)

a	b	c	$\textsc{x}$
a	b	d	$\textsc{y}$
a	e	c	$\textsc{z}$

(d)

a	b	$\textsc{r} \cap (\textsc{x} \cup \textsc{y})$
a	e	$\textsc{r} \cap \textsc{z}$
a	c	$\overline{\textsc{r}} \cap (\textsc{x} \cup \textsc{z})$
a	d	$\overline{\textsc{r}} \cap \textsc{y}$

(e)

Figure 2.11: Probabilistic data integration example

$\overline{\textsc{r}}$. The probability of event $\textsc{r}$ is 0.6, as indicated in Figure 2.11(c). Now, consider the decorated query applied to the event table R shown in Figure 2.11(d); the result is the table in Figure 2.11(e). Note that the annotations in the output combine query provenance and data provenance, and the probabilities of output tuples can be computed using these annotations and the independence of the event probabilities in Figure 2.11(c). □

The use of query provenance in the example above models the so-called *per-table* semantics of uncertain mappings described in [41]. However, unlike in [41], here we were able to simultaneously model uncertainties in *both data and mappings*, using semiring annotations as a unifying framework.

Chapter 3

Update Exchange in Orchestra

One of the most elusive goals of the data integration field has been supporting sharing across large, heterogeneous populations. While data integration and its variants (e.g., data exchange [45] and warehousing) are being adopted in corporations or small confederations, little progress has been made in integrating broader communities. Yet the need for sharing data across large communities is increasing: most of the physical and life sciences, especially those related to biology and astronomy, have become data-driven as they have attempted to tackle larger questions. The field of bioinformatics, for instance, has a plethora of different databases, each providing a different perspective on a collection of organisms, genes, proteins, diseases, and so on. Associations exist between the different databases' data (e.g., links between genes and proteins, or gene homologs between species). Unfortunately, data in this domain is surprisingly difficult to integrate, primarily because conventional data integration techniques require the development of a single global schema and complete global data consistency. Designing one schema for an entire community like systems biology is arduous, involves many revisions, and requires a central administrator[1]. Even more problematic is the fact that the data or associations in different databases are often contradictory, forcing individual biologists to choose values from databases they personally consider most authoritative [112]. Such inconsistencies are not handled by data integration tools, since there is no consensus or "clean" version of the data. Thus, scientists simply make their databases publicly downloadable, so users can copy and convert them into a local format (using custom Perl scripts or other ad hoc measures). Meanwhile the original data sources continue to be edited. In some cases the data providers publish weekly or monthly lists of updates (*deltas*) to help others keep synchronized. Today, few participants, except those with direct replicas, can actually exploit such deltas — hence, once data has been copied to a dissimilar database, it begins to diverge from the original.

---

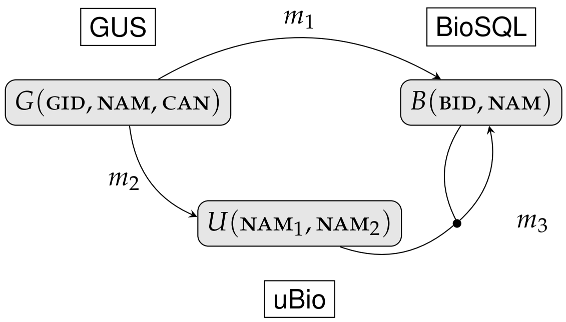
[1] Even recent architectures like peer data management [10, 89, 110, 73], which relax administration by supporting multiple mediated schemas, assume data is consistent.

To address the needs of scientists, we have been developing an extremely flexible scheme for sharing data among different participants, the *collaborative data sharing system* (abbreviated CDSS) [82]. Rather than providing a global view of all data, this scheme instead facilitates import and export among autonomous databases. The CDSS provides a principled architecture that extends the data integration approach to encompass today's scientific data sharing practices and requirements: Our goal with the CDSS is to provide the basic capability of *update exchange*, which publishes a participant's updates to "the world" at large, and then maps others' updates to the participant's local schema. As part of the mapping process, update exchange should also filter *which* updates to apply according to the local administrator's unique trust policies. Data sharing 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. As edits are made to this database, they are logged. Declarative *schema mappings* specify one database's relationships to other participants, much as in peer data management systems [73]; 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 is bidirectional and 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). As a prerequisite to assessing trust, the CDSS records the derivation of the updates, i.e., their provenance or lineage [17, 34, 25, 9]. After the update exchange, the participant has an up-to-date data instance, incorporating trusted changes made by participants transitively reachable via schema mappings. Update exchange introduces a number of novel challenges that we address in this paper:

- Building upon techniques for exchanging **data** using networks of schema mappings, we map **updates** across participants' schemas: this facilitates incremental view maintenance in a CDSS, and it also generalizes peer data management [73] and data exchange [118, 45].

- We allow each participant to perform modifications that "override" data imported from elsewhere (i.e., remove or replace imported data in the local instance), or add to it (i.e., add new data to the local database instance).

- We address additional situations in which a set of participants wants greater sharing, such that an update (including a deletion) by any participant should be propagated to *all* participants — i.e., updates affect both source and derived data.


GUS
m_1
BioSQL
$G(\textsc{gid}, \textsc{nam}, \textsc{can})$
$B(\textsc{bid}, \textsc{nam})$
m_2
$U(\textsc{nam}_1, \textsc{nam}_2)$
m_3
uBio

$$\begin{array}{ll} (m_1) & G(g,n,c) \rightarrow \exists b\ B(b,n) \\ (m_2) & G(g,n,c) \rightarrow U(n,c) \\ (m_3) & B(b,n_1) \wedge U(n_1,n_2) \rightarrow B(b,n_2) \end{array}$$

Figure 3.1: Mappings among three bioinformatics databases

- In order to allow CDSS participants to specify what data they trust and wish to import, we propose the use of *semiring provenance* [62] to trace the derivation of each mapped update, and we filter updates with participant-specified *trust conditions*.
- We develop a complete implementation of update exchange in our ORCHESTRA CDSS prototype, with novel algorithms and encoding schemes to translate updates, maintain provenance, and apply trust conditions. We provide a detailed experimental study of the scalability and performance of our implementation.

Roadmap. Section 3.1 provides an overview of the CDSS update exchange process. Section 3.2 formalizes the system operation and our data model. Section 3.3 presents algorithms for supporting update exchange, and Section 3.4 describes how we implemented our techniques within the ORCHESTRA system. We experimentally evaluate our work in Section 3.5.

3.1 CDSS Update Exchange

The CDSS model builds upon the fundamentals of data integration and peer data management [73], but adds several novel aspects. As in a peer data management system, 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, and the conditions under which the peer trusts data. Without loss of generality, we assume that each peer has a schema disjoint from the others. In the ORCHESTRA implementation we assume relational schemas, but our model extends naturally to other data models such as XML and nested relations (see Chapter 6).

Example 3.1. Figure 3.1 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, BioSQL, wants to import data from another peer, GUS, as shown by the arc labeled m_1, but the converse is not true. Similarly, peer uBio wants to import data from GUS, along arc m_2. Additionally, BioSQL uses the taxon synonyms in U to infer new entries in its database, via mapping m_3. Finally, each participant may have a certain *trust policy* about what data it wishes to incorporate: e.g., BioSQL may only trust data from uBio if it was derived from GUS entries. The CDSS facilitates dataflow among these systems, using mappings and policies developed by the independent participants' administrators. ☐

The arcs between participants in the example are formally a set of *schema mappings*. These are logical assertions that relate multiple relations from different peer schemas. We adopt the well-known formalism of *tuple-generating dependencies* (tgds). Tgds are a popular means of specifying constraints and mappings [38, 45] in data sharing, and they can express so-called *global-local-as-view* or *GLAV* mappings [49, 73]. (In turn, GLAV generalizes the earlier global-as-view and local-as-view mapping formulations [100].) A tgd is a logical assertion of the form:

$$\forall \bar{x}, \bar{y}\ (\varphi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z}\ \psi(\bar{x}, \bar{z}))$$

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 $\bar{z}$. The 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.

Notation. We use Σ for the **union** of all peer schemas and $\Sigma_{\mathbf{P}}$ for the schema of peer **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).

Example 3.2. Refer to Figure 3.1. Peers GUS, BioSQL, uBio have one-relation schemas describing taxa IDs, names, and canonical names:$\Sigma_{\mathsf{GUS}} = \{G(id, can, nam)\}$, $\Sigma_{\mathsf{BioSQL}} = \{B(id, nam)\}$, $\Sigma_{\mathsf{uBio}} = \{U(nam, can)\}$. Among these peers are mappings $\mathcal{M} = \{m_1, m_2, m_3\}$, shown as arcs in the figure, with their definitions on the right. Observe that m_1 has an existentially-quantified variable: the value of b is unknown (and not necessarily unique). The first two mappings all have a single source and target peer, corresponding to the LHS and the RHS of the implication. In general,

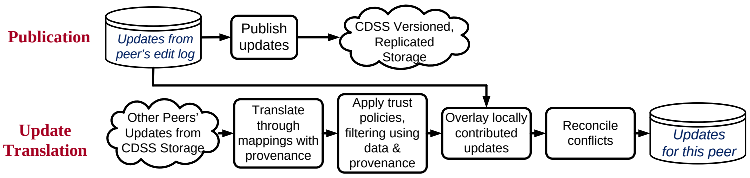

Publication
Updates from peer's edit log
Publish updates
CDSS Versioned, Replicated Storage
Update Translation
Other Peers' Updates from CDSS Storage
Translate through mappings with provenance
Apply trust policies, filtering using data & provenance
Overlay locally contributed updates
Reconcile conflicts
Updates for this peer

Figure 3.2: Dataflow at a single peer in a CDSS

relations from multiple peers may occur on either side, as in mapping m_3, which defines data in the BioSQL relation based on its own data combined with tuples from uBio. □

CDSS operation. Each peer **P** 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 [131]. 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), 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 that use the *provenance* (overview in Section 3.1) 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. See Figure 3.2 for a diagram summarizing these dataflow steps.

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. In the remainder of this section, we provide a high-level overview of these characteristic aspects of update exchange. Section 3.2 will revisit these aspects in order to define our model formally.

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. In this chapter, for simplicity of presentation we assume that these updates are mutually compatible. In our actual implementation, we treat them as *candidate* updates and use the CDSS prioritization and conflict reconciliation algorithm of [131] to determine which updates to apply.

Update Translation and Query Answers

To formalize the semantics of computing instances and query answers, we begin with a a *static* description of what should appear in each peer's data instance. Later, we will see how a practical implementation can be developed with a dynamic, *incremental* method to achieve this semantics (see Section 3.3).

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 [73]. Indeed, the certain answers semantics has been validated by over a decade of use in data integration and data exchange [45, 49, 73, 89, 102].

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, 104, 76], 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*. 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 3.2). 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 optionally return tuples with labeled nulls, i.e., a superset of the certain answers, which may be desirable for some applications.)

Example 3.3. Continuing our example, assume that the peers have the local edit logs shown in Figure 3.3(a) (where "+" signifies insertion). The update translation constructs the local instances

ΔG

+	1	2	3
+	4	5	6

ΔB

+	7	8

ΔU

+	6	5

(a) Local edit logs

G

1	2	3
4	5	6

B

7	8
$\perp_1$	2
$\perp_1$	3
$\perp_2$	5
$\perp_2$	6
$\perp_2$	9

U

6	5
2	3
5	6

(b) Local instances after update translation

Figure 3.3: Local edit logs and update translation

shown in Figure 3.3(b).

Examples of certain answers query semantics at BioSQL:

- query $ans(y,z)$:- $B(x,y), B(x,z)$ returns $\{(8,8),(2,2),(2,3),(3,2),(3,3),(5,5),(5,6),(6,5),(6,6)\}$
- query $ans(x,y)$:- $B(x,y)$ returns $\{(7,8)\}$

Moreover, if the edit log ΔU would have also contained the curation deletion $-(2,3)$ then after update translation, not only would U be missing $(2,3)$, but also B would be missing $(\perp_1, 3)$. ☐

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.

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 3.2.

Example 3.4. Some possible trust conditions in our CDSS example:

- Peer BioSQL distrusts any tuple $B(i,n)$ if the data came from GUS and $n \geq 3$, and trusts any tuple from uBio.
- Peer BioSQL distrusts any tuple $B(i,n)$ that came from mapping m_3 if $n \neq 2$.

☐

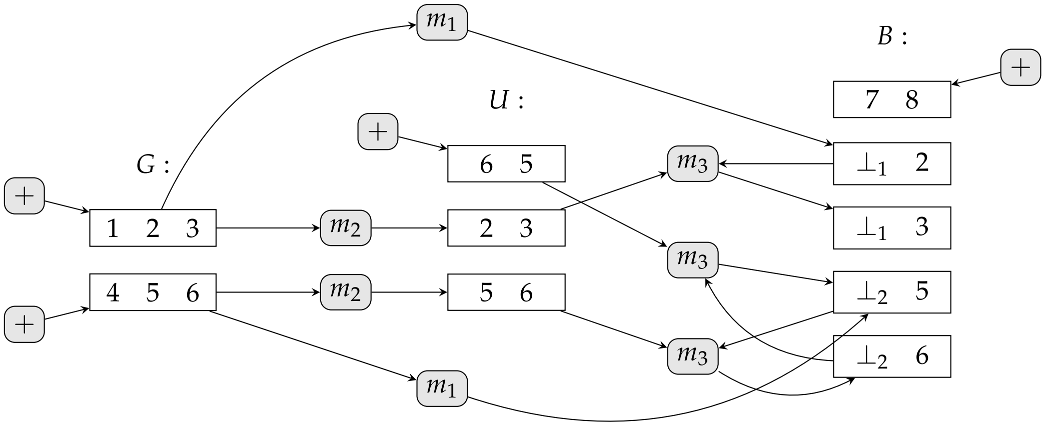


Figure 3.4: Provenance graph for bioinformatics example

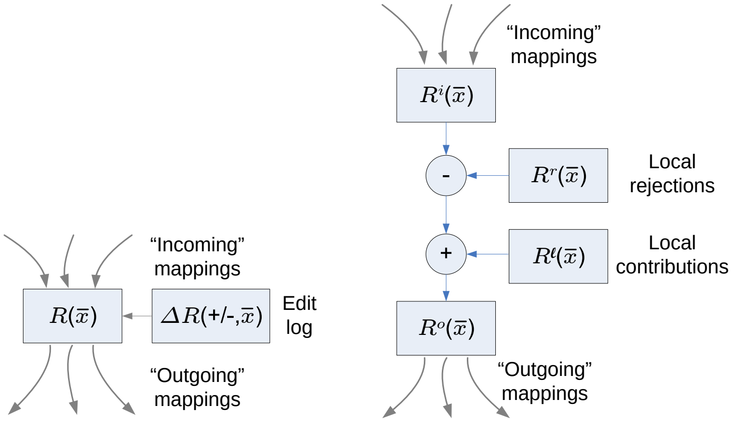
Since the trust conditions refer to other peers and to the schema mappings, the CDSS needs a precise description of how these peers and mappings have contributed to a given tuple produced by update translation. Information of this kind is commonly called *data provenance*. As we saw at the end of Example 3.3, provenance can be quite complex in a CDSS. In particular, we need a more detailed provenance model than why-provenance [17] and lineage [34] (including the extended model recently proposed in [9]). 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. We present this model more formally in Section 3.2 while here we illustrate our running example with a graphical representation of provenance.

Example 3.5. Consider the update translation from Example 3.3. The provenance graph shown in Figure 3.4 has two kinds of nodes: tuple nodes, shown as rectangles, and mapping nodes, shown as rounded boxes. Arcs connect tuple nodes to mapping nodes and mapping nodes to tuple nodes. In addition, we have nodes labeled "+" indicating insertions from the local edit logs. From this graph we can analyze the provenance of, say, $B(\perp_2, 5)$ by tracing back along the edges of the graph — in this case through m_1 to $G(4,5,6)$, and through m_3 to $U(6,5)$ and $B(\perp_2, 6)$. □

3.2 Update Exchange Formalized

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



Original relation and edit log

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

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

Figure 3.5: Effects of a peer's edit log

that recursively define relations in terms of themselves). In general, query answering is undecidable in the presence of cycles [73], so we restrict the topology of schema mappings to be at most *weakly acyclic* [39, 45]. Mapping (m_3) in Example 3.2 completes a cycle, but the set of mappings is weakly acyclic.

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 [82]). 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 3.5. Their meaning is as follows:

- R^+, the peer's *local contributions table*, contains the tuples inserted locally, unless the edit log shows they were later deleted.
- R^-, the peer's *rejections table*, contains tuples that were not inserted locally and that are rejected through a local curation deletion. (Deletions of local contributions are dealt with

simply by removing tuples from R^+).

- R^i is the peer's *input table*. It contains tuples produced by update translation, via mappings, from data at other peers.
- 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 and Example 3.3.

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^- and add those in R^+:

$$\begin{array}{ll} (i_R) & R^i(\bar{x}) \wedge \neg R^-(\bar{x}) \rightarrow R^o(\bar{x}) \\ (\ell_R) & R^+(\bar{x}) \rightarrow R^o(\bar{x}). \end{array}$$

Note that the previous description and Figure 3.5 do not incorporate the treatment of trust conditions. We show how to incorporate them at the end of Section 3.2.

Some of the new mappings in $\mathcal{M}'$ contain negation, and thus one might wonder if this affects the decidability of query evaluation.

Note that the negated atoms occur only on the LHS of the implication and every variable in the LHS also occurs in a positive atom there: we call such an assertion a *tgd with safe negation*. The results of [45, 104, 76] generalize to tgds with safe negation.

Theorem 3.6. *Let $\mathcal{M}$ be weakly acyclic and I be an instance of all local contributions and rejections tables. Then $\mathcal{M}'$ is also weakly acyclic, hence chase$_{\mathcal{M}'}(I)$ terminates in polynomial time. Moreover, if the rejections tables are empty 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 rejection tables, we use a procedure based on the chase of [45].

Definition 3.7 (Consistent system state)**.** An instance (I, J) of Σ', where I is an instance of the local rejections and contributions tables and J of the input and output tables, is **consistent** if J is equal to the output of applying $\text{chase}_{\mathcal{M}'}(I)$.

To recap, *publishing* a peer **P**'s edit log means producing a new instance of **P**'s local rejections and contributions tables, while holding the other peers' local rejections and contributions tables the same[2]. *Recomputing* **P**'s instance means computing a new instance of **P**'s input and output tables that matches the results of applying the chase 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.)

Provenance Expressions and Graphs

As explained in Section 3.1, CDSS operation needs a data provenance model that is more expressive than why-provenance or lineage. We base the model implemented in this chapter on the *provenance semiring* model proposed in Chapter 2, which has been characterized as "how-provenance" and provides provenance annotations whose properties closely mirror the relational algebra. While Chapter 2 used *systems of equations* to describe provenance, for the system implementation, an equivalent *graph-based* provenance formalism turns out to be more convenient. We will describe the graph-based formalism shortly.

Following the model of Chapter 2, the provenance of a source (base) tuple is represented by its own tuple id (we call this a *provenance token*), and that of a tuple computed through update translation is an expression over the *Datalog provenance semiring* $(\mathbb{N}^\infty[[X]], +, \cdot, 0, 1)$ of formal power series with coefficients from $\mathbb{N}^\infty$ over the set X of provenance tokens. Intuitively, $\cdot$ is used to combine the provenance of tuples in a join operation, mapping functions reflect the fact that the corresponding mapping was involved in the derivation, and $+$ is used in cases when a tuple can be derived in multiple ways (e.g., through different mappings). Using the terminology of proof-theoretic Datalog evaluation semantics (cf. Chapter 2, Definition 2.15), every summand in a provenance expression corresponds to a *derivation tree* for that tuple in the result of the Datalog program corresponding to the mappings.

Notation. Recall from Chapter 2 that a provenance-annotated relation is formally a mapping from tuples to their provenance annotations. Hence we write $R(t)$ for the provenance expression from $\mathbb{N}^\infty[[X]]$ of the tuple t in relation R. We sometimes omit $\cdot$ and use concatenation.

---

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

Example 3.8. To illustrate how the provenance expressions are formed, consider the mappings from Figure 3.1. If $G(1,2,3) = g$ then $U(2,3) = m_2 \cdot G(1,2,3) = m_2 \cdot g$ and $B(\bot_1,2) = m_1 \cdot G(1,2,3) = m_1 \cdot g$. Moreover, $B(\bot_1,3) = m_3 \cdot B(\bot_1,2) \cdot U(2,3) = m_3 \cdot m_1 \cdot g \cdot m_2 \cdot g$ Note that the expressions allow us to detect when the derivation of a tuple is "tainted" by a peer or by a mapping. Distrusting u and m_1 leads to rejecting $B(3,2)$ but distrusting u alone does not. Note also that $B(\bot_2,5) = m_1 \cdot G(4,5,6) + m_3 \cdot B(\bot_2,6) \cdot U(6,5) = m_1 \cdot g + m_3 \cdot B(\bot_2,6) \cdot u$ while $B(\bot_2,6) = m_3 \cdot B(\bot_2,5) \cdot U(5,6) = m_3 \cdot B(\bot_2,5) \cdot m_2 \cdot g$, a circular dependency corresponding to the cycle in the provenance graph. Expanding $B(\bot_2,5)$ leads to an infinite provenance expression, however, we can still perform derivation testing and trust policy enforcement using the finite graphical representation. ☐

When the mappings form cycles, it is possible for a tuple to have infinitely many derivations, as well as for the derivations to be arbitrarily large. In general, as explained in Chapter 2, the provenance of a tuple is akin to infinite *formal power series*. Nonetheless an instance's provenance is finitely representable through a *system of equations*: on the head of each equation we have a unique provenance variable **t** for every tuple t (base or derived), while in the body we have all semiring expressions that represent derivations of this tuple as an *immediate consequent* from other tuples (or provenance tokens, if t is a base tuple). Then, according to Chapter 2, the provenance of a tuple t is the value of **t** in the least fixpoint of the system formed by all these equations.

We can alternatively look at the system of provenance equations as forming a graph just like the one in Example 3.5.

Definition 3.9 (Provenance Graph)**.** The graph has two types of nodes: *tuple nodes*, one for each tuple in the system and *mapping nodes* where several such nodes can be labeled by the same mapping name. We represent an inserted value using a '+' mapping node. Every other mapping node corresponds to an instantiation of the mapping's tgds. This instantiation defines some tuples that match the LHS of the tgd and we draw edges from the corresponding tuple nodes to the mapping node (this encodes conjunction among source tuples). It also defines some tuples that match the RHS of the tgd 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 tuples are the result of direct user insertions; these are annotated with globally unique **provenance tokens**, which we represent as extra labels on the tuple nodes.

One can generate provenance expressions from the provenance graph, by traversing it recursively backwards along the arcs as in Example 3.5. It is easy to prove formally that these expressions are the same as the solutions of the system of equations.

Assigning Trust to Provenance

The provenance graph intuitively specifies the set of possible explanations for a given tuple. It is possible that a given peer not *trust* certain aspects of the CDSS: perhaps data from certain relations is of questionable value, or certain schema mappings may be judged to be of poor quality. We formalize this intuition with the notion of *trust conditions*. Section 1.1 presented several examples of trust conditions over provenance and data.

Within a CDSS, every tuple must either be a base insertion within a local contributions table, or a tuple derived from a combination of such base insertions. We assume that each peer specifies a set of tuples to initially trust; in effect, each tuple is annotated with **T**, representing that it is trusted, or **D**, indicating it is not. Additionally, each peer annotates each schema mapping m_i with **T** or **D**. As tuples are derived during update exchange, those that derive only from trusted tuples and using trusted mappings are marked as trusted. All other tuples are distrusted.

A finite provenance expression can simply be evaluated for "trustworthiness" as follows. For expressions of the form $m \cdot p_1 \cdot p_2$, we can simply map **T** to boolean **true** and **D** to **false**, then evaluate the provenance expression as a Boolean formula, where $\cdot$ represents conjunction and $+$ represents disjunction.

Trust composes along a mapping path as follows: if peer P_i has a schema mapping m_{ij} from its neighbor P_j, then P_i will receive updates via mapping m_{ij} if (1) they derive from updates trusted by P_j, and (2) the updates additionally satisfy the trust conditions imposed by P_i. In essence, the trust conditions specified by a given peer are combined (via conjunction) with the additional trust conditions specified by anyone mapping data from that peer. A peer delegates the ability to distrust tuples to those from whom it maps data.

Example 3.10. In Example 3.8 we calculated the provenances of some exchanged tuples. Suppose now that peer BioSQL trusts data contributed by GUS and itself, and hence assigns **T** to the provenance token p_3 and p_1, but does not trust uBio's tuple $(2,5)$ and so assigns **D** to p_2. Assuming that all mappings are assigned **T**, the provenance of $(3,2)$ evaluates as follows:

$$\mathbf{T} \cdot \mathbf{T} + \mathbf{T} \cdot \mathbf{T} \cdot \mathbf{D} = \mathbf{T}$$

therefore BioSQL should indeed have accepted $(3,2)$. □

In general the solutions of provenance equations may be infinite formal power series. Nonetheless Algorithm 3 evaluates them correctly with respect to Boolean trust assignments. We describe how the algorithm works in terms of the provenance graph.

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

Algorithm 3 Provenance-Trust-Eval(trust assignment α)

```
Provenance-Trust-Eval
for every mapping node labeled $m$ do
    if $\alpha(m) = \mathbf{Dm}$ then
        replace the node label with $\mathbf{D}$
    end if
    if $\alpha(m) = \mathbf{Tm}$ then
        erase the mapping node and connect incoming edges directly to outgoing edges
    end if
end for
for every tuple node labeled with provenance token $a$ do
    replace node label with $\alpha(a)$
end for
Set 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 $N = \mathbf{T}$ to OUTPUT
        end if
    end for
until no more changes are made to OUTPUT
for every remaining tuple node $N$ do
    add $N = \mathbf{D}$ to OUTPUT
end for
return OUTPUT
```

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 $\mathcal{M}'$:

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

and the definition of consistent state remains the same.

Enhanced trust models. Our implementation as described in this paper makes use of a Boolean trust model: all tuples are either trusted or distrusted. Our architecture in fact naturally expands to more complex trust models. In recent work [129] built atop Orchestra, we have shown that one can develop an extended model in which trust is modeled as a real number (where larger numbers mean greater *dis*trust) and one can use *feedback on query answers* to learn how much to distrust each source. This builds upon one of the provenance semirings mentioned in Chapter 2, the so-called *tropical semiring*. Here a conjunction in a schema mapping is converted to a *sum* of distrust levels of the tuples being joined, and a disjunction in a schema mapping returns the *minimum* level of distrust of the tuples being unioned. This requires ranked query processing and is beyond the scope of this paper.

3.3 Performing Update Exchange

We now discuss how to actually compute peer data instances in accordance with the model of the previous section. 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 Section 3.4). 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 the data it trusts, for each relation in the system, enabling future operations to be incremental. In ongoing work, we are considering more relaxed models in which portions of the computation may be distributed.

Computing Instances with Provenance

In the literature [45, 104, 76], 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 [118] used similar techniques to implement data exchange but did not consider updates or provenance.

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, 118]: Datalog queries are used to compute the certain answers [20] 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[3]. Such cases are generally represented by using existential variables in the target of a mapping tgd (e.g., (m_3) in Example 3.2). This requires *placeholder values* in the canonical universal solution. Chase-style procedures [45, 104, 76] 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 [118]. Each such function provides a unique 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, denoted Datalog. (Section 3.4 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 rules is the same as that of the *inverse rules* of [44]:

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

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 $\bar{f}(\bar{x})$.[4]

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 [76]). 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 3.11. Recall schema mapping (m_3) in Example 3.2. Its corresponding internal schema mapping:

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

□

---

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

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

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

Theorem 3.12. *If $\mathcal{M}$ is a weakly acyclic set of tgds, then $P_{\mathcal{M}}$ terminates for every edb instance $\mathcal{I}$ and its result, $P_{\mathcal{M}}(\mathcal{I})$, is a canonical universal solution.*

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

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

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

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

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

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 3.13. Since tuples in B (as shown in the graph of Example 3.5) can be derived through mappings m_1 and m_3, we can represent their provenance using two relations P_1 and P_3, using the mapping rules:

$$P_1(i,c,n) \text{ :- } G(i,c,n)$$
$$B(f(n),n) \text{ :- } P_1(i,c,n)$$
$$P_3(i,n_1,n_2) \text{ :- } B(i,n_1),U(n_1,n_2)$$
$$B(i,n_2) \text{ :- } P_3(i,n_1,n_2)$$

Recall from Example 3.8 that $B(\bot_2,5) = m_1 \cdot G(4,5,6) + m_3 \cdot B(\bot_2,6) \cdot U(6,5)$. The first part of this provenance expression is represented by the tuple $P_1(4,5,6)$ and the second by the tuple $P_3(\bot_2,6,5)$. ($P_1(4,5,6)$ can be obtained by assigning $i = 1, n = 2, c = 3$ in the rules above, and $P_3(\bot_2,6,5)$ by $i = \bot_2, n_1 = 6, n_2 = 5$.) 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 from Example 3.8, the contents of P_1 and P_3 are as follows:

P_1 :

1	2	3
4	5	6

P_3 :

$\bot_1$	2	3
$\bot_2$	5	6
$\bot_2$	6	5

□

A Primitive Operation on Provenance: 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

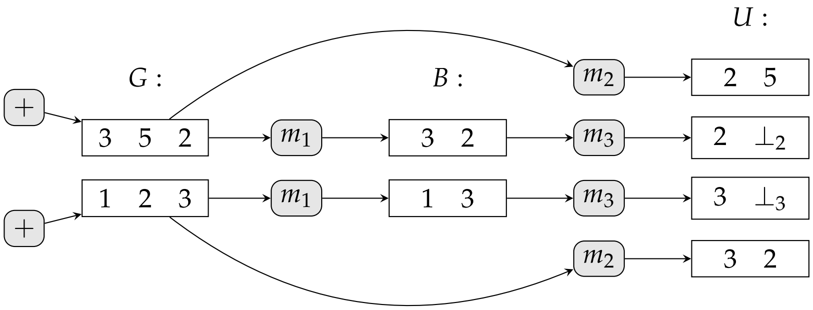


Figure 3.6: Provenance graph for derivability testing example.

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[5] of the original Datalog program over the filtered R' instances to identify the tuples that can indeed be re-derived.

Example 3.14. Consider a CDSS with the following mappings resembling those in Example 3.2:

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

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

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

and the provenance graph shown in Figure 3.6. The corresponding mapping rules are:

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

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

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

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

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

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

---

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

From these we compute the following inverse rules:

$$
\begin{array}{ll}
(r_1) & P_1'(i,c,n) \text{ :- } B'(i,n), P_1(i,c,n) \\
(r_2) & P_2'(i,c,n) \text{ :- } U'(n,c), P_2(i,c,n) \\
(r_3) & P_3'(i,n) \text{ :- } U'(n,c), P_3(i,n) \\
(r_4) & G'(i,c,n) \text{ :- } P_1'(i,c,n) \\
(r_5) & G'(i,c,n) \text{ :- } P_2'(i,c,n) \\
(r_6) & B'(i,n) \text{ :- } P_3'(i,c,n)
\end{array}
$$

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 $U' = \{(2,\bot_2),(3,2)\}$ 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 $U'(2,\bot_2)$ is still derivable but $U'(3,2)$ is not.

□

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 3.3.

Following [66] 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 [66]: 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.

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.

Algorithm 4

```
Propagate-Delete
for every $P_i$ do
    Set $R^0 \leftarrow R$
end for
Set $c \leftarrow 0$
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.
    end for
    {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 $R^-$
            end if
        end for
        Test each tuple in $R'$ for derivability from edbs; add it to $R^-$ if it fails
    end for
    Increment $c$
until no changes are made to any $R^-$
return the set of all $P_i^-$ and $R^-$
```

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 $\bar{R}^i$ update relations for the peer relations and a corresponding set P_i^- to apply to each provenance relation P_i. Algorithm 4 shows pseudocode for such an algorithm. First, the algorithm derives the deletions to apply to the provenance mapping relations; based on

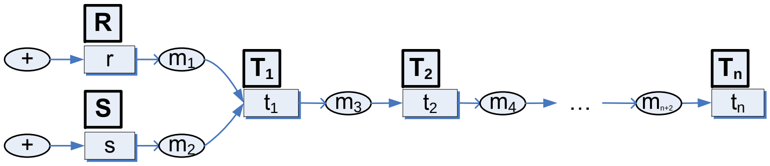


Figure 3.7: Example provenance graph showing relationships between tuples.

these, it computes a new version of the peer schema relations and their associated provenance relations (Lines 6–10). Next, it must determine whether a tuple in the instance is no longer derivable (Lines 13-22): such tuples must also be deleted. The algorithm first handles the case where the tuple is not directly derivable (Line 16), and then it performs a more extensive test for derivability from edbs (Line 22). The "existence test" is based on the derivation program described in Section 3.3, 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' 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 3.15. Revisiting Example 3.5 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 the 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 [66], 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 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 3.5 we validate this hypothesis. However, to provide an intuition of the differences, we give a brief example.

Example 3.16. Consider Figure 3.7, where nodes represent tuples and arrows represent the immediate-consequence relationship. The diagram corresponds to the mappings:

$$
\begin{array}{ll}
(m_1) & R \rightarrow T_1, \\
(m_2) & S \rightarrow T_1, \\
 & \\
(m_3) & T_1 \rightarrow T_2, \\
(m_4) & T_2 \rightarrow T_3, \\
 & \ldots \\
(m_{n+2}) & T_{n-1} \rightarrow T_n,
\end{array}
$$

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. ☐

3.4 Implementation

The ORCHESTRA system is a full implementation of a CDSS. Our initial prototype version in [131] focused on issues unrelated to update exchange (primarily those related to peer-to-peer communication and persistent storage), and hence for this paper we had to extend the system in many fundamental ways. Atop the existing catalog, communications, and persistence layers, we developed the following components in order to support incremental update exchange:

- *Wrappers* connect to DBMS data sources, obtain logs of their updates, and apply updates to them.
- *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 DBMS 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, 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 DBMS.

The majority of our technical contributions were in the update exchange engine, which comprises approximately one half of the 90,200 lines of code in ORCHESTRA. The engine parses tgds and transforms them into inverse rules, then delta rules, as described in Section 3.3. These rules are then precompiled into an executable form, resembling Datalog which we map down to a query engine through a pluggable back-end.

Provenance storage. We considered several alternative encodings for provenance relations, along the lines of 3.3. 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 find 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 [21] — 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.

Our main prototype implementation consists of a Java compilation and runtime system, which makes calls to an external query engine. The results in this paper target a SQL-based back-end. Using an off-the-shelf relational DBMS as a basis for the update exchange component is attractive for several reasons: (1) each peer is already running an DBMS, 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 DBMS. The first is that the Datalog 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 computations and updates, which must themselves be stratified and repeated in sequence. Finally, DBMSs do not directly support Skolem functions or labeled nulls.

Hence, we take a Datalog program and compile it to a combination of Java objects and SQL code. The control flow and logic for computing fixpoints is in Java,[6] using JDBC to execute SQL queries and updates on the DBMS (which is typically on the same machine). To achieve good performance, we make heavy use of prepared statements and keep the data entirely in DBMS tables. Queries return results into temporary tables, and the Java code only receives the number of operations performed, which it uses to detect fixpoint. Support for Skolem functions was implemented by adding user-defined functions and extending the schema with extra columns for each function and parameter.

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 3.5.

3.5 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 [66], 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.

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.

---

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

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 data⊢base [6], which has 25 attributes.

For each peer, it randomly partitions the 26 attributes from SWISS-PROT's schema into 2 relations, and adds a shared key attribute to preserve losslessness. Next, full mappings[7] 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 3.5) 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 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 each of the base tables at every peer). 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 split into 40000 tuples in peer relations, 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 a total of about 15000 base and derived tuples).

---

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

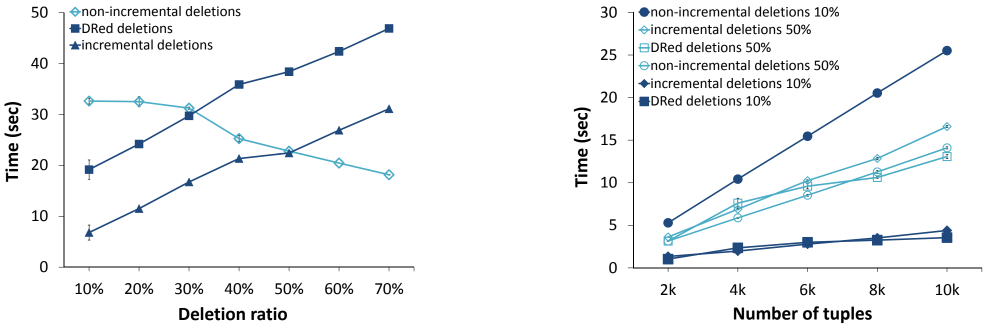
Experimental Methodology

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.

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 3.8(a) 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 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.



(a) Varying ratio of deleted tuples

(b) Varying base data set size

Figure 3.8: Relative performance of deletion propagation algorithms

Figure 3.8(b) 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.

