Foundations and Trends® in
Databases
Vol. 1, No. 4 (2007) 379–474
© 2009 J. Cheney, L. Chiticariu and W.-C. Tan
DOI: 10.1561/1900000006

now
the essence of knowledge

Provenance in Databases: Why, How, and Where

By James Cheney, Laura Chiticariu
and Wang-Chiew Tan

Contents

1 Introduction **380**

1.1 Why, How and Where: An Overview 382
1.2 Approaches in Computing Provenance: Eager vs Lazy 392
1.3 Notational Preliminaries 394

2 Why-Provenance **397**

2.1 Lineage 397
2.2 Why-Provenance 407

3 How-Provenance **416**

3.1 Provenance Semirings 417
3.2 Trio Lineage 421
3.3 Provenance Semirings and Recursion 427
3.4 How-Provenance for Schema Mappings 430

4 Where-Provenance **441**

4.1 Where-Provenance 441
4.2 Applications 450

5 Comparing Models of Provenance **453**

5.1 Relating Semiring-Based Techniques 454
5.2 Relating Lineage, Why-Provenance and Minimal Why-Provenance 458
5.3 Where-Provenance is "Contained" in Why-Provenance 459
5.4 Is Where-Provenance an Instance of How-Provenance? 460

6 Conclusions **463**

Acknowledgments **469**

References **470**

Foundations and Trends® in
Databases
Vol. 1, No. 4 (2007) 379–474
© 2009 J. Cheney, L. Chiticariu and W.-C. Tan
DOI: 10.1561/1900000006

now
the essence of knowledge

Provenance in Databases: Why, How, and Where

James Cheney[1], Laura Chiticariu[2] and Wang-Chiew Tan[3]

[1] *University of Edinburgh, UK, jcheney@inf.ed.ac.uk*
[2] *IBM Almaden Research Center, San Jose, CA, USA, chiti@almaden.ibm.com*
[3] *University of California, Santa Cruz, CA, USA, wctan@cs.ucsc.edu*

Abstract

Different notions of provenance for database queries have been proposed and studied in the past few years. In this article, we detail three main notions of database provenance, some of their applications, and compare and contrast amongst them. Specifically, we review why, how, and where provenance, describe the relationships among these notions of provenance, and describe some of their applications in confidence computation, view maintenance and update, debugging, and annotation propagation.

1

Introduction

Provenance information describes the origins and the history of data in its life cycle. Such information (also called *lineage*) is important to many data management tasks. Historically, databases and other electronic information sources were trusted because they were under centralized control: it was assumed that trustworthy and knowledgeable people were responsible for the integrity of data in databases or repositories. As argued by Lynch [49], this assumption is no longer valid for online data. Today, data is often made available on the Internet with no centralized control over its integrity: data is constantly being created, copied, moved around, and combined indiscriminately. Because information sources (or different parts of a single large source) may vary widely in terms of quality, it is essential to provide provenance and other context information which can help end users judge whether query results are trustworthy.

Data warehouses [17] and curated databases [10] are typical examples where provenance information is essential. In both data warehouses and curated databases, tremendous (and often manual) effort is usually expended in the construction of the resulting database — in the former, in specifying the extract-transform-load (ETL) process and in the

latter, in incrementally adding and updating the database. In a sense, provenance adds value to the data by explaining how it was obtained. Hence, it is of utmost importance to understand the provenance of data in the resulting database, in order to check the correctness of an ETL specification or assess the quality and trustworthiness of curated data.

Provenance has been studied in several different areas of data management, such as scientific data processing [8, 29, 53] and database management systems [15, 57]. We focus on provenance for data residing in a database management system. A number of notions of provenance in databases have been proposed in the literature. The most common forms of database provenance describe relationships between data in the source and in the output, for example, by explaining *where* output data came from in the input [58, 13], showing inputs that explain *why* an output record was produced [27, 13] or describing in detail *how* an output record was produced [43]. Besides being interesting in their own right for understanding the behavior of queries, these forms of provenance have been used in the study of classical database problems, such as view update [14] and the expressiveness of update languages [11]. More recently, they have also been used in the study of annotation propagation [7, 11, 58] and updates across peer-to-peer systems [42].

In this article, we focus on these three existing notions of why-, how- and where-provenance in databases. We shall describe them, discuss their applications, and compare and contrast these different notions in the subsequent sections. In the rest of this introductory section, we provide a high-level overview of these different notions of provenance, and introduce notation that will be used throughout the rest of the article. Sections 2, 3 and 4 focus on why-, how- and where-provenance, respectively, including formal details and applications. Section 5 discusses the relationships among the approaches, including proofs or disproofs of some "folklore" properties which have been stated in the literature but not (to our knowledge) carefully formalized and proved. Finally, Section 6 concludes with a brief discussion of additional related work and research challenges.

We emphasize that there are numerous other notions of provenance that are not described in this article. For example, provenance is also an active topic of research in scientific workflow management system

community and in the file and storage systems community. This article focuses on provenance within databases, and we refer the interested reader to the surveys [8, 53], and a recent tutorial [29] for a discussion on provenance research in general, as well as in the workflow community. Recent workshops [33, 35] also provide insight into the different views of provenance by diverse research communities.

Even in database settings, there is work that does not fit neatly into the why-, where- and how-provenance framework we focus on here, including early work such as Wang and Madnick's Polygen model [58] and Woodruff and Stonebraker's work on lineage [61], as well as Cui et al.'s lineage model [27] and more recent work on the Trio system [6]. We have chosen to focus on the why-, where-, and how-provenance framework because there is now enough related research on these models area to justify a critical review and comparison. We have recast lineage and a simplification of the Trio model as instances of our framework, but the Polygen, Woodruff–Stonebraker, and full Trio models seem to resist this categorization. Our classification should therefore be viewed as a preliminary attempt towards a full understanding of provenance in databases. We return to this issue in Section 6.

1.1 Why, How and Where: An Overview

1.1.1 Why-Provenance

Cui et al. [27] were among the first to formalize a notion of provenance, of data in the context of relational databases, called *lineage*. They associated each tuple t present in the output of a query with a set of tuples present in the input, called the *lineage* of t. Intuitively, the lineage of t is meant to collect all of the input data that "contributed to" t or helped to "produce" t. To illustrate, we use a simple example database of an online travel portal shown in Figure 1.1, where the labels $t_1, \ldots, t_8$ are used to identify the tuples. Consider the query Q_1[1] shown below, which asks for all travel agencies that offer external boat tours and their corresponding phone numbers by joining Agencies with ExternalTours

[1] Throughout the paper, we use SQL, relational algebra, and Datalog notation interchangeably, as convenient.

Agencies

	name	based_in	phone
t_1:	BayTours	San Francisco	415-1200
t_2:	HarborCruz	Santa Cruz	831-3000

ExternalTours

	name	destination	type	price
t_3:	BayTours	San Francisco	cable car	$50
t_4:	BayTours	Santa Cruz	bus	$100
t_5:	BayTours	Santa Cruz	boat	$250
t_6:	BayTours	Monterey	boat	$400
t_7:	HarborCruz	Monterey	boat	$200
t_8:	HarborCruz	Carmel	train	$90

Fig. 1.1 Our example database: an online travel portal.

on the name attribute, selecting tours by boat, and projecting on the name and phone attributes:

Q_1:
SELECT a.name, a.phone
FROM Agencies a, ExternalTours e
WHERE a.name = e.name AND e.type='boat'

Result of Q_1:

name	phone
BayTours	415-1200
HarborCruz	831-3000

The result of Q_1 executed on our example database in Figure 1.1 is shown above on the right. According to Cui et al., the lineage of the output tuple (HarborCruz, 831-3000) is $\{$Agencies(t_2), ExternalTours(t_7)$\}$, where Agencies(t_2) and ExternalTours(t_7) denote the subinstances of Agencies and ExternalTours consisting of tuples t_2 and t_7, respectively. Intuitively, the two source tuples witness the existence of the tuple of interest, (HarborCruz, 831-3000), according to Q_1. Furthermore, each of the two source tuples justify the existence of the HarborCruz tuple. In other words, the source tuples t_2 and t_7 form a "proof" or "witness" for the HarborCruz output tuple according to Q_1, and no other source tuples are part of the witness since they do not contribute to the HarborCruz output tuple. Technically speaking, by "witness" we mean a subset of the input database records that is sufficient to ensure that a given output tuple appears in the result of a query.

As another example, the lineage of the output tuple (BayTours, 415-1200) is the union of the lineage of the intermediate

tuples — (BayTours, San Francisco, 415-2000, Santa Cruz, boat, $100) and (BayTours, San Francisco, 415-2000, Monterey, boat, $250) — before the projection operator is applied on name and phone. The union of the lineage of these two intermediate tuples gives {Agencies(t_1), ExternalTours(t_5,t_6)}. Observe that this lineage representation is not as precise as one may like as it does not specify that t_5 and t_6 do not need to coexist together in order to witness the BayTours output tuple. Indeed, $\{t_1, t_5\}$ and respectively, $\{t_1, t_6\}$ are two different witnesses for the BayTours tuple. This illustrates that not every tuple in the lineage is "necessary" for the output (BayTours, 415-1200) to be produced.

This intuition was formalized by Buneman et al. [13] who introduced the notion of *why-provenance* that captures the different witnesses. Their work is in the context of a semi-structured data model with a query language that is appropriate for that data model, but we shall restrict our discussion to the relational model with select–project–join queries here.

Like lineage, why-provenance is based on the idea of providing information about the witnesses to a query. Recall that a witness is a subset of the database records that is sufficient to ensure that a given record is in the output. There may be a large number of such witnesses because many records are "irrelevant" to the presence of an output record of interest. In fact, the number of witnesses can easily be exponential in the size of the input database. To avoid this problem, the definition of why-provenance restricts attention to a smaller number of witnesses. According to [13], the why-provenance of an output tuple t in the result of a query Q applied to a database D is defined as the *witness basis* of t according to Q. The witness basis is a particular set of witnesses which can be calculated efficiently from Q and D. This is generally much smaller than the full witness set. However, every witness contains an element of the witness basis, so the witness basis can be viewed as a compact representation of the set of all witnesses.

Going back to our example, the why-provenance of (BayTours, 415-1200) in the result of Q_1 is the set $\{\{t_1, t_5\}, \{t_1, t_6\}\}$. There are two witnesses, corresponding to $\{t_1, t_5\}$ and $\{t_1, t_6\}$, respectively. Intuitively, this tells us that the output tuple is witnessed by source tuples in two different ways according to Q_1: the first uses the tuples t_1 and

Instance I:

R

	A	B
t:	1	2
t':	1	3
t'':	4	2

Two equivalent queries:

$Q : Ans(x,y) :- R(x,y).$

$Q' : Ans(x,y) :- R(x,y), R(x,z).$

Output of $Q(I)$, $Q'(I)$:

A	B
1	2
1	3
4	2

Fig. 1.2 Example queries, input and output.

Instance I:

R

	A	B
t:	1	2
t':	1	3
t'':	4	2

Output of $Q(I)$

A	B	*why*
1	2	$\{\{t\}\}$
1	3	$\{\{t'\}\}$
4	2	$\{\{t''\}\}$

Output of $Q'(I)$

A	B	*why*
1	2	$\{\{t\},\{t,t'\}\}$
1	3	$\{\{t'\},\{t,t'\}\}$
4	2	$\{\{t''\}\}$

Fig. 1.3 Example showing that why-provenance is sensitive to query rewriting.

t_5, while the second uses the tuples t_1 and t_6. Observe that $\{t_1, t_5, t_6\}$ is not a minimal witness, since the query Q_1 requires witnesses to consist of exactly one tuple from Agencies, and one tuple from ExternalTours according to the `FROM` clause of Q_1.

The preceding discussion suggests that the witness basis may be tied to the structure of the query and it is therefore sensitive to how a query is formulated. To illustrate, consider the instance I and two equivalent queries Q and Q' shown in Figure 1.2. For conciseness, we use the Datalog conjunctive query notation to express Q and Q' here and throughout the paper as convenient. Consider the output tuple $(1,2)$ in the result of Q (and Q') applied to I shown in Figure 1.3. The witness basis of this output tuple is $\{\{t\}\}$, according to Q and I. However, even though Q' is equivalent to Q, the witness basis of the output tuple $(1,2)$ according to Q' and I is $\{\{t\},\{t,t'\}\}$.

Although equivalent queries may have different witness bases, Buneman et al. [13] showed that a subset of the witness basis, called the *minimal witness basis*, is invariant under equivalent queries. The minimal witness basis consists of all the *minimal* witnesses in the witness basis, where a witness is minimal if none of its proper subinstances is also a witness in the witness basis. For example, $\{t\}$ is a minimal witness for the output tuple $(1,2)$ in Figure 1.2. However, $\{t, t'\}$ is not a

minimal witness since $\{t\}$ is a subinstance of it and it is a witness to (1,2). Hence, the minimal witness basis is $\{\{t\}\}$ for this example. In a subsequent work by [14], minimal witnesses were used in the study of variants of the view deletion problem, which is that of finding source tuples to remove in order to delete a tuple from the view for select-project–join–union queries.

1.1.2 How-Provenance

Why-provenance describes the source tuples that witness the existence of an output tuple in the result of the query. However, it leaves out some information about *how* an output tuple is derived according to the query. To illustrate, consider the query Q_2 of Figure 1.4 which asks for all cities where tours are offered (assuming all agencies offer tours in the city they are headquartered). The result of Q_2 on the example database in Figure 1.1 is shown in the right of Figure 1.4. (Ignore the additional tags on the output tuples for now.) For the output tuple (San Francisco, 415-1200) in the result of Q_2, its why-provenance is $\{\{t_1\}, \{t_1,t_3\}\}$. This description tells us that t_1 alone, and t_1 with t_3 are each sufficient to witness the existence of the output tuple according to Q_2. However, it does not tell us about the structure of the proof that t_1 (as well as t_1 and t_3) help witness the output tuple according to Q_2. Although arguably obvious from the description of the query Q_2, the why-provenance does not tell us that the source tuple t_1 contributes twice to the output tuple: (1) t_1 contributes to the intermediary result of the inner query, and (2) it combines with that intermediary result to witness the output tuple. This intuition is formalized in [43] using

Q_2:
```
SELECT  e.destination, a.phone
FROM    Agencies a,
        (SELECT name,
           based_in AS destination
        FROM Agencies a
        UNION
        SELECT name, destination
        FROM ExternalTours ) e
WHERE   a.name = e.name
```

Result of Q_2:

destination	phone	
San Francisco	415-1200	$t_1 \cdot (t_1 + t_3)$
Santa Cruz	831-3000	t_2^2
Santa Cruz	415-1200	$t_1 \cdot (t_4 + t_5)$
Monterey	415-1200	$t_1 \cdot t_6$
Monterey	831-3000	$t_1 \cdot t_7$
Carmel	831-3000	$t_1 \cdot t_8$

Fig. 1.4 A query and its output tagged with semiring provenance.

Instance I:

R

	A	B
t:	1	2
t':	1	3
t'':	4	2

Output of $Q(I)$

A	B	*how*
1	2	t
1	3	t'
4	2	t''

Output of $Q'(I)$

A	B	*how*
1	2	$t^2 + t \cdot t'$
1	3	$(t')^2 + t \cdot t'$
4	2	$(t'')^2$

Fig. 1.5 Example showing that how-provenance is sensitive to query rewriting.

provenance semirings. Intuitively, the provenance of the output tuple (San Francisco, 415-1200) is represented as a polynomial, which for this example is $t_1^2 + t_1 \times t_3$. The polynomials for each output tuple are shown on the right of the result of Q_2. The polynomial hints at the structure of the proofs by which the output tuple is derived. In this example, the polynomial describes that the output tuple is witnessed in two distinct ways: once using t_1 twice, and the other using t_1 and t_3. As we shall show, one can derive the why-provenance of an output tuple from its how-provenance polynomial. However, this example shows that the converse is not always possible.

It is easy to see that how-provenance is also sensitive to query formulations, since how-provenance is more general than why-provenance. Going back to our example queries shown on the top of Figure 1.2, Figure 1.5 illustrates that the how-provenance of the tuple $(1,2)$ in the output of $Q(I)$ is t according to Q, and respectively, $t^2 + t \times t'$ according to Q'.

Green et al. [43] formalize a notion of how-provenance for relational algebra in terms of an appropriate "provenance semiring", and extend their approach to handle recursive datalog. Subsequently, an interesting application of how-provenance appears in the context of ORCHESTRA [42, 44], a collaborative data sharing system in a network of peers interconnected through schema mappings. An extension of the semiring model of Green et al. [43] to schema mappings is used in ORCHESTRA to efficiently support trust-based filtering of updates, and incremental maintenance of peers' databases with updates in the system.

Earlier, Chiticariu and Tan proposed a notion of provenance over schema mappings called *routes* [21], and used it as a basis for SPIDER, a system for debugging schema mappings [3]. Given a schema mapping that relates a source and a target schema, routes describe *how* data in

the source instance is related to data in the target instance through the schema mapping. Hence, in retrospect, routes can be classified as a form of how-provenance over schema mappings.

1.1.3 Where-Provenance

Why-provenance describes all combinations of source tuples that witness the existence of an output tuple in the result of a query. In turn, how-provenance describes how the source tuples witness the output tuple. Buneman et al. also introduced a different notion of provenance, called *where-provenance* [13]. Intuitively, where-provenance describes where a piece of data is copied from. While why-provenance is about the relationship between source and output tuples, where-provenance describes the relationship between source and output *locations*. In the relational setting, a location is simply a column of a tuple in a relation, which precisely refers to a "cell" in a relation. The where-provenance of a value that resides in some location l in $Q(D)$ consists of *locations* of D from which the value in l was copied according to Q. Naturally, this requires that all the values that reside in the source locations of the where-provenance of l are equal to the value that resides at l. For example, the where-provenance of the value "HarborCruz" in the second output tuple in the result of Q_1 is the location (Agencies, t_2, name) (or simply, (t_2, name)) in our example database, since "HarborCruz" was copied from the name attribute of the tuple t_2 in the Agencies relation, according to Q_1.

Where-provenance is also not invariant under equivalent queries. To illustrate, consider the queries Q_1 (repeated from earlier) and Q'_1. The only difference between Q_1 and Q'_1 is in the select clause. The first attribute of the select clause of Q_1 is a.name, whereas the first attribute of the select clause of Q'_1 is e.name.

Q_1:
```
SELECT  a.name, a.phone
FROM    Agencies a, ExternalTours e
WHERE   a.name = e.name
        AND e.type='boat'
```

Q'_1:
```
SELECT  e.name, a.phone
FROM    Agencies a, ExternalTours e
WHERE   a.name = e.name
        AND e.type='boat'
```

Clearly, the queries Q_1 and Q'_1 are equivalent in that they produce the same resulting tuples for any given input database. However, the where-provenance of the output value "HarborCruz" is different under the two queries. As explained earlier, the where-provenance of "HarborCruz" in the output according to Q_1 is the location (t_2, name), since "HarborCruz" is copied from the name attribute of the tuple t_2 in Agencies. With Q'_1, however, the where-provenance of "HarborCruz" is (t_7, name), since "HarborCruz" is copied from the name attribute of t_7 in ExternalTours. Arguably, the where-provenance of "HarborCruz" according to Q_1 and Q'_1 is identical once we take the equality "a.name = e.name" into consideration in Q_1. However, as we shall discuss later with DBNotes, where-provenance is still not invariant under equivalent queries even after such "equality checks" are incorporated.

According to Buneman et al.'s definition of why and where-provenance [13], if a value v of a location of an output tuple t is not constructed by a query Q, then it must have been copied from values that reside in some source locations of a witness of t according to Q. As a consequence, the where-provenance of v consists of locations that can be found in tuples of the why-provenance of t. (We prove this more carefully in Section 5, Proposition 5.11.) For example, consider Q'_1 that was described earlier and the output tuple (BayTours, 415-1200), which we denote as t. The why-provenance of t according to Q'_1 is $\{\{t_1, t_5\}, \{t_1, t_6\}\}$. The where-provenance of the value "BayTours" at location (t, name) consists of two locations (t_5, name) and (t_6, name). Indeed, these two locations are among the locations of tuples in the witnesses of t. Observe that although t_1 is part of every witness for t according to Q'_1, the locations of t_1 are not among the locations in the where-provenance of (t, name). On the other hand, the where-provenance of (t_1, phone) is (t, phone). But in general, it is possible for the why-provenance to contain tuples whose locations contribute nothing to the where-provenance of any part of the output.

One interesting application of where-provenance has been in the study of annotation–propagation and update languages [7, 14, 58]. Annotation–propagation is closely related to provenance: a given notion of provenance can be viewed as a method for propagating annotations

from the input to the output, whereas a given annotation–propagation semantics can be viewed as a form of provenance by placing distinct annotations on each part of the input and observing where they end up in the output.

The idea of forwarding provenance information during query execution was first explored in the Polygen system [58]. In Polygen, operational rules for forwarding information about the source databases, as well as the intermediate databases that contributed to the creation of an output piece of data, are defined for basic relational operators. The where-provenance propagation rules are similar to the rules that propagate *origin source tags* (i.e., references to original sources) in Polygen [58].

In Buneman et al.'s work on annotation propagation [14], as implemented in DBNotes [7], the where-provenance of a location in the result of a query determines the set of all annotations in the source database to be associated with that output location. This approach assumes queries involve select, project, join, and union only; the where-provenance of an output location is described through a set of *propagation rules*, one for each relational operator (i.e., select, project, join, union). In contrast to Polygen, these approaches propagate arbitrary annotations through a query, and not only information about source and intermediary databases.