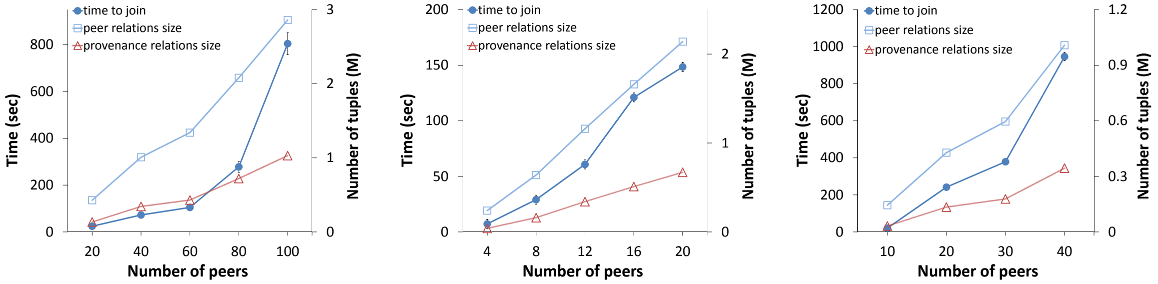
Cost and Overhead of Computing Instances

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 3.9(a) shows the results for 2,000 tuples of integer data; Figure 3.9(b) for a *larger* set of base instances with 10,000 tuples of integer data; and Figure 3.9(c) 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



(a) Integer dataset - 2k base size (b) Integer dataset - 10k base size (c) String dataset - 2k base size

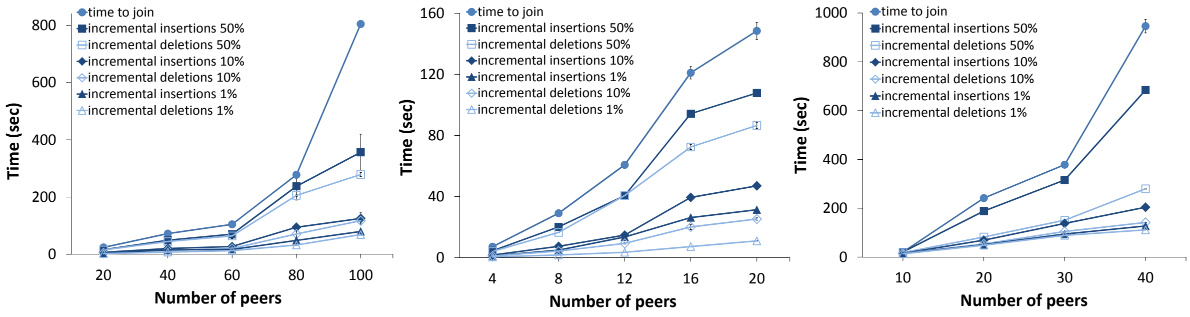
Figure 3.9: Time to join and relation instance sizes

tuples) is relatively small compared with the total instance size, averaging around 25% of the total number of tuples in the relations. In Section 3.5 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 peer relation tuple.

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.

Incremental Update Exchange vs. Number of Peers

Next, we look at how incremental insertion and deletion scale with the number of peers. Figure 3.10(a) 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 (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



(a) Integer dataset - 2k base size (b) Integer dataset - 10k base size (c) String dataset - 2k base size

Figure 3.10: Scalability of incremental update propagation algorithms

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 3.9(b)) 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.

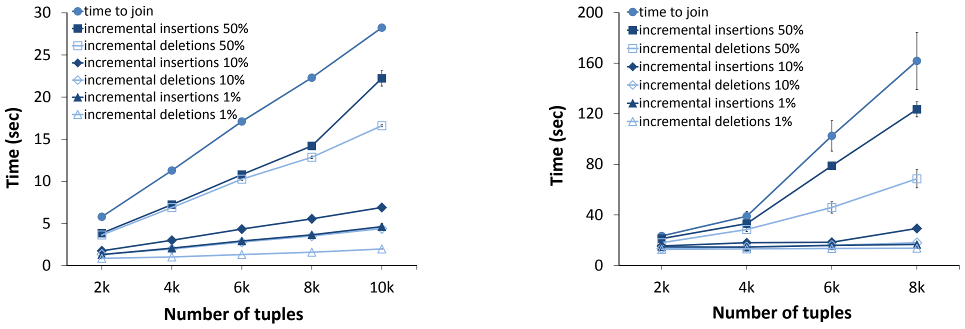
Our experiments to this point considered data with "small" attributes — integers or CLOBs. If we consider large attributes (long VARCHARs), the cost of insertions goes dramatically up (since insertions must contain *all attributes*), whereas the cost of deletions can remain similar (since deletions can be done purely with key attributes). We see this validated in Figures 3.10(c), where the gap between deletion and insertion cost widens.

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 3.11(a), and string data in Figure 3.11(b). 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.

Performance vs. Mappings

Our last 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,



(a) Integer dataset

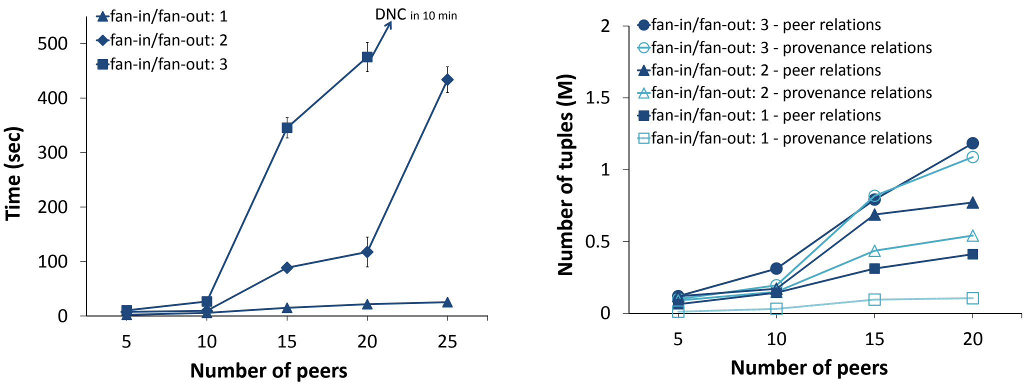
(b) String dataset

Figure 3.11: Scalability for increasing base data sizes

more alternative derivations and hence provenance 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 3.12(a) 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 3.12(b), 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.



(a) Effect on instance size and performance

(b) Effect on provenance storage size

Figure 3.12: Effects of mapping fan-in/fan-out on storage overhead and performance

Chapter 4

Optimizing Queries on Annotated Relations

The use of annotated relations in CDSS presents new challenges in query reformulation and optimization as queries that are semantically *equivalent* when posed over ordinary relations may become *inequivalent* when posed over annotated relations. Indeed, this phenomenon was already observed for the case of bag semantics [24, 81], where, e.g., adding a "redundant" self-join to a query actually changes the query's meaning. The need to compare query equivalence for different kinds of provenance annotations was also emphasized from early on in [17, 18] and reiterated in [13]. A central theme of this chapter is to compare different provenance-annotated semantics among themselves and with the standard set and bag semantics. The comparison is done w.r.t. containment[1] and equivalence of conjunctive queries (CQs) and unions of conjunctive queries (UCQs), leading to four different hierarchies among these semantics. Whether the steps in these hierarchies are strict or not is always informative and sometimes surprising.

We consider in this chapter five different kinds of provenance information that can be captured using semiring annotations. These range from the very simple *lineage* of [35], in which a tuple in the output is annotated with a set of tuple ids of all "contributing" source tuples, to the *why-provenance* of [17], in which output tuples are annotated with a *set of sets* of contributing source tuples, to the *provenance polynomials* $\mathbb{N}[X]$ used in ORCHESTRA. Recall that provenance polynomials are as "general" as any other commutative semiring, hence this is the most informative form of provenance annotations.

Another central theme of this chapter is to establish the complexity of containment and equivalence of CQs/UCQs for various semirings. Some of these are classical results [22, 123] and one other (for distributive lattices) has been established already in [62]. This chapter focuses primarily on the provenance semirings.

---

[1] We define inclusion of K-relations by the *natural order* present in the semirings of interest to us (see Section 4.1).

		$\mathbb{B}$	PosBool(X)	Lin(X)	Why(X)	Trio(X)	$\mathbb{B}[X]$	$\mathbb{N}[X]$	$\mathbb{N}$
CQs	cont	NP-C	NP-C	NP-C	NP-C	NP-C	NP-C	NP-C	? (Π_2^p-hard)
	equiv	NP-C	NP-C	NP-C	GI-C	GI-C	GI-C	GI-C	GI-C
UCQs	cont	NP-C	NP-C	NP-C	NP-C	in PSPACE	NP-C	in PSPACE	undec
	equiv	NP-C	NP-C	NP-C	NP-C	GI-C	NP-C	GI-C	GI-C

In the table, non-shaded boxes indicate contributions of this dissertation. NP is short for NP-complete. GI is short for GI-complete (i.e., complete for the class of problems polynomial time reducible to graph isomorphism).

Figure 4.1: Complexity of containment and equivalence

A priori, it is not clear that containment and equivalence for queries on relations with provenance annotations should even be decidable, as bag containment is known to be undecidable for UCQs [81], and $\mathbb{N}[X]$ seems related to bags. Nevertheless, we are able to show that containment is decidable for all the forms of provenance annotations we consider, for both CQs and UCQs (with the exception of containment of UCQs with Trio-style lineage, which we leave open). We also establish interesting connections with the same problems for bag semantics. In particular our contributions are:

- We show that the various forms of provenance annotations we consider are related by *surjective semiring homomorphisms*, which yields easy bounds on their relative behavior with respect to query containment.
- We show that for UCQs, $\mathbb{N}[X]$- containment implies K-containment for *any* semiring K, and for any *positive* K (a very large class that includes all the semirings we consider in this dissertation, see Section 4.1), K-containment implies containment under the usual set semantics.
- For the case of CQs without self-joins, we show that for any positive K, K-equivalence is the same as *isomorphism*, and thus its complexity is complete for the class GI of problems polynomial time reducible to *graph isomorphism*.[2]
- We show that containment of CQs and UCQs is decidable for lineage, why-provenance, $\mathbb{B}[X]$, and $\mathbb{N}[X]$ annotations. The decision procedures involve interesting variations on the concept of *containment mappings*, or (in the case of $\mathbb{N}[X]$-containment and Trio(X)-containment of UCQs) establishing a *small counterexample property* (see Sections 4.4 and 4.4).

---

[2] Graph isomorphism is known to be in NP, but is not known or believed to be either NP-complete or in PTIME, see [92].

We also identify the complexity in each case as NP-complete (with the exception of $\mathbb{N}[X]$-containment and $\mathsf{Trio}(X)$-containment of UCQs, where we give PSPACE upper bounds).

- We show that for why-provenance, $\mathbb{B}[X]$, and $\mathbb{N}[X]$, equivalence of CQs implies isomorphism, and the complexity is therefore somewhat lower than for containment (GI-complete). $\mathbb{N}[X]$-equivalence of UCQs is also shown to be the same as isomorphism and GI-complete. Lineage-equivalence of CQs and why-prov. and $\mathbb{B}[X]$-equivalence of UCQs are shown to remain NP-complete.
- We show that for CQs, why-prov. containment implies bag-containment, and bag containment implies lineage-containment. We also show that for UCQs $\mathbb{N}[X]$-equivalence is the same as bag equivalence hence providing a proof that the latter is the same as isomorphism and therefore GI-complete.

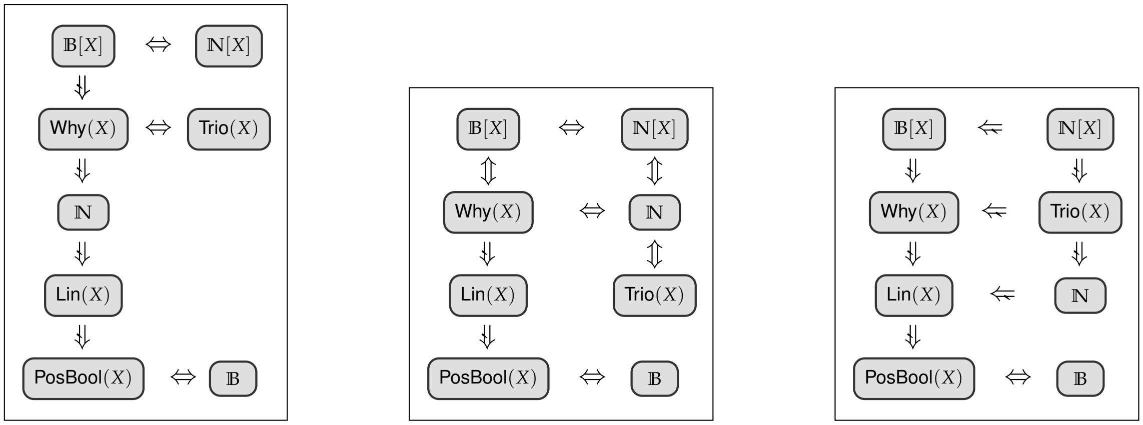
Figure 4.1 summarizes the complexity results mentioned above (for completeness we include previously known results in the shaded boxes). Figure 4.2 summarizes the logical relationships for containment/equivalence among the various semirings we consider.

The rest of this chapter is organized as follows. We define containment of queries on K-relations in terms of the *natural order* in Section 4.1 and discuss the connections with semiring homomorphisms. We review the background concepts of containment mappings and canonical databases in Section 4.2. We derive the bounds on containment based on surjective semiring homomorphisms in Section 4.3. We present the main results on containment and equivalence in Section 4.4. We finish with a discussion of containment and equivalence for Datalog queries in Section 4.5.

4.1 Preliminaries

We assume a Datalog-style representation of CQs and UCQs. We use single letters $P, Q, \ldots$ for CQs and overlined letters $\bar{P}, \bar{Q}, \ldots$ for UCQs. The semantics of CQs and UCQs on K-relations can be given by translating them (via a standard procedure) to $\mathcal{RA}^+$ queries, then applying Definition 2.2 (Chapter 2). However, it will be more convenient here to use an equivalent definition based on the notion of *valuations*. A valuation is a function $\nu : \mathsf{vars}(Q) \to \mathbb{D}$ extended to be the identity on constants. Valuations operate component-wise on tuples in the expected way. Let Q be a CQ

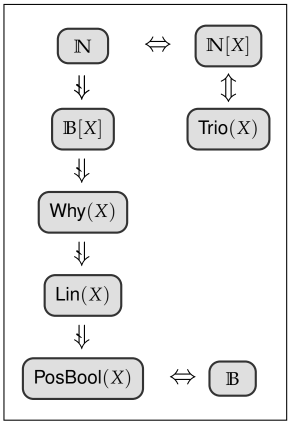
$$Q(\bar{u}) \quad \text{:-} \quad R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)$$



(a) CQ containment

(b) CQ equivalence

(c) UCQ containment



(d) UCQ equivalence

In the figures above, $K_1 \Rightarrow K_2$ indicates that K_1-containment (equivalence) implies K_2-containment (equivalence). A ticked arrow "$\not\Rightarrow$" indicates that the implication is strict.

Figure 4.2: Logical implications of containment and equivalence

and let I be a K-instance of the same schema. The *result of evaluating Q on I* is the K-relation defined

$$[\![Q]\!]^I(t) \overset{\text{def}}{=} \bigoplus_{\substack{\nu \text{ s.t.} \\ \nu(\bar{u})=t}} \bigotimes_{i=1}^{n} R_i{}^I(\nu(\bar{u}_i)) \tag{4.1}$$

and the sums and products are in K. A valuation ν which maps $\bar{u}$ to t such that the product in (4.1) is non-zero is called a *derivation* of t, and we say that it *justifies* the associated product. The meaning of (4.1) is unchanged if we assume the sum ranges only over derivations of t.

We extend this semantics to UCQs as follows. If $\bar{Q} = (Q_1, \ldots, Q_n)$ is a UCQ, then the *result of evaluating $\bar{Q}$* on a K-instance I is the K-relation defined

$$[\![\bar{Q}]\!]^I(t) \overset{\text{def}}{=} \bigoplus_{i=1}^{n} [\![Q_i]\!]^I(t) \tag{4.2}$$

We define containment of K-relations and queries over K-instances in terms of the *natural order*. Let $(K, \oplus, \otimes, 0, 1)$ be a semiring and define $a \leq b \overset{\text{def}}{\iff} \exists c\ a \oplus c = b$. When $\leq$ is a partial order we say that K is *naturally-ordered*. $\mathbb{B}, \mathbb{N}$, $\mathsf{PosBool}(X)$, and all of the semirings for provenance from Chapter 2 are naturally ordered. For $\mathsf{PosBool}(X)$ the natural order corresponds to logical entailment: $\varphi \leq \psi$ iff $\varphi \models \psi$. For $\mathbb{B}[X]$ we have $a \leq b$ iff every monomial in a also appears in b. For $\mathbb{N}[X]$ we have $a \leq b$ iff every monomial in a also appears in b with an equal or greater coefficient. Thus $2x^2y \leq 5x^2y + 2z$ but $x + 2y \not\leq 5x + 3y^2$. For lineage and why-provenance the natural order corresponds to set inclusion (n.b. for why-provenance, this is only set inclusion "at the outer level" – e.g., $\{\{x\}\} \leq \{\{x\}, \{y, z\}\}$ but $\{\{x\}, \{y, z\}\} \not\leq \{\{x, y\}, \{y, z\}\}$).

Definition 4.1. Let K be a naturally-ordered semiring and let R_1, R_2 be two K-relations. We define containment of R_1 in R_2 by

$$R_1 \leq_K R_2 \overset{\text{def}}{\iff} \forall t\ R_1(t) \leq R_2(t)$$

We define containment of queries P, Q with respect to K-relation semantics by

$$P \sqsubseteq_K Q \overset{\text{def}}{\iff} \text{for every } K\text{-instance } I,\ [\![P]\!]^I \leq_K [\![Q]\!]^I$$

When $K = \mathbb{B}$ ($K = \mathbb{N}$) we get the usual notion of query containment with respect to set (bag) semantics. For $\mathsf{PosBool}(X)$, we get the *structural containment* and *structural equivalence* of [126].[3]

4.2 Containment Mappings

In characterizing K-containment of CQs we will use variations on the notion of *containment mappings*. Let P, Q be conjunctive queries, and let h be a mapping $h : \mathsf{vars}(Q) \to \mathsf{vars}(P) \cup \mathsf{Const}(P)$ extended to be the identity on constants (we will typically use the shorthand $h : Q \to P$). We define h to operate component-wise on tuples, atoms, and CQs by replacing each occurrence of a variable x with $h(x)$. We say that $h : Q \to P$ is a *containment mapping* if $h(\mathsf{head}(Q)) = \mathsf{head}(P)$ and for every atom $R_i(\bar{u})$ in the body of Q the atom $R_i(h(\bar{u}))$ occurs in the body of P.

We will also make use of the notion of the *canonical database* (or *tableau*) for a query. This is the instance $\mathsf{can}(Q)$ obtained by viewing the body of a CQ Q as a database; i.e., $\mathsf{can}(Q) \models R(\bar{x})$ iff $R(\bar{x}) \in \mathsf{body}(Q)$. In doing this we blur the distinction between variables and domain values. When a query has duplicate atoms in the body, this does not result in duplicate tuples in the canonical database.

The classical result of [22] relates containment mappings, canonical databases, and containment of CQs under set semantics:

---

[3] There are reasonable alternatives to the natural order for incomplete databases, such as considering various orders on the sets of *possible worlds* they represent.

Theorem 4.2 ([22]). *For CQs P, Q the following are equivalent:*

1. $P \sqsubseteq_{\mathbb{B}} Q$
2. $[\![P]\!]^{\mathsf{can}(P)} \leq_{\mathbb{B}} [\![Q]\!]^{\mathsf{can}(P)}$
3. *there is a containment mapping $h : Q \to P$*

We will also exploit the device of canonical databases, but for the provenance models we will use various *abstractly-tagged* versions. The *abstractly-tagged version* $\mathsf{ab}_K(R)$ of a K-relation R is obtained by annotating each tuple in the support of R with its own tuple id from X. For $\mathbb{N}[X]$, $\mathbb{B}[X]$, and $\mathsf{Trio}(X)$ this is simply a fresh variable x from X. For lineage the variable is nested in a singleton set, $\{x\}$, and for why-provenance the variable is doubly-nested, $\{\{x\}\}$. We will use the shorthand $\mathsf{can}_K(Q)$ to mean $\mathsf{ab}_K(\mathsf{can}(Q))$. Abstractly-tagged instances will also play a role outside of the context of canonical databases (cf. Lemmas 4.20 and 4.31).

4.3 Bounds from Semiring Homomorphisms

Recall from Section 2.4 that the provenance models considered here are related by surjective semiring homomorphisms, as summarized there in Figure 2.7. It turns out that we can use this fact to yield some easy bounds on their "relative behavior" with respect to query containment (and therefore also equivalence). To make this precise, we write $K_1 \Rightarrow K_2$ to mean that for all UCQs $\bar{Q}_1, \bar{Q}_2$, if $\bar{Q}_1 \sqsubseteq_{K_1} \bar{Q}_2$ then $\bar{Q}_1 \sqsubseteq_{K_2} \bar{Q}_2$. Then we have the following:

Lemma 4.3. *For naturally-ordered semirings K_1, K_2, if there exists a surjective homomorphism $h : K_1 \to K_2$, then $K_1 \Rightarrow K_2$.*

Proof. Suppose that $h : K_1 \to K_2$ is a surjective semiring homomorphism and that $\bar{Q}_1 \sqsubseteq_{K_1} \bar{Q}_2$. Consider an arbitrary K_2-instance J. We want to show that $[\![\bar{Q}_1]\!]^{J)} \leq_{K_2} [\![\bar{Q}_2]\!]^J$. Since h is surjective, there exists a K_1-instance I such that $J = h(I)$. Since $\bar{Q}_1 \sqsubseteq_{K_1} \bar{Q}_2$ we have that $[\![\bar{Q}_1]\!]^I \leq_{K_1} [\![\bar{Q}_2]\!]^I$. By Proposition 4.37, this implies $h([\![\bar{Q}_1]\!]^I) \leq_{K_2} h([\![\bar{Q}_2]\!]^I)$. But by Proposition 2.5, $h([\![\bar{Q}_1]\!]^I) = [\![\bar{Q}_1]\!]^{h(I)} = [\![\bar{Q}_1]\!]^J$, and likewise, $h([\![\bar{Q}_2]\!]^I) = [\![\bar{Q}_2]\!]^{h(I)} = [\![\bar{Q}_2]\!]^J$. It follows that $[\![\bar{Q}_1]\!]^J \leq_{K_2} [\![\bar{Q}_2]\!]^J$. Since J was chosen arbitrarily, it follows that $\bar{Q}_1 \sqsubseteq_{K_2} \bar{Q}_2$, as required. $\square$

This immediately yields the following:

Theorem 4.4. *If there is a path downward from K_1 to K_2 in Figure 2.7, then $K_1 \Rightarrow K_2$.*

We shall see in Section 4.4 which of the implications are strict (as indicated by the ticked arrows "$\mathrel{\Rightarrow\!\!\!\!\!\!\nearrow}$" in Figure 4.2).

We also note that using similar reasoning, it is possible to establish bounds for containment/equivalence of UCQs for *arbitrary* semirings:

Theorem 4.5. *For all* K*,* $\mathbb{N}[X] \Rightarrow K$*. For all* positive K*,* $K \Rightarrow \mathbb{B}$*.*

Proof. $\mathbb{N}[X] \Rightarrow K$ follows from similar reasoning as in Proposition 4.3, but using the universality of the provenance polynomials rather than the existence of surjective semiring homomorphisms to establish the relationship. $K \Rightarrow \mathbb{B}$ follows immediately from Proposition 4.3 using the definition of positive semiring. □

The definition of positive semiring is given in the Appendix. This is a large class of semirings: $\mathbb{B}$, $\mathbb{N}$, $\mathsf{PosBool}(X)$, and all of the semirings for provenance we have considered so far in this dissertation are positive. For the special case of CQs containing no self-joins, the bounds of Theorem 4.5 collapse to a uniform condition for equivalence:

Corollary 4.6. *If CQs* P, Q *contain no self-joins, then for any positive* K*, we have* $P \equiv_K Q$ *iff* $P \cong Q$*.*

Therefore, for conjunctive queries without self-joins, every "$\Rrightarrow$" in Figure 4.2(b) becomes a "$\Leftrightarrow$".

4.4 Main Results

We are now ready to present our main results on containment and equivalence.

For all but the provenance polynomials, the decision procedures for containment of CQs (and the accompanying complexity results) extend easily to UCQs because of the following general fact which was first noted for the case of set semantics in [123]:

Proposition 4.7. *If a semiring* K *is idempotent (i.e., addition in* K *is idempotent), then for all UCQs* $\bar{P}, \bar{Q}$*, we have* $\bar{P} \sqsubseteq_K \bar{Q}$ *iff for every CQ* P *in* $\bar{P}$ *there is a CQ* Q *in* $\bar{Q}$ *such that* $P \sqsubseteq_K Q$*. As a consequence, checking* K*-containment of UCQs is polynomially equivalent to checking* K*-containment of CQs.*

The semirings for lineage, why-provenance, minimal witness provenance, and $\mathbb{B}[X]$-provenance are all idempotent. $\mathbb{N}[X]$ and $\mathsf{Trio}(X)$ are not idempotent, nor is the semiring of natural numbers used for bag semantics (and the failure of Proposition 4.7 for bag semantics was noted in [24]).

We also note that for idempotent semirings, containment and equivalence of UCQs are easily inter-reducible (and polynomially equivalent). This again generalizes a well-known fact for set semantics [123]:

Proposition 4.8. *For UCQs* $\bar{Q}_1, \bar{Q}_2$ *and idempotent* K *we have*

1. *$\bar{Q}_1 \sqsubseteq_K \bar{Q}_2$ iff $\bar{Q}_1 \cup \bar{Q}_2 \equiv_K \bar{Q}_2$*
2. *$\bar{Q}_1 \equiv_K \bar{Q}_2$ iff $\bar{Q}_1 \sqsubseteq_K \bar{Q}_2$ and $\bar{Q}_2 \sqsubseteq_K \bar{Q}_1$*

(The second item is just the definition of K-equivalence of UCQs.)

Lineage

Theorem 4.9. *For CQs P, Q the following are equivalent:*

1. *$P \sqsubseteq_{\mathsf{Lin}(X)} Q$*
2. *$[\![P]\!]^{\mathsf{can}_{\mathsf{Lin}(X)}(P)} \leq_{\mathsf{Lin}(X)} [\![Q]\!]^{\mathsf{can}_{\mathsf{Lin}(X)}(P)}$*
3. *for every atom $A(\bar{x}) \in \mathsf{body}(P)$ there is a containment mapping $h : Q \to P$ with $A(\bar{x})$ in the image of h*

Proof. (1) $\Rightarrow$ (2) is trivial. For (2) $\Rightarrow$ (3), let $t = \mathsf{head}(P)$ and let $I = \mathsf{can}_{\mathsf{Lin}(X)}(P)$. Observe that $[\![P]\!]^I(t) = \{x_1, \ldots, x_n\}$ where $\{x_1\}, \ldots, \{x_n\}$ are all the annotations occurring in I. Since $[\![P]\!]^I \leq_{\mathsf{Lin}(X)} [\![Q]\!]^I$, and $\{x_1, \ldots, x_n\}$ is the top element in the subsemiring of $\mathsf{Lin}(X)$ generated by $\{x_1\}, \ldots, \{x_n\}$, it follows that $[\![Q]\!]^I(t) = \{x_1, \ldots, x_n\}$ as well. Now, for each such x_i, there must be a justifying valuation $\nu : Q \to I$ mapping some atom $R(\bar{u})$ in the body of Q to a tuple $R(\bar{x})$ such that $R^I(\bar{x}) = \{x_i\}$. But this valuation ν may also be viewed as a containment mapping from Q to P with $R(\bar{x})$ in the image.

For (3) $\Rightarrow$ (1), consider an arbitrary $\mathsf{Lin}(X)$-instance I and output tuple t. We want to show that for all $x \in X$, if $x \in [\![P]\!]^I(t)$, then $x \in [\![Q]\!]^I(t)$. If $x \in [\![P]\!]^I(t)$, then there exists a derivation $\nu : \mathsf{vars}(P) \to \mathbb{D}$ of t in $[\![P]\!]^I$ such that $x \in R^I(\nu(\bar{u}))$ for some atom $R(\bar{u})$ in the body of P. By assumption, there exists a containment mapping $h : Q \to P$ such that $R(\bar{u}) = R(h(\bar{v}))$ for some atom $R(\bar{v})$ in the body of Q. But then $\nu \circ h$ is a derivation of t in $[\![Q]\!]^I$, and moreover, $x \in R((\nu \circ h)(\bar{v}))$. It follows that $x \in [\![Q]\!]^I(t)$, as required. ☐

It is easy to find examples of CQs P, Q such that there is a containment mapping $h : Q \to P$, but condition (3) above is not satisfied, e.g.:

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

There is no containment mapping $h : P \to Q$ with $R(x,z)$ in the image of h, so $P \not\sqsubseteq_{\mathsf{Lin}(X)} Q$. However, one can find containment mappings $h' : P \to Q$ and $h'' : Q \to P$ in both directions, so by Theorem 4.2, $P \equiv_{\mathbb{B}} Q$. This justifies the "$\Rightarrow$" between lineage and $\mathsf{PosBool}(X)/\mathbb{B}$ in Figures 4.2(a)–(d).

Note that the above example seems to contradict[4] Theorem 4.8 of [35] which claims that $P \equiv_{\mathsf{Lin}(X)} Q$ iff $P \equiv_{\mathbb{B}} Q$. In fact, the contradiction is explained by the fact that the definition of lineage given in that paper only makes sense for CQs without self-joins. We have already seen (Corollary 4.6) that for this class of queries, K-equivalence is the same as isomorphism, for any positive K (including the lineage semiring).

Also, condition (3) of Theorem 4.9 was identified previously in [24] as a necessary (but not sufficient) condition for bag containment of CQs. This justifies the "$\not\Rightarrow$" between $\mathbb{N}$ and lineage in Figure 4.2(a).

While the conditions for checking lineage containment and set containment of CQs or UCQs are different, the complexity turns out to be the same:

Corollary 4.10. *Checking $\mathsf{Lin}(X)$-containment or $\mathsf{Lin}(X)$-equivalence of CQs or UCQs is* NP*-complete.*

Proof. For CQs, membership in NP follows from part (3) of Theorem 4.9. To establish NP-hardness, we use a variation on the reduction from graph 3-coloring showing NP-hardness of ordinary set-containment of CQs [22]. As there, we use the fact that a directed graph $G = (V, E)$ is 3-colorable iff there exists a graph homomorphism $h : G \to C$, where C is the complete directed graph (without self-loops) on 3 vertices. Next, given a directed graph $G = (V, E)$, we encode G as the Boolean CQ Q_G whose body contains an atom $E(u, v)$ for each edge in the graph, and we similarly encode C as a Boolean CQ Q_C. To complete the reduction showing NP-hardness of ordinary set containment, it suffices to observe that there exists a graph homomorphism $h : G \to C$ iff there exists a containment mapping $h' : Q_G \to Q_C$. With $\mathsf{Lin}(X)$-containment, there is one more step: we assume w.l.o.g. that $\mathsf{vars}(Q_G)$ and $\mathsf{vars}(Q_C)$ are disjoint, and we let Q'_G be the Boolean CQ whose body is the conjunction of the bodies of Q_G and Q_C. It is clear that there exists a containment mapping $h : Q_G \to Q_C$ iff there exists a containment mapping $h' : Q'_G \to Q_C$. Moreover, any containment mapping $h' : Q'_G \to Q_C$ has in its image every atom in the body of Q_C, thus satisfying condition (3) of Theorem 4.9. It follows that checking $\mathsf{Lin}(X)$-containment of CQs is NP-hard. The extensions to UCQs and $\mathsf{Lin}(X)$-equivalence follow immediately using Proposition 4.7. □

Why-Provenance

To characterize $\mathsf{Why}(X)$-containment of CQs, we define the concept of *onto containment mappings*. A mapping $h : Q \to P$ is an *onto containment mapping* if it is a containment mapping and $\mathsf{body}(P) \leq_{\mathbb{N}} h(\mathsf{body}(Q))$.

---

[4] The example and this observation are due to James Cheney and Wang-Chiew Tan.

Theorem 4.11. *For CQs P, Q the following are equivalent:*

1. $P \sqsubseteq_{\mathsf{Why}(X)} Q$
2. $[\![P]\!]^{\mathsf{can}_{\mathsf{Why}(X)}(P)} \leq_{\mathsf{Why}(X)} [\![Q]\!]^{\mathsf{can}_{\mathsf{Why}(X)}(P)}$
3. *there is an onto containment mapping $h : Q \to P$*

Proof. (Sketch) Similar to the proof of Theorem 4.9. ☐

The existence of an onto containment mapping is a strictly stronger requirement than condition (3) of Theorem 4.9. For example, consider the queries

$$\begin{array}{rcl} P(x) & \text{:-} & R(x,y), R(x,x) \\ Q(u) & \text{:-} & R(u,v) \end{array}$$

There is no onto containment mapping from Q to P, hence $P \not\sqsubseteq_{\mathsf{Why}(X)} Q$, but one can find containment mappings satisfying condition (3) of Theorem 4.9 in both directions, so $P \equiv_{\mathsf{Lin}(X)} Q$. This justifies the "$\not\Rightarrow$" between why-prov. and lineage in Figure 4.2(a)-(d).

We note that the existence of onto containment mappings was identified in [24] as a sufficient (but not necessary) condition for bag containment of CQs. This justifies the "$\not\Rightarrow$" between $\mathsf{Why}(X)$ and $\mathbb{N}$ in Figure 4.2(a).

The existence of onto containment mappings in both directions leads to a simple characterization of $\mathsf{Why}(X)$-equivalence of CQs:

Theorem 4.12. *For CQs P, Q, $P \equiv_{\mathsf{Why}(X)} Q$ iff $P \cong Q$.*

Proof. Clearly isomorphism implies K-equivalence for any K, in particular for why provenance. In the other direction, if $P \equiv_{\mathsf{Why}(X)} Q$ by Theorem 4.11 there must exist onto containment mappings $h : Q \to P$ and $g : P \to Q$. But since both mappings are surjective they must also be injective. It follows that $P \cong Q$. ☐

It was shown in [24] that bag equivalence of CQs is also the same as isomorphism, hence the "$\Leftrightarrow$" between $\mathbb{N}$ and $\mathsf{Why}(X)$ in Figure 4.2(b). Also, note that there are $\mathsf{Lin}(X)$-equivalent CQs which are not isomorphic, for example:

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

Thus we have the "$\not\Rightarrow$" between $\mathsf{Why}(X)$ and $\mathsf{Lin}(X)$ in Figure 4.2(b).

For UCQs $\bar{P}, \bar{Q}$, we note that Theorem 4.12 does not imply that for UCQs $\bar{P} \equiv_{\mathsf{Why}(X)} \bar{Q}$ iff $\bar{P} \cong \bar{Q}$ (and indeed this is not the case).

Corollary 4.13. *Checking* $\mathsf{Why}(X)$*-containment for CQs or UCQs and* $\mathsf{Why}(X)$*-equivalence for UCQs is* NP*-complete. Checking* $\mathsf{Why}(X)$*-equivalence for CQs is* GI*-complete.*

Proof. (Sketch) Similar to the proof for Corollary 4.10. ☐

$\mathbb{B}[X]$-Provenance

To characterize $\mathbb{B}[X]$-containment of CQs we will need another variation on containment mappings, which we call *exact containment mappings*. A mapping $h : Q \to P$ is an *exact containment mapping* if $h(Q) = P$, i.e., $h(\mathsf{head}(Q)) = \mathsf{head}(P)$, and the bag of atoms $h(\mathsf{body}(Q))$ is identical to the bag of atoms $\mathsf{body}(P)$. Note that there is an exact containment mapping from Q to P iff P can be obtained from Q (up to isomorphism) by *unifying* variables in Q.

Theorem 4.14. *For CQs* P, Q*, the following are equivalent:*

1. $P \sqsubseteq_{\mathbb{B}[X]} Q$
2. $[\![P]\!]^{\mathsf{can}_{\mathbb{B}[X]}(P)} \leq_{\mathbb{B}[X]} [\![Q]\!]^{\mathsf{can}_{\mathbb{B}[X]}(P)}$
3. *there is an exact containment mapping* $h : Q \to P$

Proof. (1) ⇒ (2) is trivial, and (3) ⇒ (2) is straightforward to check. For (2) ⇒ (3), we assume for simplicity that $\mathsf{body}(P)$ contains no duplicate atoms (the argument can be extended to work without this assumption). Now suppose

$$[\![P]\!]^{\mathsf{can}_{\mathbb{B}[X]}(P)} \leq_{\mathbb{B}[X]} [\![Q]\!]^{\mathsf{can}_{\mathbb{B}[X]}(P)}.$$

Then in particular

$$[\![P]\!]^{\mathsf{can}_{\mathbb{B}[X]}(P)}(t) \leq [\![Q]\!]^{\mathsf{can}_{\mathbb{B}[X]}(P)}(t),$$

where t is the tuple of distinguished variables in $\mathsf{head}(P)$. Also, the polynomial $[\![P]\!]^{\mathsf{can}_{\mathbb{B}[X]}(P)}(t)$ contains as a term (i.e., with Boolean coefficient true) the product $x_1 \cdots x_n$ of all tuple ids in $\mathsf{can}_{\mathbb{B}[X]}(P)$. Since containment holds, the polynomial $[\![Q]\!]^{\mathsf{can}_{\mathbb{B}[X]}(P)}(t)$ must also contain the same term. Working backwards, there must be some valuation $\nu : \mathsf{vars}(Q) \to \mathbb{D}$ justifying the term. Moreover, in order to yield all variables $x_1, \ldots, x_n$ in the term, ν must map the atoms of $\mathsf{body}(Q)$ surjectively onto the tuples of $\mathsf{can}_{\mathbb{B}[X]}(P)$; and in order for all the exponents in the term to all equal one, the mapping of atoms to tuples must be injective. It follows that ν is an exact containment mapping from Q to P. ☐

Every exact containment mapping is also an onto containment mapping, but the converse is not true. For example, the mapping $h : Q \to P$ which sends w to u, z to v, and everything else to itself in

$$\begin{aligned} P(x,y) &\text{ :- } R(x,y), S(u,v) \\ Q(x,y) &\text{ :- } R(x,y), S(u,v), S(w,z) \end{aligned}$$

is an onto containment mapping, but not an exact containment mapping. This justifies the "$\not\Rightarrow$" between $\mathbb{B}[X]$ and $\mathsf{Why}(X)$ in Figure 4.2(a),(c). To justify the "$\not\Rightarrow$" between $\mathbb{B}[X]$ and $\mathsf{Why}(X)$ in Figure 4.2(d), consider P, Q as above and define the UCQs $\bar{P} = (P)$ and $\bar{Q} = (P, Q)$. Then $\bar{P} \equiv_{\mathsf{Why}(X)} \bar{Q}$ but $\bar{P} \not\equiv_{\mathbb{B}[X]} \bar{Q}$.