DBNotes [7, 22] is an annotation management system for relational databases which builds upon previous ideas in where-provenance and Polygen. DBNotes propagates annotations from source locations to output locations based on where-provenance. Queries in DBNotes are also select–project–join–union queries except that they are expressed in declarative SQL-like expressions, and not relational operators as in [14]. Like [14], every location can be associated with zero or more annotations and these annotations are propagated to the output when a query is executed. The *default* annotation–propagation behavior of queries in DBNotes forwards annotations to an output location based on the where-provenance of that output location. For a simple example, consider the annotated relation I_{a} shown in Figure 1.6 and the queries Q and Q' from Figure 1.2. Each of the six locations of I_{a} is associated with one annotation, denoted as a_i, where $1 \leq i \leq 6$. The execution of

Annotated instance I^a:
R

	A	B
t:	1^{a_1}	2^{a_2}
t':	1^{a_3}	3^{a_4}
t'':	4^{a_5}	2^{a_6}

Output of $Q(I^a)$ (DEFAULT propagation):

A	B
1^{a_1}	2^{a_2}
1^{a_3}	3^{a_4}
4^{a_5}	2^{a_6}

Output of $Q'(I^a)$ (DEFAULT propagation):

A	B
$1^{a_1,a_3}$	2^{a_2}
$1^{a_1,a_3}$	3^{a_4}
4^{a_5}	2^{a_6}

Output of $Q(I^a)$, $Q'(I^a)$ (DEFAULT-ALL propagation):

A	B
$1^{a_1,a_3}$	$2^{a_2,a_6}$
$1^{a_1,a_3}$	3^{a_4}
4^{a_5}	$2^{a_2,a_6}$

Fig. 1.6 Example showing that where-provenance is sensitive to query rewriting.

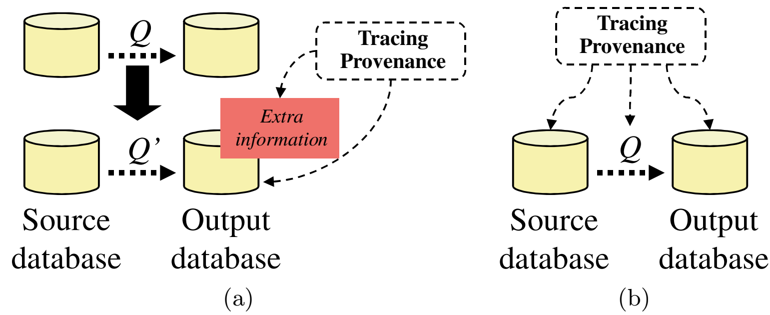
Q and respectively, Q' on I_{a} under the default propagation scheme produces the two annotated instances shown in Figure 1.6. In the output of Q, the annotation a_1 propagates from the value "1" of the source tuple t to the output value "1" of (1, 2) in $Q(I_{\mathrm{a}})$. This is because the value "1" of (1, 2) in $Q(I_{\mathrm{a}})$ is copied from the value "1" of t according to Q. In the case of Q', however, the value "1" of (1, 2) in $Q'(I_{\mathrm{a}})$ is copied from "1" of t or "1" of t' in I_{a}. Hence, two annotations, a_1 and a_3, appear with the value "1" of (1, 2) in $Q'(I_{\mathrm{a}})$. This simple example illustrates once more that where-provenance is sensitive under equivalent query formulations: while Q and Q' are equivalent, they produce different annotated results. In fact, the query Q'': $Ans(x,y) :- R(x,y), R(z,y)$ is also equivalent to Q and it propagates both a_2 and a_6 to the values "2" in the output, whereas the two copies of value "1" in the output is annotated with a_1 and respectively, a_3.

If a query Q propagates annotations under the *default-all* propagation scheme in DBNotes, then equivalent formulations of Q are guaranteed to produce identical annotated results. In the default-all scheme, annotations are propagated based on where data is copied from according to *all* equivalent queries of Q. Hence, this propagation scheme can be perceived as a "better" method for propagating annotations for Q. The result of executing Q (or Q' or Q'') on I_{a} under the default-all scheme is shown in Figure 1.6. Observe that all annotations relevant for an output value are associated under the same output value in the default-all behavior, regardless of how the query is formulated. For this example, both "1"s in the default-all output are associated with a_1 and a_3. This is because Q', which is an equivalent query of Q, associates both annotations with the value "1". Similarly, both "2"s in the default-all output are associated with a_2 and a_6. This is because Q'' associates

both annotations with the value "2". In fact, given Q, DBNotes generates a finite set of equivalent queries (in this case, $\{Q, Q', Q''\}$) that captures all the relevant annotations that would be propagated by any equivalent query of Q.

1.2 Approaches in Computing Provenance: Eager vs Lazy

Along with our discussion of the three notions of provenance, we shall also give an overview of a few recent systems where provenance is an integral component, and describe the algorithms for computing provenance implemented in these systems. Figure 1.7(c) illustrates a classification of the systems we shall discuss in some detail in this paper, based on the approach each takes in computing provenance. There are two approaches for computing provenance: the *eager approach* and the *lazy approach*. In this article, we describe the basic ideas behind the two approaches, and defer the discussion of system implementation details to Sections 2–4.



	Eager Approach	*Lazy Approach*
Why-provenance		WHIPS (i.e., lineage) [25, 27]
How-provenance	ORCHESTRA [42, 44] Trio [2, 5, 52]	SPIDER [3, 21]
Where-provenance	DBNotes [7, 22]	

(c)

Fig. 1.7 Approaches in computing data provenance: (a) the eager approach; (b) the lazy approach. (c) A classification of recent systems for computing data provenance.

Figures 1.7(a) and 1.7(b) illustrate the two possible approaches for computing provenance. In the *eager approach* (also known as the *bookkeeping* or *annotation* approach), the query is re-engineered so that extra annotations are carried over to the output database during the transformation, to help answer provenance. As a consequence, the provenance of a piece of output data can usually be derived by examining the output database and the extra information. In the *lazy approach* (also known as *non-annotation* approach), provenance is computed when needed — by examining the source data, the output data, and the transformation. In contrast to the eager approach, the lazy approach does not require the re-engineering of the transformation for the purpose of carrying additional information to the output database.

Both approaches have advantages, as well as disadvantages, and they are appropriate in different scenarios. Since additional information is carried over and stored along with actual data in the output database, an eager approach involves a performance overhead during the execution of the transformation, as well as a space overhead for storing the extra information in the output. Recently, various schemes aimed at reducing the amount of extra information stored have been investigated in [31, 38, 54], and the problem of compressing or approximating provenance has been explored in [9, 16, 50]. However, the eager approach has the advantage that if the right additional information is propagated, provenance may be derived directly from the output database and the extra information, without examining the source database. Hence, an eager approach is useful in scenarios where the source data may become unavailable after the transformation. The lazy approach does not require the re-engineering of the transformation. Hence, it has the advantage that it can be readily deployed on an existing system without changes to the system, and furthermore, it does not incur any performance or storage overhead during the execution of the transformation. Thus, a lazy approach is useful when storage space is an issue, or, when it is not possible to modify the implementation of the query execution system. A disadvantage of the lazy approach is that deriving provenance usually involves sophisticated techniques for reasoning about the source database, the output database, and the

transformation. Hence, the lazy approach cannot be used if the source data becomes unavailable.

Although most existing work takes a distinctively eager or lazy approach, it might be interesting to consider hybrid approaches that take advantage of the best characteristics of both the approaches. In fact, the WHIPS lineage-tracking system [25, 27] combines eager and lazy ideas in its handling of queries involving negation and aggregation.

1.3 Notational Preliminaries

In this section, we introduce (largely standard) notation we shall use in the rest of the paper.

Let $\mathbf{D}$ be a finite domain of data values $\{d_1,\ldots,d_n\}$ and $\mathcal{U}$ a collection of *field names* (or attribute names). We will use the symbols U,V for (finite) subsets of $\mathcal{U}$. A record (or tuple) $t,t_1,t_2,t',u,\ldots$ is a function $U \to \mathbf{D}$, written as $(A_1{:}d_1,\ldots,A_n{:}d_n)$. A tuple assigning values to each field name in U is called U-tuple; e.g. $(A_1{:}d_1,\ldots,A_n{:}d_n)$ is a $\{A_1,\ldots,A_n\}$-tuple. We write *Tuple* for the set of all tuples and U-*Tuple* for the set of all U-tuples. We write $t \bullet A$ for the value of the A-field of t, $t[U]$ for the restriction of tuple t over $V \supseteq U$ to field names in U, and $t[A \mapsto B]$ for the result of renaming field A to B in t (assuming B is not already present in t). We sometimes write $(A{:}e(A))_{A\in U}$ to define a tuple $t{:}U$ such that $t \bullet A = e(A)$ for each $A \in U$.[2] Here, $e(A)$ is an expression parameterized by an unknown field name A. For example, if $t{:}V$ then we can express the projection $t[U]$ using this notation as $(A{:}t \bullet A)_{A\in U}$.

A *relation* or table $r{:}U$ is a finite set of tuples over U. Let $\mathcal{R}$ be a finite collection of *relation names.* A *schema* $\mathbf{R}$ is a mapping $(R_1{:}U_1,\ldots,R_n{:}U_n)$ from $\mathcal{R}$ to finite subsets of $\mathcal{U}$. A *database (or instance)* $I{:}(R_1{:}U_1,\ldots,R_n{:}U_n)$ is a function mapping each $R_i{:}U_i \in \mathbf{R}$ to a relation r_i over U_i.

We also define *tuple locations* as tuples tagged with relation names, written (R,t). We write $TupleLoc = \mathcal{R} \times Tuple$ for the set of all tagged tuples. We can view a database instance I equivalently as a finite set

[2] For readers familiar with lambda-calculus notation for function definition, note that $(A{:}e(A))_{A\in U}$ is equivalent to $\lambda A \in U \cdot e(A)$.

$\{(R,t) \mid t \in I(R)\} \subseteq \mathit{TupleLoc}$ of such tagged tuples according to a standard translation. We will also sometimes consider *field locations* that refer to a particular field of a tagged tuple. Formally, such a location is just a triple $(R,t,A) \in \mathcal{R} \times \mathit{Tuple} \times U$. We write *FieldLoc* for the set of all locations.

We will use the following notation for (monotone) relational algebra queries:

$$Q ::= R \mid \{t\} \mid \sigma_\theta(Q) \mid \pi_U(Q) \mid Q_1 \bowtie Q_2 \mid Q_1 \cup Q_2 \mid \rho_{A \mapsto B}(Q)$$

Here, $\{t\}$ is a *singleton constant* $\{t\}$. Selections σ_θ filter a relation by retaining tuples satisfying some predicate θ. We leave the form of predicates unspecified (but typically include field equality tests $A = B$ and $A = d$). Projections $\pi_U(Q)$ replace each tuple t in a relation with $t[U]$, discarding any other fields. Join (or natural join) and union are standard; renaming is written $\rho_{A \mapsto B}(Q)$.