Like $\mathsf{Why}(X)$-equivalence, $\mathbb{B}[X]$-equivalence of CQs turns out to be the same as isomorphism:

Theorem 4.15. *For CQs P, Q, $P \equiv_{\mathbb{B}[X]} Q$ iff $P \cong Q$.*

This justifies the "$\Leftrightarrow$" between $\mathsf{Why}(X)$ and $\mathbb{B}[X]$ in Figure 4.2(b).

Checking for the existence of an exact containment mapping turns out to have the same complexity as checking for the existence of a containment mapping:

Corollary 4.16. *Checking $\mathbb{B}[X]$-containment of CQs or UCQs, or $\mathbb{B}[X]$-equivalence of UCQs, is* NP*-complete. Checking $\mathbb{B}[X]$-equivalence of CQs is* GI*-complete.*

Proof. (Sketch) GI-completeness of checking $\mathbb{B}[X]$-equivalence of CQs follows from Theorem 4.15 and the fact that we can view directed graphs as Boolean CQs (and vice versa), as in the proof of Corollary 4.10. The proofs of the other results are again variations on the proof of Corollary 4.10. □

Provenance Polynomials

We now prove the results for $\mathbb{N}[X]$-containment. For CQs, this turns out to be the same as for $\mathbb{B}[X]$-containment (thus justifying the "$\Leftrightarrow$" between $\mathbb{N}[X]$ and $\mathbb{B}[X]$ in Figure 4.2(a)):

Theorem 4.17. *For CQs P, Q the following are equivalent:*

1. $P \sqsubseteq_{\mathbb{N}[X]} Q$
2. $[\![P]\!]^{\mathit{can}_{\mathbb{N}[X]}(P)} \leq_{\mathbb{N}[X]} [\![Q]\!]^{\mathit{can}_{\mathbb{N}[X]}(P)}$
3. *there is an exact containment mapping $h : Q \to P$*

Proof. (1) $\Rightarrow$ (2) is trivial, and (2) $\Rightarrow$ (3) is exactly the same as in Theorem 4.14. For (3) $\Rightarrow$ (1) some additional care is required because addition in $\mathbb{N}[X]$ is not idempotent. We need to make sure that the coefficient of an arbitrary term in the polynomial $[\![Q]\!]^I(t)$, for some arbitrary $\mathbb{N}[X]$-instance I and tuple t, is at least as large as the coefficient of the same term in the polynomial $[\![P]\!]^I(t)$. To check this, it suffices to observe that for any valuations $\nu, \nu' : \mathsf{vars}(P) \to \mathbb{D}$ justifying a term in $[\![P]\!]^I(t)$, the valuations $\nu \circ h$ and $\nu' \circ h$ justify the same monomial in $[\![Q]\!]^I(t)$; and moreover if $\nu \neq \nu'$ then $\nu \circ h \neq \nu' \circ h$. Hence every justification for $[\![P]\!]^I(t)$ corresponds to a unique justification for $[\![Q]\!]^I(t)$. Since addition in $\mathbb{N}$ is monotone this implies the required inequality for the term coefficients. $\square$

Since $\mathbb{N}[X]$-containment of CQs holds exactly when $\mathbb{B}[X]$-containment holds, the same is true for $\mathbb{N}[X]$-equivalence:

Theorem 4.18. *Let P, Q be two CQs. Then $P \equiv_{\mathbb{N}[X]} Q$ iff $P \cong Q$.*

This justifies the "⇔" between $\mathbb{N}[X]$ and $\mathbb{B}[X]$ (and therefore also $\mathsf{Why}(X)$ and $\mathbb{N}$) in Figure 4.2(b).

Next we consider $\mathbb{N}[X]$-containment of UCQs. Using similar reasoning as in Theorem 4.17, it is not hard to see that a weaker version of the Sagiv-Yannakakis property for set-containment of UCQs [123] holds for $\mathbb{N}[X]$:

Lemma 4.19. *For UCQs $\bar{P}, \bar{Q}$, if $\bar{P} \sqsubseteq_{\mathbb{N}[X]} \bar{Q}$, then for every $P_i \in \bar{P}$ there exists $Q_j \in \bar{Q}$ s.t. $P_i \sqsubseteq_{\mathbb{N}[X]} Q_j$.*

Proof. (Sketch) Similar reasoning as in Theorem 4.17, using the abstractly-tagged canonical database for $\bar{P}$. $\square$

A natural question to ask is whether the lemma above can be strengthened to require that each $P_i \in \bar{P}$ correspond to a *unique* $Q_j \in \bar{Q}$; as this is clearly also a sufficient condition for containment, this would therefore yield a decision procedure for containment. However, the strengthened version is not true: consider the UCQs $\bar{P} = (P_1, P_2)$ and $\bar{Q} = (Q_1, Q_2)$ where

$$\begin{array}{lcllcl} P_1 & \text{:-} & R(x,y), R(x,x) & Q_1 & \text{:-} & R(x,y), R(u,u) \\ P_2 & \text{:-} & R(x,y), R(y,y) & Q_2 & \text{:-} & R(x,x), R(x,x) \end{array}$$

Both P_1 and P_2 are $\mathbb{N}[X]$-contained in Q_1, but neither is $\mathbb{N}[X]$-contained in Q_2; nevertheless, one can show that $\bar{P} \sqsubseteq_{\mathbb{N}[X]} \bar{Q}$.

Another natural idea is to check containment of $\bar{P}$ in $\bar{Q}$ by evaluating both queries on the canonical database for $\bar{P}$, in analogy with Theorem 4.17; unfortunately, one can easily find counterexamples showing that this procedure is unsound.

However, we are able to show that $\mathbb{N}[X]$-containment of UCQs is decidable in PSPACE by establishing a "small counterexample" property. In particular we show that if $\bar{P} \not\sqsubseteq_{\mathbb{N}[X]} \bar{Q}$, then $[\![\bar{P}]\!]^I \not\leq_{\mathbb{N}[X]} [\![\bar{Q}]\!]^I$ for some $\mathbb{N}[X]$-instance I whose size is bounded by the sizes of $\bar{P}$ and $\bar{Q}$.

We begin by bounding the sizes of the *annotations* of the required counterexample, by observing that if a counterexample exists, then its abstractly-tagged version is also a counterexample:

Lemma 4.20. *For any naturally-ordered semiring K, if $\bar{P}, \bar{Q} \in UCQ$ and $\bar{P}(I) \not\leq_K \bar{Q}(I)$ for some K-instance I, then $\bar{P}(\mathbf{ab}_{\mathbb{N}[X]}(I)) \not\leq_{\mathbb{N}[X]} \bar{Q}(\mathbf{ab}_{\mathbb{N}[X]}(I))$.*

Proof. Straightforward argument using Proposition 2.5, the universality property of $\mathbb{N}[X]$, and Proposition 4.37. ☐

Of course, the lemma holds in particular for $K = \mathbb{N}[X]$. Next, we show that the *number of tuples* in a counterexample can also be bounded:

Theorem 4.21. $\bar{P} \not\sqsubseteq_{\mathbb{N}[X]} \bar{Q}$ *iff* $[\![\bar{P}]\!]^I \not\leq_{\mathbb{N}[X]} [\![\bar{Q}]\!]^I$ *for some abstractly-tagged* $\mathbb{N}[X]$*-instance I containing at most k tuples, where k is the maximum number of atoms occurring in the body of a CQ in $\bar{Q}$.*

Proof. "$\Leftarrow$" is trivial. For "$\Rightarrow$", suppose $\bar{P} \not\sqsubseteq_{\mathbb{N}[X]} \bar{Q}$. Then for some $\mathbb{N}[X]$-instance I, we have $[\![\bar{P}]\!]^I \not\leq_{\mathbb{N}[X]} [\![\bar{Q}]\!]^I$. By Lemma 4.20, we may assume that I is abstractly-tagged. Choose some tuple t such that $[\![\bar{P}]\!]^I(t) \not\leq [\![\bar{Q}]\!]^I(t)$. There must be some monomial μ in the polynomial $[\![\bar{P}]\!]^I(t)$ with coefficient a such that the same monomial μ in the polynomial $[\![\bar{Q}]\!]^I(t)$ has coefficient b and $a > b$. Now let I' be the $\mathbb{N}[X]$-instance obtained from I by discarding any tuples whose annotations do not occur in μ. Note that I' has at most k tuples. Moreover, the coefficients for μ in the polynomials for $[\![\bar{P}]\!]^{I'}(t)$ and $[\![\bar{Q}]\!]^{I'}(t)$ are unchanged. Hence $[\![\bar{P}]\!]^{I'}(t) \not\leq [\![\bar{Q}]\!]^{I'}(t)$, and therefore, $\bar{P}(I) \not\leq_{\mathbb{N}[X]} \bar{Q}(I)$. This completes the proof. ☐

Theorem 4.21 leads immediately to a decision procedure for checking $\mathbb{N}[X]$-containment of UCQs: simply test $\bar{P}(I) \leq_{\mathbb{N}[X]} \bar{Q}(I)$ for all instances I containing at most k tuples over, say, the first nk values of the domain, where n is the maximum arity of a relation in the schema. (If $\bar{P}$ and $\bar{Q}$ contain constants, these must be included among the values considered as well.) Moreover, one can check that this can be done using only polynomial space:

Corollary 4.22. $\mathbb{N}[X]$*-containment of UCQs is decidable in* PSPACE*.*

The exact complexity of the problem remains open.

Finally, what about $\mathbb{N}[X]$-equivalence of UCQs? Theorem 4.21 tells us that it is decidable, but not much else. However, it turns out we can use Theorem 4.17 along with Lemma 4.19 to show that, as with CQs, $\mathbb{N}[X]$-equivalence of UCQs is the same as isomorphism.

Theorem 4.23. *For UCQs $\bar{P}, \bar{Q}$, we have $\bar{P} \equiv_{\mathbb{N}[X]} \bar{Q}$ iff $\bar{P} \cong \bar{Q}$.*

In the proof we make use of the following simple proposition which states that removing $\mathbb{N}[X]$-equivalent CQs from $\mathbb{N}[X]$-equivalent UCQs yields $\mathbb{N}[X]$-equivalent UCQs:

Proposition 4.24. *Let $\bar{P}, \bar{Q} \in UCQ$ and suppose $\bar{P} \equiv_{\mathbb{N}[X]} \bar{Q}$. Then for all $P \in \bar{P}, Q \in \bar{Q}$, if $P \cong Q$, then $\bar{P}' \equiv_{\mathbb{N}[X]} \bar{Q}'$, where $\bar{P}'$ ($\bar{Q}'$) is the UCQ obtained from $\bar{P}$ ($\bar{Q}$) by removing P (Q).*

Proof. (of Theorem 4.23) "$\Leftarrow$" is trivial. For "$\Rightarrow$" we argue by induction on $|\bar{P}| + |\bar{Q}|$. In the base case, $|\bar{P}| + |\bar{Q}| = 0$, and the queries are trivially $\mathbb{N}[X]$-equivalent and isomorphic. In the inductive case, consider $\bar{P} = (P_1, \ldots, P_n)$ and $\bar{Q} = (Q_1, \ldots, Q_m)$ with $n + m > 0$, and assume inductively that for all $\bar{P}', \bar{Q}'$ s.t. $|\bar{P}'| + |\bar{Q}'| < n + m$, if $\bar{P}' \equiv_{\mathbb{N}} \bar{Q}'$ then $\bar{P}' \cong \bar{Q}'$. If $\bar{P} \equiv_{\mathbb{N}} \bar{Q}$, then using Lemma 4.19, one can show that there exists some non-empty sequence $i_1, \ldots, i_{2k}$ such that $P_{i_1} \sqsubseteq_{\mathbb{N}[X]} Q_{i_2} \sqsubseteq_{\mathbb{N}[X]} \cdots \sqsubseteq_{\mathbb{N}[X]} P_{i_{2k-1}} \sqsubseteq_{\mathbb{N}[X]} Q_{i_{2k}}$ and $Q_{i_{2k}} \sqsubseteq_{\mathbb{N}[X]} P_{i_1}$. It follows that all the CQs in the sequence are $\mathbb{N}[X]$-equivalent, and hence (by Theorem 4.21) isomorphic. In particular, we have $P_{i_1} \cong Q_{i_1}$. Denote by $\bar{P}'$ the UCQ obtained by removing P_{i_1} from $\bar{P}$, and denote by $\bar{Q}'$ the UCQ obtained by removing Q_{i_1} from $\bar{Q}$. By Proposition 4.24, we have $\bar{P}' \equiv_{\mathbb{N}} \bar{Q}'$. Using the induction hypothesis, this implies $\bar{P}' \cong \bar{Q}'$. Since $\bar{P}' \cong \bar{Q}'$ and $P_{i_1} \cong Q_{i_1}$, it follows that $\bar{P} \cong \bar{Q}$ as required. □

Since $\mathbb{B}[X]$ is idempotent, but $\mathbb{N}[X]$ is not, it is easy to find examples of $\bar{P}, \bar{Q}$ where $\bar{P} \equiv_{\mathbb{B}[X]} \bar{Q}$ but $\bar{P} \not\equiv_{\mathbb{N}[X]} \bar{Q}$, e.g., $\bar{P} = (P)$ and $\bar{Q} = (P, P)$ where P is an arbitrary CQ. This justifies the "$\not\Leftarrow$" between $\mathbb{N}[X]$ and $\mathbb{B}[X]$ in Figure 4.2(c) and Figure 4.2(d).

Bag Semantics

In this section, we discuss some further connections between provenance annotations and bag semantics.

We note that by Theorem 4.5, $\mathbb{N}[X]$-containment of UCQs implies bag-containment. Since the former is decidable and the latter is not, it follows that there exist UCQs for which bag-containment holds but $\mathbb{N}[X]$-containment does not. This justifies the "$\not\Leftarrow$" between $\mathbb{N}[X]$ and $\mathbb{N}$ in Figure 4.2(d). Also, we can show that:

Proposition 4.25. *For containment of UCQs, we have*

1. $\mathbb{N} \not\Rightarrow \mathbb{B}[X]$ *and* $\mathbb{B}[X] \not\Rightarrow \mathbb{N}$
2. $\mathbb{N} \not\Rightarrow \mathsf{Why}(X)$ *and* $\mathsf{Why}(X) \not\Rightarrow \mathbb{N}$
3. $\mathbb{N} \Rightarrow \mathsf{Lin}(X)$

This justifies the "$\not\Leftarrow$" between $\mathbb{N}$ and lineage in Figure 4.2(c) and shows that $\mathbb{N}$ is incomparable there with $\mathbb{B}[X]$ and $\mathsf{Why}(X)$.

Next, the "$\Leftrightarrow$" between $\mathbb{N}$ and $\mathbb{N}[X]$ in Figure 4.2(d) follows from the following result:

Theorem 4.26. *For UCQs $\bar{P}, \bar{Q}$ we have $\bar{P} \equiv_{\mathbb{N}} \bar{Q}$ iff $\bar{P} \equiv_{\mathbb{N}[X]} \bar{Q}$*

Proof. $\mathbb{N}[X] \Rightarrow \mathbb{N}$ follows from Theorem 4.5. We prove $\mathbb{N} \Rightarrow \mathbb{N}[X]$ by contrapositive. Suppose $\bar{P} \not\equiv_{\mathbb{N}[X]} \bar{Q}$. Then for some $\mathbb{N}[X]$-instance I and tuple t, we have $\bar{P}(I)(t) = A$ and $\bar{Q}(I)(t) = B$ and $A \neq B$. Since A and B are non-identical polynomials, one can always find a valuation $\nu : X \to \mathbb{N}$ such that $\mathsf{Eval}_\nu(A) \neq \mathsf{Eval}_\nu(B)$. By Proposition 2.5, we have $\bar{P}(\nu(I)(t)) \neq \bar{Q}(\nu(I)(t))$. Since $\nu(I)$ is an $\mathbb{N}$-instance, it follows that $\bar{P} \not\equiv_{\mathbb{N}} \bar{Q}$. Therefore $\mathbb{N}[X] \not\Rightarrow \mathbb{N}$, as required. □

By Theorem 4.23 it follows from the above that bag equivalence of UCQs is also the same as isomorphism. Prior to receiving the reviews of the paper upon which this chapter is based, it seemed to us that the community considers the decidability of equivalence of UCQs under bag semantics an open problem. However, one of the referees pointed out (as related work on bag-set semantics) the papers [30, 31]. [30] stated the result that bag-set equivalence of UCQs (called *disjunctive queries* there) is the same as isomorphism, and added as an observation that this also holds for bag semantics. The outline of the proof of the bag-set semantics result is provided in [27] and although bag semantics is not discussed further there we have observed that, in fact, results on bag-set semantics do correspond to results on bag semantics via the following *transfer lemma*:

Lemma 4.27. *There exists a mapping $\varphi : CQ \to CQ$ (which we extend to UCQs by applying it componentwise on CQs), a mapping f from bag instances to set instances, and a mapping g from set instances to bag instances, such that for any UCQ $\bar{Q}$, bag instance I, and set instance J, we have*

1. $[\![\bar{Q}]\!]^I = [\![\varphi_{\bar{Q}}]\!]^{f(I)}$
2. $[\![\varphi_{\bar{Q}}]\!]^J = [\![\bar{P}]\!]^{g(J)}$

where $\varphi_{\bar{Q}}$ is the result of φ applied to $\bar{Q}$.

Proof. We first define $\varphi : \mathrm{CQ} \to \mathrm{CQ}$, as follows. Let Q be a CQ over schema Σ, and let Σ' be the schema obtained from Σ by replacing each n-ary relational predicate R with an $n+1$-ary relational predicate R'. Then φ maps Q to the CQ φ_Q over schema Σ' obtained from Q by replacing each join atom $R(x_1, \ldots, x_k)$ with an atom $R'(x_1, \ldots, x_k, u)$ where u is a fresh variable. For example, if Q is the CQ

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

then φ_Q is the CQ

$$\varphi_Q(x,y) \quad \text{:-} \quad R(x,y,u), R(y,z,v)$$

We extend φ to map a UCQ $\bar{Q}$ to a UCQ $\varphi_{\bar{Q}}$ by applying it componentwise on CQs.

Next we define an encoding f of bag-instances over schema Σ to bag-set instances over schema Σ', and an encoding g of bag-set instances over Σ' to bag-instances over Σ, as follows:

- f maps a tuple t in relation R with multiplicity k to k distinct tuples $t'_1, \ldots, t'_k$ in R' each obtained from t by appending a fresh constant in the last column
- g assigns to tuple t in R the multiplicity k where k is the number of tuples t' in R' such that t and t' agree on all columns of t

It is straightforward to verify that φ, f, and g satisfy the conditions required by the Lemma. $\square$

Lemma 4.27 implies that bag-containment (bag-equivalence) of CQs/UCQs is polynomial time reducible to bag-set containment (bag-set equivalence). Moreover, for UCQs $\bar{P}, \bar{Q}$, the transformation φ defined in the proof of Lemma 4.27 satisfies $\bar{P} \cong \bar{Q}$ iff $\varphi(\bar{P}) \cong \varphi(\bar{Q})$. Thus Lemma 4.27 transfers to bag semantics the isomorphism results for equivalence under bag-set semantics.

Trio

For CQs, $\mathsf{Trio}(X)$-containment turns out to coincide with $\mathsf{Why}(X)$-containment:

Theorem 4.28. *For CQs P, Q we have $P \sqsubseteq_{\mathsf{Trio}(X)} Q$ iff $P \sqsubseteq_{\mathsf{Why}(X)} Q$.*

Therefore, Theorem 4.11 (Theorem 4.12) applies to $\mathsf{Trio}(X)$-containment (equivalence) as well, and we have a "$\Leftrightarrow$" between $\mathsf{Trio}(X)$ and $\mathsf{Why}(X)$ in Figure 4.2(a) and Figure 4.2(b).

To establish the decidability of $\mathsf{Trio}(X)$-equivalence of UCQs, we note that $\mathsf{Trio}(X)$ contains an embedded copy of $\mathbb{N}$, hence $\mathsf{Trio}(X) \Rightarrow \mathbb{N}$ for UCQs. Combined with Theorem 4.26 this implies:

Theorem 4.29. *For UCQs $\bar{P}, \bar{Q}$ we have $\bar{P} \equiv_{\mathsf{Trio}(X)} \bar{Q}$ iff $\bar{P} \cong \bar{Q}$.*

This justifies the "$\Leftrightarrow$" between $\mathsf{Trio}(X)$ and $\mathbb{N}$ in Figure 4.2(d).

To establish decidability in PSPACE of $\mathsf{Trio}(X)$-containment of UCQs, we follow a similar argument as used for $\mathbb{N}[X]$ in Section 4.4. However, this time the argument is complicated by the fact that unlike $\mathbb{N}[X]$, $\mathsf{Trio}(X)$ cannot easily "simulate its own computations" by factoring them through calculations involving abstractly-tagged databases. As a simple example, consider the CQs

$$Q_1(x,y) \text{ :- } R(x,z), R(z,y) \qquad Q_2(x,y) \text{ :- } R(x,y)$$

and consider the following $\mathsf{Trio}(X)$-relation R and its abstractly-tagged version:

$$R \stackrel{\text{def}}{=} \begin{array}{|cc|c|} \hline \mathrm{a} & \mathrm{a} & 2 \\ \mathrm{b} & \mathrm{c} & pq \\ \hline \end{array} \qquad \mathsf{ab}_{\mathsf{Trio}(X)}(R) = \begin{array}{|cc|c|} \hline \mathrm{a} & \mathrm{a} & r \\ \mathrm{b} & \mathrm{c} & s \\ \hline \end{array}$$

Note that $[\![Q_1]\!]^R(\mathrm{a},\mathrm{a}) = 4$ while $[\![Q_2]\!]^R(\mathrm{a},\mathrm{a}) = 2$. On the other hand, $[\![Q_1]\!]^{\mathsf{ab}_{\mathsf{Trio}(X)}(R)}(\mathrm{a},\mathrm{a}) = [\![Q_2]\!]^{\mathsf{ab}_{\mathsf{Trio}(X)}(R)}(\mathrm{a},\mathrm{a}) = p$ (recall that $\mathsf{Trio}(X)$ "drops exponents" in its computations). Hence we do not have enough information to recover the answers over the original table.

To overcome this limitation, we introduce variation of abstractly-tagged instances called *k-abstractly tagged instances*. The tags in these instances are *sums* of k fresh variables from X. As we shall see, these turn out to suffice for $\mathsf{Trio}(X)$ to "simulate its own computations" for UCQs of degree $\leq k$.

To make this precise, we order the variables of $X = \{x_1, x_2, \dots\}$, and we define the set $S_k \subseteq \mathsf{Trio}(X)$ of sums of k fresh variables from X:

$$S_k \stackrel{\text{def}}{=} \{(x_1 + \cdots + x_k), (x_{k+1} + \cdots + x_{2k}), \dots\}$$

The *k-abstractly tagged version* $\mathsf{ab}_k(I)$ of a $\mathsf{Trio}(X)$-instance I is the $\mathsf{Trio}(X)$-instance obtained by replacing each tuple's annotation in I with a fresh element of S_k. For example, considering again the $\mathsf{Trio}(X)$-relation R from earlier, we have

$$\mathsf{ab}_k(R) = \begin{array}{|cc|c|} \hline \mathrm{a} & \mathrm{a} & r+s \\ \mathrm{b} & \mathrm{c} & u+v \\ \hline \end{array}$$

Now, we define $\mathsf{Trio}_k(X) \subseteq \mathsf{Trio}(X)$ to be the set of annotations from $\mathsf{Trio}(X)$ obtained via calculations of degree $\leq k$ involving elements of S_k:

$$\mathsf{Trio}_k(X) \stackrel{\text{def}}{=} \{\mathsf{Eval}_\nu(a) : a \in \mathbb{N}[X], \nu : X \to S_k \text{ s.t. } a \text{ has degree} \leq k \text{ and } \nu \text{ is injective } \}$$

Proposition 4.30. *Suppose valuation $\nu : X \to S_k$ is injective. Then for any $a, b \in \mathbb{N}[X]$ of degree $\leq k$, $a = b$ iff $\mathsf{Eval}_\nu(a) = \mathsf{Eval}_\nu(b)$. As a consequence, any element $c \in \mathsf{Trio}_k(X)$ decomposes uniquely into a sum of products of elements of S_k.*

To illustrate, consider the previous example, and note that $[\![Q_1]\!]^{\mathsf{ab}_k(R)}(\mathrm{a}, \mathrm{a}) = r + 2rs + s$ which decomposes uniquely to the expression $(r + s)(r + s)$ involving only tags from $\mathsf{ab}_k(R)$. On the other hand, $[\![Q_2]\!]^{\mathsf{ab}_k(R)}(\mathrm{a}, \mathrm{a}) = r + s$ and now we have enough information to recover the results of $[\![Q_1]\!]^R$ and $[\![Q_2]\!]^R$ (by replacing $r + s$ with 2 in the calculations).

We are now ready to state the analog of Lemma 4.20 for $\mathsf{Trio}(X)$:

Lemma 4.31. *Suppose $\bar{P}, \bar{Q} \in UCQ$ have degree $\leq k$, and suppose I is a $\mathsf{Trio}(X)$-instance such that $[\![\bar{P}]\!]^I \not\leq_{\mathsf{Trio}(X)} [\![\bar{Q}]\!]^I$. Then $[\![\bar{P}]\!]^{\mathsf{ab}_k(I)} \not\leq_{\mathsf{Trio}(X)} [\![\bar{Q}]\!]^{\mathsf{ab}_k(I)}$.*

Proof. (Sketch) Let $I' = \mathsf{ab}_k(I)$ and let $\nu : S_k \to \mathsf{Trio}(X)$ be the valuation which records how the annotations from I were replaced by elements of S_k. (Thus $\nu(I') = I$.) We claim that ν extends to a mapping $h_\nu : \mathsf{Trio}_k(X) \to \mathsf{Trio}(X)$ that behaves like a semiring homomorphism, so long as the computations stay within $\mathsf{Trio}_k(X)$:

1. $h_\nu(0) = 0$ and $h_\nu(1) = 1$

2. for all $a, b \in \mathsf{Trio}_k(X)$, $h_v(a + b) = h_v(a) + h_v(b)$

3. for all $a, b \in \mathsf{Trio}_k(X)$, if $a \cdot b \in \mathsf{Trio}_k(X)$ then $h_v(a \cdot b) = h_v(a) \cdot h_v(b)$

Using Lemma 4.31, we can show that h_v exists and is uniquely determined by the above properties. Since $\bar{P}, \bar{Q}$ have degree $\leq k$, it is not hard to see that all annotations in $[\![\bar{P}]\!]^{I'}$ and $[\![\bar{Q}]\!]^{I'}$ are in $\mathsf{Trio}_k(X)$. Also, it is not hard to show that h_v enjoys two relevant properties of proper semiring homomorphisms with respect to query evaluation:

- h_v is compatible with the natural order: $\forall a, b \in \mathsf{Trio}_k(X)$, if $a \leq_{\mathsf{Trio}(X)} b$ then $h_v(a) \leq_{\mathsf{Trio}(X)} h_v(b)$
- Query evaluation commutes with h_v: $h_v([\![\bar{P}]\!]^{I'}) = [\![\bar{P}]\!]^{h_v(I')} = [\![\bar{P}]\!]^{I}$ and $h_v([\![\bar{Q}]\!]^{I'}) = [\![\bar{Q}]\!]^{h(I')} = [\![\bar{Q}]\!]^{I}$

As a consequence, we have that $[\![\bar{P}]\!]^{I'} \leq_{\mathsf{Trio}(X)} [\![\bar{Q}]\!]^{I'}$ implies $[\![\bar{P}]\!]^{I} \leq_{\mathsf{Trio}(X)} [\![\bar{Q}]\!]^{I}$. $\square$

Thus, we have established a bound on the size of annotations of the counterexamples that need to be considered. The last step is to additionally bound the number of tuples and establish our "small counterexample property":

Theorem 4.32. *For UCQs $\bar{P}, \bar{Q}$ of degree $\leq k$, if $\bar{P} \not\sqsubseteq_{\mathsf{Trio}(X)} \bar{Q}$, then $[\![\bar{P}]\!]^{I} \not\leq_{\mathsf{Trio}(X)} [\![\bar{Q}]\!]^{I}$ for a k-abstractly tagged $\mathsf{Trio}(X)$-instance I containing at most ℓ tuples, where ℓ is the maximum number of atoms occurring in a CQ in $\bar{Q}$.*

Proof. (Sketch) Very similar to the proof for Theorem 4.21. $\square$

Corollary 4.33. *$\mathsf{Trio}(X)$-containment of UCQs is decidable in* PTIME.

Finally, we note that one can find examples of UCQs showing that $\mathbb{N}[X] \not\Rightarrow \mathsf{Trio}(X)$ and $\mathsf{Trio}(X) \not\Rightarrow \mathbb{N}$, as indicated in Figure 4.2(d).

4.5 Datalog

We have seen that for UCQs, containment under various provenance semantics turns out to be decidable, even for models such as $\mathbb{N}[X]$ and $\mathsf{Trio}(X)$ which are closely related to bag semantics. Is it possible that these decidability results extend to Datalog? We leave this question open for most of the provenance models considered in this dissertation, but give two results in this section suggesting that the answer is likely negative, even for the easier question of the two, equivalence.

Under set semantics, Shmueli [128] has shown that equivalence (and therefore also containment) of Datalog queries is undecidable. This result immediately extends to K-containment and K-equivalence of Datalog programs for K a distributive lattice by dint of the following observation:

Theorem 4.34. *For any distributive lattice K and Datalog programs Q_1, Q_2, we have $Q_1 \sqsubseteq_K Q_2$ iff $Q_1 \sqsubseteq_{\mathbb{B}} Q_2$.*

This generalizes an earlier result [56] (later rediscovered in [62]) from UCQs to Datalog programs. Since $\mathsf{PosBool}(X)$ is a distributive lattice, this settles the question negatively for that provenance model.

For the other provenance models, it is possible that undecidability of Datalog-equivalence may be established by reduction from the same problem for set semantics, along the lines of the following:

Theorem 4.35. *$\mathbb{N}^\infty$-equivalence of Datalog programs is undecidable.*

Proof. By reduction from the same problem for set semantics. If Q is a Datalog program, let Q' be the Datalog obtained from Q by renaming the output predicate to Q', and adding one more rule:

$$Q'(x_1, \ldots, x_n) \text{ :- } Q'(x_1, \ldots, x_n)$$

where n is the arity of the output predicate. This rule effectively "amplifies" the multiplicity of any output tuple in the query result to ∞. Now given Datalog programs Q_1, Q_2, let Q_1' and Q_2' be the corresponding Datalog programs derived from them as above. We claim that $Q_1 \equiv_{\mathbb{B}} Q_2$ iff $Q_1 \equiv_{\mathbb{N}^\infty} Q_2$. To see this, consider an arbitrary input $\mathbb{N}^\infty$-instance I, and let J be the set instance corresponding to its support. For any output tuple t, observe that

$$\begin{aligned} [\![Q_i]\!]^J(t) = 0 &\quad \Longleftrightarrow \quad [\![Q_i']\!]^I(t) = 0 \\ [\![Q_i]\!]^J(t) = 1 &\quad \Longleftrightarrow \quad [\![Q_i']\!]^I(t) = \infty \end{aligned}$$

Since any set instance J corresponds to the support of some $\mathbb{N}^\infty$-instance I, this establishes the claim. □

Appendix

Definition 4.36 (Semiring homomorphism)**.** Let K_1, K_2 be semirings. A mapping $h : K_1 \to K_2$ is called a *semiring homomorphism* if $h(0) = 0$, $h(1) = 1$, and for all $a, b \in K_1$, we have $h(a + b) = h(a) + h(b)$ and $h(a \cdot b) = h(a) \cdot h(b)$.

Proposition 4.37. *Let K_1, K_2 be naturally-ordered commutative semirings. If $h : K_1 \to K_2$ is a semiring homomorphism then for all $a, b \in K_1$, $a \leq_{K_1} b \implies h(a) \leq_{K_2} h(b)$. If h is also surjective, then for all $a, b \in K_1$, $a \leq_{K_1} b \iff h(a) \leq_{K_2} h(b)$.*

Proof. Straightforward calculation. ☐

Given a semiring K define $\dagger : K \to \mathbb{B}$ as follows:

$$\begin{aligned} \dagger(0) &\stackrel{\text{def}}{=} \textsf{false} \\ \dagger(a) &\stackrel{\text{def}}{=} \textsf{true} \text{ when } a \neq 0 \end{aligned}$$

Proposition 4.38. *The following are equivalent:*

1. $\dagger$ *is a semiring homomorphism*
2. *K satisfies*
   a) $0 \neq 1$
   b) $a + b = 0$ *implies* $a = 0$ *or* $b = 0$
   c) $ab = 0$ *implies* $a = 0$ *or* $b = 0$

A semiring K is called *positive* if it satisfies either of the (equivalent) statements in Proposition 4.38.

Definition 4.39 (Congruence relation)**.** If K is a semiring and $\approx$ is an equivalence relation on K, then we say that $\approx$ is a *congruence relation* on K if $a \approx a'$ and $b \approx b'$ implies $a + b \approx a' + b'$ and $a \cdot b \approx a' \cdot b'$.

Definition 4.40 (Quotient semiring)**.** Let K be a semiring and let $\approx$ be a congruence relation on K. If $a \in K$ then denote the equivalence class of a in $\approx$ by $a/\approx$. Then the *quotient of K by $\approx$* is the semiring whose domain is the set $K/\approx$ of equivalence classes of $\approx$, $0 \stackrel{\text{def}}{=} 0_K/\approx$, $1 \stackrel{\text{def}}{=} 1_K/\approx$, $(a/\approx) + (b/\approx) = (a + b)/\approx$, and $(a/\approx) \cdot (b/\approx) \stackrel{\text{def}}{=} (a \cdot b)/\approx$.

Chapter 5

Ring-Annotated Relations and Differences

In this chapter we study the reformulation (rewriting) of relational queries that contain the *difference* operator. Our goal for query reformulation is to optimize by reusing existing information, such as materialized views. Since the objective is optimization, we focus on *exact* reformulation, which finds only equivalent rewritings of the query. [1]

Query reformulation using views is well understood for *positive* fragments of relational languages, such as conjunctive queries (CQs) or unions of CQs (UCQs), under both set and bag semantics (see, e.g., [23, 101]). As we shall discuss in more detail in the preamble to Section 5.3 in both cases (bag and set semantics), complete procedures for finding UCQ rewritings using UCQ views exist, using finite search spaces. Also, in both cases UCQ equivalence is decidable. In fact, in the same discussion we argue that whenever a (reasonable) finite search space procedure exists, query equivalence must also be decidable.

It follows that the initial outlook on doing reformulations involving the difference operator is glum because even without views the equivalence of relational algebra ($\mathcal{RA}$) queries is undecidable, for both set and bag semantics. [2] Hence, we cannot hope for the approaches to UCQ reformulation under bag or set semantics to extend to the entire $\mathcal{RA}$.

However, here are at least three reasons not to give up easily. With reformulation of queries that include difference:

- *Optimization using materialized views* could be done over a broader space of plans (even if the original query and view were just CQs/UCQs!). Rewritings could *subtract* one view from a larger view in order to return a query answer.
- *View adaptation* [65], the act of updating a materialized view instance when the view defini-

---

[1] In data integration, one is also interested in *maximally contained rewritings*, see e.g., [71].

[2] The latter follows, e.g., from the undecidability of bag-containment of unions of conjunctive queries (UCQs) [81], since for UCQs Q, Q' we have Q is contained in Q' iff $Q - Q'$ is equivalent to the empty answer query.

tion has changed, could be seen as a reformulation using views. Here, the updated view can be recomputed based on the old contents of the view, by adding and/or subtracting queries over the base data and possibly other views.

- *Incremental view maintenance* [66] could be seen as a reformulation using views, since insertions and deletions could be treated as unions and differences.
- *Mapping evolution* and *update exchange* in CDSS could be attacked using reformulation-based techniques, as the two problems are closely related to view adaptation and incremental view maintenance.

These are all highly relevant problems in databases.

Thus we ask the following natural question: is there a slightly less expressive class of queries than $\mathcal{RA}$, — still including difference, and hence still providing the benefits cited above — for which reformulation can be handled effectively. In this chapter we do this via an excursion through a non-standard semantics that is of interest in its own right: what we term *$\mathbb{Z}$-relations*. These are relations whose tuples are annotated with integers (positive or negative) and the positive $\mathcal{RA}$ operators are defined on them according to the semiring-annotated semantics introduced in Chapter 2. In addition, difference has an obvious, natural definition on $\mathbb{Z}$-relations.

$\mathbb{Z}$-relations are a natural representation for the *updates* to source relations (collections of tuple insertions and deletions, a.k.a. *deltas*) which must be propagated in incremental view maintenance applications. Indeed, both data and updates can be uniformly represented using $\mathbb{Z}$-relations, and "application" of a delta to a relation corresponds to simply computing a union. We discuss this further in Section 5.1.

It turns out that reformulation of $\mathcal{RA}$ queries using $\mathcal{RA}$ views can be solved effectively with respect to the $\mathbb{Z}$-semantics since here equivalence of $\mathcal{RA}$ queries with respect to a set of $\mathcal{RA}$ views is *decidable*. Moreover, we obtain practically useful results about the class of $\mathcal{RA}$ queries for which the reformulation with respect to $\mathbb{Z}$-semantics remains valid with respect to bag semantics. Although membership in this class of queries is necessarily undecidable, there are many useful cases with simple sufficient conditions for membership, in the three classes of applications outlined above.

For applications such as CDSS which require rich *provenance* information, we propose the use of another kind of ring-annotated relation, *$\mathbb{N}[X]$-relations*, for uniformly representing data and updates. These are like the provenance polynomials $\mathbb{N}[X]$ of Chapter 2, but with integer coefficients.

The main contributions of the chapter are:

- We show that under $\mathbb{Z}$-semantics every $\mathcal{RA}$ query is equivalent to the difference of two queries in $\mathcal{RA}^+$. The latter are selection/projection/join/union queries, forming the *positive* relational algebra, and equivalent in expressiveness to UCQs. Then the decidability of equivalence of $\mathcal{RA}$ queries under $\mathbb{Z}$-semantics is a corollary of the decidability of equivalence of UCQs.

- It follows that in reformulation using views under $\mathbb{Z}$-semantics we can work with differences of unions of conjunctive queries (DUCQs). We give a terminating, confluent, sound and complete rewrite system such that if two DUCQs are equivalent under a set of views then they can be rewritten to the same query (modulo isomorphism). This leads to our procedure for exploring the space of reformulations (using the opposites of the rewrite rules).

- In contrast to CQs/UCQs under set semantics, there is no inherent or natural, instance-independent notion of "minimality" for DUCQs under $\mathbb{Z}$-semantics that would yield a finite reformulation search space. We bound the search under a simple cost model, which is an abstraction of the one used in a query optimizer.

- Next we examine when we can use the $\mathbb{Z}$-semantics reformulation strategy to obtain results that work for the bag semantics. We show that the reformulation procedure is closed for queries/views in this class. We also give simple membership conditions.

- We also show how to extend our results to queries with *built-in predicates*, i.e., inequalities and non-equalities.

- Finally, we discuss the use of $\mathbb{N}[X]$-relations for updates, and we show that $\mathbb{Z}$-equivalence and $\mathbb{Z}[X]$-equivalence of relational algebra queries coincide. A consequence is that our $\mathbb{Z}$-semantics reformulation strategy is also sound and complete for $\mathbb{Z}[X]$-semantics.

The chapter is structured as follows. We discuss motivating applications in Section 5.1. We define the semantics of $\mathcal{RA}$ on $\mathbb{Z}$-relations, establish the decidability of $\mathbb{Z}$-equivalence of $\mathcal{RA}$ queries and introduce DUCQs in Section 5.2. We introduce the rewrite system for queries using views in Section 5.3. We present reformulation algorithms and strategies in Section 5.4. We discuss reformulation for bag semantics/set semantics via $\mathbb{Z}$-semantics in Section 5.5. We extend our $\mathbb{Z}$-equivalence results to $\mathcal{RA}$ with built-in predicates in Section 5.6. We discuss $\mathbb{Z}$[X]-relations in Section 5.7.

5.1 Applications of Differences

In this section, we illustrate the three motivating applications mentioned in the introduction, and show how these problems are closely related. Given a uniform way of representing *data* along with *changes to the data*, we can consider each of these problems to be a case of *query reformulation* or *query rewriting*. We shall propose $\mathbb{Z}$-relations as a unifying representation for this purpose.

Optimizing queries using views [23, 101]. Given a query Q and a set of materialized views $\mathcal{V}$, the goal is to speed up computation of Q by (possibly) rewriting Q using views in $\mathcal{V}$. Sometimes, a view may be "nearly" applicable for answering a query, but cannot be used unless difference is allowed in the rewriting. For example, consider a view V with paths of length 2 and 3 in R:

$$\begin{aligned} V(x,y) &\ \text{:-}\ R(x,z), R(z,y) \\ V(x,y) &\ \text{:-}\ R(x,u), R(u,v), R(v,y) \end{aligned}$$

and a query Q for paths of length 3:

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

To answer Q using V, we might compute paths of length 2, then subtract those paths from V under bag semantics. If our end goal is set semantics, we would add a duplicate elimination step at the end.

View adaptation [65]. Here, we have a set of base relations, an existing materialized view, and an updated view definition, and we want to refresh the materialized view instance to reflect the new definition. For example, the materialized view:

$$\begin{aligned} V(x,y,z) &\ \text{:-}\ R(x,y), R(x,z) \\ V(x,y,z) &\ \text{:-}\ R(x,y), R(y,z) \\ V(x,y,z) &\ \text{:-}\ R(x,y), R(y,z), y = z \end{aligned}$$

might be redefined by deleting the second rule and projecting out the middle column:

$$\begin{aligned} V'(x,z) &\ \text{:-}\ R(x,y), R(x,z) \\ V'(x,z) &\ \text{:-}\ R(x,y), R(y,z), y = z \end{aligned}$$

In this case, the computation of $V'(x,z)$ might be sped up under bag semantics by computing the second rule $V(x,y,z)$:- $R(x,y), R(y,z)$, subtracting the tuples of the result, and projecting only x and z. (Again, duplicate removal could be done at the end to get a set-semantics answer.)

Incremental view maintenance [66]. We are given a source database, a materialized view, and a set of changes to be applied to the source database (tuple insertions or deletions), and the goal is to compute the corresponding change to the materialized view. This can then be applied to the existing materialized view to obtain the new version. For example, consider a source relation R and materialized view V, with definitions and instances:

$$R : \boxed{\begin{array}{cc} a & b \\ c & b \\ b & c \\ b & a \end{array}} \qquad V(x,y) \text{ :- } R(x,z), R(z,y) : \boxed{\begin{array}{cc} a & a \\ a & c \\ b & b \\ c & c \end{array}}$$

Suppose we update R by deleting (b,a) and inserting (c,d); to maintain V, we must insert a new tuple (b,d). Note that deleting (b,a) does *not* result in deleting (b,b) from V, because this tuple can still be derived by joining (b,c) with (c,b). Yet if we now delete (c,b) from R, then (b,b) and (c,c) must be deleted from V.

In order to solve the incremental view maintenance problem, Gupta et al. [66] proposed recording in V along with each tuple the *number of derivations* of that tuple, i.e., the *multiplicity* of the tuple under bag semantics. To represent changes to a (bag) relation, they introduced the concept of *delta relations*, essentially bag relations with associated *signs*: "$+$" indicates an insertion and "$-$" a deletion. Finally, in order to propagate updates, they proposed the device of *delta rules*. A set of delta rules for V correspond to the UCQ:

$$\begin{array}{lcl} V^{\Delta}(x,y) & \text{:-} & R(x,z), R^{\Delta}(z,y) \\ V^{\Delta}(x,y) & \text{:-} & R^{\Delta}(x,z), R'(z,y) \end{array}$$

Here R' denotes the updated version of R, obtained by *applying* the delta R^{Δ} to R, i.e., computing $R' \overset{\text{def}}{=} R^{\Delta} \cup R$ where union on delta relations sums the (signed) tuple multiplicities. By computing V^{Δ} and then applying it to V, we obtain the updated version of V, namely V'. Note that there may actually be more than one possible set of delta rules for V, e.g.:

$$\begin{array}{lcl} V^{\Delta}(x,y) & \text{:-} & R^{\Delta}(x,z), R(z,y) \\ V^{\Delta}(x,y) & \text{:-} & R'(x,z), R^{\Delta}(z,y) \end{array}$$

We would like to choose among the possible delta rules sets, as well as simply computing V' "from scratch" via the query:

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

based on the expected costs of the various plans. We model the relation R^{Δ} with $\mathbb{Z}$-relations, presented in the next section. We consider this to again be a variant of optimizing queries using views: the goal is to compute the view with deltas applied, V', given not only the base data R and R^{Δ}, but also the existing materialized view V, and the materialized view R' resulting from applying the updates in R^{Δ} to R. (We can compute V' either before or after updating R to R'.) Since every delta relation includes deletions, this version of the reformulation problem also incorporates a form of difference.

A unified treatment of these three applications requires methods for representing data and changes via an excursion to an alternative semantics ($\mathbb{Z}$-relations), performing query reformulation in this context, and proving sufficient conditions for cases in which the results agree with bag and set semantics.

5.2 Z-Relations

We use here the named perspective [2] 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 $U\text{-}\mathsf{Tup}$. (Usual) relations over U are subsets of $U\text{-}\mathsf{Tup}$ (we will also refer to these as $\mathbb{B}$-*relations*).

A *bag relation* over attributes U is a mapping $R : U\text{-}\mathsf{Tup} \to \mathbb{N}$ from U-tuples to their associated *multiplicities*. Tuples with multiplicity 0 are those "not present" in R, and we require of a bag relation that its *support* defined by $\mathsf{supp}(R) \overset{\mathtt{def}}{=} \{t \mid R(t) \neq 0\}$ is finite.

A $\mathbb{Z}$-*relation* over attributes U is a mapping $R : U\text{-}\mathsf{Tup} \to \mathbb{Z}$ of finite support. In other words, it is a bag relation where multiplicities may be positive or negative.

A *bag instance* ($\mathbb{Z}$-instance) is a mapping from predicate symbols to bag relations ($\mathbb{Z}$-relations). A *set instance* (or $\mathbb{B}$-*instance*) is a mapping from predicate symbols to $\mathbb{B}$-relations.

We define the semantics of the relational algebra on $\mathbb{Z}$-instances in accordance with the semiring-annotated semantics introduced in Chapter 2. The operations of the **positive algebra** ($\mathcal{RA}^{+}$) are as in Definition 2.2. For the time being we assume selection predicates correspond to equalities $A = B$ of attributes or equalities $A = c$ of attributes with domain values. (We extend this to include inequality predicates in Section 5.6.)

Observe that if we start with $\mathbb{Z}$-relations with just positive multiplicities, i.e. $\mathbb{N}$-relations, the results of the operations defined above are also $\mathbb{N}$-relations (leading to Lemma 5.1 below) and the resulting semantics is in fact the usual bag semantics. $\mathbb{B}$-relations correspond to reading the "$+$" as disjunction and the "$\cdot$" as conjunction, essentially "eliminating duplicates", hence we get the usual set semantics.

We extend the above definition to the full relational algebra ($\mathcal{RA}$) on $\mathbb{Z}$-instances by defining the difference operator in the obvious way:

difference If $[\![Q_1]\!]^I, [\![Q_2]\!]^I : U\text{-}\mathsf{Tup} \to K$ then $[\![Q_1 - Q_2]\!]^I : U\text{-}\mathsf{Tup} \to K$ is defined by

$$[\![Q_1 - Q_2]\!]^I(t) \stackrel{\text{def}}{=} [\![Q_1]\!]^I(t) \oplus -[\![Q_2]\!]^I(t)$$

For bag semantics, the subtraction in the definition above is replaced by *proper subtraction* (negative numbers are truncated to 0). For $\mathbb{B}$-relations this becomes the usual set difference in set semantics.

Every bag instance is also a $\mathbb{Z}$-instance. Relational queries on set or bag instances can be evaluated under bag semantics or under $\mathbb{Z}$-semantics. To disambiguate we use the notation $\mathsf{Eval}_K QI$ to mean the evaluation of Q on bag instance I under K-semantics, for $K \in \{\mathbb{N}, \mathbb{Z}\}$.

For $Q, Q' \in \mathcal{RA}$ and $K \in \{\mathbb{B}, \mathbb{N}, \mathbb{Z}\}$ we say that Q and Q' are *K-equivalent* (denoted $Q \equiv_K Q'$) if for every K-instance I, $[\![Q]\!]^I = [\![Q']\!]^I$. The following simple but useful observation relates $\mathbb{N}$-equivalence and $\mathbb{Z}$-equivalence of positive queries:

Lemma 5.1. *If $Q, Q' \in \mathcal{RA}^+$ then $Q \equiv_{\mathbb{Z}} Q'$ iff $Q \equiv_{\mathbb{N}} Q'$*

Proof. "$\Rightarrow$" follows from the fact that every bag instance is also a $\mathbb{Z}$-instance and the two semantics agree for positive queries on bag instances. "$\Leftarrow$" follows from the fact that bag equivalent positive queries, when transformed into UCQs, are isomorphic (see Section 7.5 in [58]). ☐

Normal Form and Decidability

Equivalence of relational queries under set semantics has long been known to be undecidable [123]. We have also seen (cf. footnote in the preamble to this chapter) that bag equivalence of relational queries is undecidable, via an easy reduction from containment of UCQs (which makes essential use of proper subtraction). In contrast, the form of subtraction used in $\mathbb{Z}$-relations turns out to be surprisingly well-behaved: we will show in this section is that $\mathbb{Z}$-equivalence of relational queries is actually *decidable*. The key idea is that in contrast to bag and set semantics, under $\mathbb{Z}$-semantics, every relational query is equivalent to a single difference of positive queries:

Theorem 5.2 (Normalization)**.** *For any $Q \in \mathcal{RA}$ we can effectively find $A, B \in \mathcal{RA}^+$ such that $Q \equiv_{\mathbb{Z}} A - B$.*

Proof. For any $Q \in \mathcal{RA}$ we define a *difference normal form* denoted by $\mathsf{diffNF}(Q)$ by structural recursion on Q as follows:

- If R is a predicate symbol then $\mathsf{diffNF}(R) = R - \emptyset$.
- If $\mathsf{diffNF}(Q) = A - B$ then

$$\begin{array}{rrcl@{\qquad}rrcl}
 & (A-B)-C & \equiv_{\mathbb{Z}} & A-(B\cup C) & & \sigma_P(A-B) & \equiv_{\mathbb{Z}} & \sigma_P(A)-\sigma_P(B) \\
 & A\bowtie(B-C) & \equiv_{\mathbb{Z}} & (A\bowtie B)-(A\bowtie C) & (!) & (A-B)\cup C & \equiv_{\mathbb{Z}} & (A\cup C)-B \\
(!) & A-(B-C) & \equiv_{\mathbb{Z}} & (A\cup C)-B & (!) & \pi_X(A-B) & \equiv_{\mathbb{Z}} & \pi_X(A)-\pi_X(B) \\
 & (A-B)\bowtie C & \equiv_{\mathbb{Z}} & (A\bowtie C)-(B\bowtie C) & & \rho_\beta(A-B) & \equiv_{\mathbb{Z}} & \rho_\beta(A)-\rho_\beta(B) \\
(!) & A\cup(B-C) & \equiv_{\mathbb{Z}} & (A\cup B)-C & & & &
\end{array}$$

Figure 5.1: Algebraic identities for the difference operator under $\mathbb{Z}$-semantics

  - $\mathsf{diffNF}(\pi_X(Q)) = \pi_X(A) - \pi_X(B)$, and
  - $\mathsf{diffNF}(\sigma_P(Q)) = \sigma_P(A) - \sigma_P(B)$.
- If $\mathsf{diffNF}(Q_1) = A_1 - B_1$ and $\mathsf{diffNF}(Q_2) = A_2 - B_2$ then
  - $\mathsf{diffNF}(Q_1 \bowtie Q_2) = (A_1 \bowtie A_2 \;\cup\; B_1 \bowtie B_2) - (A_1 \bowtie B_2 \;\cup\; B_1 \bowtie A_2)$,
  - $\mathsf{diffNF}(Q_1 \cup Q_2) = (A_1 \cup A_2) - (B_1 \cup B_2)$, and
  - $\mathsf{diffNF}(Q_1 - Q_2) = (A_1 \cup B_2) - (A_2 \cup B_1)$.

Clearly $\mathsf{diffNF}(Q)$ has the form $A - B$ with $A, B \in \mathcal{RA}^+$. It is straightforward to show by induction on Q that $Q \equiv_{\mathbb{Z}} \mathsf{diffNF}(Q)$ using the algebraic $\mathbb{Z}$-semantics identities in Figure 5.1. Note that in general $\mathsf{diffNF}(Q)$ may be of size exponential in the size of Q.

The given definition of $\mathsf{diffNF}(Q)$ proliferates redundant occurrences of $\emptyset$. We will assume that simplifications are made based on identities such as $A \cup \emptyset \equiv A$ or $A \bowtie \emptyset \equiv \emptyset$ (true for $\mathbb{Z}$-semantics, bag semantics, set semantics, and in fact all semiring-annotated relations semantics). As a result, it is easy to check that for $\mathcal{RA}^+$ queries Q we have $\mathsf{diffNF}(Q) = Q - \emptyset$. $\square$

Note that the identities in Figure 5.1 that are flagged by (!) *fail* in fact for both set and bag semantics, and indeed Theorem 5.2 fails for set or bag semantics.

Corollary 5.3. *$\mathbb{Z}$-equivalence of $\mathcal{RA}$ queries is decidable.*

Proof. Let $Q, Q' \in \mathcal{RA}$. By Theorem 5.2, $Q \equiv_{\mathbb{Z}} Q'$ iff $\mathsf{diffNF}(Q) \equiv_{\mathbb{Z}} \mathsf{diffNF}(Q')$. Let $A, B, C, D \in \mathcal{RA}^+$ such that $\mathsf{diffNF}(Q) = A - B$ and $\mathsf{diffNF}(Q') = C - D$. But $A - B \equiv_{\mathbb{Z}} C - D$ iff $A \cup D \equiv_{\mathbb{Z}} B \cup C$ iff $A \cup D \equiv_{\mathbb{N}} B \cup C$. But $A \cup D$ and $B \cup C$ are in $\mathcal{RA}^+$ and hence equivalent to UCQs so the result follows from the decidability of UCQ bag equivalence [58]. $\square$

Corollary 5.3 can be used to show a PSPACE upper bound on the complexity of checking $\mathbb{Z}$-equivalence of $\mathcal{RA}$ queries; however, the exact complexity remains open. A lower bound is given by Proposition 5.4 which shows that the problem is at least GI-hard.[3]

DUCQs

As we shall below, our reformulation algorithm uses $\mathcal{RA}$ queries in difference normal form. In, fact as with CQs and UCQs it is notationally convenient to use a Datalog-style syntax for the queries on which reformulation operates directly. We define *differences of unions of conjunctive queries* (DUCQs) for this purpose, for example:

$$\begin{array}{rcl} Q(x,z) & \text{:-} & R(x,y), S(y,z) \\ Q(x,y) & \text{:-} & R(x,y), R(y,x), R(x,x) \\ -Q(x,y) & \text{:-} & R(x,y), T(y,y) \end{array}$$

The $-$ marks a "negated" CQ, so this example corresponds (roughly) to the algebraic form

$$Q = (\pi(R \bowtie S) \cup (R \bowtie R \bowtie R)) - (R \bowtie T)$$

DUCQs are related to the *elementary differences* of [123]. Under $\mathbb{Z}$-semantics, any $\mathcal{RA}$ query can be written equivalently as a DUCQ; this follows from Theorem 5.2 and the fact that any $\mathcal{RA}^+$ query can be rewritten as a UCQ. The semantics of DUCQs on bag relations/$\mathbb{Z}$-relations/set relations can be given formally by translation to $\mathcal{RA}$.

Although we do not have a precise complexity for the $\mathbb{Z}$-equivalence of $\mathcal{RA}$ queries, we can fully characterize the complexity of checking $\mathbb{Z}$-equivalence of DUCQs:

Proposition 5.4. *For DUCQs Q, Q' checking $Q \equiv_{\mathbb{Z}} Q'$ is* GI*-complete.*

Proof. Following similar reasoning as in the proof of Corollary 5.3, we have $A \cup -B \equiv_{\mathbb{Z}} C \cup -D$ iff $A \cup D \equiv_{\mathbb{Z}} B \cup C$ iff $A \cup D \equiv_{\mathbb{N}} B \cup C$. But by the results of [30], this holds iff $A \cup D$ and $B \cup C$ are isomorphic. Thus $\mathbb{Z}$-equivalence of DUCQs is polynomial time interreducible with the GI-complete problem of checking isomorphism of UCQs. □

5.3 Reformulation Using Views

Let Σ be a relational schema and $\mathcal{V}$ a finite set of views over Σ. Let $\Sigma_{\mathcal{V}}$ be the schema consisting of the view names (disjoint from Σ). A $\Sigma \cup \Sigma_{\mathcal{V}}$ instance is *$\mathcal{V}$-compatible* if it consists of a Σ-instance I

---

[3] GI is the class of problems polynomial time reducible to graph isomorphism. Graph isomorphism is known to be in NP, but is not known or believed to be either NP-complete or in PTIME.

and a $\Sigma_{\mathcal{V}}$-instance J such that $J = [\![\mathcal{V}]\!]^I$. Orthogonally, these instances may consist of $\mathbb{Z}$-relations, $\mathbb{N}$-relations or $\mathbb{B}$-relations. For $K \in \{\mathbb{B}, \mathbb{N}, \mathbb{Z}\}$ we say that two relational queries Q, Q' over $\Sigma \cup \Sigma\mathcal{V}$ are *K-equivalent under* $\mathcal{V}$, denoted $Q \equiv_K^{\mathcal{V}} Q'$, if Q and Q' agree on all *cV-compatible* K-instances.

Given a Σ-query Q, a *reformulation of* Q *using* $\mathcal{V}$ is a $\Sigma \cup \Sigma\mathcal{V}$-query Q' such that $Q \equiv_K^{\mathcal{V}} Q'$. We also call Q' an *equivalent rewriting* using $\mathcal{V}$.

Query reformulation algorithms typically work by *effectively enumerating* certain queries, call this a *search space*, filtering the queries that are not equivalent rewritings, and finding a minimum-cost query (according to some cost model). There are three requirements for designing such algorithms; (1) the search space must be *finite* so the algorithm terminates, (2) the returned rewriting should be actually equivalent so the algorithm is *sound*, and (3) if equivalent rewritings exist then at least one of them should be found so the algorithm is *complete*. A more subtle requirement is that the returned rewriting has smaller cost than *any* rewriting, even when there are infinitely many ones.

In the case of set semantics one can construct infinitely many rewritings of CQs even without views, simply by adding to their bodies atoms that can be homomorphically mapped to existing ones [2]. This can be dealt with by considering only queries that have no non-trivial endomorphism, call them *locally minimal*. It was shown in [101] that the space of locally minimal CQ reformulations of a CQ query Q using CQ views $\mathcal{V}$ is finitely bounded. Further work focused on pruning and efficiently exploring this search space (looking for not just equivalent rewritings but also maximally contained rewritings) [102, 44, 119, 111]. Even if certain classes of integrity constraints (capturing views and more) are considered, a corresponding notion of minimality can be used to define a finite search space for rewritings, using the *chase and backchase* technique [38]. For CQs and bag semantics a finite search space exploration is described in [23]. Interestingly, the UCQ and/or bag-semantics analogs of the complexity results in [101] do not seem to appear anywhere (to the best of our knowledge). For bag semantics, we attempt to remedy this omission in Subsection 5.3 below.

In the reformulation procedures we have mentioned, the search spaces are constructed and enumerated combinatorially, thus including non-equivalent rewritings. One takes advantage of the decidability of $\equiv_K^{\mathcal{V}}$ to filter them out. But such decidability is not just sufficient, it is also *necessary*. Indeed, all such reasonable approaches describe a total recursive function that associates to each Σ-query Q a finite set *searchSpace*(Q) of $\Sigma \cup \Sigma_{\mathcal{V}}$-queries. Moreover, procedures like local minimization can be abstracted by another total recursive function that associates to each Σ-query Q and each $\Sigma \cup \Sigma_{\mathcal{V}}$-query Q' another $\Sigma \cup \Sigma_{\mathcal{V}}$-query $\mu(Q, Q')$ such that $Q' \equiv_K \mu(Q, Q')$. And finally, one shows that $Q' \equiv_K^{\mathcal{V}} Q$ if and only if $\mu(Q, Q') \in$ *searchSpace*(Q). Since the latter is decidable, so is $\equiv_K^{\mathcal{V}}$.

In particular, K-equivalence of Σ-queries must also be decidable which is why under bag or set semantics we cannot hope to extend the ideas that have worked for positive queries to rewritings of relational queries with difference.

We have seen that shifting to $\mathbb{Z}$-semantics however leads to the decidability of $\mathbb{Z}$ equivalence of $\mathcal{RA}$ queries (hence DUCQs). According to the discussion above, next we need to explore $\mathbb{Z}$-equivalence under a set of views and we do so in Section 5.3 using an uniquely terminating *term rewrite system*.

In the same subsection we discuss adding the reverse of the term rewrite rules as the basis for a reformulation algorithm. In the last two subsections we consider limiting our search to *diff-irredundant rewritings*, and we develop a *cost model* and a procedure for limiting the search space further.

Complexity of Bag Reformulation

In the case of bag semantics, UCQ queries and UCQ views there are only finitely many rewritings (modulo symbol renaming). This is essentially because under bag semantics, positive query equivalence is the same as isomorphism (see [24] for CQs and [30] for UCQs). For CQs, that is, select-project-join queries, an algorithm for exploring the resulting finite search space in conjunction with System-R style query optimization was given in [23].

We consider here, for UCQs and bag semantics, the analogs of the complexity results shown in [101] for the the existence of rewritings of CQs using CQ views under set semantics.

Theorem 5.5. *Given a query $Q \in UCQ$ and set of views $\mathcal{V} \subseteq UCQ$ it is* NP*-complete to determine whether:*

(i) There exists a UCQ rewriting under bag semantics of Q using at least one predicate in cV (the problem is NP*-hard even for CQs).*

(ii) There exists a UCQ rewriting under bag semantics of Q that is complete*, i.e., using only predicates in $\mathcal{V}$ (the problem is* NP*-hard even for CQs).*

Proof. (Sketch) Membership in NP in both cases is easy to see.

(i) NP-hardness when Q and V are CQs (and hence also for the case where Q and V are UCQs) is established by a straightforward reduction from the *subgraph isomorphism* problem: given graphs G_1, G_2 is there a subgraph of G_1 which is isomorphic to G_2? (In contrast to graph isomorphism, the subgraph isomorphism problem is known to be NP-complete.)

(ii) NP-hardness when Q and V are CQs is again established by a reduction from the subgraph isomorphism problem. This time, we work with a slightly specialized version of the problem: if n is the number of edges in G_1, and m is the number of edges in G_2, we assume that $m - n$ does not divide m. It can be shown that this version of the problem remains NP-complete. Next, given directed graphs G_1 and G_2 satsifying this assumption, we construct a Boolean CQ Q encoding the edges of G_2, a CQ view V_1 encoding the edges of G_1, and an additional CQ view V_2 defined

$$V_2(x_1, \ldots, x_{2(m-n)}) \text{ :- } E(x_1, x_2), E(x_3, x_4), \ldots, E(x_{2(m-n)-1}, x_{2(m-n)}).$$

We finish the reduction by showing that there exists a complete rewriting of Q using $\{V_1, V_2\}$ iff there is a subgraph of G_1 which is isomorphic to G_2.

□

By the way, for DUCQs, the first question in the theorem has a trivial answer: a DUCQ Q can always be rewritten as the equivalent $(V \cup -V) \cup Q$ (equivalent under bag semantics, $\mathbb{Z}$-semantics, and even set semantics). As for the second question in the theorem, for DUCQs, we do not even know if this problem is decidable.

Term Rewriting System and Decidability of Equivalence under Views

For the dual purposes of checking equivalence under views and — as we shall see later — enumerating reformulations we introduce here a *term rewriting system* [91] for DUCQs under $\mathbb{Z}$-semantics. We fix a relational schema Σ and a set of views $\mathcal{V}$ given by DUCQs over the relations in Σ. The terms of our rewrite system are the DUCQs over any combination of the source predicates in Σ and the view predicates in $\mathcal{V}$.

In the rewrite rules we use an auxiliary *view unfolding relation* on CQs, defined $Q \rightarrow_V Q'$, if Q' can be obtained from a CQ Q by *unfolding* (in the standard way) a single occurrence of the view predicate V in Q. We extend $\rightarrow_V$ to work with UCQ views by unfolding repeatedly (once for each CQ in the view) and producing a UCQ as output (with the same number of rules as the view). For example, if V is the UCQ view:

$$\begin{array}{lcl} V(x,y) & \text{:-} & R(x,z), R(z,y) \\ V(x,y) & \text{:-} & R(x,u), R(u,v), R(v,y) \end{array}$$

and Q is the CQ:

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

then $Q \to_V Q'$ where Q' is the UCQ:

$$\begin{aligned} Q'(x,y) &\ \text{:-}\ V(x,z), R(z,w), R(w,y) \\ Q'(x,y) &\ \text{:-}\ V(x,z), R(z,u), R(u,v), R(v,y) \end{aligned}$$

Finally we extend $\to_V$ to work with DUCQs by propagating the $-$ marks (e.g., if the second CQ in V above were marked $-$, then the second CQ in Q' above would also be marked $-$).

Now we define a rewrite relation $\to$ on terms as follows:

$$\frac{P, Q, R \in \text{DUCQ} \qquad V \in \mathcal{V} \qquad P \to_V Q}{P \cup R \to Q \cup R} \qquad (\textsc{Unfold})$$

$$\frac{A, B \in \text{CQ} \qquad Q \in \text{DUCQ} \qquad A \cong B}{Q \cup A \cup (-B) \to Q} \qquad (\textsc{Cancel})$$

Next we establish the salient properties of our rewrite system. We denote the transitive reflexive closure of $\to$ by $\xrightarrow{*}$. A term is a *normal form* if no rewrite rule applies to it. A *reduction sequence* $Q_1 \to \cdots \to Q_n$ is *terminating* if Q_n is a normal form.

Proposition 5.6. *The rewrite system above is* uniquely terminating, *i.e., it satisfies the following two properties:*

1. ***(confluence)*** *For all $Q, Q_1, Q_2 \in$ DUCQ if $Q \xrightarrow{*} Q_1$ and $Q \xrightarrow{*} Q_2$ then there exist $Q_3, Q_3' \in$ DUCQ such that $Q_1 \xrightarrow{*} Q_3$ and $Q_2 \xrightarrow{*} Q_3'$ and $Q_3 \cong Q_3'$.*

2. ***(termination)*** *Every reduction sequence $Q_1 \to Q_2 \to \cdots$ eventually must terminate.*

Proposition 5.7. *The rewrite system above is* sound and complete *w.r.t. $\mathbb{Z}$-equivalence with respect to $\mathcal{V}$, i.e., for any $Q_1, Q_2 \in$ DUCQ we have*

1. ***(soundness)*** *If $Q_1 \xrightarrow{*} Q_2$ then $Q_1 \equiv_{\mathbb{Z}}^{\mathcal{V}} Q_2$.*

2. ***(completeness)*** *If $Q_1 \equiv_{\mathbb{Z}}^{\mathcal{V}} Q_2$ then there exist Q_1', Q_2' such that $Q_1 \xrightarrow{*} Q_1'$ and $Q_2 \xrightarrow{*} Q_2'$ and $Q_1' \cong Q_2'$.*

Corollary 5.8. *$\mathbb{Z}$-equivalence of DUCQs (and thus $\mathcal{RA}$ queries) with respect to a set of DUCQ (and thus $\mathcal{RA}$) views $\mathcal{V}$ is decidable.*

The corollary holds for $\mathcal{RA}$ queries and views because we can always convert them first to DUCQs. Note that converting them to DUCQs may increase the size exponentially, and there can be a separate exponential blowup when unfolding the views in the DUCQs. We leave open the exact complexity of the problems in Corollary 5.8.

In addition to their use for showing decidability, the term rewrite rules can be very valuable in the search for reformulations. By the soundness property above, using the rules guarantees that we only explore equivalent rewritings. By the completeness property, if an equivalent rewriting exists, it will be reachable by a sequence of rewrite steps or — and this is the main difficulty we will have to face — *converse* rewrite steps.

Thus, in Section 5.4 we develop an enumeration algorithm for $\mathbb{Z}$-equivalent reformulations, which uses the above rewrite system combined with the converse of the Unfold and Cancel operations, called Fold and Augment, respectively. Fold rewrites a query by removing some of its CQs and replacing them with an equivalent view (recall that under bag semantics two UCQs are equivalent iff they are isomorphic.) In general we may need to Augment first before Fold is applicable.

$$\frac{P, Q, R \in \text{DUCQ} \qquad V \in \mathcal{V} \qquad P \rightarrow_V Q}{Q \cup R \rightarrow P \cup R} \tag{Fold}$$

$$\frac{A, B \in \text{CQ} \qquad Q \in \text{DUCQ} \qquad A \cong B}{Q \rightarrow Q \cup A \cup (-B)} \tag{Augment}$$

We observe that in isolation Augment can add/subtract arbitrary CQsto the query, regardless of whether this will turn out to be "useful" (by enabling a subsequent Fold). Hence we do not use Augment directly, but rather define a compound Augment-Fold operation, which augments the query *only* with the CQs necessary to perform a Fold operation with a given view. (The Augment step of Augment-Fold may be skipped if Fold can be directly applied to the query.) Additionally, we restrict Augment-Fold to apply only if there exists at least one rule in common between the query and the view.

After applying Augment-Fold, we may be able to Cancel some rules introduced into the query. Hence, we will also always apply Cancel after an Augment-Fold repeatedly until it is no longer applicable. We denote the rule which corresponds to this sequence of Augment-Fold followed by repeated Cancel by Augment-Fold-Cancel, and write $Q \leftharpoonup Q'$ if Q' can be obtained from Q by an application of Augment-Fold-Cancel. . We denote the reflexive transitive closure of $\leftharpoonup$ by $\overset{*}{\leftharpoonup}$. Note that $Q \overset{*}{\leftharpoonup} Q'$ implies $Q \overset{*}{\leftarrow} Q'$ but the converse does not hold in general.

Proposition 5.9. *If $Q \in DUCQ$ then checking whether there exists $Q' \in DUCQ$ s.t. $Q \rightarrow Q'$ (resp. $Q \leftharpoonup Q'$) is* GI*-complete (resp.* NP*-complete).*

Diff-Irredundant Rewritings

While Unfold and Cancel only give terminating sequences of rewritings, the addition of the converse rules destroys this property. For example Augment (which we denote $\leftarrow$) can be used to "grow" a DUCQ arbitrarily by adding more and more CQs: for example, $Q \leftarrow Q \cup A \cup -A \leftarrow Q \cup A \cup B \cup -A \cup -B \leftarrow \cdots$.

Therefore the space of queries reachable in this way is infinite. How do we cut it down to a finite size? Given the success of considering locally minimal rewritings for set semantics, we could try to find a similar notion of minimality for DUCQs, hopefully one that is compatible with useful cost models. In this subsection, we define such a notion based on removing certain redundant computations, and we consider whether the space of non-redundant rewritings is finite.

Definition 5.10. A DUCQ $Q = A - B$ is said to be *diff-redundant* if for some subset of CQs $A' \subseteq A$ and $B' \subseteq B$, we have $A' \equiv_{\mathbb{Z}}^{\mathcal{V}} B'$. In this case the pair of terms A', B' is said to be *diff-redundant*.

In the example above, $Q \cup A \cup -A$ and $Q \cup A \cup B \cup -A \cup -B$ are both diff-redundant.

A diff-redundant DUCQ can be *minimized* by repeatedly finding and removing diff-redundant pairs of terms until a *diff-irredundant* query is obtained. In the examples above, this leads back to Q (assuming Q itself is irredundant). More generally:

Proposition 5.11. *If Q, Q' are DUCQs that do not contain view predicates and $Q \equiv_{\mathbb{Z}} Q'$, then minimizing Q and minimizing Q' produces the same query (up to isomorphism).*

However, when queries may contain view predicates, this property fails dramatically:

Theorem 5.12. *The set $\mathsf{irr}_{\mathcal{V}}(Q)$ of diff-irredundant rewritings of a DUCQ Q with respect to a set of views $\mathcal{V}$ is in general infinite.*

Proof. If R and S are binary relations, then denote by RS the *relational composition* of R and S, i.e., the query:

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

We will use exponents to mean repeated relational composition (e.g., $R^3 = RRR$), and to simplify notation we use here $+$ for union.

Now let $Q = R^2$, and let $\mathcal{V}$ contain the single view $V = R + R^3$. Then we have:

$$\begin{aligned} Q &= R^2 \\ &\equiv_{\mathbb{Z}}^{\mathcal{V}} VR - R^4 \\ &\equiv_{\mathbb{Z}}^{\mathcal{V}} VR - VR^3 + R^6 \\ &\equiv_{\mathbb{Z}}^{\mathcal{V}} VR - VR^3 + VR^5 - R^8 \end{aligned}$$

and more generally:

$$Q \quad \equiv_{\mathbb{Z}}^{\mathcal{V}} \quad (-1)^n R^{2(n+1)} + \sum_{i=1}^{n} (-1)^{i+1} V R^{2i-1}$$

for all $n \geq 0$. Clearly, there are infinitely many rewritings of this form. Moreover, one can check that every such rewriting is diff-irredundant. It follows that $\mathsf{irr}_{\mathcal{V}}(Q)$ is infinite. $\square$

Thus, considering diff-irredundant queries does not suffice to establish a finite bound on the set of possible rewritings for a DUCQ.