The precise semantics $Q(I)$ of a query Q evaluated against an instance I is described below. We review this standard definition only because we will be considering a number of variations on it later.

$$\begin{aligned}
(\{t\})(I) &= \{t\} \\
R(I) &= I(R) \\
(\sigma_\theta(Q))(I) &= \{t \in Q(I) \mid \theta(t)\} \\
(\pi_U(Q))(I) &= \{t[U] \mid t \in Q(I)\} \\
(Q_1 \bowtie Q_2)(I) &= \{t \mid t[U_1] \in Q_1(I), \quad t[U_2] \in Q_2(I)\} \\
(Q_1 \cup Q_2)(I) &= Q_1(I) \cup Q_2(I) \\
(\rho_{A \mapsto B}(Q))(I) &= \{t[A \mapsto B] \mid t \in Q(I)\}
\end{aligned}$$

Here, we assume that Q has the set of attributes V, denoted as $Q{:}V$, that $U \subseteq V$ in the case of projection, and that $Q_1{:}U_1, Q_2{:}U_2$ in the case of join.

As mentioned earlier, when convenient we also employ Datalog notation using nameless tuples, and assume familiarity with the standard translation between SPJRU queries and unions of conjunctive Datalog queries. For example, the query $\{(A(x,y) :- R(x,y), S(x,z)), (A(x,x) :- R(x,x))\}$ is equivalent to $(R \bowtie S) \cup \sigma_{A=B}(R)$, where we assume schema $R(A,B)$ and $S(A,C)$.

We also employ the following convention regarding partial functions (which is standard in, for example, programming language semantics). Formally, we can view a partial function $f: X \to Y$ as a total function $f: X \to Y \cup \{\bot\}$, where $\bot$ is a special, fresh constant not already present in Y, called "undefined". We write $Y_\bot$ to abbreviate $Y \cup \{\bot\}$, and we define $\mathrm{dom}(f) = \{x \in X \mid f(x) \neq \bot\}$.

One advantage of this convention is that it permits unambiguous definitions of operations with different behavior regarding undefinedness. For example, we will later make use of *strict* and *lazy* union operations. Strict union $\cup_\mathrm{S}$ is defined as the union of two sets if both are defined, and undefined otherwise (that is, $X \cup_\mathrm{S} \bot = \bot$), whereas lazy union $\cup_\mathrm{L}$ differs from strict union in that it is undefined only if both sets are undefined (that is, $X \cup_\mathrm{L} \bot = X$). We will define these operations more carefully later.

The various provenance semantics we shall consider will be defined by interpreting the language of relational queries over other classes of structures besides relations. A familiar example of this technique is interpreting queries over bags (multisets) instead of set relations. The semiring provenance semantics discussed earlier directly generalizes both relation and multiset semantics, and several other provenance semantics are instances of the semiring semantics.

2

Why-Provenance

In this section, we start by discussing *lineage*, a notion of provenance for relational databases formalized by Cui et al. [27]. We also give an overview of their algorithms for computing lineage implemented in the WHIPS data warehouse system [25]. After this, we describe the *why-provenance* model of Buneman et al. [13], which is a refinement of the lineage model. Why-provenance is characterized by *witness bases*. Intuitively, every witness in the witness basis is a proof for why an output tuple exists. Naturally, the elimination of all such proofs will cause the output tuple to disappear. A variant of the notion of witness basis is subsequently used in the study of the view deletion problem by Buneman et al. [14], which we shall also describe in this section.

2.1 Lineage

In the work of Cui et al. [27], the *lineage* of an output record is based on identifying a subset of input records relevant to the output record. Intuitively, an input record is relevant to an output record if it contributed to the existence of that output record. The following definition, which

makes precise what relevant means, is paraphrased from Definition 4.1 of [27]:

Definition 2.1 (Lineage for a relational operator [27, Definition 4.1]). Let Op be any relational operator over relations $R_1, \ldots, R_n$, The *lineage* of a record $t \in Op(R_1, \ldots, R_n)$ is a sequence $\langle R'_1, \ldots, R'_n \rangle$ of subsets $R'_i \subseteq R_i$, such that:

(1) $Op(R'_1, \ldots, R'_n) = \{t\}$.
(2) For each $1 \leq i \leq n$ and for each $t_i \in R'_i$, we have $Op(R'_1, \ldots, R'_{i-1}, \{t_i\}, \ldots, R'_n) \neq \emptyset$.
(3) $\langle R'_1, \ldots, R'_n \rangle$ is *maximal* among subsets of $R_1, \ldots, R_n$ satisfying (1) and (2).

Intuitively, (1) ensures that the lineage is "relevant" to t, (2) ensures that no "irrelevant" records are included in the lineage, and (3) ensures that the lineage is "complete". In addition, Cui et al. provided operational definitions of the lineage of a tuple for each basic relational operator (i.e., selection, projection, join, union, renaming, difference, and aggregation operators). The operational definitions are shown below. Cui et al. proved that the operational definitions below coincide with Definition 2.1 for queries consisting of individual operators.

Theorem 2.1 ([27, Theorem 4.4], paraphrased).

(1) If $t \in \sigma_\theta(R)$ then the lineage of t is $\langle \{t\} \rangle$.
(2) If $t \in \pi_U(R)$ then the lineage of t is $\langle \{t' \in R \mid t'[U] = t\} \rangle$.
(3) If $t \in R_1 \bowtie \cdots \bowtie R_n$ then the lineage of t is $\langle \{t_1 \in R_1 \mid t_1 = t[U_1]\}, \ldots, \{t_n \in R_n \mid t_n = t[U_n]\} \rangle$, where $U_1, \ldots, U_n$ are the attributes of $R_1, \ldots, R_n$, respectively.
(4) If $t \in R_1 \cup \cdots \cup R_n$ then the lineage of t is $\langle \{t_1 \in R_1 \mid t_1 = t\}, \ldots, \{t_n \in R_n \mid t_n = t\} \rangle$.
(5) If $t \in R_1 \backslash R_2$ then the lineage of t is $\langle \{t\}, R_2 \rangle$.
(6) If $t \in \alpha_{G,\mathrm{aggr}(B)}(R)$ (where $\alpha_{G,\mathrm{aggr}(B)}(R)$ denotes grouping by fields G and aggregating the B fields by aggr) then the lineage of t is $\langle \{t' \in R \mid t'[G] = t[G]\} \rangle$.

It is worth pointing out that in Cui et al.'s definition, it is assumed that queries have no repeated relation names (and thus, no self-joins). Queries involving the same relation more than once are treated by considering each occurrence of the relation as a separate "copy". For example, a self-join $R \bowtie R$ is rewritten into $(R \text{ AS } R_1) \bowtie (R \text{ AS } R_2)$ and is considered to operate on two separate copies of the instance R, corresponding to R_1 and R_2, respectively.

For complex queries that are composed of a sequence of more than one relational operator, lineage is defined inductively. Let $Op_1 \circ \cdots \circ Op_n$ be a relational algebra query where Op_i, $1 \leq i \leq n$, denotes a relational operator. Let I be a database instance, let V be the resulting output of applying $Op_2 \circ \cdots \circ Op_n$ to I, and let t be a tuple in V. Roughly speaking, the lineage of t in I with respect to $Op_1 \circ \cdots \circ Op_n$ is a subinstance I^* of I such that (1) I^* is the lineage of a subinstance V^* of V in I according to $Op_2 \circ \cdots \circ Op_n$, and (2) V^* is the lineage of t in V according to Op_1. The definition shown below is paraphrased from Definition 4.5 of [27].

Definition 2.2 (Lineage for a view [27, Definition 4.5]). Let Q be a (complex) SPJRU query over $R_1, \ldots, R_n$.

- If $Q = R_i$, then the lineage of $t \in R_i$ is $R_i(t)$.
- If $Q = Op(Q_1, \ldots, Q_k)$ where each Q_i is an SPJRU query over $R_1, \ldots, R_n$, then let $S_1 = Q_1(I), \ldots, S_k = Q_k(I)$, and suppose the lineage of t in $Op(S_1, \ldots, S_k)$ is L. For each $S_i(t') \in L$, let $L_{S_i(t')}$ be the lineage of t' in Q_i with respect to $Q_i(I)$. Then the lineage of t in $Q(I)$ is the union of all such sets $L_{S_i(t')}$.

To illustrate, consider the example database from Figure 1.1. The query Q_1 from Section 1.1.1 asking for travel agencies that offer external boat tours and their corresponding phone numbers can be expressed in relational algebra as $\pi_{\text{name,phone}}(\sigma_{\text{type='boat'}}(\text{Agencies} \bowtie \text{ExternalTours}))$. The query, together with its result when applied to the travel portal database, is reproduced.

Q_1:
SELECT a.name, a.phone
FROM Agencies a, ExternalTours e
WHERE a.name = e.name
AND e.type='boat'

Result of Q_1:

name	phone
BayTours	415-1200
HarborCruz	831-3000

The lineage of the tuple (BayTours, 415-1200) is the union of the lineages of intermediary tuples (BayTours, San Francisco, 415-2000, Santa Cruz, boat, \$250) and (BayTours, San Francisco, 415-2000, Monterey, boat, \$400) according to $\sigma_{\text{type='boat'}}(\text{Agencies} \bowtie \text{ExternalTours})$ obtained before the projection operator is applied. In turn, the lineage of the first intermediary tuple is the lineage of the tuple (BayTours, San Francisco, 415-2000, Santa Cruz, boat, \$250) in the intermediary result of Agencies $\bowtie$ ExternalTours whose lineage is in turn $\langle\{\text{Agencies}(t_1)\}, \{\text{ExternalTours}(t_5)\}\rangle$. Similarly, the lineage of the second intermediary tuple is the lineage of the tuple (BayTours, San Francisco, 415-2000, Monterey, boat, \$400) in the intermediary result of Agencies $\bowtie$ ExternalTours whose lineage is in turn $\langle\{\text{Agencies}(t_1)\}, \{\text{ExternalTours}(t_6)\}\rangle$. Thus, by unioning the lineages of the two intermediary tuples we obtain the lineage of (BayTours, 415-1200) in the result of Q_1 as $\langle\{\text{Agencies}(t_1)\}, \{\text{ExternalTours}(t_5, t_6)\}\rangle$.

2.1.1 A Compositional Definition

The original definition of lineage is somewhat difficult to work with or to relate to other forms of provenance, and it omits some common features such as renaming and constant queries. In this section, we develop an equivalent definition that defines lineage directly in terms of propagating annotations from the input to the output. The definition is compositional in the sense that we define lineage via annotation–propagation behavior of each relational operator independently.