A Cost Model

Another notion of minimality we might consider is the *global minimality* of [101], where the goal is to minimize the *total number of atoms* in a CQ reformulated using views. We find this notion problematic for several reasons. First, in contrast to the classical CQ minimization techniques (where minimizing the number of atoms coincides with computing the *core* of the query), the mathematical justification is unclear. Second, it is not clear how global minimality should be extended to UCQs/DUCQs (total number of atoms in all CQs?). Third, in the practical applications of interest to us, the real goal is to minimize the *cost* of executing a query, and the number of atoms is often not indicative of query performance. This is because of many factors, especially the different costs of computing with different source relations (which may be of drastically different cardinalities, have different indexes, etc.).

To illustrate this last point, recall that in Section 5.1 we had a view maintenance example, where the original view was defined as:

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

and our updated view could be defined as:

$$V'(x,y) \quad \text{:-} \quad R'(x,y), R'(z,y)$$

where $R'(x,y)$ represents $R(x,y) \cup R^{\Delta}(x,y)$. The reformulation problem has two materialized views: the original V in terms of the base R, and R' (R after updates are applied, which we can compute either before or after computing V'). If R^{Δ} is *large*, then V' may be most efficient to compute in terms of R', as specified in the second rule above. Alternatively, if R^{Δ} is *small*, then it may be more efficient to compute V' using V and delta rules:

$$\begin{aligned} V'(x,y) \quad &\text{:-} \quad V(x,y) \\ V'(x,y) \quad &\text{:-} \quad R^{\Delta}(x,z), R(z,y) \\ V'(x,y) \quad &\text{:-} \quad R(x,z), R^{\Delta}(z,y) \end{aligned}$$

Note that the latter query has three rules compared to only one in the first case, and five atoms to only two. Yet, intuitively, for small R^{Δ}s, the second case is more efficient.

In practical DBMS implementations, a *cost model* [125] predicts query performance given known properties (such as cardinalities) of the input relations. Here we seek an abstract cost model that captures the essence of the detailed models implemented in real optimizers. Our cost model, $\mathsf{cost} : \mathcal{RA} \rightarrow \mathbb{N}$, is instance-dependent, and makes use of external calls to a *cardinality estimation function*, $\mathsf{card} : \mathcal{RA} \rightarrow \mathbb{N}$, which returns the estimated number of tuples in the result of a subquery (for the current database instance). We define cost inductively on $\mathcal{RA}$ expressions:

$$\begin{aligned}
\mathsf{cost}(R) &= \mathsf{card}(R)\\
\mathsf{cost}(\pi E) &= \mathsf{cost}(E)\\
\mathsf{cost}(\sigma E) &= \mathsf{cost}(E)\\
\mathsf{cost}(E_1 \cup E_2) &= \mathsf{cost}(E_1) + \mathsf{cost}(E_2)\\
\mathsf{cost}(E_1 - E_2) &= \mathsf{cost}(E_1) + \mathsf{cost}(E_2)\\
\mathsf{cost}(E_1 \bowtie E_2) &= \mathsf{cost}(E_1) + \mathsf{cost}(E_2)\\
&\quad + \mathsf{card}(E_1 \bowtie E_2)
\end{aligned}$$

This cost model intuitively focuses on the cost of *producing* new tuples in expensive operations, namely joins and tablescans. (Union in $\mathbb{Z}$-semantics is inexpensive, as it is in bag-semantics; difference in $\mathbb{Z}$-semantics is in fact a union operation that negates counts, and hence it is also inexpensive.) Our cost model essentially computes a lower bound on the amount of work a join must perform: it considers tuple creation cost but ignores the cost of matching tuples. (For an index nested loops join or a hash join, matching is in fact inexpensive, so this is a fairly realistic lower bound.) Our model satisfies the *principle of optimality* [32] required by a real query optimizer cost model.

The cost model above is based on queries represented as relational algebra expressions (with a specific order of evaluation). However it can be extended to CQs by defining the cost of a CQ as the cost of the cheapest equivalent algebraic expression,[4] and to UCQs and DUCQs by summing the costs of all the CQs in the UCQ or DUCQ. Assuming every source relation is non-empty, a simple observation is that any CQ has cost at least 1, and any DUCQ Q has cost at least $n = |Q|$ (where $|Q|$ is the number of CQs in the DUCQ).

Now we can establish a bound on the search space in which we wish to look for rewritings of minimum cost:

---

[4] This is in fact done by dynamic programming in real query optimizers.

Proposition 5.13 (completeness under cost)**.** *For $Q \in DUCQ$, let $\mathcal{V}$ be a set of views, let $k = \textsf{cost}(Q)$, and let w be the maximum number of CQs in Q or in a view in $\mathcal{V}$. Assume w.l.o.g. that Q does not contain any view predicates (they can always be removed by applying* Unfold *repeatedly). Then any DUCQ of minimum cost among those equivalent to Q using $\mathcal{V}$ can be reached from Q in at most $O(kw^k)$ steps of* (Fold + Augment)*-rewriting.*

Proof. Suppose that Q' is a DUCQ reformulation of Q of minimum cost. Then using our observations above, Q' contains at most k CQs each containing at most k view predicates. We can rewrite Q' into Q by a sequence of u Unfold steps. (Read in the reverse direction, this will yield a sequence of v Augment steps followed by a sequence of u Fold steps, rewriting Q into Q'.)

Consider a single CQ in Q'. We claim that Unfold can be applied repeatedly to it at most w^k times, replacing the single CQ with at most w^k CQs. Doing this for all of the $\leq k$ CQs in Q', we have $u \leq kw^k$ Unfold steps, and the result is a DUCQ containing $\leq kw^k$ CQs.

Next, since each application of Cancel removes 2 CQs, it follows that Cancel can be applied at most $\lfloor kw^k/2 \rfloor$ times. Thus, the total length of the sequence rewriting Q' to Q is at most $kw^k + \lfloor kw^k/2 \rfloor = O(kw^k)$. □

This yields a finite bound (which can be effectively computed from Q, $\mathcal{V}$, and $\textsf{cost}$) on the region of the rewrite space that must be explored in order to find a rewriting of minimum cost (for a given instance).

5.4 Finding Query Rewritings

Now that we understand the conditions under which reformulation can be bounded to a finite search space, we develop an enumeration algorithm for exploring the space of possible query reformulations, for the problems of *optimizing queries using views*, *view adaptation*, and *view maintenance*. Recall from Section 5.1 that the *optimizing queries using views* problem takes a set of materialized views, plus a query to be reformulated, as input. *View adaptation* takes a single materialized view (the "old" view definition and materialized instance), with the modified view definition as the query to be reformulated. *View maintenance* takes the pre-updated view instance and updated versions of the base relations as input views, with the maintained view (i.e., the view computed over the updated base relations) as the query to be reformulated.

Reformulation Algorithm

From Section 5.3 we have a set of rewrite rules, namely Unfold, Cancel, and Augment-Fold-Cancel. Algorithm 5 shows how these can be composed to enumerate the space of plans. The

Algorithm 5 *reformulate*(query q, set of views V) returns query

```
for each view $v$ in $V$ do
    $v$ := normalize($v$)
end for
Let $q_i$ := normalize($q$)
Let $Q := \emptyset$
Let $F := \{(q_i, 0)\}$
for each $(q, k)$ in $F$ do
    Remove $(q, k)$ from $F$ and add it to $Q$
    for each $v$ in $V$ do
        Let $q'$ := AUGMENT-FOLD-CANCEL($q$,$v$)
        if $q'$ not in $Q$ and not prune$(q', k+1)$ then
            Add $(q', k+1)$ to $F$
        end if
    end for
end for
return the $q$ in $Q$ with lowest cost
```

normalize function (omitted) repeatedly applies Unfold and Cancel to the input query until they are no longer applicable. Next, the main loop simply applies Augment-Fold-Cancel to rewrite portions of each "frontier" query q in F, in terms of any views that overlap with q.

The *prune* function determines whether the rewritten query q' should be added to the frontier set, or disregarded during exploration. For the rewrite algorithm to be guaranteed to terminate and find an optimal rewriting according to the cost model, it suffices to define *prune*(q, k) to return **true** iff k is less than the bound given by Proposition 5.13. (In practice, we would add additional heuristics to *prune* to limit the search space, sacrificing the guarantee of a minimum-cost rewriting.) Once the full space has been explored, *reformulate* returns the rewriting from Q with the lowest cost according to our cost model.

Rewriting in a Query Optimizer

In principle, one could search the space of query rewritings by building a layer above the query optimizer, which enumerates possible rewritings; then separately optimizes each. However, a more efficient approach is to *extend an existing optimizer* to incorporate the rewriting system into its enumeration. Unlike with rewriting of conjunctive queries, most existing cost-based optimizers cannot easily be extended to DUCQ rewritings. The System-R optimizer [125] only does cost-based optimization of *joins*, instead relying on heuristics for applying unions and differences. Starburst [67] can rewrite queries with unions and differences, but only at its heuristics-based *query rewrite* stage. Starburst only has limited facilities for cost-based comparison of alternative rewritings.

Fortunately, the Volcano [55] optimizer generator (as well as its successors) can be modified to incorporate our rewrite scheme within an optimizer. Volcano models a query initially as a plan of *logical operators* representing the algebraic operations, including unions and differences; it uses *transformation rules* to describe algebraic equivalences that can be used to find alternate plans. *Implementation rules* describe how to rewrite a logical operator (or set of operators) into a series of physical algorithms, which have associated costs.

Our rewrite rules of Section 5.3 can be expressed as transformation rules for Volcano, and we would not need to change any implementation rules. However, we would also need to modify Volcano's pruning algorithm: we must place a finite bound on the size of the rewritings explored, as with our *prune* function in the previous subsection.

5.5 Applications to Bag and Set Semantics

Having obtained a sound and complete algorithm for reformulation of $\mathcal{RA}$ queries using $\mathcal{RA}$ views under $\mathbb{Z}$-semantics, we wish to see for which $\mathcal{RA}$ queries/views this same algorithm actually provides reformulations under bag semantics. Thus we are naturally led to study the following class of queries:

Definition 5.14. We denote by $\overline{\mathcal{RA}}$ the class of all queries $Q \in \mathcal{RA}$ such that for all bag-instances I, $\mathsf{Eval}_{\mathbb{N}} Q I = \mathsf{Eval}_{\mathbb{Z}} Q I$.

Right away we see that:

Lemma 5.15. *For any $Q_1, Q_2 \in \overline{\mathcal{RA}}$ we have $Q_1 \equiv_{\mathbb{Z}} Q_2$ implies $Q_1 \equiv_{\mathbb{N}} Q_2$.*

Then

Lemma 5.16. *If $A, B \in \mathcal{RA}^+$ then $A - B \in \overline{\mathcal{RA}}$ if and only if $B \sqsubseteq_{\mathbb{N}} A$.*

Proof. "$\Rightarrow$": suppose $A - B \in \overline{\mathcal{RA}}$ and consider an arbitrary bag instance I and tuple t. Since $\mathsf{Eval}_{\mathbb{Z}} A - B I(t) = \mathsf{Eval}_{\mathbb{N}} A - B I(t) \geq 0$, and by definition, $\mathsf{Eval}_{\mathbb{Z}} A - B I(t) = \mathsf{Eval}_{\mathbb{Z}} A I(t) - \mathsf{Eval}_{\mathbb{Z}} B I(t)$, it follows that $\mathsf{Eval}_{A} \mathbb{Z} I(t) \geq \mathsf{Eval}_{\mathbb{Z}} B I(t)$. Since A and B are positive queries, by Lemma 5.1, $\mathsf{Eval}_{\mathbb{Z}} A I = \mathsf{Eval}_{\mathbb{N}} A I$ and $\mathsf{Eval}_{\mathbb{Z}} B I = \mathsf{Eval}_{\mathbb{N}} B I$ and therefore $\mathsf{Eval}_{\mathbb{N}} A I(t) \geq \mathsf{Eval}_{\mathbb{N}} B I(t)$. Since I and t were chosen arbitrarily, it follows that $B \sqsubseteq_{\mathbb{N}} A$.

"$\Leftarrow$": suppose $A \sqsubseteq_{\mathbb{N}} B$ and consider an arbitrary bag instance I. By Lemma 5.1 $\mathsf{Eval}_{\mathbb{N}} A I = \mathsf{Eval}_{\mathbb{Z}} A I$ and $\mathsf{Eval}_{\mathbb{N}} B I = \mathsf{Eval}_{\mathbb{Z}} B I$. But since $\mathsf{Eval}_{\mathbb{N}} B I \leq \mathsf{Eval}_{\mathbb{N}} A$, it follows that $\mathsf{Eval}_{\mathbb{Z}} A - B I = \mathsf{Eval}_{\mathbb{N}} A - B I \geq 0$. Since I was chosen arbitrarily, it follows that $A - B \in \overline{\mathcal{RA}}$. ☐

Proposition 5.17. *If $Q \in \overline{\mathcal{RA}}$ then $\mathsf{diffNF}(Q) \in \overline{\mathcal{RA}}$ and $Q \equiv_{\mathbb{N}} \mathsf{diffNF}(Q)$.*

Proof. Suppose $Q \in \overline{\mathcal{RA}}$, let $\mathsf{diffNF}(Q) = A - B$ (with $A, B \in \mathcal{RA}^+$), and choose an arbitrary bag instance I and tuple t. Then $\mathsf{Eval}_{\mathbb{N}} Q I(t) = \mathsf{Eval}_{\mathbb{Z}} Q I(t) = \mathsf{Eval}_{\mathbb{Z}} A - B I(t) = \mathsf{Eval}_{\mathbb{Z}} A I(t) - \mathsf{Eval}_{\mathbb{Z}} B I(t) \geq 0$. It follows that $\mathsf{Eval}_{\mathbb{Z}} A I(t) \geq \mathsf{Eval}_{\mathbb{Z}} B I(t)$ and therefore (using Lemma 5.1) that $\mathsf{Eval}_{\mathbb{Z}} A - B I(t) = \mathsf{Eval}_{\mathbb{N}} A - B I(t)$. Since I and t were chosen arbitrarily, it follows that $Q \equiv_{\mathbb{N}} A - B$, as required, and also that $B \sqsubseteq_{\mathbb{N}} A$. By Lemma 5.16 this in turn implies that $A - B \in \overline{\mathcal{RA}}$. ☐

Corollary 5.18. *Bag-equivalence of $\overline{\mathcal{RA}}$ queries is decidable.*

Next we show that if we start with a DUCQ in $\overline{\mathcal{RA}}$ and a set of DUCQ views also in $\overline{\mathcal{RA}}$, then the exploration of the space of reformulations prescribed by the algorithm of Section 5.4 examines only queries in $\overline{\mathcal{RA}}$ which are $\equiv_{\mathbb{N}}$ under the views to the original DUCQ.

Suppose that $\mathcal{V}$ is a set of views in $\overline{\mathcal{RA}}$ expressed as DUCQs.

Proposition 5.19. *If Q is a DUCQ in $\overline{\mathcal{RA}}$ and Q' is a DUCQ obtained from Q either by a step of augmentation or by a step of folding with a view from $\mathcal{V}$ then $Q' \in \overline{\mathcal{RA}}$ and Q' is $\equiv_{\mathbb{N}}$ to Q under $\mathcal{V}$.*

Therefore, the reformulation algorithm can be used with $\overline{\mathcal{RA}}$ queries and views. Unfortunately, but not unexpectedly, it is undecidable whether an $\mathcal{RA}$ query or even a DUCQ is actually in $\overline{\mathcal{RA}}$ (Lemma 5.16 provides a reduction from bag-containment of UCQs). But there are interesting classes of queries for which membership in $\overline{\mathcal{RA}}$ is guaranteed. The simplest but still very useful case is based on the observation that $\mathcal{RA}^+ \subset \overline{\mathcal{RA}}$. It follows that our algorithm is also complete for finding DUCQ reformulations of UCQs using UCQ views, as in the example in Section 5.1.

Next we identify a subclass of $\overline{\mathcal{RA}}$ for which, while still undecidable in general, membership may be easier to check in certain cases.

Definition 5.20. We denote by $\widehat{\mathcal{RA}}$ the class of of all $\mathcal{RA}$ queries Q such that for every occurrence $A - B$ of the difference operator in Q, we have $B \sqsubseteq_{\mathbb{N}} A$.

Theorem 5.21 (Normalization for $\widehat{\mathcal{RA}}$). *For any $Q \in \widehat{\mathcal{RA}}$ can find $A, B \in \mathcal{RA}^+$ such that $Q \equiv_{\mathbb{N}} A - B$.*

Proof. Straightforward induction on Q, using the same algebraic identities as in Theorem 5.2. Although these identities fail under bag semantics for $\mathcal{RA}$ queries, they hold for $\widehat{\mathcal{RA}}$ queries. ☐

Corollary 5.22. *Bag-equivalence of $\widehat{\mathcal{RA}}$ queries is decidable.*

Theorem 5.23. $\widehat{\mathcal{RA}} \subset \overline{\mathcal{RA}}$

Proof. Checking $\widehat{\mathcal{RA}} \subseteq \overline{\mathcal{RA}}$ is a straightforward application of Theorem 5.21. To see that the inclusion is proper, consider the query $Q \stackrel{\text{def}}{=} (R - (R \cup R)) - (R - (R \cup R))$. Q is both $\mathbb{Z}$-equivalent and $\mathbb{N}$-equivalent to the unsatisfiable query $\emptyset$; it follows that Q is in $\overline{\mathcal{RA}}$. However, since $R \cup R \not\sqsubseteq_{\mathbb{N}} R$, it is clear that $Q \notin \widehat{\mathcal{RA}}$. $\square$

Although $\overline{\mathcal{RA}}$ contains queries which are not in $\widehat{\mathcal{RA}}$ (because of their syntactic structure), it turns out that, semantically, $\widehat{\mathcal{RA}}$ *captures* $\overline{\mathcal{RA}}$, as the following theorem makes precise:

Theorem 5.24. *For all $Q \in \overline{\mathcal{RA}}$ there exists $Q' \in \widehat{\mathcal{RA}}$ such that $Q \equiv_{\mathbb{N}} Q'$.*

Proof. For any $Q \in \overline{\mathcal{RA}}$, we show that there must exist $A, B \in \mathcal{RA}^+$ such that $B \sqsubseteq_{\mathbb{N}} A$ and $Q \equiv_{\mathbb{N}} A - B$. Fix a $Q \in \overline{\mathcal{RA}}$. By Theorem 5.2, there exist $A, B \in \mathcal{RA}^+$ such that $Q \equiv_{\mathbb{Z}} A - B$. We argue by contradiction that $B \sqsubseteq_{\mathbb{N}} A$. Suppose $B \not\sqsubseteq_{\mathbb{N}} A$. Then there exists a bag instance I and tuple t such that $\mathsf{Eval}_{\mathbb{N}} A I(t) < \mathsf{Eval}_{\mathbb{N}} B I$. Then $\mathsf{Eval}_{\mathbb{Z}} A - B I(t) < 0$, i.e., $\mathsf{Eval}_{\mathbb{Z}} A - B I = \mathsf{Eval}_{\mathbb{Z}} Q I$ is not even a bag-instance. However, this is a contradiction, because by assumption (since $Q \in \overline{\mathcal{RA}}$), we must have $\mathsf{Eval}_{\mathbb{Z}} Q I = \mathsf{Eval}_{\mathbb{N}} Q I$. Finally, we argue that $Q \equiv_{\mathbb{N}} A - B$. To see this, fix an arbitrary bag instance I. We want to show that $\mathsf{Eval}_{\mathbb{N}} Q I = \mathsf{Eval}_{\mathbb{N}} A - B I$. Since $Q \in \overline{\mathcal{RA}}$, we have $\mathsf{Eval}_{\mathbb{N}} Q I = \mathsf{Eval}_{\mathbb{Z}} Q I$. Since $Q \equiv_{\mathbb{Z}} A - B$ we have $\mathsf{Eval}_{\mathbb{Z}} Q I = \mathsf{Eval}_{\mathbb{Z}} A - B I$. Finally, since $A - B \in \widehat{\mathcal{RA}}$, by Theorem 5.23 we have $\mathsf{Eval}_{\mathbb{Z}} A - B I = \mathsf{Eval}_{\mathbb{N}} A - B I$. This completes the proof. $\square$

Membership in $\widehat{\mathcal{RA}}$ is also undecidable. However in some practical situations, such as incremental view maintenance of $\mathcal{RA}^+$ views using delta rules [66], the difference operator is used in a very controlled way where the containment requirement is satisfied (e.g., it is just necessary for the system to enforce that only tuples actually present in source tables are ever deleted from the sources).

We are also interested in reformulating and answering queries under $\mathbb{Z}$-semantics, but then "eliminating duplicates" to obtain the answer under set semantics. Even for $\widehat{\mathcal{RA}}$ queries, this is not in general straightforward: for example, consider the query $Q = (R \cup R) - R$. Under set semantics, this is equivalent to the unsatisfiable query, while under bag semantics or $\mathbb{Z}$-semantics, it is equivalent to the identity query R. We can, however, restrict the use of negation in $\widehat{\mathcal{RA}}$ further, to obtain another fragment of $\mathcal{RA}$ suitable for this purpose.

Definition 5.25. An $\mathcal{RA}$ query Q over a schema Σ is said to be a *base-difference* query if $A - B$ can appear in Q only when A and B are both base relations (names in Σ). Further, a base-difference query Q is said to be *positive-difference* wrt a set-instance I if for each $A - B$ appearing in Q we have $A^I \supseteq B^I$ (where A^I is the relation in I that corresponds to $A \in \Sigma$.)

Although the use of negation in base-difference queries considered on instances wrt which they are positive-difference is highly restricted, it still captures the form needed for incremental view maintenance, where negation just relates old and new versions of source relations via the tables of deleted and inserted tuples.

For conversion between bag semantics / $\mathbb{Z}$-semantics and set semantics we also define the *duplicate elimination* operator $\delta : \mathbb{Z} \to \mathbb{B}$ which maps 0 to false and everything else (positive or negative) to true. Conversely, we can view any set instance as a bag/$\mathbb{Z}$ instance by replacing false with 0 and true with 1. With this we can state the salient property of base-difference queries:

Proposition 5.26. *. Let $Q \in \mathcal{RA}$ be a base-difference query and let I be a set instance w.r.t. which Q is positive-difference. Then, we can compute $[\![Q]\!]^I$ by viewing I as a $\mathbb{Z}$-instance, computing $[\![Q]\!]^I$ under $\mathbb{Z}$ semantics and finally applying δ.*

Consequently, the optimization techniques in this chapter (which replaces a query with a $\mathbb{Z}$-equivalent one) will also apply to set semantics, provided we restrict ourselves to base-difference queries applied to instances w.r.t. which they are positive-difference.

5.6 Built-in Predicates

To this point, our approach to query rewriting has assumed equality predicates only. Clearly, any practical implementation would also consider inequality ($<, \leq$) and non-equality ($\neq$) predicates. In this section we discuss the extensions necessary to support such *built-in predicates*.

We assume our domain $\mathbb{D}$ comes equipped with a dense linear order $<$, and we define $\mathcal{RA}^<$, $\widehat{\mathcal{RA}}^<$, CQ$^<$, etc. as the previously defined classes of queries extended to allow use of the predicates $<$, $\leq$, $=$, and $\neq$. In general, the predicates in a CQ$^<$ induce only a partial order on the variables. We shall call a CQ$^<$ *total* if the predicates in the query induce a total order on the variables, and *partial* otherwise. To facilitate syntactic comparison of queries we shall assume w.l.o.g. for total CQ$^<$s that a minimal number of predicate atoms are used, i.e., if the predicates induce the total order $x < y < z$ then the predicate atoms $x < y$ and $y < z$ and no others appear in the query. A UCQ$^<$ or DUCQ$^<$ is *total* if all of its CQ$^<$s are total, and *partial* if it contains a partial CQ$^<$.

As in [30, 29], we note that a partial CQ$^<$ Q can always be converted into an equivalent total UCQ$^<$, denoted $\mathsf{lin}(Q)$, that contains one CQ$^<$ for each *linearization* of the partial order on the variables. For example:

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

can be rewritten:

$$\begin{aligned}
Q(x,y) &\ \ \text{:-}\ \ R(x,y), R(y,z), x = z, x < y \\
Q(x,y) &\ \ \text{:-}\ \ R(x,y), R(y,z), x < y, y = z \\
Q(x,y) &\ \ \text{:-}\ \ R(x,y), R(y,z), x < y, y < z \\
Q(x,y) &\ \ \text{:-}\ \ R(x,y), R(y,z), x < z, z < y
\end{aligned}$$

Likewise a partial $\text{UCQ}^<$ ($\text{DUCQ}^<$) Q can be converted into an equivalent total $\text{UCQ}^<$ ($\text{DUCQ}^<$) $\mathsf{lin}(Q)$ by replacing each partial $\text{CQ}^<$ with its equivalent total $\text{UCQ}^<$. Note that if Q is already total, then $Q = \mathsf{lin}(Q)$.

Theorem 5.27. *For all $Q, Q' \in UCQ^<$ the following are equivalent:*

1. $Q \equiv_{\mathbb{N}} Q'$
2. $Q \equiv_{\mathbb{Z}} Q'$
3. $\mathsf{lin}(Q) \cong \mathsf{lin}(Q')$

Proof. (2) ⇒ (1) because every bag instance is a $\mathbb{Z}$-instance. (1) ⇒ (3) is stated in [30] for bag-set semantics but it also holds for bag semantics (see Section 7.5 in [58]). (3) ⇒ (1) follows from the observation that $Q \equiv_{\mathbb{Z}} \mathsf{lin}(Q)$. □

Corollary 5.28. *$\mathbb{Z}$-equivalence of $\mathcal{RA}^<$ queries is decidable and so is bag-equivalence of $\widehat{\mathcal{RA}}^<$ queries.*

This leads to an approach to enumerating rewritings of queries with predicates w.r.t. views: linearize the queries and views into total UCQs/DUCQs as above and reformulate using the linearized representations. As an optional final step, the reformulated query could then be "de-linearized" to a partial query.

5.7 Z[X]-relations

We have seen that $\mathbb{Z}$-relations are useful for reformulating and answering queries even when the end goal is the result under bag semantics or set semantics.

For CDSS, where the end goal of reformulation is the query answer under provenance annotated semantics, we find a natural ring-annotated counterpart to $\mathbb{N}[X]$-relations in $\mathbb{Z}[X]$-*relations*, which are relations annotated with elements of the commutative ring $(\mathbb{Z}[X], +, \cdot, 0, 1)$ of polynomials over variables X with (positive or negative) integer coefficients. Like Z-relations, $\mathbb{Z}[X]$-relations give a convenient way to uniformly represent data and updates. (See Example 1.16 in Chapter 1 for an example.)

We extend the semantics of positive relational algebra queries on semiring-annotated relations (Definition 2.2) to $\mathbb{Z}[X]$-relations exactly as we did for $\mathbb{Z}$-relations, using the usual subtraction of polynomials.

Proposition 5.29.

- *If I is an $\mathbb{N}[X]$-instance, and $Q \in \mathcal{RA}^+$, then evaluating Q on I under $\mathbb{N}[X]$-semantics yields the same result as evaluating Q on I under $\mathbb{Z}[X]$-semantics.*
- *For $Q, Q' \in \mathcal{RA}^+$, we have $Q \equiv_{\mathbb{N}[X]} Q'$ iff $Q \equiv_{\mathbb{Z}[X]} Q'$.*

What about query equivalence, query reformulation, etc. with $\mathbb{Z}[X]$-semantics? It turns out that for such questions, $\mathbb{Z}[X]$-semantics and $\mathbb{Z}$-semantics are essentially interchangable:

Theorem 5.30. *For $Q, Q' \in \mathcal{RA}$ we have $Q \equiv_{\mathbb{Z}} Q'$ iff $Q \equiv_{\mathbb{Z}[X]} Q'$.*

As a consequence, $\mathbb{Z}[X]$-equivalence of $\mathcal{RA}$ queries is also decidable, as is $\mathbb{Z}[X]$-equivalence of $\mathcal{RA}$ queries with respect to a set $\mathcal{V}$ of $\mathcal{RA}$ views. Moreover, the term-rewrite system based approach to reformulating $\mathcal{RA}$ queries under $\mathbb{Z}$-semantics using materialized $\mathcal{RA}$ views is also sound and complete with respect to $\mathbb{Z}[X]$-semantics.

Chapter 6

Semiring-Annotated XML

In Chapter 2, we saw that many of the mechanisms for evaluating queries over annotated relations—e.g., incomplete and probabilistic databases, databases with multiplicities (bags), and those carrying provenance annotations—can be unified in a general framework based on *commutative semirings*. Intuitively, one of the semiring operations models alternative uses of data while the other models its joint (or dependent) use. Chapter 2 presented semantics for positive relational algebra (i.e., unions of conjunctive queries) and positive Datalog for relations decorated with annotations from a semiring. It also identified a canonical notation for provenance annotations using semiring polynomials (and formal power series) that captures, abstractly, computations in arbitrary semirings and therefore serves as a good representation for CDSS implementations such as ORCHESTRA.

This work has opened up a number of interesting avenues for investigation but its restriction to the relational model is limiting. One of the main areas that motivates work on provenance and CDSS is scientific data processing. In these applications, relational data sources are often combined with data extracted from hierarchical repositories of files. XML provides a natural model for tree-structured, heterogeneous sources, but current systems for managing XML data do not provide mechanisms for decorating XML with provenance annotations and for propagating annotated data through queries. A major goal of this work is to extend the framework for semiring-annotated relations described in Chapter 2 to handle annotated XML data. This lays the groundwork for a future extension of the CDSS paradigm to XML.

Besides provenance, our work is also motivated by applications to incomplete and probabilistic XML data. Incomplete XML has not received much attention so far, but significant work has been done on probabilistic XML. For example, in [126], the uncertainty associated with data obtained by probing the "hidden web" (i.e., data hidden behind query forms and web services) is represented using XML trees whose nodes are annotated with boolean expressions composed of independent

Bernoulli event variables.

Starting from these motivations, we develop an extension of the semiring annotation framework to XML and its premier query language, XQuery [42]. Because dealing with lists and ordered XML does not seem to be related to the way we use semirings, we focus on an *unordered* variant of XML. Chapter 2 provided strong evidence that the idea of using semirings to represent annotations is robust. In this chapter, we describe two new results that add to this body of evidence:

- We define the semantics for a large fragment of first-order, positive XQuery—practically all of the features that do not depend on order—on semiring-annotated XML in two different ways, and show that these agree. The first approach goes by translation to an extension of the nested relational calculus [19] (*NRC*),[1] while the second uses an encoding that "shreds" XML data into a *child* relation between node identifiers, and a corresponding translation of XPath into Datalog.
- We prove a general theorem showing that the semantics of queries commutes with the applications of semiring homomorphisms.

By instantiating our semantics using annotations formulated as polynomials over a fixed set of variables with coefficients in $\mathbb{N}$, we obtain our main contribution: a provenance framework for unordered XML data and a large class of XQuery views. We believe that this framework has practical potential: it captures an intuitive notion of provenance useful for scientific applications, and the size of the provenance polynomials is bounded by $O(|D|^{|q|})$ where D is the XML database and q is the XQuery program that defines the view.

Additionally, we illustrate two important applications of annotated XML: a security application that shows how to transfer *confidentiality policies* from a database to a view by organizing the clearance levels as a commutative semiring, and general *strong representation systems* for incomplete and probabilistic annotated databases that use the provenance polynomials themselves as annotations. The correctness of these systems follows from the commutation with homomorphisms theorem.

In outline, the chapter is organized as follows. Section 6.1 reviews the notion of commutative semiring annotations. Section 6.2 introduces the unordered XML data model (UXML) and the corresponding fragment of XQuery (UXQuery), and describes our extension of these formalisms with semiring annotations. We defer a formal discussion of the semantics of UXQuery to Section 6.5, but illustrate its behavior on several examples. We describe applications to security and incomplete and probabilistic data in Sections 6.3 and 6.4. The main technical results are collected in Section 6.5. There we review *NRC*, describe its extension to trees (6.5), define its semantics (6.5), give the compilation of UXQuery into this language (6.5), and state the commutation with homomorphism theorems (6.6). Section 6.7 presents an alternative definition for a fragment of UXQuery, via an encoding of UXML into relations and a translation of XPath into Datalog.

---

[1] Since *NRC* is used by itself in various contexts [14, 77], this semantics is of interest even without the connection to XML.

6.1 Semiring Annotations

As shown in Chapter 2, commutative semirings and relational data fit together naturally: when each tuple in a relation is tagged with an element of K, the semantics of standard query languages can be generalized to propagate the annotations in a way that captures bag semantics, probabilistic and incomplete relations, and standard notions of provenance. An (imperfect) intuition for the meaning of these annotations is as follows: 0 means that the tuple is not present or available; $k_1 + k_2$ means that the tuple can be produced from the data described by k_1 or that described by k_2; and the annotation $k_1 \cdot k_2$ means that it requires both the data described by k_1 and that described by k_2. The annotation 1 means that exactly one copy of the tuple is available "without restrictions." In the relational setting, we have seen that the axioms of commutative semirings are forced by standard equivalences on the (positive) relational algebra (Proposition 2.4). In this chapter, we show that commutative semirings also suffice for a variety of annotated nested data and their associated query languages.

We develop our theory for arbitrary commutative semirings, but use specific semirings in various applications:

- $(\mathbb{B}, \vee, \wedge, \mathsf{false}, \mathsf{true})$: set-based data;
- $(\mathbb{N}, +, \cdot, 0, 1)$: bag-based data;
- Positive boolean expressions: incomplete/probabilistic data (see [62] and Section 6.4);
- Confidentiality levels: see Section 6.3;
- Lineage and why-provenance (it turns out that these are different and correspond to different semirings, see [13]);
- $(\mathbb{N}[X], +, \cdot, 0, 1)$: a "universal" semiring of multivariate polynomials with coefficients in $\mathbb{N}$ and indeterminates in X.

As we have already argued in this dissertation, the polynomials in $\mathbb{N}[X]$ provide a very general and informative notion of provenanceand, in fact, capture the generality of *all* commutative semiring calculations: any function $X \longrightarrow K$ can be uniquely extended to a semiring homomorphism $\mathbb{N}[X] \longrightarrow K$. This fact is relevant to querying since (as in Chapter 2) by Theorem 6.5 and Corollary 6.6 below, our semantics for query answering commutes with applying homomorphisms to annotated data. This yields the principal result of our framework: a comprehensive notion of provenance for unordered XML and a corresponding fragment of XQuery.

6.2 Annotated and Unordered XML

We fix a commutative semiring K and consider XML data modified so that instead of lists of trees (sequences of elements) there are *sets* of trees. Moreover, each tree belonging to such a set is decorated with an annotation $k \in K$. Since *bags* of elements can be obtained by interpreting the annotations as multiplicities (by picking K to be $(\mathbb{N}, +, \cdot, 0, 1)$), the only difference compared to standard XML is the absence of ordering between siblings.[2] We call such data *K-annotated unordered XML*, or simply *K-UXML*. Given a domain $\mathcal{L}$ of *labels*, the usual mutually recursive definition of XML data naturally generalizes to *K-UXML*:[3]

- A *value* is either a label in $\mathcal{L}$, a tree, or a K-set of trees;
- A *tree* consists of a label together with a finite (possibly empty) K-set of trees as its "children";
- A finite *K-set of trees* is a function from trees to K such that all but finitely many trees map to 0.

In examples, we illustrate K-UXML data by adding annotations as a superscript notation on the label at the root of the (sub)tree. By convention *omitted annotations* correspond to the "neutral" element $1 \in K$.[4] Note that a tree gets an annotation only as a member of a K-set. To annotate a single tree, we place it in a singleton K-set. When the semiring of annotations is $(\mathbb{B}, \vee, \wedge, \mathsf{false}, \mathsf{true})$ we have essentially unannotated unordered XML; we write UXML instead of $\mathbb{B}$-UXML.

In Figure 6.1, two K-UXML data values are displayed as trees. The source value can be written in document style as

$<\mathsf{a}^{z}>$

$\quad <\mathsf{b}^{x_1}>\ \mathsf{d}^{y_1}\ </>$

$\quad <\mathsf{c}^{x_2}>\ \mathsf{d}^{y_2}\ \mathsf{e}^{y_3}\ </>$

---

[2] For simplicity, we also omit attributes and model atomic values as the labels on trees having no children.

[3] In the XQuery data model, sets of labels are also values; it is straightforward to extend our formal treatment to include this.

[4] Items annotated with 0 are allowed by the definition but are useless because our semantics interprets 0 as "not present/available".