We define lineage as a *partial function* mapping a query $Q : \mathbf{R} \to U$, an $\mathbf{R}$-instance I and a U-tuple t to either an $\mathbf{R}$-instance $\mathsf{Lin}(Q, I, t)$, or a special constant $\perp$ (meaning "undefined"). We will prove that this definition is equivalent to that of Cui et al. for SPJU queries without renaming or constants, and we will use it later in the paper in Section 5 to compare lineage with other forms of provenance.

Cui et al. also considered queries without constants, and so the lineage of an output tuple is always a nonempty set. Also, lineage was defined only for tuples actually present in the output. In what follows, we generalize the definitions of [27] to distinguish between two scenarios, where,

(1) A tuple has "empty" lineage provided it is present in the output but it was constructed by the query, e.g. using a constant expression.
(2) A tuple has "no" lineage provided it is not present in the output.

The function $\mathsf{Lin}(Q,I,t)$ is partial because we want to distinguish between these two cases, rather than using the empty set to ambiguously handle both cases. We will use the symbol $\emptyset$ for "empty lineage" and $\bot$ for "no lineage"; alternatively we say that the lineage is *undefined* if $\mathsf{Lin}(Q,I,t) = \bot$.

Once we take into account the possibility of no lineage, some care is needed in how we combine lineages of intermediate tuples in join, union, and projection operations. First, consider a join query such as $Q_1 \bowtie Q_2$. The lineage of a tuple t in $Q_1 \bowtie Q_2$ is the union of the lineages of $t[U_1]$ and $t[U_2]$ provided $t[U_1] \in Q_1$ and $t[U_2] \in Q_2$; however, if either $t[U_1]$ or $t[U_2]$ is undefined, then t cannot be in the result so its lineage should also be undefined. Thus, we will handle joins using a *strict union* operation, defined as follows:

$$\begin{aligned} \bot \cup_{\mathrm{S}} X = X \cup_{\mathrm{S}} \bot &= \bot \\ X \cup_{\mathrm{S}} Y &= X \cup Y \quad (X \neq \bot \neq Y) \end{aligned}$$

For the union operation, consider a query $Q_1 \cup Q_2$. If t is in both Q_1 and Q_2, then the lineage of t is the union of its lineages in the subqueries. However, if t is in only one subquery, then its lineage should be the lineage of the subquery in which it is defined. The lineage of t in $Q_1 \cup Q_2$ is only undefined if it is undefined in both subqueries. To handle this behavior, we use a *lazy union* operation:

$$\begin{aligned} \bot \cup_{\mathrm{L}} X = X \cup_{\mathrm{L}} \bot &= X \\ X \cup_{\mathrm{L}} Y &= X \cup Y \quad (X \neq \bot \neq Y) \end{aligned}$$

Finally, for projection, if t is not in the query result, then there can be no tuples in the subquery that project down to the given tuple. In this case we want the result to be $\bot$. On the other hand, if multiple tuples t project to $t[U]$, then we want to combine their lineage annotations lazily. Thus, we define a lazy flattening operation as follows:

$$\begin{aligned}
\textstyle\bigcup_{\mathrm{L}} \emptyset &= \bot \\
\textstyle\bigcup_{\mathrm{L}} \{X\} &= X \\
\textstyle\bigcup_{\mathrm{L}} (X \cup Y) &= \textstyle\bigcup_{\mathrm{L}} X \cup_{\mathrm{L}} \textstyle\bigcup_{\mathrm{L}} Y
\end{aligned}$$

Having made these auxiliary definitions we are now ready to define lineage for arbitrary queries as follows.

$$\begin{aligned}
\mathsf{Lin}(\{u\}, I, t) &= \begin{cases} \emptyset, & \text{if } t = u \\ \bot, & \text{otherwise} \end{cases} \\
\mathsf{Lin}(R, I, t) &= \begin{cases} \{R(t)\}, & \text{if } t \in I(R) \\ \bot, & \text{otherwise} \end{cases} \\
\mathsf{Lin}(\sigma_\theta(Q), I, t) &= \begin{cases} \mathsf{Lin}(Q, I, t), & \text{if } \theta(t) \\ \bot, & \text{otherwise} \end{cases} \\
\mathsf{Lin}(\pi_U(Q), I, t) &= \textstyle\bigcup_{\mathrm{L}} \{\mathsf{Lin}(Q, I, u) \mid u \in Q(I), u[U] = t\} \\
\mathsf{Lin}(\rho_{A \mapsto B}(Q), I, t) &= \mathsf{Lin}(Q, I, t[B \mapsto A]) \\
\mathsf{Lin}(Q_1 \bowtie Q_2, I, t) &= \mathsf{Lin}(Q_1, I, t[U_1]) \cup_{\mathrm{S}} \mathsf{Lin}(Q_1, I, t[U_2]) \\
\mathsf{Lin}(Q_1 \cup Q_2, I, t) &= \mathsf{Lin}(Q_1, I, t) \cup_{\mathrm{L}} \mathsf{Lin}(Q_2, I, t)
\end{aligned}$$

Again, we assume $Q_1{:}U_1$ and $Q_2{:}U_2$ in the case for join, $Q{:}V \supseteq U$ for projection and $Q_1, Q_2{:}U$ for union.

Our definition of lineage for queries consisting of Selection–Projection–Join–Union (SPJU) operations without constants or renaming is, essentially, equivalent to that of Cui et al.'s. We define lineage as a subinstance of the input, whereas they defined lineage as a vector of subsets of the input tables; this is a minor difference in presentation.

Proposition 2.2 Let Q be a constant-free SPJU query, I be an instance and t be a tuple with $t \in Q(I)$. Then

(1) If $Q = \sigma_\theta(R)$ then $\mathsf{Lin}(\sigma_\theta(R), I, t) = \{R(t)\}$.
(2) If $Q = \pi_U(R)$ then $\mathsf{Lin}(\pi_U(R), I, t) = \{R(u) \in I \mid u[U] = t\}$.

(3) If $Q = R \bowtie S$ then $\mathsf{Lin}(R \bowtie S, I, t) = \{R(t[U]), S(t[V])\}$, where $R{:}U, S{:}V$.
(4) If $Q = R \cup S$ then $\mathsf{Lin}(R \cup S, I, t) = \{R(t) \mid R(t) \in I\} \cup \{S(t) \mid S(t) \in I\}$.

Proof. For each part, the proof is straightforward by unwinding definitions. □

Moreover, it seems clear that our compositional definition of lineage is equivalent (modulo the representations of lineages) to the transitive definition of lineage for composite views given by Cui et al. We omit a detailed proof. Hence, our definition of lineage (for constant-free queries) is equivalent to their characterization [27, Theorem 4.4].

We next establish a basic property relating our definition of lineage to ordinary evaluation. The following proposition says that the lineage of a tuple t with respect to an SPJRU query Q and instance I is defined if and only if t is in the result of $Q(I)$; moreover, if it exists, the lineage is a "witness" to $t \in Q(I)$.

Proposition 2.3 Let Q be an SPJRU query, I be a database instance, and t be a tuple.

(1) If $\mathsf{Lin}(Q, I, t) = \bot$ then $t \notin Q(I)$.
(2) If $\mathsf{Lin}(Q, I, t) = J \neq \bot$ then $J \subseteq I$ and $t \in Q(J)$.

Proof. Both parts are straightforward by induction on the structure of Q. □

2.1.2 An Application: Tracing Lineage in Data Warehouses

Cui et al. propose and implement a lazy approach for computing lineage over SPJU queries with set difference and aggregates in the context of the WHIPS data warehouse system [25, 60]. Every time the lineage of

a tuple t in the result of a query Q is sought, their approach generates one or multiple "reverse" queries that when applied to the database I return the lineage of t in I according to Q.

For an SPJ query Q, a single reverse query Q_{r} is sufficient for obtaining the lineage of any tuple t in $Q(I)$. The idea behind the algorithm for constructing Q_{r} is as follows. First, Q is rewritten into an equivalent *canonical* expression of the PSJ form $\pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_n))$. Then, the reverse query Q_{r} is obtained as $\sigma_{C \wedge (\bigwedge_{A_i \in A} t \bullet A_i = R_j \cdot A_i)}(R_1 \bowtie \cdots \bowtie R_n)$, where the condition $\bigwedge_{A_i \in A} t \bullet A_i = R_j \cdot A_i$ specifies that the value of attribute A_i in some relation R_j among $R_1, \ldots, R_n$ must equal the value of $t \bullet A_i$, for every attribute $A_i \in A$ projected by Q. Finally, the lineage of t is obtained via an additional postprocessing step that splits the result of $Q_{\mathrm{r}}(I)$ into a set of relation instances I'_i of R_i by projecting on the attributes of R_i, for $1 \leq i \leq n$.

To exemplify, consider again the SPJ query Q_1 above and suppose the lineage of the tuple (BayTours, 415-1200) in the result of Q_1 is sought. The relational algebra expression $\pi_{\text{name,phone}}$ $(\sigma_{\text{type='boat'}}(\text{Agencies} \bowtie \text{ExternalTours}))$ given earlier is in fact the SPJ canonical form of Q_1. Hence, the reverse query Q_{r} is obtained as $\sigma_{\text{type='boat'} \wedge \text{name='BayTours'} \wedge \text{phone='415-1200'}}(\text{Agencies} \bowtie \text{ExternalTours})$. Applying Q_{r} to the database instance in Figure 1.1 results in tuples (BayTours, San Francisco, 415-2000, Santa Cruz, boat, \$250) and (BayTours, San Francisco, 415-2000, Monterey, boat, \$400). The lineage of (BayTours, 415-1200) is finally obtained by projecting the output of Q_{r} on the attributes of Agencies and respectively, ExternalTours relations, resulting in instances Agencies(t_1) and ExternalTours(t_5, t_6), respectively.

The correctness of the algorithm for computing lineage outlined above is justified by the fact that lineage is invariant for equivalent formulations of an SPJ query without repeated relation names (Theorem 4.8 in [27]). Hence, it is safe to transform the original query into its canonical form and use it to construct the reverse query that retrieves the lineage, since lineage is not affected by this transformation. However, as the next proposition shows, the invariance no longer holds for certain queries that involve self-joins.

Proposition 2.4 There exist two equivalent PJR queries Q and Q', an instance I, and a tuple $t \in Q(I)$, where $\mathsf{Lin}(Q,I,t) \neq \mathsf{Lin}(Q',I,t)$.