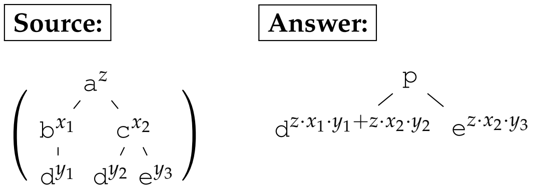


Figure 6.1: Simple `for` Example.

$$
\begin{array}{rcll}
l & \in & \mathcal{L} & \text{(labels)} \\
k & \in & K & \text{(annotations)} \\
p & ::= & l & \text{(label)} \\
 & | & \$x & \text{(variable)} \\
 & | & () & \text{(empty set)} \\
 & | & (p) & \text{(singleton set)} \\
 & | & p,p & \text{(set union)} \\
 & | & \texttt{for}\ \$x\ \texttt{in}\ p\ \texttt{return}\ p & \text{(iteration)} \\
 & | & \texttt{let}\ \$x := p\ \texttt{return}\ p & \text{(binding)} \\
 & | & \texttt{if}\ (p{=}p)\ \texttt{then}\ p\ \texttt{else}\ p & \text{(conditional)} \\
 & | & \texttt{element}\ p\ \{p\} & \text{(tree)} \\
 & | & \texttt{name}(p) & \text{(tree label)} \\
 & | & \texttt{annot}\ k\ p & \text{(scalar multiplication)} \\
 & | & p/s & \text{(navigation)} \\
s & ::= & ax::nt & \text{(step)} \\
ax & ::= & \texttt{self} \mid \texttt{child} \mid \texttt{descendant} & \text{(axes)} \\
nt & ::= & l \mid * & \text{(test)}
\end{array}
$$

Figure 6.2: *K*-UXQuery Syntax.

< />

where we have abbreviated leaves <*l*>< /> as *l*.

We propose a query language for *K*-UXML called *K-UXQuery*. Its syntax, listed in Figure 6.2, corresponds to a core fragment of XQuery [42] with one exception: the new construct `annot` *k p* allows queries to modify the annotations on sets. With `annot` *k p* any *K*-UXML value can be built with the *K*-UXQuery constructs.

We use the following types for *K*-UXML and *K*-UXQuery:

$$t \quad ::= \quad label \mid tree \mid \{tree\}$$

where *label* denotes $\mathcal{L}$, *tree* denotes the set of all trees and $\{tree\}$ denotes the set of all finite *K*-sets of trees. Typing rules for the operators of *K*-UXQuery are given in Figure 6.3.

At the end of this section we discuss this syntax in more detail, and in Section 6.5 we present a formal semantics that uses the operations of the semiring to combine annotations. In the rest of

$$\Gamma \vdash l : \mathit{label} \qquad \frac{\Gamma(x) = t}{\Gamma \vdash \$x : t} \qquad \Gamma \vdash () : \{\mathit{tree}\} \qquad \frac{\Gamma \vdash p : \mathit{tree}}{\Gamma \vdash (p) : \{\mathit{tree}\}} \qquad \frac{\Gamma \vdash p_1 : \{\mathit{tree}\} \qquad \Gamma \vdash p_2 : \{\mathit{tree}\}}{\Gamma \vdash p_1, p_2 : \{\mathit{tree}\}}$$

$$\frac{\Gamma \vdash p_1 : \{\mathit{tree}\} \qquad \Gamma, x : \mathit{tree} \vdash p_2 : \{\mathit{tree}\}}{\Gamma \vdash \texttt{for } \$x \texttt{ in } p_1 \texttt{ return } p_2 : \{\mathit{tree}\}} \qquad \frac{\Gamma \vdash p_1 : t_1 \qquad \Gamma, x : t_1 \vdash p_2 : t_2}{\Gamma \vdash \texttt{let } \$x := p_1 \texttt{ return } p_2 : t_2}$$

$$\frac{\Gamma \vdash p_1 : \mathit{label} \qquad \Gamma \vdash p_2 : \mathit{label} \qquad \Gamma \vdash p_3 : t \qquad \Gamma \vdash p_4 : t}{\Gamma \vdash \texttt{if } (p_1 = p_2) \texttt{ then } p_3 \texttt{ else } p_4 : t} \qquad \frac{G \vdash p_1 : \mathit{label} \qquad G \vdash p_2 : \{\mathit{tree}\}}{\Gamma \vdash \texttt{element } p_1 \ \{p_2\} : \mathit{tree}}$$

$$\frac{\Gamma \vdash p_1 : \mathit{tree}}{\Gamma \vdash \texttt{name}(p_1) : \mathit{label}} \qquad \frac{\Gamma \vdash p : \{\mathit{tree}\}}{\Gamma \vdash p/\mathit{ax}::\mathit{nt} : \{\mathit{tree}\}} \qquad \frac{\Gamma \vdash k \in K \qquad \Gamma \vdash p : \{\mathit{tree}\}}{\Gamma \vdash \texttt{annot } k \ p : \{\mathit{tree}\}}$$

Figure 6.3: K-UXQuery Typing Rules.

this section, however, we illustrate the semantics informally on some simple examples to introduce the basic ideas. We start with very simple queries demonstrating how the individual operators work, and build up to a larger example corresponding to a translation of a relational algebra query.

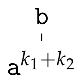
As a first example, define the following queries:

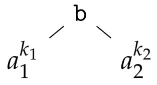
$$p_1 \overset{\text{def}}{=} \texttt{element } a_1 \ \{()\} \qquad \text{and} \qquad p_2 \overset{\text{def}}{=} \texttt{element } a_2 \ \{()\}$$

Each p_i constructs a tree labeled with a_i and having no children—i.e., a leaf node. The query (p_1) produces the singleton K-set in which p_1 is annotated with $1 \in K$. The query `annot` k_1 (p_1) produces the singleton K-set in which p_1 is annotated with $k_1 \cdot 1 = k_1$. We can also construct a union of K-sets:

$$q \overset{\text{def}}{=} \texttt{annot } k_1 \ (p_1), \texttt{annot } k_2 \ (p_2)$$

The result computed by q depends on whether a_1 and a_2 are the same label or different labels. If $a_1 = a_2 = \texttt{a}$, then p_1 and p_2 are the same tree and so the query then `element b` $\{q\}$ produces the left tree below. If $a_1 \neq a_2$, then the same query produces the tree on the right.





Next, let us examine a query that uses iteration:

$$\begin{aligned} p = \texttt{element p } \{ \ & \texttt{for \$t in \$S return} \\ & \quad \texttt{for \$x in (\$t)/* return} \\ & \qquad \texttt{(\$x)/* } \} \end{aligned}$$

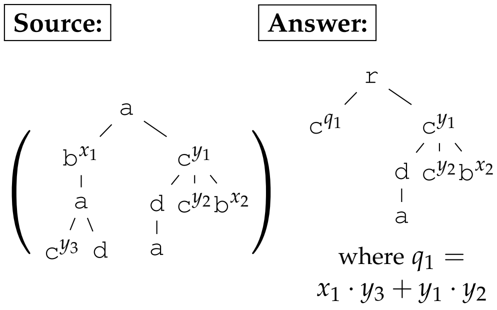
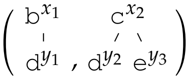


Figure 6.4: XPath Example.

If $S is the (source) set on the left side of Figure 6.1, then the answer produced by p is the tree on the right in the same figure.[5] Operationally, the query works as follows. First, the outer `for`-clause iterates over the set given by $S. As $S is a singleton in our example, $t is bound to the tree whose root is labeled a and annotation in $S is z. Next, the inner `for`-clause iterates over the set of trees given by ($t)/∗:



It binds $x to each of these trees, evaluates the `return`-clause in this extended context, and multiplies the resulting set by the annotation on $x. For example, when $x is bound to the b child, the `return`-clause produces the singleton set $(\mathtt{d}^{y_1})$. Multiplying this set by the annotation x_1 yields $(\mathtt{d}^{x_1 \cdot y_1})$. After combining all the sets returned by iterations of this inner `for`-clause, we obtain the set $(\mathtt{d}^{x_1 \cdot x_1 + x_2 \cdot y_2}, \mathtt{e}^{x_2 \cdot y_3})$. The final answer for p is obtained by multiplying this set by z. Note that the annotation on each child in the answer is the sum, over all paths that lead to that child in $t, of the product of the annotations from the root of $t to that child, thus recording *how* it arises from subtrees of $S.

Next we illustrate the semantics of XPath descendant navigation (shorthand //). Consider the query

$$r = \texttt{element r} \; \{ \; \texttt{\$T//c} \; \}$$

which picks out the set of subtrees of elements of $T whose label is c. A sample source and corresponding answer computed by r are shown in Figure 6.4. In Section 6.5 we define the semantics of the descendant operator using structural recursion and iteration. It has the property that the annotation for each subtree in the answer is the sum of the products of annotations for each path from the root to an occurrence of that subtree in the source, like the answer shown here.

---

[5] Actually this query is equivalent to the shorter "grandchildren" XPath query $S/∗/∗; we use the version with a `for`-clauses to illustrate the semantics of iteration.

Source and Answer as K-Relations:

R

A B C	
$a\ b\ c$	x_1
$d\ b\ e$	x_2
$f\ g\ e$	x_3

S

B C	
$b\ c$	x_4
$g\ c$	x_5

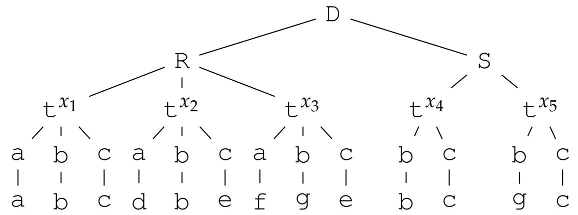
Q

A C	
$a\ c$	$x_1^2 + x_1 \cdot x_4$
$a\ e$	$x_1 \cdot x_2$
$d\ c$	$x_1 \cdot x_2 + x_2 \cdot x_4$
$d\ e$	x_2^2
$f\ c$	$x_3 \cdot x_5$
$f\ e$	x_3^2

Query:

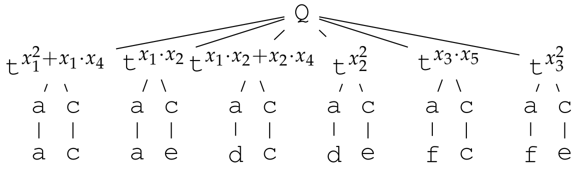
```
let $r := $d/R/∗,
    $rAB := for $t in $r return <t> { $t/A,$t/B } < />,
    $rBC := for $t in $r return <t> { $t/B,$t/C } < />,
    $s := $d/S/∗
return
 <Q> { for $x in $rAB,$y in ($rBC,$s)
       where $x/B=$y/B
       return <t> { $x/A,$y/C } < /> } < />
```

Source as UXML:



D
R
S
t^{x_1}
t^{x_2}
t^{x_3}
t^{x_4}
t^{x_5}
a b c a b c a b c b c b c
a b c d b e f g e b c g c

Answer as UXML:



Q
$t^{x_1^2+x_1 \cdot x_4}$
$t^{x_1 \cdot x_2}$
$t^{x_1 \cdot x_2+x_2 \cdot x_4}$
$t^{x_2^2}$
$t^{x_3 \cdot x_5}$
$t^{x_3^2}$
a c a c a c a c a c a c
a c a e d c d e f c f e

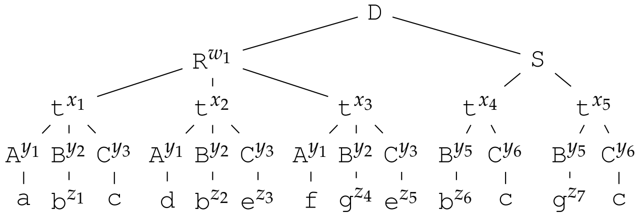
Figure 6.5: Relational (encoded) example.

Now we turn to a larger example, which demonstrates how K-UXQuery behaves on an encoding of a database of relations whose tuples are annotated with elements of K, i.e., the *K-relations* of Chapter 2. As a sanity check, we verify that our semantics for K-UXQuery on this data agrees with the semantics given for the positive relational algebra given earlier. Consider the following relational algebra query

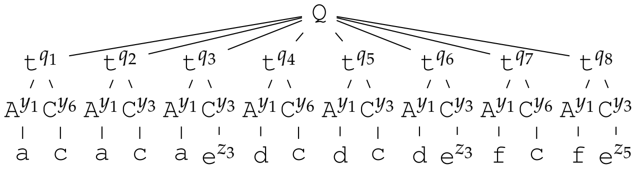
$$Q = \pi_{AC}(\pi_{AB}(R) \bowtie (\pi_{BC}(R) \cup S))$$

and suppose that we evaluate it over K-relations $R(\text{A}, \text{B}, \text{C})$ and $S(\text{B}, \text{C})$ shown at the top of Figure 6.5. The result is the K-relation $Q(\text{A}, \text{C})$, also shown at the top of Figure 6.5. For example, the annotation on $\langle d\ c\rangle$ in Q is a sum of products $x_1 \cdot x_2 + x_2 \cdot x_4$, which records that the tuple can be obtained by joining two R-tuples or, alternatively, by joining an R-tuple and an S-tuple. The rest of Figure 6.5 shows the K-UXML tree that is obtained by encoding the relations R and S, the

Source:


D
R^w1
S
t^x1 t^x2 t^x3 t^x4 t^x5
A^y1 B^y2 C^y3 A^y1 B^y2 C^y3 A^y1 B^y2 C^y3 B^y5 C^y6 B^y5 C^y6
a b^z1 c d b^z2 e^z3 f g^z4 e^z5 b^z6 c g^z7 c

Answer:


Q
t^q1 t^q2 t^q3 t^q4 t^q5 t^q6 t^q7 t^q8
A^y1 C^y6 A^y1 C^y3 A^y1 C^y3 A^y1 C^y6 A^y1 C^y3 A^y1 C^y3 A^y1 C^y6 A^y1 C^y3
a c a c a e^z3 d c d c d e^z3 f c f e^z5

$$
\begin{aligned}
\text{where } q_1 &= w_1 \cdot x_1 \cdot x_4 \cdot y_2 \cdot y_5 \cdot z_1 \cdot z_6 \\
q_2 &= w_1^2 \cdot x_1^2 \cdot y_2^2 \cdot z_1^2 \\
q_3 &= w_1^2 \cdot x_1 \cdot x_2 \cdot y_2^2 \cdot z_1 \cdot z_2 \\
q_4 &= w_1 \cdot x_2 \cdot x_4 \cdot y_2 \cdot y_5 \cdot z_2 \cdot z_6 \\
q_5 &= w_1^2 \cdot x_1 \cdot x_2 \cdot y_2^2 \cdot z_1 \cdot z_2 \\
q_6 &= w_1^2 \cdot x_2^2 \cdot y_2^2 \cdot z_2^2 \\
q_7 &= w_1 \cdot x_3 \cdot x_5 \cdot y_2 \cdot y_5 \cdot z_4 \cdot z_7 \\
q_8 &= w_1^2 \cdot x_3^2 \cdot y_2^2 \cdot z_4^2
\end{aligned}
$$

Figure 6.6: Extended Annotations Example.

corresponding translation of the view definition into K-UXQuery, and the K-UXML view that is computed using K-UXQuery. Observe that the result is the encoding of the K-relation Q.

The next proposition shows that this equivalence holds in general. Fix a semiring K, a set of attributes U, and domain of values $\mathbb{D}$. Recall from Chapter 2 that a K-relation is a function from tuples to K with finite support–i.e., such that all but finitely many tuples map to $0 \in K$—where a tuple t is a function from U to $\mathbb{D}$. The K-UXML encoding of a tuple $t \in U \to \mathbb{D}$ is a tree with root labeled `t` (the label `t` is a fixed constant), and a child u for each $u \in U$ leading to a leaf labeled $t(u) \in \mathbb{D}$. All of the annotations in the encoding of a tuple are $1 \in K$. The K-UXML encoding of a K-relation R is a tree with root labeled `R` (again, the label `R` is a fixed constant), and a child for each tuple t. The annotation on each tuple is $R(t) \in K$. Finally, the corresponding translation of positive relational algebra queries into K-UXQuery is given in Figure 6.7.[6] (Note that the translation assumes a translation of selection predicates P into K-UXQuery.)

Proposition 6.1. *Let Q be be a query in positive relational algebra, and R a K-relation. Let v be the K-UXML encoding of R, and let $p = (\!|Q|\!)$ be the translation of Q into K-UXQuery. Then $p(v)$, computed*

---

[6] This translation works on a single relation; extending it to whole instances, as in the example in Figure 6.5, is straightforward.

```
$(\!|\emptyset|\!)$                          $\stackrel{\text{def}}{=}$ element R {()}
$(\!|Q_1 \cup Q_2|\!)$                       $\stackrel{\text{def}}{=}$ element R {((($(\!|Q_1|\!)$)/*,($(\!|Q_2|\!)$)/*)}
$(\!|\pi_{\{v_1,\ldots,v_k\}}(Q_1)|\!)$      $\stackrel{\text{def}}{=}$ element R {
                                               for $t in (($(\!|Q_1|\!)$)/* return
                                               (element t {
                                                 for $u in ($t)/* return
                                                   if name($u) = $v_1$ then $u
                                                   else ...
                                                   else if name($u) = $v_k$ then $u
                                                   else ()
                                               })
                                             }
$(\!|\sigma_P(Q_1)|\!)$                      $\stackrel{\text{def}}{=}$ element R {
                                               for $t in (($(\!|Q_1|\!)$)/* return
                                                 if $(\!|P|\!)$($t) then $t else ()
                                               })
                                             }
$(\!|Q_1 \times Q_2|\!)$                     $\stackrel{\text{def}}{=}$ element R {
                                               for $t_1 in (($(\!|Q_1|\!)$)/* return
                                               for $t_2 in (($(\!|Q_2|\!)$)/* return
                                                 (element t {(($t_1)/*,($t_2)/*)})
                                             }
$(\!|\rho_{(u_i:=u_j)}(Q_1)|\!)$             $\stackrel{\text{def}}{=}$ element R {
                                               for $t in (($(\!|Q_1|\!)$)/* return
                                               (element t {
                                                 for $u in ($t)/* return
                                                   if (name($u)=$u_i$) then element $u_j$ {($u)/*} else $u
                                               })
                                             }
```

Figure 6.7: Translation from positive relational algebra to *K*-UXQuery

according to K-UXQuery, encodes $[\![Q]\!]^R$*, the K-relation computed according Definition 2.2.*

Proof. By straightforward induction on Q. ☐

In a *K*-relation, annotations *only* appear on tuples. In our model for annotated UXML data, however, every internal node carries an annotation (recall that, according to our convention, every node in Figure 6.5 depicted with no annotation carries the "neutral" element $1 \in K$). Therefore, we have more flexibility in how we annotate source values—besides tuples, we can place annotations on the values in individual fields, on attributes on the relations themselves, and even on the whole database! It is interesting to see how, even for a query that is essentially relational, these extra annotations participate in the calculations. We have worked this out in the final example of this section, see Figure 6.6. The query is the same as in Figure 6.5 but the source data has additional annotations. Note how the expressions annotating the tuple nodes in the answer involve many non-tuple annotations from the source.

So far we have assumed that the annotations belong to an arbitrary commutative semiring K and we looked at the expressions that equate $q_1, \ldots, q_8$ in Figure 6.6 as calculations in K. However, if we work with the semiring of polynomials $K = (\mathbb{N}[X], +, \cdot, 0, 1)$ where we think of the source annotations as indeterminates ("provenance tokens") and take

$$X := \{w_1, x_1, \ldots, x_5, y_1, \ldots, y_6, z_1, \ldots, z_7\}$$

then the expressions that equate $q_1, \ldots, q_8$ are the *provenance polynomials* that annotate the tuple nodes in the answer. This kind of provenance shows, for example, that some of the tuples in the answer use source data annotated with z_1 or y_5 although these do not appear explicitly in the annotations of the answer attributes or values in the tuples. The annotations in a particular semiring K can then be computed by evaluating these polynomials in K. Corollary 6.6 (commutation with homomorphisms) guarantees that the result will be the same as that obtained via the semantics on K-UXML values.

Note also that we can obtain the answer shown in Figure 6.5 simply by setting all the indeterminates except for $x_1, \ldots, x_5$ to 1 and then simplifying using the semiring laws. When we set these indeterminates to 1, some subtrees which were distinguished by annotations become now identified (q_1 and q_2, q_4 and q_5); this explains the sums in the annotations of the answer in Figure 6.5.

The semantics in Section 6.5 allows us to prove the following upper bound:

Proposition 6.2. *If v is a UXML value annotated with indeterminates from a set X and p is a UXQuery, then computing $p(v)$ according to the $\mathbb{N}[X]$-UXQuery semantics produces an $\mathbb{N}[X]$-UXML value such that the size of any of the provenance polynomials that annotate $p(v)$ is $O(|v|^{|p|})$.*

Proof. □

K-UXQuery vs. XQuery

Although UXQuery only contains core operators, more complicated syntactic features such as `where`-clauses that we used in the examples above can be *normalized* into core queries using standard translations [42]. For example, the `where`-clause `where $x/B=$y/B` from Figure 6.5 normalizes to:

```
for $a in $x/B/* return
for $b in $y/B/* return
  if (name($a)=name($b)) then ... else ()
```

Our language includes only the downward XPath axes, since the other axes can be compiled into this fragment [117]. To simplify our formal system, we also do not identify a value with the singleton set containing it. This is inessential but it simplifies the compilation in Section 6.5. In examples we often elide the extra set constructor when it is clear from context—e.g., we wrote `$x/A` above, not `($x)/A`.

Unlike these minor differences, we made two essential restrictions in the design of K-UXQuery. The first has to do with order—we omit `orderby` and other operators whose semantics depends on position, since these do not make sense on unordered data. The second essential restriction is to *positive* queries—e.g., the conditional expression only tests the equality of labels; see Section 6.5 for further discussion.

6.3 A Security Application

We can model *confidentiality policies* using commutative semirings. For example, the total order $\mathcal{C}: \mathsf{P} < \mathsf{C} < \mathsf{S} < \mathsf{T} < 0$ describes the following levels of "clearance": P = public, C = confidential, S = secret, and T = top-secret. It is easy to see that $(\mathcal{C}, \min, \max, 0, \mathsf{P})$ is a commutative semiring.[7] We add 0 as a separate element. It is needed because items in a K-UXML set with annotation 0 are interpreted as *not* belonging to the set (i.e., 0 is so secret, it isn't even there!), and we do not want to lose data tagged as T completely.

Our framework solves the following problem. Suppose that an XML database has been manually annotated with security information specifying what clearance one must have for each data subtree they wish to see. Now we use XQuery to produce views of this database. We would like to compute automatically clearance annotations for the data in a view, based on *how* that data was obtained from the already annotated data in the original database. The two operations of the clearance semiring correspond to *alternatives* in obtaining the view data, in which case the minimum clearance among them suffices, and to *joint necessities*, in which case the maximum clearance among them is needed.

We give an example that shows that our annotated XML model is a particularly flexible framework for such clearance specifications. Consider the source data in Figure 6.6, which in fact encodes a relational database but where we have much more annotation flexibility than in the [62] model where only tuples are annotated. We annotate with elements from $\mathcal{C}$ as follows $w_1 := \mathsf{C}$ (the entire relation `R` is confidential), $x_2 := \mathsf{S}$ (in addition, this tuple is secret), and $y_5 := \mathsf{T}$ (all values of attribute `B` in relation `S` are top-secret), and finally, the rest of the annotations are P (which plays the role of 1 in this semiring).

---

[7] Note that the natural order (cf. Chapter 2) on this semiring is actually the opposite of the clearance order.

Q

A C	
$a\ c$	$w_1 \cdot y_5 + w_1^2 = \mathsf{C} \cdot \mathsf{T} + \mathsf{C}^2 = \mathsf{C}$
$a\ e$	$w_1^2 \cdot x_2 = \mathsf{C}^2 \cdot \mathsf{S} = \mathsf{S}$
$d\ c$	$w_1 \cdot x_2 \cdot y_5 + w_1^2 \cdot x_2 = \mathsf{C} \cdot \mathsf{S} \cdot \mathsf{T} + \mathsf{C}^2 \cdot \mathsf{S} = \mathsf{S}$
$d\ e$	$w_1^2 \cdot x_2^2 = \mathsf{C}^2 \cdot \mathsf{S} = \mathsf{S}$
$f\ c$	$w_1 \cdot y_5 = \mathsf{C} \cdot \mathsf{T} = \mathsf{T}$
$f\ e$	$w_1^2 = \mathsf{C}^2 = \mathsf{C}$

Figure 6.8: Security Clearance Example.

The result of the view/query in Figure 6.5 when applied to this data is the $\mathcal{C}$-UXML encoding of a relation in which only the annotations on the tuples are different from $1 = \mathsf{P}$ (this is because the query projects out the attribute `B`, otherwise we could have had non-P annotations inside the tuples). We show this answer as an annotated relation in Figure 6.8. We also show there the polynomials that would annotate the tuples if we would do the calculations in the provenance semiring $\mathbb{N}[w_1, x_2, y_5]$. These help understand *how* the resulting clearances are computed since it is a consequence of Corollary 6.6 (commutation with homomorphisms) that by evaluating the provenance polynomials in $\mathcal{C}$ under the valuation $w_1 := \mathsf{C}, x_2 := \mathsf{S}, y_5 := \mathsf{T}$ we get the same result as the $\mathcal{C}$-UXQuery semantics.

Going back to the security application, for the data in the view, confidential clearance gives access to the first and last tuple, secret clearance to all but the fifth tuple, etc. Note how the top-secret annotation of the attribute `B` in `S` affects just three of the tuples in the answer and how in two of those cases the tuples are still available to lower clearances because they can be also produced with data from `R` only.

In the example above the semiring of clearances is a total order but this can be generalized to non-total orderings, provided they form a *distributive lattice*. The distributivity ensures that views that we consider equivalent actually compute the same clearance for the results. This follows from the following proposition which generalizes a similar result in [56] (later rediscovered in [62]) for relations and positive relational algebra.

Proposition 6.3. *If two UXQueries are equivalent on all UXML inputs and K is a distributive lattice then the queries are equivalent on all K-annotated UXML inputs.*

Proof. □

One way to establish the end-to-end correctness of an access control scheme is by proving a *non-interference* property [122]. Informally, non-interference guarantees that the low-security parts of the output of a computation will not depend on high-security data in the input. In our framework, the non-interference property follows from Theorem 6.5, which shows that semiring homomorphisms commutes with query evaluation. To see this, fix a clearance level l, and let h be the "erasing" semiring homomorphism—i.e., from $\mathcal{C}$ to $\mathbb{B}$—that maps clearances strictly above l to 0 and everything else to 1. By Theorem 6.5 we have that, for an input K-UXML document consisting of a mixture of high- and low-security data, erasing the high-security parts and evaluating the query yields the same result as evaluating the query on the full document and then erasing the high-security parts.

6.4 Incomplete and Probabilistic K-UXML

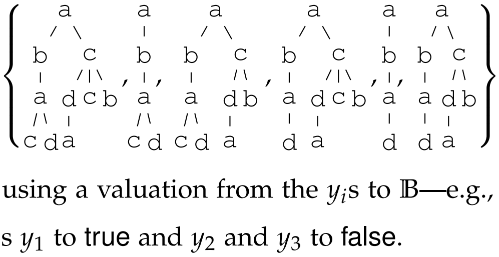
Commutative semirings can also be used to model incomplete and probabilistic databases for unordered XML data, even with repetitions. An incomplete UXML database is a set of *possible worlds*, each of which is itself a UXML (i.e. a $\mathbb{B}$-UXML) database. For repetitions (multiplicities) the possible worlds are $\mathbb{N}$-UXML databases. More generally, we treat here incomplete K-UXML databases for arbitrary commutative semirings K. It turns out that by using provenance annotations we can construct a powerful system for representing and querying incomplete K-UXML databases.

Recall that provenance polynomials are elements of the commutative semiring $(\mathbb{N}[X], +, \cdot, 0, 1)$—i.e. polynomial expressions over variables X with natural number coefficients. For any commutative semiring K, provenance polynomials are "universal" in the sense that any function $f : X \to K$ (we call f a *valuation*) extends uniquely to a semiring homomorphism $f^\star : \mathbb{N}[X] \to K$. We exploit this to construct a representation system for incomplete K-UXML data. We first fix a semiring K, and a set of variables X. We call a v in $\mathbb{N}[X]$-UXML a *representation*. Next we define a function Rep_K that maps a representation v in $\mathbb{N}[X]$-UXML to the *set* of K-UXML instances that can be obtained by applying K-valuations to the variables in X—i.e., $\mathrm{Rep}_K(v)$ is $\{f^\star(v) : f : X \to K\}$, the set of *possible worlds* v represents.

As an example, let v be the source tree in Figure 6.4. To streamline the example, we will set the x_1 and x_2 annotations to 1, leaving just the annotations y_1, y_2, y_3 on the subtrees labeled `c`.

For $K = \mathbb{B}$, the set of possible worlds represented by v is the following set of UXML values:

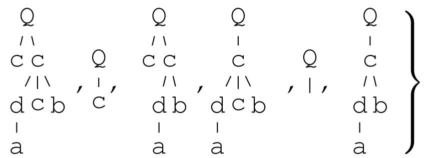
$\mathrm{Rep}_{\mathbb{B}}(v) =$



Each tree in $\mathrm{Rep}_{\mathbb{B}}(v)$ is obtained using a valuation from the y_is to $\mathbb{B}$—e.g., for the rightmost tree in this display, the valuation maps y_1 to true and y_2 and y_3 to false.

Now consider querying such an incomplete UXML database. In general, given an XQuery p, we would like the answer to be (semantically) the set of all K-UXML instances obtained by evaluating p over each K-UXML instance in the set of possible worlds represented by v—i.e., $p(\mathrm{Rep}_K(v))$ is $\{p(v') : v' \in \mathrm{Rep}_K(v)\}$. Returning to the representation v above and using p, the query in Figure 6.4, we have:

$$p(\mathrm{Rep}_{\mathbb{B}}(v)) =$$

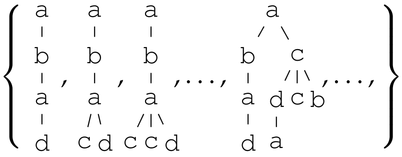


As usual in incomplete databases, we do not wish to return this set, which may be large in general. Instead, we would like a *representation* of it. By Corollary 6.6 below, it turns out that such a representation is obtained by evaluating p over v with $\mathbb{N}[X]$-UXQuery semantics. In general, we have that $p(\mathrm{Rep}_K(v)) = \mathrm{Rep}_K(p(v))$. Indeed, the specific answer for this example shown in Figure 6.4 is the representation of $p(\mathrm{Rep}_{\mathbb{B}}(v))$. Using the terminology of incomplete databases, we say that $\mathbb{N}[X]$-UXML is a *strong representation system* [79, 2] for K-UXQuery and K-UXML data.

For simpler K, the full power of $\mathbb{N}[X]$ may not be needed. For example, when $K = \mathbb{B}$, we can use annotations from the semiring $(\mathsf{PosBool}(B), \vee, \wedge, \mathsf{false}, \mathsf{true})$ of positive Boolean expressions over a set B of variables (i.e., the expressions involve only B, disjunction, conjunction, and constants for true and false).[8] This corresponds to an XML analogue of the Boolean c-tables [79] used in incomplete databases. Valuations $\nu : B \to \mathbb{B}$ extend uniquely to homomorphisms $\nu^\star : \mathsf{PosBool}(B) \to \mathbb{B}$, so the definition above of $\mathrm{Rep}_{\mathbb{B}}$ still makes sense. Indeed, it follows (again from the commutation with homomorphisms in Section 6.6) that $\mathsf{PosBool}(B)$-UXML is a *strong representation system* for UXQuery and $\mathbb{B}$-UXML (i.e., ordinary UXML) and that we can transform an $\mathbb{N}[B]$-UXML representation into $\mathsf{PosBool}(B)$-UXML representation by applying the obvious homomorphism. It can be shown that PosBool works not just for $\mathbb{B}$ but for incomplete L-UXML for any distributive lattice L, in particular the ones used for the security application in Section 6.3.

Another instance of our general result is that $\mathbb{N}[X]$-UXML also provides a strong representation system for UXML with repetitions. For example, if we let v be the same tree as above, and pick $K = \mathbb{N}$, then the set of possible worlds is the following:

$$\mathrm{Rep}_{\mathbb{N}}(v) =$$



---

[8] We also identify those expressions which yield the same truth value for all Boolean assignments of the variables in B (to permit simplifications).

Note that children may be repeated—e.g., the third tree in this display has a subtree with two children `c`; this is obtained from a valuation that maps y_2 to 2.

Probabilistic data can also be modeled using semiring annotations. Again we use as representations $\mathbb{N}[X]$-UXML values and *all* the worlds corresponding to valuations $f : X \to K$. But now we consider such a valuation as the conjunction of independent events, $\{f(x) = k\}$ one for each x. The probability of each independent event can be computed from some probability distribution on K. For example, if $K = \mathbb{B}$ we can use Bernoulli distributions, if $K = \mathbb{N}$ we can use $\mathbf{Pr}[f(x) = n] = 1/2^n$ for $n > 0$, and 0 for $f(x) = 0$, etc. It follows again that we have a *strong representation system* this time for probability distributions on all the possible instances. For $K = \mathbb{B}$, more generally for distributive lattices, it suffices again to use PosBool expressions. Since tree pattern queries are expressible in UXQuery, we get the query evaluation algorithm in [126] as a particular case.

6.5 Semantics via Complex Values

In this section we develop our formal semantics for K-UXQuery by translation into a data model and query language for complex values. Trees can be understood as data values built recursively using pairing and collection constructions (see e.g., [16, 121]). For UXML trees, the collections are sets. This suggests defining trees as *complex values*, as data values built using pairing and sets, nested arbitrarily.

We develop our semantics in several steps. First, we generalize the semantics of *NRC* to handle semiring-annotated values. We then extend the calculus with a recursive tree type and structural recursion operator on trees. This operator is needed to express the `descendant` operator of K-UXQuery.[9] Finally, we use this calculus as a compilation target for K-UXQuery.

Annotated Complex Values

We start from the (positive) Nested Relational Calculus [19]. The types of *NRC* are:

$$t \quad ::= \quad \mathit{label} \mid t \times t \mid \{t\}$$

Complex values are built with the following constructors:

$$v \quad ::= \quad l \mid (v, v) \mid \{v\} \mid v \cup v \mid \{\}$$

We abbreviate $\{v_1\} \cup \cdots \cup \{v_n\}$ as $\{v_1, \ldots, v_n\}$—e.g., $(l_1, \{l_2, l_3\})$ is a complex value of type $\mathit{label} \times \{\mathit{label}\}$.

---

[9] When the nesting depth of the XML documents is bounded, the structural recursion operator (and the recursive tree type) are not needed, see [37].

$$
\begin{array}{rcll}
e & ::= & l & \text{(atom)} \\
 & | & x & \text{(variable)} \\
 & | & \{\} & \text{(empty set)} \\
 & | & \{e\} & \text{(singleton)} \\
 & | & e \cup e & \text{(union)} \\
 & | & \cup(x \in e)\ e & \text{(big-union)} \\
 & | & \mathit{let}\ x = e\ \mathit{in}\ e & \text{(let)} \\
 & | & \mathit{if}\ e = e\ \mathit{then}\ e\ \mathit{else}\ e & \text{(if)} \\
 & | & (e, e) & \text{(pair)} \\
 & | & \pi_1(e) & \text{(first projection)} \\
 & | & \pi_2(e) & \text{(second projection)}
\end{array}
$$

$$
\Gamma \vdash l : \mathit{label} \qquad \frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad \Gamma \vdash \{\} : \{t\} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \{e\} : \{t\}} \qquad \frac{\Gamma \vdash e_1 : \{t\} \qquad \Gamma \vdash e_2 : \{t\}}{\Gamma \vdash e_1 \cup e_2 : \{t\}}
$$

$$
\frac{\Gamma \vdash e_1 : \{t_1\} \qquad \Gamma, x : t_1 \vdash e_2 : \{t_2\}}{\Gamma \vdash \cup(x \in e_1)\ e_2 : \{t_2\}} \qquad \frac{\Gamma \vdash e_1 : \mathit{label} \qquad \Gamma \vdash e_2 : \mathit{label} \qquad \Gamma \vdash e_3 : t \qquad \Gamma \vdash e_4 : t}{\Gamma \vdash \mathit{if}\ e_1 = e_2\ \mathit{then}\ e_3\ \mathit{else}\ e_4 : t}
$$

$$
\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad \frac{\Gamma \vdash e : t_1 \times t_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \pi_i(e) : t_i}
$$

Figure 6.9: Syntax and Typing rules for *NRC*

The syntax and typing rules for the fragment of *NRC* we consider is given in Figure 6.9. The restriction to *positive* expressions is embodied in the typing rule for conditionals—we only compare label values. It is shown in [19] that equality tests for arbitrary sets can be used to define non-monotonic operations (i.e., difference, intersection, membership, and nesting). This restriction is essential for the semantics of *NRC* on annotated complex values because semirings do not contain features for representing negation.

The crucial *NRC* operation is the big-union operator: $\cup(x \in e_1)\ e_2$. It computes the union of the family of sets defined by e_2 indexed by x, where x takes each value in the set e_1. For example, the first relational projection is expressed as follows

$$\mathsf{project}_1\ R \overset{\text{def}}{=} \cup(x \in R)\ \{\pi_1(x)\}.$$

Next we show how to decorate complex values with semiring annotations, and give a semantics that works on annotated values. Again we fix a commutative semiring $(K, +, \cdot, 0, 1)$. Dealing with complex values annotated with elements from K requires a different semantics for the type $\{t\}$. The usual semantics is the set of finite subsets of $[\![t]\!]$. Instead, the semantics of $[\![\{t\}]\!]_K$ is defined as the set of functions $f : [\![t]\!]_K \to K$ with *finite support*, i.e., such that $\mathsf{supp}(f) := \{a \in [\![t]\!]_K \mid f(a) \neq 0\}$ is finite. We call elements of $[\![\{t\}]\!]_K$ *K-collections*. With $K = \mathbb{B}$ we obtain the usual semantics as finite subsets; with $K = \mathbb{N}$ we get bags.