Proof. Let R be a binary relation $R(A,B)$ and $Q = \pi_{AB}(R \bowtie \rho_{B \mapsto C}(R))$ and $Q' = R$. Obviously, $Q \equiv Q'$. Let I be the instance $\{R(1,2), R(1,3)\}$ and hence, $Q(I)$ consists of the tuples $\{R(1,2), R(1,3)\}$. Then, we have $\mathsf{Lin}(Q,I,(1,2)) = \{R(1,2), R(1,3)\}$, whereas $\mathsf{Lin}(Q',I,(1,2)) = \{R(1,2)\}$. □

The proposition holds in general even when there are no self-joins but when repeated relations are allowed. In particular, the above proposition also holds for certain JU queries where a relation name occurs multiple times in the query.

Proposition 2.5 There exist two equivalent JU queries Q and Q', an instance I, and a tuple $t \in Q(I)$, where $\mathsf{Lin}(Q,I,t) \neq \mathsf{Lin}(Q',I,t)$.

Proof. Let R be a binary relation $R(A,B)$ and S be a binary relation $S(A,B)$. Let $Q = R \cup (R \bowtie S)$ and $Q' = R$. Suppose I is $\{R(1,2), R(1,3), S(1,2)\}$, then we have $\mathsf{Lin}(Q,I,(1,2)) = \{R(1,2), S(1,2)\}$, whereas $\mathsf{Lin}(Q',I,(1,2)) = \{R(1,2)\}$. □

Cui et al. generalize the reverse-query algorithm to handle the general class of queries they consider, where aggregates and the set union and difference operators are also allowed. In this general case, however, it may not be possible to trace lineage by using a single query. To illustrate, consider the (rather contrived) query Q_2 below that asks for average boat ticket prices above $250 per destination.

```
Q2: SELECT     AVG(price) AS avg_price
    FROM       ExternalTours
    WHERE      type='boat'
    GROUP BY   destination
    HAVING     avg_price > $250
```

The result of Q_2 consists of a single tuple $t = (\$300)$ (i.e., the average ticket price for boat trips to Monterey). Note that \$300 is a newly computed value and it does not appear in the input database. Hence, we cannot trace t's lineage by only examining the input database. Intuitively, we need to know that t has been obtained from the tuple (Monterey, \$300) in the intermediate result of the aggregation. The algorithm for tracing lineage over SPJU queries with aggregates and set difference is based on this very idea. First, the original query is rewritten into a canonical form consisting of alternating D-segments and AUSPJ-segments. Here D stands for Difference, so a D-segment consists of the application of a difference operator. Similarly, AUSPJ stands for Aggregation–Union–Selection–Projection–Join, and an AUSPJ-segment consists of a sequence of these operators in this order. As before, the rewriting rules ensure lineage preservation for every output tuple. Then, an intermediate view is defined for each segment. Each intermediate view can be computed eagerly and stored at view materialization time, or it can be computed lazily, only when needed for tracing purposes. Finally, lineage is recursively traced over each segment and corresponding intermediate view until the input database is reached.

As an example, the canonical form of Q_2 above is $\pi_{\text{avg_price}}$ $(\sigma_{\text{avg_price}>\$250}(\alpha_{\text{destination},\text{AVG(price)}}(\sigma_{\text{type='boat'}}(\text{ExternalTours}))))$. It consists of two AUSPJ segments corresponding to sequences $\pi_{\text{avg_price}} - \sigma_{\text{avg_price}>\$250}$ and $\alpha_{\text{destination},\text{AVG(price)}} - \sigma_{\text{type='boat'}}$, respectively. Any of the SPJU operators may be missing from a segment, but note that the aggregate operator specifies the cut-off point between the segments. The result of the second segment, consisting of tuples (Santa Cruz, \$250) and (Monterey, \$300), is stored as an intermediate view AllDestinations. Intuitively, this intermediate view provides the missing clues in determining the lineage of output tuples, being eliminated from the final output due to projecting on avg_price. Lineage tracing for the output tuple $t = (\$300)$ proceeds as follows. First, t's lineage is traced across the first segment, back to the intermediate view, by executing the reverse query $\sigma_{\text{avg_price}=\$300}(\text{AllDestinations})$. This leads to the intermediate tuple (Monterey, \$300). We now recursively trace the lineage of this intermediate tuple through the

second segment, back to the input database, by executing the query $\sigma_{\text{type='boat'}\wedge\text{destination='Monterey'}}$(ExternalTours). This leads to source tuples t_6 and t_7, which together constitute the lineage of t.

2.2 Why-Provenance

As discussed in Section 1, while the notion of lineage defined by Cui et al. exposes a (small) collection of input tuples which contribute to the creation of an output tuple t, it is not as precise as one might like as it does not capture the notion of individual witnesses for the output tuple. For instance, the lineage of the output tuple (BayTours, 415-1200) consisting of {Agencies(t_1), ExternalTours(t_5, t_6)} does not specify that t_5 and t_6 need not coexist in order to witness the BayTours output tuple. In fact, proper subinstances of {Agencies(t_1), ExternalTours(t_5, t_6)} are sufficient to prove the output tuple. Indeed, $\{t_1, t_5\}$ and respectively, $\{t_1, t_6\}$ are two different witnesses for the BayTours tuple.

Buneman et al. [13] defined the notion of *why-provenance* in terms of a deterministic semistructured data model and query language. Why-provenance is based on identifying subinstances of the input that "witness" a part of the output. In this article, we will reformulate why-provenance in terms of the relational model and relational algebra (SPJU) query language. Formally, given a database instance I, a query Q and a tuple $t \in Q(I)$, a *witness* for t with respect to Q is a subinstance I' of I such that $t \in Q(I')$.

Definition 2.3 (Witness [13]). Let I be a database instance, Q be a query over I, and t be a tuple in $Q(I)$. An instance $I' \subseteq I$ is a *witness for t with respect to Q* if $t \in Q(I')$.

Note that the notion of witness from Definition 2.3 above is not tied to the structure of the query Q. Intuitively, any subinstance of the database that produces t is a witness for t, and in particular, both the lineage of t, and the entire database I are witnesses for t. The set of witnesses for t with respect to a database instance I and a query Q is thus:

$$\mathsf{Wit}(Q,I,t) = \{J \subseteq I \mid t \in Q(J)\}.$$

Observe that if Q is a monotone query, then $\mathsf{Wit}(Q,I,t)$ is closed under upwards inclusion. It is finite (provided I is finite), but potentially exponentially large due to the possibility of witnesses containing "irrelevant" tuples. Since we only consider monotone queries, it follows immediately that $\mathsf{Wit}(Q,I,t) = \emptyset$ if and only if $t \notin Q(I)$, whereas $\emptyset \in \mathsf{Wit}(Q,I,t)$ holds if and only if $\forall J \subseteq I \cdot t \in Q(J)$.

Buneman et al. defined the *why-provenance* of an output tuple t in the result of a query Q applied to a database instance I as a particular subset of $\mathsf{Wit}(Q,I,t)$ called the *witness basis*. For a Datalog-style query, each witness in the witness basis corresponds intuitively to the leaves of a "proof tree". Hence, we shall sometimes call them *proof-witnesses*[1] in this paper since an instantiation of the operator tree of a relational algebra query can be seen as a proof tree. We write $\mathcal{P}(\mathcal{P}(\mathit{TupleLoc}))$ for the set of sets of tuples; recall that normally *TupleLoc* is a finite set so $\mathcal{P}(\mathcal{P}(\mathit{TupleLoc}))$ is also finite. We define proof-witnesses for relational algebra queries as follows, adapting Buneman et al.'s definition to the relational model and relational algebra.

Definition 2.4 (Why-Provenance, i.e., Witness Basis [13, Definition 6]). Let Q be an SPJU query. Let I be a database instance and t be a tuple in $Q(I)$. Then, the *why-provenance* (or *witness basis*) of t according to Q and I, denoted as $\mathsf{Why}(Q,I,t)$, is a subset of $\mathcal{P}(\mathcal{P}(\mathit{TupleLoc}))$ defined as follows:

$$\begin{aligned}
\mathsf{Why}(\{t\},I,\{u\}) &= \begin{cases} \{\emptyset\}, & \text{if } (t = u), \\ \emptyset, & \text{otherwise.} \end{cases} \\
\mathsf{Why}(R,I,t) &= \begin{cases} \{\{(R,t)\}\}, & \text{if } (t \in R(I)), \\ \emptyset, & \text{otherwise.} \end{cases} \\
\mathsf{Why}(\sigma_\theta(Q),I,t) &= \begin{cases} \mathsf{Why}(Q,I,t), & \text{if } \theta(t), \\ \emptyset, & \text{otherwise.} \end{cases} \\
\mathsf{Why}(\pi_U(Q),I,t) &= \bigcup\{\mathsf{Why}(Q,I,u) \mid u \in Q(I), t = u[U]\} \\
\mathsf{Why}(\rho_{A \mapsto B}(Q),I,t) &= \mathsf{Why}(Q,I,t[B \mapsto A]) \\
\mathsf{Why}(Q_1 \bowtie Q_2,I,t) &= \mathsf{Why}(Q_1,I,t[U_1]) \uplus \mathsf{Why}(Q_2,I,t[U_2]) \\
\mathsf{Why}(Q_1 \cup Q_2,I,t) &= \mathsf{Why}(Q_1,I,t) \cup \mathsf{Why}(Q_2,I,t))
\end{aligned}$$

[1] Following Val Tannen's suggestion.

Here, $\Cup$ takes all the pairwise unions of two collections, playing a role similar to strict union in our definition of lineage. That is, $S \Cup \bot = \bot \Cup S = \bot$, and $S \Cup T = \{s \cup t \mid s \in S, t \in T\}$ otherwise.

The proof-witnesses are derived from the syntax of Q and so the size of each proof-witness is bounded by the size of Q. In particular, the proof-witnesses exclude tuples that plainly have nothing to do with t being produced by Q. For this reason, the witness basis itself tends to be small as well, compared to the set of all witnesses. Nevertheless, the witness basis "represents" the set of all witnesses in a suitable sense, as we shall show (Theorem 2.7).

Going back to our example, the why-provenance (i.e., witness basis) of (BayTours, 415-1200) in the result of Q_1 is the set $\{\{t_1, t_5\}, \{t_1, t_6\}\}$. There are two witnesses in the basis, corresponding to $\{t_1, t_5\}$ and $\{t_1, t_6\}$, respectively. This indicates that the output tuple is witnessed by input tuples in two different ways according to Q_1: once using t_1 and t_5, and once with t_1 and t_6. Note that the definition requires witnesses in the basis to consist of exactly one tuple from Agencies, and one tuple from ExternalTours according to the `FROM` clause of Q_1. Hence, the set $\{t_1, t_5, t_6\}$, which is essentially the lineage of the BayTours tuple, is not in the witness basis.

Similar to lineage, it is easy to establish the following property about why-provenance. The following proposition states that every tuple t in the result of an SPJRU query Q on an instance I has a non-empty witness basis. Furthermore, every proof witness for t according to Q and I is a witness for t according to Q.

Proposition 2.6 Suppose Q is an SPJRU query, I is a database instance, and t is a tuple.

(1) If $\mathsf{Why}(Q, I, t) = \emptyset$ then $t \notin Q(I)$.
(2) If $J \in \mathsf{Why}(Q, I, t)$ then $J \subseteq I$ and $t \in Q(J)$.

Proof. Both parts can be proved easily by induction on the structure of Q. □

In fact, we can also show that every witness contains a proof-witness:

Theorem 2.7 Let Q be an SPJRU query, I an instance and t a tuple. If $J \in \mathsf{Wit}(Q,I,t)$, then there exists J' such that $J \supseteq J' \in \mathsf{Why}(Q,I,t)$.

Proof. By induction on Q.

Case 1. Suppose $Q = \{u\}$. If $J \in \mathsf{Wit}(\{u\},I,t)$ then $t = u$ and $J = \emptyset$, and $J = J' = \emptyset \in \mathsf{Why}(\{u\},I,t)$ works.

Case 2. Suppose $Q = R$. If $J \in \mathsf{Wit}(R,I,t)$ then J must contain $\{(R,t)\}$, the unique proof witness for t.

Case 3. Suppose $Q = \sigma_\theta(Q')$. Let $J \in \mathsf{Wit}(\sigma_\theta(Q'),I,t)$ be given. Then by definition, we have $J \subseteq I$ and $t \in (\sigma_\theta(Q'))(J)$. Hence $\theta(t)$ holds and $t \in Q'(J)$, so $J \in \mathsf{Wit}(Q',I,t)$ also. By induction we can obtain J' such that $J \supseteq J' \in \mathsf{Why}(Q',I,t)$. By definition, since $\theta(t)$ holds we have $\mathsf{Why}(Q',I,t) = \mathsf{Why}(\sigma_\theta(Q'),I,t)$. Hence, $J' \in \mathsf{Why}(\sigma_\theta(Q'),I,t)$, as desired.

Case 4. Suppose $Q = \pi_U(Q')$. Let $J \in \mathsf{Wit}(\pi_U(Q'),I,t)$ be given. By definition, we have $J \subseteq I$ and $t \in (\pi_U(Q'))(J)$. So for some $u \in Q'(J)$, we have $t = u[U]$. Hence $J \in \mathsf{Wit}(Q',I,u)$, so by induction, we can obtain J' with $J \supseteq J' \in \mathsf{Why}(Q',I,u)$. Note that by monotonicity we have $u \in Q'(J) \subseteq Q'(I)$. Since $t = u[U]$ and $u \in Q'(I)$, by definition we have that

$$\begin{aligned}\mathsf{Why}(Q',I,u) &\subseteq \bigcup\{\mathsf{Why}(Q',I,u) \mid u \in Q(I), t = u[U]\}\\ &= \mathsf{Why}(\pi_U(Q'),I,t)\ .\end{aligned}$$

This implies that $J' \in \mathsf{Why}(\pi_U(Q'),I,t)$, as desired.

Case 5. Suppose $Q = Q_1 \bowtie Q_2$, and $Q_1{:}U_1$ and $Q_2 : U_2$. Let $J \in \mathsf{Wit}(Q_1 \bowtie Q_2,I,t)$ be given. By definition, $J \subseteq I$ and $t \in (Q_1 \bowtie Q_2)(J)$, so we must have $t[U_1] \in Q_1(J)$ and $t[U_2] \in Q_2(J)$. Let $t_1 = t[U_1]$ and $t_2 = t[U_2]$. Thus $J \in \mathsf{Wit}(Q_1,I,t_1)$ and $J \in \mathsf{Wit}(Q_2,I,t_2)$, so by induction we have J_1', J_2' with $J \supseteq J_i' \in \mathsf{Why}(Q_i,I,t_i)$ for $i \in \{1,2\}$. Observe that by monotonicity $t_i \in Q_i(J) \subseteq Q_i(I)$ for $i \in \{1,2\}$. By definition, we have $J' = J_1' \cup J_2' \in \mathsf{Why}(Q_1 \bowtie Q_2,I,t)$. Clearly $J' \subseteq J$ so we are done.

Case 6. Suppose $Q = Q_1 \cup Q_2$. Let $J \in \mathsf{Wit}(Q_1 \cup Q_2, I, t)$ be given. By definition, $J \subseteq I$ and $t \in (Q_1 \cup Q_2)(J)$, so we must have $t \in Q_1(J)$ or $t \in Q_2(J)$. Without loss of generality consider the former case; the latter is symmetric. Then $J \in \mathsf{Wit}(Q_1, I, t)$ so there exists J' such that $J \supseteq J' \in \mathsf{Why}(Q_1, I, t) \subseteq \mathsf{Why}(Q_1 \cup Q_2, I, t)$. □

This is a key property, and has not, to our knowledge, been established previously. Together with monotonicity, it implies that the witness basis "represents" the (often much larger) witness set:

Corollary 2.8. Let Q be a query, I an instance and t a tuple.

$$\mathsf{Wit}(Q, I, t) = \{J \subseteq I \mid \exists J' \in \mathsf{Why}(Q, I, t).\ J' \subseteq J\}$$

Proof. The $\subseteq$ direction is Theorem 2.7. For the $\supseteq$ direction, if J contains an element of the witness basis, then J contains a witness, and by monotonicity J is itself a witness. □

Since the witness basis is tied to the structure of the query, it is unsurprising that why-provenance is sensitive to how a query is written. An example illustrating this issue was given in Section 1.1. Recall the equivalent queries Q and Q' shown in Figure 1.3 and consider the tuple (1, 2) in the result of Q (or Q') applied to the database instance I (also shown in the figure). According to Q, the why-provenance of the output tuple (1, 2) consists of the witness {(1, 2)}. According to Q', however, the why-provenance for the output tuple consists of the witness {(1, 2), (1,3)}. Buneman et al. also considered a *minimal witness basis* variant of why-provenance, which, unlike the witness basis, is invariant under query equivalence. A minimal witness basis consists of *minimal witnesses*, which are defined next.

We say that a set $s \in S$ is a *minimal element* of a collection S if for any $s' \in S$ with $s' \subseteq s$, we have $s' = s$. Equivalently, an element of S is minimal if there exists no element $s \in S$ such that $s' \subsetneq s$.

Definition 2.5. Let Q be a query, I be a database instance, and t be a tuple in $Q(I)$. A minimal witness is a minimal element of $\mathsf{Wit}(Q, I, t)$.

We write $\mathsf{MWit}(Q,I,t)$ for the set of all minimal witnesses, that is,

$$\mathsf{MWit}(Q,I,t) = \{J \in \mathsf{Wit}(Q,I,t) \,|\, J \text{ minimal in } \mathsf{Wit}(Q,I,t)\}$$

Definition 2.6. Let Q be a query, I be a database instance, and t be a tuple in $Q(I)$. The *minimal witness basis* for t with respect to the (SPJRU) query Q and I, denoted as $\mathsf{MWhy}(Q,I,t)$, is the set of all minimal elements of $\mathsf{Why}(Q,I,t)$, that is,

$$\mathsf{MWhy}(Q,I,t) = \{J \in \mathsf{Why}(Q,I,t) \mid J \text{ minimal in } \mathsf{Why}(Q,I,t)\}$$

Going back to the example shown in Figure 1.3, the witness {(1, 2)} for the output tuple (1, 2) in the result of Q' is a minimal witness, whereas {(1, 2), (1, 3)} is not. In fact, the minimal witness basis for (1, 2) according to Q' (and also Q) is the singleton set {{(1, 2)}}. In contrast to the witness basis, which is tied to the structure of the query, the set of minimal witnesses is based on the semantics of the query. On the other hand, the minimal witness basis is defined in terms of the witness basis, which is not invariant up to query equivalence. Nevertheless, the minimal witness basis is invariant under equivalent rewritings of a query [13, Theorem 3]. In fact, the minimal witness basis coincides with the set of minimal witnesses, as we shall now show.

First, observe that every minimal witness is in fact a proof-witness (and is therefore in the why-provenance):

Corollary 2.9 (cf. [55, Proposition 4.2.3]). Let Q be a query, I an instance and t a tuple. Then $\mathsf{MWit}(Q,I,t) \subseteq \mathsf{Why}(Q,I,t)$.

Proof. Let $J \in \mathsf{MWit}(Q,I,t)$ be given. Then by Theorem 2.7 we can choose J' such that $J \supseteq J' \in \mathsf{Why}(Q,I,t)$. Since $\mathsf{Why}(Q,I,t) \subseteq \mathsf{Wit}(Q,I,t)$, we have $J' \in \mathsf{Wit}(Q,I,t)$. By minimality, $J = J'$. □

Having established that all minimal witnesses are included in the why-provenance, it is natural to ask if the minimal witnesses coincide with the minimal elements of the why-provenance. This is the case; indeed,

we can calculate the minimal witnesses for t with respect to Q and I just by considering minimal witnesses in the witness basis of t with respect to Q and I.

Theorem 2.10. Let Q be an SPJRU query, I an instance, and t a tuple. Then $\mathsf{MWit}(Q,I,t) = \mathsf{MWhy}(Q,I,t)$.

Proof. To show $\mathsf{MWit}(Q,I,t) \subseteq \mathsf{MWhy}(Q,I,t)$, suppose $J \in \mathsf{MWit}(Q,I,t)$. Then by the fact that $\mathsf{MWit}(Q,I,t) \subseteq \mathsf{Why}(Q,I,t)$, which was just shown above, we know $J \in \mathsf{Why}(Q,I,t)$. We need to show that J is minimal in $\mathsf{Why}(Q,I,t)$. Let $J' \in \mathsf{Why}(Q,I,t)$ such that $J' \subseteq J$ be given. Then $J' \in \mathsf{Why}(Q,I,t) \subseteq \mathsf{Wit}(Q,I,t)$, so $J' = J$ by the minimality of J in $\mathsf{Wit}(Q,I,t)$. Hence, $J \in \mathsf{MWhy}(Q,I,t)$.

For the $\supseteq$ direction, suppose $J \in \mathsf{MWhy}(Q,I,t)$. Then $J \in \mathsf{Why}(Q,I,t) \subseteq \mathsf{Wit}(Q,I,t)$. We need to show that J is minimal in $\mathsf{Wit}(Q,I,t)$. Suppose $J' \in \mathsf{Wit}(Q,I,t)$ with $J' \subseteq J$. Then by Theorem 2.7, we know that there exists $J'' \in \mathsf{Why}(Q,I,t)$ with $J'' \subseteq J'$. But since J is minimal in $\mathsf{Why}(Q,I,t)$ and $J'' \subseteq J$, we have $J'' = J' = J$. Hence $J \in \mathsf{MWit}(Q,I,t)$. □

Note that the definition of $\mathsf{MWit}(Q,I,t)$ is independent of the structure of the query. Since $\mathsf{MWhy}(Q,I,t) = \mathsf{MWit}(Q,I,t)$, and MWit is clearly invariant under query equivalence by definition, MWhy is also invariant under query equivalence.