K-complex values are obtained by arbitrarily nesting pairing and K-collections. We define new semantics for the *NRC* constructors: the singleton constructor $[\![\{v\}]\!]_K$ is the function that maps $[\![v]\!]_K$ to 1 and everything else to 0; $[\![\{\}]\!]_K$ is the constant function that maps everything to 0; and $[\![v_1 \cup v_2]\!]_K$ is the pointwise K-addition of $[\![v_1]\!]_K$ and $[\![v_2]\!]_K$. In order to express all K-collections in the calculus, we extend *NRC* with an operation for multiplying the annotations on the elements of K-collections by the "scalar" k in K. It is written $k\,e$ and has the following typing rule:

$$\frac{\Gamma \vdash k \in K \qquad \Gamma \vdash e : \{t\}}{\Gamma \vdash k\,e : \{t\}}$$

We call the calculus extended with this operator NRC_K. The set of K-complex values are constructed using:

$$v \quad ::= \quad l \mid (v,v) \mid k\,\{v\} \mid v \cup v \mid \{\}$$

and, as above, we abbreviate K-collections using the following notation: $\{v_1^{k_1}, \ldots, v_n^{k_n}\} \overset{\text{def}}{=} k_1\,\{v_1\} \cup \cdots \cup k_n\,\{v_n\}$. Determining the right semantics for the $\cup(x \in e_1)\;e_2$ operation is more challenging.

Let e_1 have type $\{t_1\}$ and e_2 have type $\{t_2\}$ (whenever x has type t_1). Let $X = [\![t_1]\!]_K$ and $Y = [\![t_2]\!]_K$. Then $[\![e_1]\!]_K$ is a function $f : X \to K$ with finite support $\mathsf{supp}(f) = \{x_1, \ldots, x_n\}$. In general e_2 depends on x so for each x_i we have a corresponding semantics for e_2, i.e., a function $g_i : Y \to K$. Using this function we define for each $y \in Y$

$$[\![\cup(x \in e_1)\;e_2]\!]_K(y) \overset{\text{def}}{=} \sum_{i=1}^{n} f(x_i) \cdot g_i(y)$$

Since each g_i has finite support, so does $[\![\cup(x \in e_1)\;e_2]\!]_K$.

The semantics of the other operations inherited from positive *NRC* is straightforward (it is essential that the equality test does not involve K-collections and therefore additional annotations). For example,

$$\begin{aligned} \mathsf{flatten}\,\{\{a^p, b^r\}^u, \{b^s\}^v\} &= \{a^{u \cdot p}, b^{u \cdot r + v \cdot s}\} \\ \{a^p, b^r\} \times \{c^u\} &= \{(a,c)^{p \cdot u}, (b,c)^{r \cdot u}\} \end{aligned}$$

where $R \times S \overset{\text{def}}{=} \cup(x \in R)\cup(y \in S)\;(x,y)$.

We take the fact that the semantics of NRC_K is an instance of the general approach to collection languages promoted in e.g., [19, 99, 12] as evidence for the robustness of our semantics. Appendix 6.7 gives a set of equational axioms for NRC_K that follow from the general approach just mentioned. These axioms also form a foundation for query optimization for NRC_K and K-UXQuery (e.g., see [120]).

As positive *NRC* strictly extends the positive relational algebra (RA+), the following sanity check is also in order.

Proposition 6.4. *Let NRC(RA+) be the usual encoding of projection, selection, cartesian product and union in (positive) NRC. The semantics of NRC(RA+) on K-complex values representing K-relations coincides with the semantics of RA+ on K-relations given in Chapter 2 (Definition 2.2).*

As another sanity check, observe that $NRC_{\mathbb{N}}$ corresponds to the *positive* fragment of the *Nested Bag Calculus* [105].

Adding Trees: $\textbf{\textit{NRC}} + srt$

To represent trees, we extend the calculus with a constructor $\mathsf{Tree}(a, C)$ where a is the label and C the set of immediate subtrees, as shown in . Trees of the form $\mathsf{Tree}(a, \{\})$ are leaves. The typing rule for the tree constructor is given by:

$$\frac{\Gamma \vdash v_1 : \mathit{label} \qquad \Gamma \vdash v_2 : \{\mathit{tree}\}}{\Gamma \vdash \mathsf{Tree}(v_1, v_2) : \mathit{tree}}$$

where *tree* is a new type. It is easy to see that the values of type *tree* and $\mathit{label} \times \{\mathit{tree}\}$ are in a 1-1 correspondence. In one direction this isomorphism is witnessed by $\mathsf{Tree}(\pi_1(P), \pi_2(P))$, where P is a pair. To express the other direction, we extend the calculus with two new operations, $\mathit{tag}(-)$ and $\mathit{kids}(-)$ that return the root tag and the set of subtree children of the root, respectively. The mapping from trees to pairs is then given by $(\mathit{tag}(T), \mathit{kids}(T))$, where T is a tree. Hence, *semantically*, the *tree* type is *recursive*.[10] In the spirit of [121] we add an operation for *structural recursion* on trees:

$$\frac{\Gamma, x : \mathit{label}, y : \{t\} \vdash e_1 : t \qquad \Gamma \vdash e_2 : \mathit{tree}}{\Gamma \vdash (\mathit{srt}(x, y).\, e_1)\ e_2 : t}$$

Its semantics obeys the equation

$$\begin{aligned} &(\mathit{srt}(x,y).\, e_1)\ \mathsf{Tree}(e_2, e_3)\ = \\ &\qquad e_1[x := e_2, y := \cup(z \in e_3)\ (\mathit{srt}(x,y).\, e_1)\ z] \end{aligned} \tag{6.1}$$

where the notation $e[x := e']$ denotes substitution of e' for x in e. For example, the query

$$(\mathit{srt}(x,y).\, \{x\} \cup \mathsf{flatten}\ y)\ t$$

where $\mathsf{flatten}\ W \overset{\text{def}}{=} \cup(w \in W)\ w$ returns the set of atoms in t.

---

[10] $\mathsf{Tree}(-,-), \mathit{tag}(-)$ and $\mathit{kids}(-)$ are an instance of a standard technique for handling recursive types in functional languages.

To summarize, we extend the syntax of NRC_K with the following constructs:

$$\begin{array}{rcll} e & ::= & ... & \\ & \mid & \mathsf{Tree}(e, e) & \text{(tree)} \\ & \mid & tag(e) & \text{(tag)} \\ & \mid & kids(e) & \text{(children)} \\ & \mid & (srt(x, y).\, e)\ e & \text{(recursion)} \end{array}$$

The typing rules for these new constructs are as follows:

$$\frac{\Gamma \vdash e_1 : label \qquad \Gamma \vdash e_2 : \{tree\}}{\Gamma \vdash \mathsf{Tree}(e_1, e_2) : tree} \qquad \frac{\Gamma \vdash e_1 : tree}{\Gamma \vdash tag(e_1) : label} \qquad \frac{\Gamma \vdash e_1 : tree}{\Gamma \vdash kids(e_1) : \{tree\}}$$

$$\frac{\Gamma, x : label, y : \{t\} \vdash e_1 : t \qquad \Gamma \vdash e_2 : tree}{\Gamma \vdash (srt(x, y).\, e_1)\ e_2 : t}$$

We denote this query language by $NRC + srt$.

$$[\![l]\!]^\rho_K = l \qquad [\![x]\!]^\rho_K = \rho(x) \qquad [\![\{\}]\!]^\rho_K(x) = 0_K \qquad [\![\{e\}]\!]^\rho_K(x) = \mathit{if}\ x = [\![e]\!]^\rho_K\ \mathit{then}\ 1_K\ \mathit{else}\ 0_K$$

$$[\![e_1 \cup e_2]\!]^\rho_K(x) = [\![e_1]\!]^\rho_K(x) + [\![e_2]\!]^\rho_K(x) \qquad \frac{[\![e_1]\!]^\rho_K = s_1}{[\![\bigcup(x \in e_1)\ e_2]\!]^\rho_K(y) = \sum_{v \in \mathsf{dom}(s_1)} s_1(v) \cdot [\![e_2]\!]^{\rho[x \leftarrow v]}_K(y)}$$

$$\frac{\mathit{if}\ [\![e_1]\!]^\rho_K = [\![e_2]\!]^\rho_K\ \mathit{then}\ [\![e_3]\!]^\rho_K\ \mathit{else}\ [\![e_4]\!]^\rho_K = v}{[\![\mathit{if}\ e_1 = e_2\ \mathit{then}\ e_3\ \mathit{else}\ e_4]\!]^\rho_K = v} \qquad \frac{[\![e_1]\!]^\rho_K = v_1 \qquad [\![e_2]\!]^\rho_K = v_2}{[\![(e_1, e_2)]\!]^\rho_K = (v_1, v_2)}$$

$$\frac{[\![e]\!]^\rho_K = (v_1, v_2) \qquad i \in \{1, 2\}}{[\![\pi_i(e)]\!]^\rho_K = v_i} \qquad \frac{[\![e_1]\!]^\rho_K = l \qquad [\![e_2]\!]^\rho_K = s}{[\![\mathsf{Tree}(e_1, e_2)]\!]^\rho_K = \mathsf{Tree}(l, s)}$$

$$\frac{[\![e]\!]^\rho_K = \mathsf{Tree}(l, s)}{[\![kids(e)]\!]^\rho_K = s} \qquad \frac{[\![e]\!]^\rho_K = \mathsf{Tree}(l, s)}{[\![tag(e)]\!]^\rho_K = l} \qquad \frac{[\![e_2]\!]^\rho_K = \mathsf{Tree}(l, s) \qquad [\![\bigcup(z \in s)\ (srt(x, y).\, e_1)\ z]\!]^\rho_K = s'}{[\![(srt(x, y).\, e_1)\ e_2]\!]^\rho_K = [\![e_1]\!]^{\rho[x := l, y := s']}_K}$$

Figure 6.10: Semantic Equations for $NRC_K + srt$

The semantics of $NRC_K + srt$, given with respect to an environment to variables ρ, is summarized by the equations in Figure 6.10. The meaning of $\mathsf{Tree}(-,-), tag(-)$ and $kids(-)$ are all straightforward (similar to pairing and projections). For *srt*, we require that Equation (6.1) continues to hold. Indeed, since K-collections have finite support, even in the presence of K-annotations, values of type *tree* have a finitary recursive structure. The semantics of $\bigcup(- \in -)\ -$ and Equation (6.1) above uniquely determines the semantics of *srt*.

Compiling UXQuery to $NRC + srt$

$$a \rightsquigarrow a \qquad () \rightsquigarrow \{\} \qquad \$x \rightsquigarrow x \qquad \frac{p \rightsquigarrow e}{(p) \rightsquigarrow \{e\}} \qquad \frac{p_1 \rightsquigarrow e_1 \qquad p_2 \rightsquigarrow e_2}{p_1,p_2 \rightsquigarrow e_1 \cup e_2}$$

$$\frac{p_1 \rightsquigarrow e_1 \qquad p_2 \rightsquigarrow e_2}{\texttt{for}\ \$x\ \texttt{in}\ p_1\ \texttt{return}\ p_2 \rightsquigarrow \cup(x \in e_1)\ e_2} \qquad \frac{p_1 \rightsquigarrow e_1 \qquad p_2 \rightsquigarrow e_2}{\texttt{let}\ \$x := p_1\ \texttt{return}\ p_2 \rightsquigarrow \mathit{let}\ x = e_1\ \mathit{in}\ e_2}$$

$$\frac{p_1 \rightsquigarrow e_1 \qquad p_2 \rightsquigarrow e_2 \qquad p_3 \rightsquigarrow e_3 \qquad p_4 \rightsquigarrow e_4}{\texttt{if}\ (p_1 = p_2)\ \texttt{then}\ p_3\ \texttt{else}\ p_4 \rightsquigarrow \mathit{if}\ e_1 = e_2\ \mathit{then}\ e_3\ \mathit{else}\ e_4} \qquad \frac{p_1 \rightsquigarrow e_1 \qquad p_2 \rightsquigarrow e_2}{\texttt{element}\ p_1\ \{p_2\} \rightsquigarrow \mathsf{Tree}(e_1, e_2)}$$

$$\frac{p_1 \rightsquigarrow e_1}{\texttt{name}(p_1) \rightsquigarrow \mathit{tag}(e_1)} \qquad \frac{p \rightsquigarrow e}{\texttt{annot}\ k\ p \rightsquigarrow k\, e} \qquad \frac{p \rightsquigarrow e \qquad e \overset{s}{\rightsquigarrow} e'}{p/s \rightsquigarrow e'}$$

Figure 6.11: Compilation of UXQuery

$$e \overset{\texttt{self::}*}{\rightsquigarrow} e \qquad\qquad e \overset{\texttt{self::}a}{\rightsquigarrow} \cup(x \in e)\ \mathit{if}\ \mathit{tag}(x) = a\ \mathit{then}\ \{x\}\ \mathit{else}\ \{\}$$

$$e \overset{\texttt{child::}a}{\rightsquigarrow} \begin{array}{l} \cup(x \in e) \\ \quad \cup(y \in \mathit{kids}(x)) \\ \quad \mathit{if}\ \mathit{tag}(y) = a\ \mathit{then}\ \{y\}\ \mathit{else}\ \{\} \end{array} \qquad\qquad e \overset{\texttt{child::}*}{\rightsquigarrow} \cup(x \in e) \mathit{kids}(e)$$

$$\frac{e' = \cup(x \in e)\ \pi_1((\mathit{srt}(b,s).\,f)\ x) \qquad \text{where } f = \begin{array}{l} \mathit{let}\ \mathit{self} = \mathsf{Tree}(b, \cup(x \in s)\ \{\pi_2(x)\})\ \mathit{in} \\ \mathit{let}\ \mathit{matches} = \cup(x \in s)\ \{\pi_1(x)\}\ \mathit{in} \\ (\mathit{matches} \cup \mathit{if}\ a = b\ \mathit{then}\ \{\mathit{self}\}\{\}\ \mathit{else}\ , \mathit{self}) \end{array}}{e \overset{\texttt{descendant::}a}{\rightsquigarrow} e'}$$

$$\frac{e' = \cup(x \in e)\ \pi_1((\mathit{srt}(b,s).\,f)\ x) \qquad \text{where } f = \begin{array}{l} \mathit{let}\ \mathit{self} = \mathsf{Tree}(b, \cup(x \in s)\ \{\pi_2(x)\})\ \mathit{in} \\ \mathit{let}\ \mathit{matches} = \cup(x \in s)\ \{\pi_1(x)\}\ \mathit{in} \\ (\mathit{matches} \cup \{\mathit{self}\}, \mathit{self}) \end{array}}{e \overset{\texttt{descendant::}*}{\rightsquigarrow} e'}$$

Figure 6.12: Compilation of XPath to $NRC_K + srt$

We define the semantics of K-UXQuery on K-UXML values by translation (compilation) to $NRC_K + srt$ in Figures 6.11 and 6.12. Since K-UXML values can be expressed with the constructors in K-UXQuery it suffices to translate K-UXQuery. The compilation function is written $p \rightsquigarrow e$. Here we discuss some of the more interesting cases; more details can be found in the long version of this paper. Most of the operators in K-UXQuery have a direct analog in $NRC_K + src$ and therefore have a simple translation, for example if $p_1 \rightsquigarrow e_1$ and $p_2 \rightsquigarrow e_2$ then `for` $\$x$ `in` p_1 `return` $p_2 \rightsquigarrow \cup(x \in e_1)\ e_2$. The most interesting compilation rules concern navigation steps. The compilation of a step $ax::nt$, written $e \overset{ax::nt}{\rightsquigarrow} e'$, describes by e' the set of trees that results from applying the given step to the set of trees described by e. Navigation compilation is then used in query compilation: if $p \rightsquigarrow e$ and $e \overset{ax::nt}{\rightsquigarrow} e'$ then $p/ax::nt \rightsquigarrow e'$.

The compilation of the `descendant` axis is the only place where we make use of structural recursion: we use *srt* to recursively walk down the structure of the tree and build up a set containing all of the matching nodes. For example, in the compilation rule for `descendant::*`, the s argument accumulates a set of pairs whose first component is the set of descendants below the immediate subtree contained in the second component of the pair. At each step, the body of the *srt* expression constructs a new pair using the current node and the accumulator. The descendants are obtained by projecting the first component of the final result.

6.6 Commutation with Homomorphisms

In this section, we prove a correctness theorem, stating that the semantics commutes with semiring homomorphisms, and explore some of its broader implications.

A semiring homomorphism $h : K_1 \xrightarrow{\text{hom}} K_2$ can be *lifted* to a transformation H from NRC_{K_1}+*srt* expressions to NRC_{K_2}+*srt* expressions by replacing every occurrence of a scalar k with $h(k)$. Since every K-complex value can be expressed with the constructors of $NRC_K + srt$, this gives us in particular a transformation from K_1-complex values to K_2-complex values.

A fundamental property of $NRC + srt$ is that query evaluation on K-complex values commutes with such transformations induced by homomorphisms.

Theorem 6.5. *If $h : K_1 \xrightarrow{hom} K_2$ is a homomorphism of semirings, denote by H its lifting as explained above. Then for any NRC_{K_1}+srt query e, and any environment ρ, $H(\llbracket e \rrbracket^{\rho}_{K_1}) = \llbracket H(e) \rrbracket^{H \circ \rho}_{K_2}$.*

Proof. By induction on the structure of e. There are 14 cases.

Case l: trivial

Case x: trivial

Case $\{\}$: immediate since $h(0_{K_1}) = 0_{K_2}$

Case $\{e\}$: immediate since $h(1_{K_1}) = 1_{K_2}$

Case $e_1 \cup e_2$: immediate from induction hypothesis and $h(a + b) = h(a) + h(b)$

Case $\cup(x \in e_1)\ e_2$: the induction hypothesis states that for all ρ',

$$H(\llbracket e_1 \rrbracket^{\rho'}_{K_1}) = \llbracket H(e_1) \rrbracket^{H \circ \rho'}_{K_2} \tag{6.2}$$

$$H(\llbracket e_2 \rrbracket^{\rho'}_{K_1}) = \llbracket H(e_2) \rrbracket^{H \circ \rho'}_{K_2} \tag{6.3}$$

Suppose $[\![e_1]\!]^{\rho}_{K_1} = \{a_1^{u_1},\dots,a_n^{u_n}\}$ and $[\![H(e_1)]\!]^{H\circ\rho}_{K_2} = \{b_1^{v_1},\dots,b_m^{v_m}\}$. Define an equivalence relation $i \sim j$ iff $H(a_i) = H(a_j)$. Then

$$H([\![e_1]\!]^{\rho}_{K_1}) = \{H(a_1)^{\sum_{j\sim 1} h(u_j)},\dots,H(a_m)^{\sum_{j\sim m} h(u_j)}\} = \{b_1^{v_1},\dots,b_m^{v_m}\}$$

and let us assume without loss of generality that for all $1 \leq i \leq m$,

$$H(a_i) \quad = \quad b_i \tag{6.4}$$

$$\sum_{i\sim j} h(u_j) \quad = \quad v_i \tag{6.5}$$

Now observe that

$$\begin{aligned}
H\left([\![\cup(x\in e_1)\ e_2]\!]^{\rho}_{K_1}\right) &= H(\textstyle\sum_{i=1}^{n} u_i \cdot [\![e_2]\!]^{\rho[x\leftarrow a_i]}_{K_1}) \\
&= \textstyle\sum_{i=1}^{n} h(u_i)\cdot H([\![e_2]\!]^{\rho[x\leftarrow a_i]}_{K_1}) \\
&= \textstyle\sum_{i=1}^{n} h(u_i)\cdot [\![e_2]\!]^{H\circ(\rho[x\leftarrow a_i])}_{K_2} && \text{by (6.3)} \\
&= \textstyle\sum_{j=1}^{m} \sum_{i\sim j} h(u_i)\cdot [\![e_2]\!]^{(H\circ\rho)[x\leftarrow H(a_i)]}_{K_2} \\
&= \textstyle\sum_{j=1}^{m} v_j \cdot [\![e_2]\!]^{(H\circ\rho)[x\leftarrow b_j]}_{K_2} && \text{by (6.4), (6.5)} \\
&= [\![\cup(x\in e_1)\ e_2]\!]^{H\circ\rho}_{K_1}
\end{aligned}$$

as required.

Case *if* $e_1 = e_2$ *then* e_3 *else* e_4: immediate using induction hypothesis (the fact that the conditional tests only equality of labels is essential)

Case (e_1, e_2): immediate using induction hypothesis

Case $\pi_i(e)$: immediate using induction hypothesis

Case $\mathsf{Tree}(e_1, e_2)$: similar to (e_1, e_2)

Case $\mathit{kids}(e)$: similar to $\pi_2(e)$

Case $\mathit{tag}(e)$: similar to $\pi_1(e)$

Case $(\mathit{srt}(x,y).\, e_1)\ e_2$: the induction hypothesis states that for all ρ',

$$H([\![e_1]\!]^{\rho'}_{K_1}) \quad = \quad [\![H(e_1)]\!]^{H\circ\rho'}_{K_2} \tag{6.6}$$

$$H([\![e_2]\!]^{\rho'}_{K_1}) \quad = \quad [\![H(e_2)]\!]^{H\circ\rho'}_{K_2} \tag{6.7}$$

We prove by (inner) induction on the K_1-tree $[\![e_2]\!]^{\rho}_{K_1}$ that the claim holds for e, i.e.,

$$H([\![(\mathit{srt}(x,y).\, e_1)\ e_2]\!]^{\rho}_{K_1}) = [\![H((\mathit{srt}(x,y).\, e_1)\ e_2)]\!]^{H\circ\rho}_{K_2}$$

There are two cases.

Case $[\![e_2]\!]^{\rho}_{K_1} = \mathsf{Tree}(l, \{\})$: we have that

$$\begin{aligned} H([\![(srt(x,y).\, e_1)\ e_2]\!]^{\rho}_{K_1}) &= H([\![e_1]\!]^{\rho[x \leftarrow l, y \leftarrow \{\}]}_{K_1}) \\ &= [\![H(e_1)]\!]^{H \circ (\rho[x \leftarrow l, y \leftarrow \{\}])}_{K_2} \qquad \text{by (6.6)} \\ &= [\![H(e_1)]\!]^{(H \circ \rho)[x \leftarrow l, y \leftarrow \{\}]}_{K_2} \\ &= [\![(srt(x,y).\, e_1)\ e_2]\!]^{H \circ \rho}_{K_2} \end{aligned}$$

as required.

Case $[\![e_2]\!]^{\rho}_{K_1} = \mathsf{Tree}(l, s)$ where $s = \{a_1^{u_1}, \ldots, a_n^{u_n}\}$: similar calculations as in the big-union case, but using the (inner) induction hypothesis.

Case $k\, e$: immediate using induction hypothesis and $h(a \cdot b) = h(a) \cdot h(b)$

$\square$

A semiring homomorphism $h : K_1 \to K_2$ can similarly be lifted to a transformation H from K_1-UXQuery expressions to K_2-UXQuery expressions (and from K_1-UXML values to K_2-UXML values), again by just replacing each occurrence of a scalar k with $h(k)$. Based on our compilation semantics for K-UXQuery, we conclude from the theorem above that a similar commutation property holds for K-UXML and K-UXQuery:

Corollary 6.6. *If $h : K_1 \xrightarrow{hom} K_2$ is a semiring homomorphism, denote by H its lifting to a transformation from K_1-UXQuery to K_2-UXQuery. Then for any K_1-UXQuery query p, for any environment ρ, $H([\![p]\!]^{\rho}_{K_1}) = [\![H(p)]\!]^{H \circ \rho}_{K_2}$. (Here, we lift $[\![\cdot]\!]$ to K-UXQuery in the obvious way using our compilation to NRC_K+srt.)*

We already mentioned several applications of the commutations with homomorphisms theorem (cf. Section 6.2, Section 6.3, and Section 6.4). Another simple but practically useful application involves the "duplicate elimination" homomorphism $\dagger : \mathbb{N} \to \mathbb{B}$ defined as $\dagger(0) \stackrel{\text{def}}{=} \mathsf{false}$ and $\dagger(n+1) \stackrel{\text{def}}{=} \mathsf{true}$. Lifting $\dagger$ to K-complex values and trees or to K-UXML values we obtain that evaluation of ordinary values can be factored through that of values with multiplicities, with duplicate elimination deferred to a final step (in the style of commercial relational database systems).

6.7 Semantics via Relations

We sketch in this section an encoding of K-UXML into K-relations and an accompanying compilation of XPath into Datalog (extended with Skolem functions) which has the important property that the answer to the Datalog program corresponds to the answer to the XPath query with *identical annotations*. This provides an alternative definition of the semantics of XPath on K-UXML which agrees with that of Section 6.5. The availability of such a compilation scheme is an important concern in practice, where XML data is often "shredded" into relations, with queries over the data compiled into SQL for execution by an RDBMS [47, 127]. However, the focus here is not on practicality, but on demonstrating a basic proof-of-concept scheme.

We encode a (set of) K-UXML trees using a single K-relation $E(pid, nid, label)$. Each tuple in E corresponds to a single K-UXML node, and carries the same annotation as the K-UXML node.

As opposed to UXML in which an item is an entire tree identified by its value, in this encoding an item is identified by its node id. Thus *pid* is the identifier of the node's parent, *nid* is the identifier of the node itself, and *label* is the node's label. The special *pid* 0 is reserved and indicates that the node corresponds to a (top-level) root of a tree in the set.

Node ids are invented as needed during translation of the K-UXML into relational form. During subsequent query processing, additional node ids may be needed to represent nodes in the query result; we use Skolem functions for this purpose. Recursive Datalog rules are used to implement the XPath descendant operator. To give a flavor of the query translation, we show the rule for one important case, the `descendant` axis:

$$e \overset{\texttt{descendant}::a}{\rightsquigarrow} \qquad \begin{array}{rcl} R(n,l) & \text{:-} & E(0,n,l) \\ R(n,l) & \text{:-} & R(p,\_), E(p,n,l) \\ E'(f(p), f(n), l) & \text{:-} & E(p,n,l) \\ E'(0, f(n), \mathrm{a}) & \text{:-} & R(n, \mathrm{a}) \end{array}$$

E encodes the set of input trees and E' encodes the set of output trees. f is a Skolem function. To illustrate, the XPath query //c on the source tree in Figure 6.4 with $x_1 := 0$ (to simplify the example) yields:

$E' =$

pid	*nid*	*label*	
0	$f(2)$	c	y_1
0	$f(5)$	c	$y_1 \cdot y_2$
$f(0)$	$f(1)$	a	1
$f(1)$	$f(2)$	c	y_1
$f(2)$	$f(3)$	d	1
$f(3)$	$f(4)$	a	1
$f(2)$	$f(5)$	c	y_2
$f(2)$	$f(6)$	b	x_2

The K-UXML tree which would have been produced by executing the query directly on the input tree is encoded by the tuples reachable from the root tuples (which have *pid* 0). Note that there are also some "garbage" tuples in the table that are unreachable from any root: e.g., $(f(0), f(1), \mathtt{a})$. We summarize with the following theorem:

Theorem 6.7. *There is a 1-1 translation ϕ of K-UXML to K-relations and a translation ψ of XPath to Datalog with Skolem functions, such that for every K-UXML value v and XPath query p, we have $\phi(p(v)) = \psi(\phi(p))$.*

Proof. ☐

Monads of Semimodules as Collection Types

Let $(K, +, \cdot, 0, 1)$ be a commutative semiring. A *semimodule* over K (a K-semimodule) is an algebraic structure $(M, +, 0, \lambda)$ where $(M, +, 0)$ is a commutative monoid, and $\lambda : K \times M \to M$ is a *scalar multiplication operation*, written (as usual) $\lambda(k, x) = k\,x$ such that

$$\begin{aligned}
k\,(x + y) &= k\,x + k\,y \\
k\,0 &= 0 \\
(k_1 + k_2)\,x &= k_1\,x + k_2\,x \\
(k_1 \cdot k_2)\,x &= k_1\,(k_2\,x) \\
0\,x &= 0 \\
1\,x &= x
\end{aligned}$$

K-semimodules and their homomorphisms form a category K-**SMod**. The forgetful functor U :K$-$**SMod** $\to$ **Set** has a left adjoint that is very easy to describe: the free K-semimodule generated by a set X is the set X^K_f of functions $X \to K$ that have *finite support* (see Section 6.5 and note that in NRC_K $[\![\{t\}]\!]_K$ is precisely $([\![t]\!]_K)^K_f$), with the obvious pointwise addition and pointwise multiplication K-semimodule structure. This adjunction yields a (strong) monad on **Set**, which can be enriched [99] with a K-semimodule structure on each monad algebra. Therefore, we have a collection and aggregates query language, as in [19, 99, 12]. In fact, it is easy to see that any commutative monoid is an $\mathbb{N}$-semimodule and that the $\mathbb{B}$-semimodules are exactly the commutative-idempotent monoids, so the finite sets and finite bags collections are included here [11]. Properties like the commutation with homomorphisms theorem (6.5) have a very general category-theoretic justification, based on the fact that all the query language constructs in such query languages come from functorial constructs and natural transformations.

We can also capture some of this theory through an equational axiomatization for NRC_K

---

[11] It is not clear how to include finite lists in this semiring-based family of collection types

Proposition 6.8. *The semantics of NRC$_K$ satisfies the following equational axioms:*

- $\cup$, $\{\}$ *and multiplication with scalars from K satisfy the axioms of a* semimodule *over K.*
- $\cup(x \in e_1)\ e_2$ *satisfies the axioms:*

$$\begin{aligned}
\cup(x \in \cup(y \in R)\ S)\ T &= \cup(y \in R) \cup(x \in S)\ T \\
\cup(x \in S)\ \{x\} &= S \\
\cup(x \in \{e\})\ S &= S[x := e] \\
\cup(x \in k_1\, R_1 \cup k_2\, R_2)\ S &= k_1\, (\cup(x \in R_1)\ S)\ \cup \\
&\quad\ k_2\, (\cup(x \in R_2)\ S) \\
\cup(x \in R) \cup(y \in S)\ T &= \cup(y \in S) \cup(x \in R)\ T \\
\cup(x \in R)\ (k_1\, S_1 \cup k_2\, S_2) &= k_1\, (\cup(x \in R)\ S_1)\ \cup \\
&\quad\ k_2\, (\cup(x \in R)\ S_2)
\end{aligned}$$

(In particular, the 4th and 6th axioms state its bilinearity *w.r.t. the semimodule structure).*

Chapter 7

Related Work

7.1 Paradigms for Data Integration

We mentioned in Chapter 1 that a number of other paradigms for data integration and data sharing have been proposed over the years, of which collaborative data sharing is only the latest. In this section, we overview several of these paradigms in more detail, and in (roughly) chronological order. The goal is not to give a complete account of their various features and functionalities, but just to give a better sense of the context of the work described in this dissertation. Figure 7.2 gives a comparative summary of some of their main features.

Data Warehousing

While this dissertation has focused on the needs of scientific collaborators, another set of data integration requirements exists in business and corporate settings. Since the early 1990s, these have been addressed by the development of so-called *data warehouse* technologies [83, 80]. As defined in [80], a data warehouse is a "subject-oriented, integrated, nonvolatile, and time-variant collection of data in support of management's decisions." The emphasis in data warehousing is on answering complex *decision-support* (OLAP) queries, which are typically complicated aggregate queries posed over large volumes of historical business data.

A typical data warehouse architecture consists of a stack of several components as shown in Figure 7.1:

1. A collection of heterogeneous *data sources* at the bottom level, which may consist of an organization's day-to-day operational databases, legacy systems, unstructured data stored in text files, and so on.

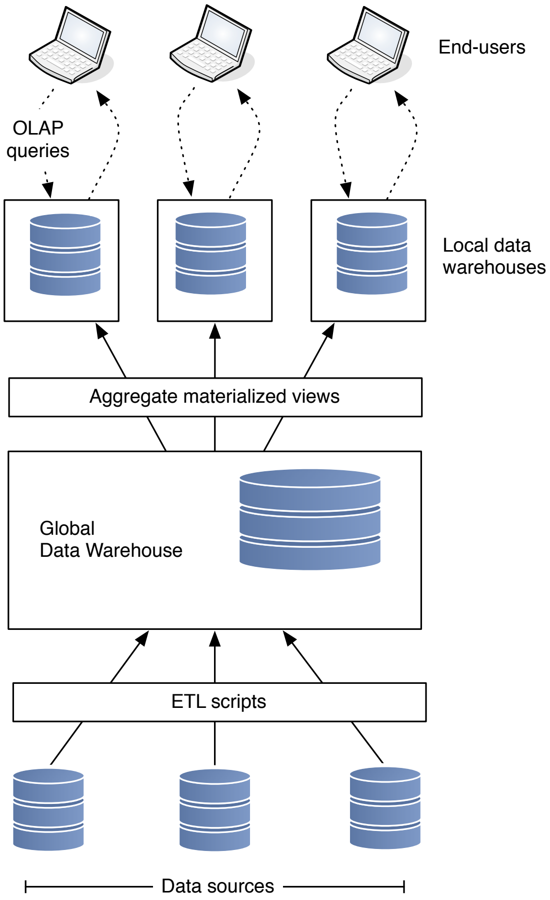

End-users
OLAP queries
Local data warehouses
Aggregate materialized views
Global Data Warehouse
ETL scripts
Data sources

Figure 7.1: Data warehouse architecture

2. A centralized "global" data warehouse, which is a database containing a historical record of data collected from the data sources.

3. Special-purpose *extract-transform-load* (ETL) scripts that extract data from data sources, transform it into the global schema of the data warehouse, and perform *cleaning* of the source data to remove inconsistencies.

4. Specialized "local" warehouses containing aggregated data derived from the global warehouse, over which end-users pose their analytical OLAP queries.

As Figure 7.1 suggests, the flow of data through a typical data warehouse is unidirectional, from source to global warehouse, and from global warehouse to local warehouse. Typically, the global and local data warehouses are refreshed only periodically (e.g., by weekly or monthly batch jobs) and data is only added, never deleted. The data warehouse architecture has two chief virtues, compared to exposing data sources directly to end-user queries: first, formulating queries

	Materialized instances	Declarative mappings	Update propagation	Mediated schema
Data warehousing	●	◐	◐	●
Virtual data integration	○	●	n/a	●
P2P data integration	○	●	n/a	○
Data exchange	●	●	○	●
Collaborative data sharing	●	●	●	○

A "●" indicates "present", a "◐" indicates "partial", and a "○" indicates "absent."

Figure 7.2: Feature comparison of data integration paradigms

over a single, clean, consistent instance is much simpler from a usability standpoint; second, a company's operational databases are isolated from the performance costs of evaluating expensive OLAP queries.

Compared to CDSS, there are several important differences. First of all, the organization of a data warehouse is strongly centralized and hierarchical, using a single mediated schema and a unidirectional flow of data. CDSS, in contrast, is highly decentralized, allows data to flow in many directions, and does not require agreement on a single mediated schema or global data consistency. The ETL process used to populate the data warehouse from data sources is similar to update exchange in its aims, but technically is quite different: ETL is usually implemented using procedural scripts in general-purpose programming languages, while CDSS uses a logic-based formalism that can be viewed as a kind of "declarative ETL." The tradeoff between these approaches is one of expressiveness versus amenability to automated reasoning tasks such as optimization of update propagation.

Virtual Data Integration

With the explosion in the 1990s of data on the web, there came a need to build systems to facilitate rapid integration of this data. This led to the development of *virtual data integration* technologies [135, 70]. Figure 7.3 shows a diagram of a typical virtual data integration architecture, as exemplified by systems like TSIMMIS [53], the Information Manifold [103], and Kleisli [15]. As with data warehousing, the bottom level of such an architecture involves a collection of *data sources*, usually imagined to be web data sources. For data sources such as web sites, which do not expose their databases directly, wrappers must be provided which extract relational or semistructured data from HTML pages. In contrast to data warehousing, the schemas of the data

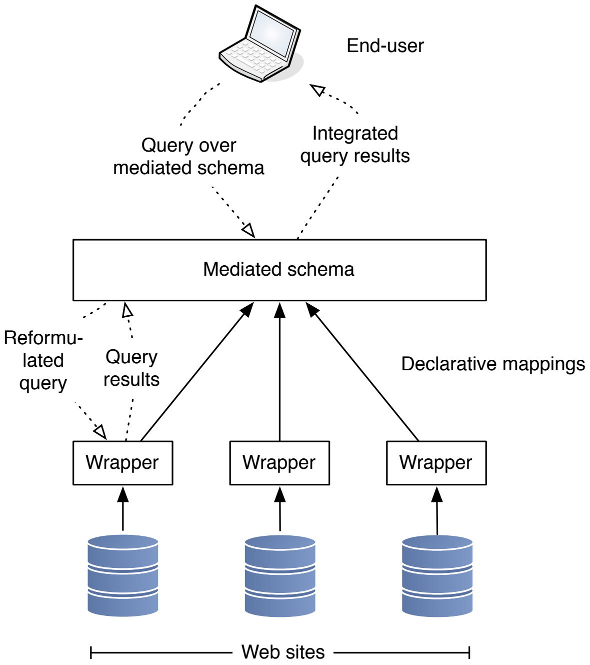


Figure 7.3: Virtual data integration architecture

sources are related to the mediated schema via *declarative mappings*, and no instance of the mediated schema is ever materialized. Rather, queries posed over the mediated schema are *reformulated* on-the-fly, according to the mappings, and answered over the remote data sources. The results from the remote sources are then collected together and returned to the user. Update propagation, a main concern in materialized approaches such as data warehousing, becomes a non-issue in virtual data integration.

Mappings in virtual data integration are expressed in a number of formalisms, such as *global-as-view* (GAV), *local-as-view* (LAV), or *global-local-as-view* (GLAV). The paper by Lenzerini [100] surveys and compares these formalisms in detail. Briefly, with GAV, the virtual mediated instance is essentially just a view (expressed using, e.g., SQL queries) over the data sources. Query posed over the mediated schema can be answered over the sources using the standard technique of view unfolding. With LAV, the data sources are thought of as views (typically conjunctive queries) over the mediated schema. End-user queries posed over the mediated schema are answered over the sources using so-called *maximally-contained query plans* [43]. An linchpin result of Abiteboul and Duschka [1] established that such plans (for conjunctive LAV mappings and Datalog queries) return precisely the *certain answers*. Finally, GLAV mappings combine features of the GAV and LAV formalisms.

While virtual data integration was originally motivated by the need to integrate data from

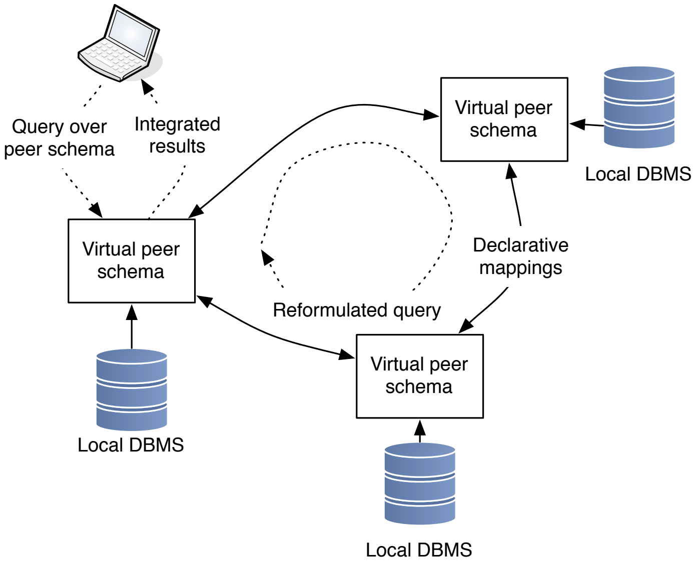

Query over peer schema
Integrated results
Virtual peer schema
Virtual peer schema
Local DBMS
Declarative mappings
Reformulated query
Virtual peer schema
Local DBMS
Local DBMS

Figure 7.4: Peer-to-peer data integration architecture

the web, the techniques developed to accomplish this can equally well be applied to data inside corporations. This has recently led to a market for commercial *enterprise information integration* systems.

Peer-to-Peer Data Integration

The reliance of virtual data integration on a global mediated schema eventually came to be seen as a limitation in more decentralized application scenarios lacking a strong central authority to play the role of the mediator. This motivated the development of *peer-to-peer data integration* (P2P data integration), as pioneered in systems such as Piazza [64, 69]. Figure 7.4 illustrates the architecture of such a system. Compared to virtual data integration, the main difference is that there is no single mediated schema. Rather, each data source ("peer") comes with its own virtual peer schema. Declarative mappings relate the data source of each peer with its peer schema, and a peer schema with schemas of other peers. End-user queries are posed over a particular peer schema, and the goal is to compute the certain answers to the queries using reformulation strategies across the network of peers. As with CDSS, technical difficulties arise in the presence of cycles in the network of mappings.

Data Exchange

We already discussed data exchange [45] in some detail in Chapter 1, but for completeness we repeat and elaborate upon some of the details here.

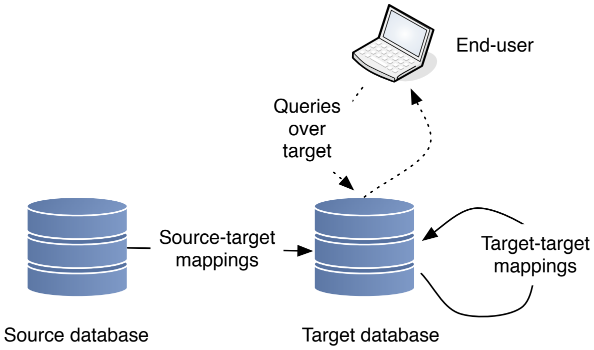


Figure 7.5: Data exchange architecture

Figure 7.5 shows a picture of a typical data exchange architecture, as exemplified by systems such as Clio [75, 118]. A data exchange setting comprises a *source schema* and a *target schema*, related by declarative *source-target* mappings (tgds). Additionally, *target-target* mappings may enforce integrity constraints on the target instance. These may be a combination of tgds and *equality-generating dependencies* (a generalization of key constraints). Given an instance of the source database, the goal is to compute a corresponding instance of the target database. In general, there may exist many instances of the target database compatible with the schema mappings, hence a special one — the *canonical universal solution* — is the one materialized. The end-user poses queries over this materialized target instance. Cycles in (target-target) mappings cause technical difficulties, hence as with CDSS, the network of mappings is required to be weakly acyclic. Propagation of updates from source to target is not considered in the classical data exchange framework, which is clearly a practical limitation.

Peer-to-Peer Data Exchange

The CDSS paradigm can be described to a first approximation as a generalization of the ideas of data exchange to peer-to-peer scenarios (with additional support for data provenance and efficient update propagation). Another recent paper [52] describes a model called *peer data exchange*, whose name is suggestive of something rather similar, but turns out to be rather different. The model of that paper considers a form of data exchange that supports mappings in two directions (from source to target, and from target to source), but source database remains fixed and the exchange of data is still done in a unidirectional fashion, from source to target. The target to source mappings are used only to further constrain the space of allowable target instances.

The Hyperion system [90, 88] also implements a form of peer-to-peer data exchange. As with CDSS, Hyperion aims to facilitate data sharing among networks of peer relational databases. In contrast to CDSS, which uses *schema*-level mappings to accomplish this data sharing, Hyperion instead addresses the problem of data sharing at the *data* level, employing so-called *mapping tables* that relate corresponding data values between sources.

Collaborative Data Integration

A recent paper [94] proposes a paradigm for data integration called *collaborative data integration* (CDI), and an implementation in a system called Youtopia, that is similar in many of its aims and technical features to CDSS. As in CDSS, CDI proposes a data exchange-style setup involving peer schemas related by networks of declarative mappings (tgds), and computation of materialized solutions. Youtopia employs an interesting new approach to resolving the technical difficulties associated with cycles in mappings, based on semi-automated user intervention. This allows the system to support mappings with arbitrary cycles. The system also supports update exchange, and incorporates sophisticated new transaction facilities. In contrast to CDSS, it does not track data provenance.

7.2 Provenance and Annotated Data Models

Lineage and why-provenance were introduced in [34, 35, 17] (the last paper uses a tree data model) but the relationship with [79] was not noticed. The papers on probabilistic databases [50, 138, 96] note the similarities with [79] but do not attempt a generalization.

Datalog with bag semantics in which derivation trees are counted was considered in several papers, among them [109, 113, 114]. 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 [40, 97], again with non-terminating algorithms. Later [115] gave an algorithm for detecting infinite multiplicities in Datalog with bag semantics and [51] gave a terminating algorithm for Datalog on probabilistic databases.

Two recent papers on provenance, although independent of our work, have a closer relationship to our approach. Like us, [25] identifies the limitations of why-provenance and proposes *route-provenance* which is also related to derivation trees. The issue of infinite routes in recursive programs is avoided by considering only *minimal* ones. [8] proposes a notion of lineage of tuples for a type of incomplete databases but does not consider recursive queries. We have seen that the annotations in that paper can be captured using a semiring, $\mathsf{Trio}(X)$.

The first attempt at a general theory of relations with annotations appears to be [81] where axiomatized *label systems* are introduced in order to study containment.

A line of work is artificial intelligence on *soft constraint satisfaction problems* (soft CSP) [11] is similar in spirit to our work. 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 or Prolog programs. Computing solutions is the same as the evaluation of join and projection in Section 2.2 and [11] also uses fixed points on semirings. There are some important differences though. The semirings used in [11] 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.

We left open the question of how to incorporate the relational algebra difference operator into the framework of semiring annotations. This problem was addressed in [54], which proposed the use of a monus operation for this purpose. The same paper also initiated the study of expressiveness of query languages on semiring-annotated relations.

7.3 Update Exchange

As already mentioned, our work on update exchange takes advantage of previous work on data integration, PDMS (e.g., [73]) and data exchange [45, 104, 118]. With our encoding in Datalog, 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 [66]. Incremental maintenance of recursive views is also considered in [108] in the context of databases with constraints using the Gabrielle-Levi fixpoint operator. The AutoMed system [110] 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 [52], 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.

[8] proposes a notion of lineage of tuples which is a combination of sets of relevant tuple ids and bag semantics. As best as we can tell, this is more detailed than why-provenance, but as we have seen in Chapter 2, 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, [8] does not support any notion of incremental translation of updates

---

[1] Another difference is that for Datalog semantics we require our semirings to be ω-continuous while [11] uses the less well-behaved fixed points given by Tarski's theorem for monotone operators on complete lattices. However, the semiring examples in [11] appear to be in fact ω-continuous.

over mappings or incompleteness in the form of tuples with labeled nulls. Our provenance model also generalizes the duplicate (bag) semantics for Datalog [114] and supports generalizations of the results in [115].

A recent paper [106] adopts an approach to incremental view maintenance similar in spirit to our work in its use of provenance annotations to speed up propagation of deletions. There, the annotations are from what is essentially $\mathsf{PosBool}(X)$ (cf. Chapter 2), the semiring of positive Boolean expressions over variables from X.

7.4 Query Containment and Equivalence

The seminal paper by Chandra and Merlin [22] introduced the fundamental concepts of containment mappings and canonical databases in showing the decidability of containment of CQs under set semantics and identifying its complexity as NP-complete. The extension to UCQs is due to Sagiv and Yannakakis [123]. We have built upon the techniques from these papers.

The papers by Ioannidis and Ramakrishnan [81] and Chaudhuri and Vardi [24] initiated the study of query containment under bag semantics. Chaudhuri and Vardi showed that bag-equivalence of CQs is the same as isomorphism, established the Π_2^p-hardness of checking bag-containment of CQs, and gave partial conditions for checking bag-containment (see Section 4.4 for further connections with our results). Chaudhuri and Vardi [24] also introduced the study of *bag-set semantics*, and showed that bag-set equivalence of CQs (without repeated atoms in the body) is the same as isomorphism. This was essentially a rediscovery of a well-known result in graph theory due to Lovász [107] (see also [74]), who showed that for finite relational structures F, G, if $|\mathrm{Hom}(F, H)| = |\mathrm{Hom}(G, H)|$ for all finite relational structures H, where $\mathrm{Hom}(A, B)$ is the set of homomorphisms $h : A \to B$, then $F \cong G$. In database terminology, this says that bag-set equivalence of Boolean CQs (without repeated atoms in the body) is the same as isomorphism. Ioannidis and Ramakrishnan showed that bag-containment of UCQs is undecidable.

In Section 4.4 we have discussed the results of Cohen et al. [30] and Cohen [27] on bag equivalence and bag-set equivalence of UCQs. The decidability of bag-containment of CQs remains open. Recent progress was made on the problem by Jayram et al. [84] who established the undecidability of checking bag-containment of CQs with inequalities.

Semiring-annotated relations are also related to the lattice-annotated relations used in *parametric databases* by Lakshmanan and Shiri [98]. That paper also studied query containment and equivalence, giving a number of positive decidability results. None of our provenance models fall into this framework (with the exception of $\mathsf{PosBool}(X)$, cf. Theorem 4.34).

We have already mentioned in Section 2.4 the paper by Grahne et al. [56], which studied containment and equivalence of positive relational queries on bilattice-annotated relations.

Tan [130] showed that query containment is decidable for CQs on relations with *where-provenance* information. Our results here on why-provenance complement the where provenance results (why-provenance and where-provenance were introduced together in [17]).

Cohen [28] recently initiated the study of query optimization under *combined semantics*, which generalizes bag semantics and bag-set semantics by enriching the relational algebra with a *duplicate elimination* operator. "Duplicate elimination" also makes sense for K-relations in the form of the *support* operator:

$$\mathsf{supp}(R) \overset{\text{def}}{=} \lambda t. \begin{cases} 0 & \text{if } R(t) = 0 \\ 1 & \text{otherwise} \end{cases}$$

For $K = \mathbb{N}$, this is duplicate elimination; for $K = \mathsf{PosBool}(X)$ it corresponds to the poss operator of [5] which returns the "possible" tuples of an incomplete relation. It would be interesting to see whether the decidability results presented here can be extended to queries using supp.

7.5 Ring-Annotated Relations and Updates

Exact query reformulation using views has been studied extensively, due to its applications in query optimization, data integration, and view maintenance, starting with the papers by Levy et al. [101] and Chaudhuri et al. [23]. The former paper established fundamental results for UCQs with built-in predicates (our $\text{UCQ}^{<}$s) under set semantics. The latter paper considered CQs with built-in predicates ($\text{CQ}^{<}$s) under bag semantics, but it did not provide a complete reformulation algorithm or consider UCQs.

The view adaptation problem was introduced in [65], which gives a case-based algorithm for adapting materialized views under changes to view definitions (under bag semantics). In contrast, our methods apply to view adaptation, but use a more general term rewrite system to develop a sound and complete query reformulation algorithm.

Our $\mathbb{Z}$-relations appeared in an early form as the *deltas* in the count incremental view maintenance algorithm for UCQs of [66]. That paper did not consider query equivalence for deltas or make a general study of query reformulation.

$\mathbb{Z}[X]$-relations made their first appearance in [54]. That paper did not consider the use of $\mathbb{Z}[X]$ to represent data updates, nor did it consider questions of query equivalence and query reformulation.

7.6 Semiring-Annotated XML

The original why/where provenance paper [17] used a tree-structured data model. However, the model was tag-deterministic and the annotations were in effect paths from the root. Its query language relies on a deep-union construct that seems incomparable with what we do. In addition, the related work in [62] surveys work on semirings, other models of provenance, and probabilistic and incomplete relations that we do not repeat here.

Among proposed models for probabilistic and incomplete XML, closest to our work is [4, 126], which uses unordered XML decorated with Boolean combinations of probabilistic events. A model for incomplete XML was developed in [3]. In both systems the query language is tree patterns, and the main focus is on handling updates and complexity results. By contrast, our goal is a general-purpose annotation framework with a richer query language in which probabilistic and incomplete XML are obtained as special cases. Other models for probabilistic XML include probabilistic interval annotations [78], probabilistic trees for data integration [134], and numeric probability annotations [116]; for incomplete XML we add the maximal matchings approach of [85].

The focus of [14] is to compare a semantics for *NRC* on annotated complex values to the semantics of an update language but the data model and the query semantics is different from ours. In particular, query-constructed values are annotated with “unknown.” Another provenance model for *NRC*, tracing operational executions for scientific dataflows, is described in [77].

Semirings are used to provide semantics for regular path queries decorated with preference annotations over graph-structured data in [57]. It is unclear whether there is any connection with our semiring-annotated *data*.

Dealing with ordered XML seems to be a troublesome issue in our framework. Unlike sets and bags, lists are not immediately representable as the functions of finite support into some commutative (or even non-commutative) semiring. Still we believe that, based on our semantics for UXML, a practical, albeit somewhat ad-hoc, provenance semantics for ordered XQuery could be devised and then tested for user acceptance.

7.7 Further Work on Orchestra

The material presented in Chapter 2 and Chapter 3 is also covered in another dissertation [86], by Grigoris Karvounarakis. (The work described in those chapters was performed in collaboration with Karvounarakis and Tannen [62] and Karvounarakis, Ives, and Tannen [60].) That dissertation also covers several major components of ORCHESTRA not discussed here, including its facilities for bi-directional mappings [87] and a query language for CDSS provenance.

Another major component of Orchestra not covered in this dissertation is its automatic, provenance-based conflict resolution facilities. That component is described in [131].

The version of Orchestra described in this dissertation performs all update exchange computations *locally* at one peer's site. (The global update store is used only to store and retrieve local curations and the final results of update exchange computations.) However, more recent work on the system [132] has developed a fully distributed, reliable query processing engine that can be used to distribute the update exchange computations across the peer network, with significant performance benefits.

Chapter 8

Conclusions and Future Directions

We have developed in this dissertation (and in collaboration with others) the theoretical foundations and the practical design and implementation of a new kind of data management system, the collaborative data sharing system (CDSS). The ORCHESTRA CDSS prototype allows collaborators in domains such as the sciences to share relational data, modulated by schema mappings, trust policies, and local curation facilities which allow collaborators to retain full control of their own data. The system includes novel facilities for efficient handling of updates, both to data and to schema mappings. The system tracks the provenance of data as it is transformed and propagated according to schema mappings, and uses this provenance information as the basis for trust policies and efficient update propagation algorithms, and makes it available to CDSS users to aid them in curating their databases.

The system is based on solid formal foundations, including a powerful new framework of semiring-annotated relations, which unifies and clarifies the relationships among a number of previous models of data provenance, and leads to a natural notion of provenance appropriate for CDSS. We addressed fundamental questions related to semiring-annotated relations, such as query containment and equivalence in the presence of provenance information. We also argued for ring-annotated relations as a uniform representation for data and updates, and we saw that this approach enables a unified perspective on update translation and mapping evolution as special cases of a more general problem, optimizing queries using materialized views, which turns out to be surprisingly tractable in this context. Finally, we showed that semiring annotations also make sense for XML and nested relational data.

We conclude the presentation with a list of several promising research directions suggested by the work in this dissertation.

8.1 Immediate Next Steps

Soft CSP and conjunctive query containment

Although in general database research and artificial intelligence CSP research sadly tend to ignore each other (witness the original submission of [62]!), some deep connections have been exhibited in [93], involving, in particular, conjunctive query containment. Looking at the relationship between our framework of semiring-annotated relations and that of [11] (cf. Chapter 7) from this perspective might provide interesting results.

Transactions

It would be worthwhile to extend CDSS to incorporate complex transaction models that give rise to data dependencies and conflicts between the update policies of different peers. Solving such problems will likely require a *transaction provenance model* in which we also would annotate updates with transaction identifiers.

Datalog and query containment

We left open the decidability of containment and equivalence of Datalog problems in the presence of several kinds of provenance annotations. A negative resolution of these problems would not be unexpected, but still nice to know. On the other hand, a positive resolution of any of them would be quite surprising and interesting.

UCQs, relational algebra, and query containment

In studying query containment and equivalence, we assumed a Datalog-style representation for UCQs, which is expressively equivalent to the positive relational algebra ($\mathcal{RA}^+$) on K-relations, but exponentially less concise. Under set semantics, it is well-known [123] that checking containment of $\mathcal{RA}^+$ queries is correspondingly harder (Π_2^p-complete rather than NP-complete). An obvious question is how the move to an algebraic representation affects the results in the presence of provenance annotations. This question appears to be open even for bag semantics, and may be rather difficult to resolve.

Query containment for annotated XML

It would clearly be worthwhile to investigate how the same questions of query containment and equivalence considered here for annotated relations play out for annotated XML, in order to better understand the interplay between provenance annotations and query optimization strategies in that context.

Incomplete XML databases

We have given very general strong representation systems for incomplete XML in Section 6.4. This opens a whole set of questions about their (relative) completeness and expressive power, along the lines of the questions considered in [63].

Practical applications of annotated XML

The framework for annotated XML we have described seems to be flexible and potentially useful in practical applications. We are thinking in particular about using semirings of confidentiality levels in an RDBMS by hiding the out-of-model calculations from users and also about recording jointly provenance, security, and uncertainty (the product of several semirings is also a semiring!).

Beyond relational CDSS

Extending CDSS to handle XML and nested relational data would have many practical benefits. Schema mappings among XML data sources have been well-studied [118, 72], and we have seen in Chapter 6 that semiring annotations also make sense for XML. Still, there would likely be significant engineering challenges in building a practical XML CDSS.

8.2 Longer-Term Directions

The tyranny of schema mappings

A fundamental assumption made in CDSS and other paradigms for data integration is the pre-existence of precise schema mappings relating the data sources. In practice, these mappings are extremely time-consuming to develop, and present a significant barrier to adoption. We feel that this is one major reason why current data integration systems have not found more widespread use. In this dissertation, we took some steps towards ameliorating the problem by allowing for efficient handling of updates to mappings, expecting that they will change over time.

Longer term, it would be worthwhile to investigate relaxing this requirement by developing language constructs and data models that allow querying over heterogeneous data sources, even when mappings are “partial” or non-existent. The recently-proposed notion of *dataspaces* [68] provides a vision for such a system, with unstructured, keyword-based search and structured, relational queries over mediated schemas at opposite ends of a continuous spectrum. Even when all sources are highly-structured (e.g., relational databases), making the query language compositional will require relaxing the relational data model to allow heterogeneous query results (e.g., tuples with different schemas) to be collected together; XML is a natural choice for this purpose. This gives additional motivation for implementing an XML CDSS as an underlying platform for more advanced dataspace-style systems.

CDSS and probabilistic data

There is also a need to extend the CDSS paradigm to incorporate *probabilistic information*. Recently, the data management community has focused on probabilistic databases as a unifying framework that allows uncertainties in data, mappings, and queries to be handled in a principled way. While not originally targeted at uncertainty management, the provenance information used in Orchestra can be applied to probabilistic data as well. The semiring-based provenance framework in Orchestra is also potentially interesting as a vehicle for designing and implementing algorithms for top-k query answering.

CDSS and privacy

Finally, CDSS participants have natural *privacy concerns*, hence there is a need for security mechanisms to control data sharing among various categories of users. Here again, the framework of semiring-annotated data offers a potential route to novel solutions: in Chapter 6, we proposed annotating XML data with security clearance levels, and showed that the semantics of XQuery on security-annotated XML obeys a fundamental *non-interference* property. A intriguing novelty of the semiring-annotations framework is the potential to mix security annotations with other kinds of annotations, such as trust levels and probabilities, yielding interesting hybrid semantics.

Bibliography

[1] Serge Abiteboul and Oliver Duschka. Complexity of answering queries using materialized views. In *PODS*, Seattle, WA, 1998.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Representing and querying XML with incomplete information. *TODS*, 31(1):208–254, 2006.

[4] Serge Abiteboul and Pierre Senellart. Querying and updating probabilistic information in XML. In *EDBT*, 2006.

[5] Lyublena Antova, Christoph Koch, and Dan Olteanu. From complete to incomplete information and back. In *SIGMOD*, 2007.

[6] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL. *Nucleic Acids Research*, 28, 2000.

[7] Catriel Beeri and Moshe Vardi. A proof procedure for data dependencies. *JACM*, 31(4), October 1984.

[8] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, and Jennifer Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.

[9] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, and Jennifer Widom. Uldbs: Databases with uncertainty and lineage. In *VLDB*, 2006.

[10] Philip A. Bernstein, Fausto Giunchiglia, Anastasios Kementsietsidis, John Mylopoulos, Luciano Serafini, and Ilya Zaihrayeu. Data management for peer-to-peer computing: A vision. In *WebDB*, June 2002.

[11] Stefano Bistarelli. *Semirings for Soft Constraint Solving and Programming*. Springer, 2004.

[12] P. Buneman and V. Tannen. A structural approach to query language design. In *The Functional Approach to Data Management Modeling, Analyzing, and Integrating Heterogenous Data*. Springer, 2004.

[13] Peter Buneman, James Cheney, Wang-Chiew Tan, and Stijn Vansummeren. Curated databases. In *PODS*, 2008.

[14] Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. In *ICDT*, 2007.

[15] Peter Buneman, Susan B. Davidson, Kyle Hart, Chris Overton, and Limsoon Wong. A data transformation system for biological data sources. In *VLDB*, 1995.

[16] Peter Buneman, Mary F. Fernandez, and Dan Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1):76–110, 2000.

[17] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[18] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. On propagation of deletions and annotations through views. In *PODS*, 2002.

[19] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *TCS*, 149(1):3–48, 1995.

[20] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Logical foundations of peer-to-peer data integration. In *PODS*, 2004.

[21] 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 *WebDB*, 2000.

[22] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[23] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseak Shim. Optimizing queries with materialized views. In *ICDE*, 1995.

[24] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of *Real* conjunctive queries. In *PODS*, 1993.

[25] Laura Chiticariu and Wang-Chiew Tan. Debugging schema mappings with routes. In *VLDB*, 2006.

[26] Noam Chomsky and Marcel-Paul Schützenberger. The algebraic theory of context-free languages. *Computer Programming and Formal Systems*, pages 118–161, 1963.

[27] Sara Cohen. Containment of aggregate queries. *SIGMOD Record*, 34(1):77–85, 2005.

[28] Sara Cohen. Equivalence of queries combining set and bag-set semantics. In *PODS*, 2006.

[29] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Deciding equivalences among conjunctive aggregate queries. *J. ACM*, 54(2), 2007.

[30] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *PODS*, 1999.

[31] Sara Cohen, Yehoshua Sagiv, and Werner Nutt. Equivalences among aggregate queries with negation. *ACM TOCL*, 6(2):328–360, April 2005.

[32] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill / MIT Press, 1990.

[33] Peter Crawley and Robert P. Dilworth. *Algebraic Theory of Lattices*. Prentice Hall, 1973.

[34] Yingwei Cui. *Lineage Tracing in Data Warehouses*. PhD thesis, Stanford University, 2001.

[35] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM TODS*, 25(2), 2000.

[36] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.

[37] Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. Well-definedness and semantic type-checking in the nested relational calculus and XQuery. In *ICDT*, 2005.

[38] A. Deutsch, L. Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1), 2006.

[39] Alin Deutsch and Val Tannen. Reformulation of XML queries and constraints. In *ICDT*, 2003.

[40] F. Dong and L. V. S. Lakshmanan. Deductive databases with incomplete information. In *Symposium on Logic Programming*, 1992.

[41] Xin Dong, Alon Y. Halevy, and Cong Yu. Data integration with uncertainty. In *VLDB*, 2007.

[42] Denise Draper, Peter Fankhauser, Mary Fernandez, Ashok Malhotra, Michael Rys, Jerome Simeon, and Philip Wadler. XQuery 1.0 formal semantics. Available from `http://www.w3.org/TR/xquery-semantics/`, 12 November 2003. W3C working draft.

[43] Oliver M. Duschka. *Query Planning and Optimization in Information Integration*. PhD thesis, Stanford University, Stanford, California, 1998.

[44] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *PODS*, 1997.

[45] Ronald Fagin, Phokion Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. *Theoretical Computer Science*, 336, 2005.

[46] Hao Fan and Alexandra Poulovassilis. Using schema transformation pathways for data lineage tracing. In *BNCOD*, volume 1, 2005.

[47] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), September 1999.

[48] J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated XML: Queries and provenance. In *PODS*, Vancouver, B.C., June 2008.

[49] Marc Friedman, Alon Y. Levy, and Todd D. Millstein. Navigational plans for data integration. In *AAAI '99*, 1999.

[50] N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *TOIS*, 14(1), 1997.

[51] Norbert Fuhr. Probabilistic datalog — a logic for powerful retrieval methods. In *SIGIR*, 1995.

[52] Ariel Fuxman, Phokion G. Kolaitis, Renée J. Miller, and Wang-Chiew Tan. Peer data exchange. In *PODS*, 2005.

[53] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey Ullman, and Jennifer Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2), March 1997.

[54] Floris Geerts and Antonella Poggi. On BP-complete query languages on K-relations. In *Workshop on Logic in Databases*, 2008.

[55] Goetz Graefe and William J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[56] Gösta Grahne, Nicolas Spyratos, and Daniel Stamate. Semantics and containment of queries with internal and external conjunctions. In *ICDT*, 1997.

[57] Gösta Grahne, Alex Thomo, and William W. Wadge. Preferentially annotated regular path queries. In *ICDT*, 2007.

[58] Todd J. Green. Containment of conjunctive queries on annotated relations. In *ICDT*, 2009.

[59] Todd J. Green, Zachary G. Ives, and Val Tannen. Reconcilable differences. In *ICDT*, 2009.

[60] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *VLDB*, Vienna, Austria, September 2007.

[61] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange. Submitted for publication in ACM TODS, 2009.

[62] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, Beijing, China, June 2007.

[63] Todd J. Green and Val Tannen. Models for incomplete and probabilistic information. In *EDBT Workshops*, 2006.

[64] Steven Gribble, Alon Halevy, Zachary Ives, Maya Rodrig, and Dan Suciu. What can databases do for peer-to-peer? In *WebDB*, June 2001.

[65] Ashish Gupta, Inderpal Singh Mumick, Jun Rao, and Kenneth A. Ross. Adapting materialized views after redefinitions: Techniques and a performance study. *Information Systems*, 26(5):323–362, 2001.

[66] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[67] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in Starburst. In *SIGMOD*, 1989.

[68] Alon Halevy, Michael Franklin, and David Maier. Principles of dataspace systems. In *PODS*, 2006.

[69] Alon Halevy, Zachary Ives, Jayant Madhavan, Peter Mork, Dan Suciu, and Igor Tatarinov. The Piazza peer data management system. *TKDE*, 16(7), July 2004.

[70] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: the teenage years. In *VLDB*, 2006.

[71] Alon Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4), 2001.

[72] Alon Y. Halevy, Zachary G. Ives, Peter Mork, and Igor Tatarinov. Piazza: Data management infrastructure for semantic web applications. In *12th World Wide Web Conference*, May 2003.

[73] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema mediation in peer data management systems. In *ICDE*, March 2003.

[74] Pavol Hell and Jaroslav Nešetřil. *Graphs and Homomorphisms*. Oxford University Press, 2004.

[75] Mauricio A. Hernandez, Renée J. Miller, and Laura M. Haas. Clio: A semi-automatic tool for schema mapping. In *SIGMOD*, 2001.

[76] André Hernich and Nicole Schweikardt. CWA-solutions for data exchange settings with target dependencies. In *PODS*, 2007.

[77] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. A formal model of dataflow repositories. In *DILS*, 2007.

[78] Edward Hung, Lise Getoor, and V. S. Subrahmanian. Probabilistic interval XML. *ACM TOCL*, 8(4), 2007.

[79] Tomasz Imielinski and Witold Lipski. Incomplete information in relational databases. *JACM*, 31(4), 1984.

[80] William H. Inmon. *Building the Data Warehouse*. Wiley, New York, 3rd edition, 2002.

[81] Yannis E. Ioannidis and Raghu Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *TODS*, 20(3):288–324, 1995.

[82] Zachary Ives, Nitin Khandelwal, Aneesh Kapur, and Murat Cakir. ORCHESTRA: Rapid, collaborative sharing of dynamic data. In *CIDR*, January 2005.

[83] Matthias Jarke, Maurizio Lenzerini, Yannis Vassiliou, and Panos Vassiliadis. *Fundamentals of Data Warehouses*. Springer, 2000.

[84] T. S. Jayram, Phokion G. Kolaitis, and Erik Vee. The containment problem for real conjunctive queries with inequalities. In *PODS*, 2006.

[85] Yaron Kanza, Werner Nutt, and Yehoshua Sagiv. Queries with incomplete answers over semistructured data. In *PODS*, 1999.

[86] Grigoris Karvounarakis. *Provenance in Collaborative Data Sharing*. PhD thesis, University of Pennsylvania, 2009.

[87] Grigoris Karvounarakis and Zachary G. Ives. Bidirectional mappings for data and update exchange. In *WebDB*, 2008.

[88] Anastasios Kementsietsidis and Marcelo Arenas. Data sharing through query translation in autonomous sources. In *VLDB*, 2004.

[89] Anastasios Kementsietsidis, Marcelo Arenas, and Renée J. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *SIGMOD*, June 2003.

[90] Anastasios Kementsietsidis, Marcelo Arenas, and Renee J. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *SIGMOD*, 2003.

[91] Jan Willem Klop. *Handbook of Logic in Computer Science*, volume 2, chapter 1. Oxford University Press, 1992.

[92] Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The Graph Isomorphism Problem: its Structural Complexity*. Birkhäuser Verlag, 1993.

[93] P. Kolaitis and M. Vardi. Conjunctive-query containment and constraint satisfaction. In *PODS*, 1998.

[94] Łucja Kot and Christoph Koch. Cooperative update exchange in the Youtopia system. *PVLDB*, 2(1):193–204, 2009.

[95] W. Kuich. Semirings and formal power series. In *Handbook of formal languages*, volume 1. Springer, 1997.

[96] Laks V. S. Lakshmanan, Nicola Leone, Robert Ross, and V. S. Subrahmanian. Probview: a flexible probabilistic database system. *TODS*, 22(3), 1997.

[97] Laks V. S. Lakshmanan and Fereidoon Sadri. Probabilistic deductive databases. In *Symposium on Logic Programming*, 1994.

[98] Laks V. S. Lakshmanan and Nematollaah Shiri. A parametric approach to deductive databases with uncertainty. *IEEE Trans. Knowl. Data Eng.*, 13(4):554–570, 2001.

[99] S. Kazem Lellahi and Val Tannen. A calculus for collections and aggregates. In *Category Theory and Computer Science*, 1997.

[100] Maurizio Lenzerini. Tutorial - data integration: A theoretical perspective. In *PODS*, 2002.

[101] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *PODS*, 1995.

[102] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, 1996.

[103] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, 1996.

[104] Leonid Libkin. Data exchange and incomplete information. In *PODS*, 2006.

[105] Leonid Libkin and Limsoon Wong. Query languages for bags and aggregate functions. *JCSS*, 55(2):241–272, 1997.

[106] Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, and Boon Thau Loo. Recursive computation of regions and connectivity in networks. In *ICDE*, 2009.

[107] László Lovász. Operations with structures. *Acta Mathematica Hungarica*, 18(3–4):321–328, 1967.

[108] James J. Lu, Guido Moerkotte, Joachim Schue, and V.S. Subrahmanian. Efficient maintenance of materialized mediated views. In *SIGMOD*, 1995.

[109] M. Maher and R. Ramakrishnan. Déjà vu in fixpoints of logic programs. In *NACLP*, 1989.

[110] Peter J. McBrien and Alexandra Poulovassilis. P2P query reformualtion over both-as-view data transformation rules. In *DBISP2P*, 2006.

[111] Prasenjit Mitra. An algorithm for answering queries efficiently using views. In *Proceedings of the Australasian Database Conference*, 2001.

[112] Peter Mork, Ron Shaker, Alon Halevy, and Peter Tarczy-Hornoch. PQL: A declarative query language over dynamic biological schemata. In *American Medical Informatics Association (AMIA) Symposium, 2002*, November 2002.

[113] Inderpal Singh Mumick. *Query Optimization in Deductive and Relational Databases*. PhD thesis, Stanford University, 1991.

[114] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *VLDB Journal*, 1990.

[115] Inderpal Singh Mumick and Oded Shmueli. Finiteness properties of database queries. In *Fourth Australian Database Conference*, 1993.

[116] Andrew Nierman and H. V. Jagadish. ProTDB: Probabilistic data in XML. In *VLDB*, 2002.

[117] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking forward. In *EDBT Workshops*, 2002.

[118] Lucian Popa, Yannis Velegrakis, Renée J. Miller, Mauricio A. Hernández, and Ronald Fagin. Translating web data. In *VLDB*, 2002.

[119] Rachel Pottinger and Alon Levy. A scalable algorithm for answering queries using views. In *VLDB*, 2000.

[120] Christopher Ré, Jéôme Simèon, and Mary Fernández. A complete and efficient algebraic compiler for xquery. In *ICDE*, page 14, 2006.

[121] Edward L. Robertson, Lawrence V. Saxton, Dirk Van Gucht, and Stijn Vansummeren. Structural recursion on ordered trees and list-based complex objects. In *ICDT*, 2007.

[122] A. Sabelfeld and A. C. Myers. Language-based Information-flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[123] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.

[124] Anish Das Sarma, Omar Benjelloun, Alon Halevy, and Jennifer Widom. Working models for uncertain data. In *ICDE*, 2006.

[125] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[126] Pierre Senellart and Serge Abiteboul. On the complexity of managing probabilistic XML data. In *PODS*, 2007.

[127] Jayavel Shanmugasundaram, H. Gang, Kristin Tufte, Chun Zhang, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, 1999.

[128] O. Shmueli. Equivalence of datalog queries is undecidable. *J. of Logic Programming*, 15, 1993.

[129] Partha Pratim Talukdar, Marie Jacob, Mohammad Salman Mehmood, Koby Crammer, Zack Ives, Fernando Pereira, and Sudipto Guha. Learning to create data-integrating queries. In *VLDB*, 2008.

[130] Wang-Chiew Tan. Containment of relational queries with annotation propagation. In *DBPL*, September 2003.

[131] Nicholas E. Taylor and Zachary G. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD*, 2006.

[132] Nicholas E. Taylor and Zachary G. Ives. Reliable storage and querying for collaborative data sharing systems. Submitted for publication, 2009.

[133] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, 1989.

[134] Maurice van Keulen, Ander de Keijzer, and Wouter Alink. A probabilistic XML approach to data integration. In *ICDE*, 2005.

[135] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.

[136] Cong Yu and Lucian Popa. Semantic adaptation of schema mappings when schemas evolve. In *VLDB*, 2005.

[137] Lotfi A. Zadeh. Fuzzy sets. *Inf. Control*, 8(3), 1965.

[138] Esteban Zimányi. Query evaluation in probabilistic relational databases. *TCS*, 171(1-2), 1997.