Corollary 2.11. Let Q, Q' be SPJRU queries, where $Q \equiv Q'$. Let I be an instance and t a tuple. Then $\mathsf{MWhy}(Q,I,t) = \mathsf{MWhy}(Q',I,t)$.

A natural question that arises at this point regards the relationships between lineage, why-provenance (i.e., the witness basis) and the minimal witness basis. We shall not pursue this further here, but will address this question in some detail in Section 5. In particular, we shall show in Section 5.1 that for the case of SPJRU queries, both lineage and the minimal witness basis can be computed from why-provenance; in fact all of these can be viewed as instances of the semiring model. However, neither lineage, nor why-provenance can be obtained from the minimal witness basis.

2.2.1 Minimal Witnesses and the View Deletion Problem

In subsequent work, Buneman et al. [14] make explicit the connection between minimal witnesses and the view deletion problem. Let I be a database instance and $V = Q(I)$ be a view defined over I. The *view deletion problem* is to find a set of tuples ΔI to remove from I so as to delete a given tuple t in the view V. This problem makes sense for monotone queries, where source tuples must be removed in order to delete a tuple from the output. In translating the deletion of t from the view into deletions over the input database I, an obvious starting point is to examine the input tuples that witness the existence of t in I according to Q. Intuitively, all minimal witnesses of t must be "destroyed" to delete t from the view, where a witness is destroyed if one of the tuples in the witness is deleted from I. Thus, minimal witnesses and the view deletion problem are closely connected.

Only in very restricted settings does a unique set ΔI that causes the deletion of the desired tuple in the view exist. To see this, consider a view that is the result of a join between two relations R_1 and R_2. Let t be a tuple in the view and assume for simplicity that t has a single minimal witness consisting of tuples t_1 of R_1 and t_2 or R_2. Clearly, there are at least two choices of sets of tuples to delete from the source that result in the deletion of t from the view (i.e., $\{t_1\}$, $\{t_2\}$, or any subset of source tuples that contains t_1 or t_2). In [14], Buneman et al. study the alternative problem of finding a *minimal* update to I that will cause the desired deletion of t from the view. They consider two variants of the problem, which correspond to minimizing two different objective functions: (1) the number of side-effects that the deletions in I cause in the view (in addition to the deletion of t); and (2) the number of tuples deleted from I, regardless of the number of side-effects in V.

The first variant leads to the *view side-effect problem* stated as follows: *given a source database I, a query Q, the view $V = Q(I)$ and a tuple $t \in V$, find a subset $\Delta I \subseteq I$ whose removal will delete t from V while minimizing the number of other tuples deleted from the view.* In other words, we wish to minimize $|\Delta V|$, where $\Delta V = (V \backslash Q(I \backslash \Delta I)) \backslash \{t\}$ is called the set of side-effects on V. If $\Delta V = \emptyset$ we say that the deletion ΔI on I is *side-effect free*. Note that the query,

source database, as well as the tuple are part of the input to the view side-effect problem. Buneman et al. showed a dichotomy in the complexity of the problem of deciding whether there exists a side-effect-free deletion for the class of SPJRU queries. The problem is NP-hard for queries involving join and either projection or union [14, Theorem 2.1]. For this class of queries, a tuple in the output of the query may have many witnesses (due to projection or union), and there may be many possible ways of destroying each witness (due to join). The difficulty here consists in reasoning about how a set of source deletions affects the existence of other view tuples in order to minimize the side-effects on the view. On the other hand, the problem of finding a side-effect free deletion can always be solved in polynomial time for the subclass of queries that do not simultaneously involve join, and either projection or union (i.e., SPU and SJ queries) [14, Theorems 2.3 and 2.4].

The second variant leads to the *source side-effect problem* stated as follows: *given a source database I, a query Q, the view $V = Q(I)$ and a tuple $t \in V$, find the smallest subset $\Delta I \subseteq I$ whose removal will delete t from V.* Buneman et al. show that the problem is NP-hard for queries involving join and either projection or union [14, Theorems 2.5 and 2.6], and it is polynomial-time solvable for the remaining queries in the class [14, Theorems 2.8 and 2.9]. Again, we treat the query Q, instance I and tuple t as part of the input in these results.

It is worth pointing out that view deletions, and more generally, the view update problem has been studied extensively in the past. In earlier research on the general view update problem [4, 24, 30, 46], an update (i.e., deletion, insertion, or modification) to the view can be translated as a combination of different types of updates to the source. Dayal and Bernstein [30] also identify *clean sources*, which, in the context of the view deletion problem for SPJU views, corresponds to side-effect free deletions. Cui and Widom [26] give an algorithm that finds an exact side-effect free deletion whenever there exists one, using lineage.

Cong et al. [23] show that both the view side-effect and the source side-effect problems become tractable in the case of *key-preserving* SPJ views [23, Theorems 3.1 and 3.4]. Intuitively, an SPJ query Q is *key-preserving* if it retains a key for every input relation involved in Q.

3

How-Provenance

The why-provenance of an output tuple provides a set of witnesses for that output tuple. However, it does not provide additional information on *how* the output tuple is actually derived. In this section, we begin by discussing the relational algebra provenance semirings framework of Green et al. [43], where the notion of *how-provenance* was first articulated. We also discuss the lineage component of the Trio system [2, 6, 59] for managing relational data with uncertainty and lineage. Trio tracks data provenance and uses it to compute confidence levels for uncertain values in the output of a query, among other things. Although described as capturing "*where* data comes from" in [5, 52], Trio's notion of lineage can be viewed as a form of how-provenance, as we shall explain in this section.

After this, we describe an extension of provenance semirings for recursive datalog programs also proposed by Green et al. [43]. We then leave the realm of relational algebra and datalog, and turn our attention to schema mappings. In this context, we discuss an extension of provenance semirings implemented in the ORCHESTRA collaborative data sharing system [42, 44]. We also discuss *routes*, a notion of provenance over schema mappings proposed by Chiticariu and Tan [21], prior to the work of Green et al. on ORCHESTRA [42, 43, 44], in the context of the

SPIDER system [3] for debugging schema mappings. Routes are also a form of how-provenance over schema mappings, hence their treatment in this section.

3.1 Provenance Semirings

The notion of *how-provenance* was introduced by Green et al. [43]. In this work, the classical *semiring* algebraic structure is used in devising a general framework for uniformly treating various extensions to relational algebra such as handling bag semantics or incomplete and probabilistic databases. Moreover, Green et al. show that their semiring framework is appropriate for capturing a notion of data provenance that is more general than why-provenance and which they call *how-provenance*. (We will discuss the precise meaning of "more general" later in Section 5.)

The idea behind the semiring framework of Green et al. [43] is to distinguish two basic transformations that source tuples undergo as a result of applying a relational query to a source database: they can be either joined together as an effect of a join, or merged together, via union or projection. In Section 1, we have already seen an example of how this observation can be exploited to provide an explanation of *how* an output tuple is derived in the result of a query. For convenience, the example is reproduced in Figures 3.1(a, b). Consider the query Q and the tags shown to the right of the result of Q in Figure 3.1(b). Intuitively, these abstract tags describe how each output tuple is produced in the result of Q, in terms of the abstract tags (or identifiers) of the source tuples. For example, the tag $t_1 \cdot (t_1 + t_3)$ describes that the first output tuple (San Francisco, 415-1200) is produced by joining $(\cdot)$ t_1 with the result of unioning $(+)$ t_1 and t_3. Similarly, the tag t_2^2 describes that the second output tuple (Santa Cruz, 831-3000) was created by joining t_2 with itself. Green et al. [43] observe that these abstract tags describing how source tuples are combined (i.e., unioned and/or joined) to produce an output tuple are in fact polynomials in a commutative semiring $(K, 0, 1, +, \cdot)$.

Let us now formally introduce the framework of [43] for mapping relational algebra operations into operations in the semiring. Let K be

Agencies

	name	based_in	phone
t_1:	BayTours	San Francisco	415-1200
t_2:	HarborCruz	Santa Cruz	831-3000

ExternalTours

	name	destination	type
t_3:	BayTours	San Francisco	cable car
t_4:	BayTours	Santa Cruz	bus
t_5:	BayTours	Santa Cruz	boat
t_6:	BayTours	Monterey	boat

(a)

Q:

```
SELECT   e.destination, a.phone
FROM     Agencies a,
         (SELECT name,
            based_in AS destination
         FROM Agencies a
         UNION
         SELECT name, destination
         FROM ExternalTours ) e
WHERE    a.name = e.name
```

Result of Q:

destination	phone	
San Francisco	415-1200	$t_1 \cdot (t_1 + t_3)$
Santa Cruz	831-3000	t_2^2
Santa Cruz	415-1200	$t_1 \cdot (t_4 + t_5)$
Monterey	415-1200	$t_1 \cdot t_6$

(b)

Fig. 3.1 (a) Example source database with two $\mathbb{N}[t_1,\ldots,t_6]$ relations; (b) the $\mathbb{N}[t_1,\ldots,t_6]$ relation resulting from evaluating the query Q in the provenance semiring.

a set containing a distinguished element 0. A *K-relation* models a relation as a function R on all possible tuples, where R maps tuples in the relation to nonzero elements of K, and tuples that are not in the relation to the special element 0. Formally, a *K-relation* over a finite set of attributes U is a function R: *U-Tuples* $\to K$, such that its support defined as $\text{supp}(R) = \{t \,|\, R(t) \neq 0\}$ is finite. Here, *U-Tuples* denotes the set of all tuples with attributes U. A K-relation R:*U-Tuple* $\to K$ corresponds to a finite relation whose elements are tagged with elements of K. In particular, $\mathbb{B}$-relations correspond to ordinary tables, and $\mathbb{N}$-relations to multisets or bags. Indeed, the K-relational model we are about to discuss subsumes both classical set-based semantics and bag-based semantics of the relational algebra, as well as several other interesting semantics such as probabilistic and incomplete information.

We sometimes write K-relations concretely as ordinary relations in which each tuple carries an annotation. For example

$\{(A\!:\!1, B\!:\!2)^x, (A\!:\!17, B\!:\!42)^{y+xy}\}$ denotes the K-relation r satisfying $r(A\!:\!1, B\!:\!2) = x$, $r(A\!:\!17, B\!:\!42) = y + xy$, and $r(t) = 0$ for all other tuples. In this representation, the semiring annotations of any duplicate tuples are summed, for example, $\{(A\!:\!1, B\!:\!2)^p, (A\!:\!1, B\!:\!2)^q\}$ is equivalent to $\{(A\!:\!1, B\!:\!2)^{p+q}\}$.

The basic relational algebra operators are mapped into operations on an algebraic structure $(K, 0, 1, +, \cdot)$ as follows.

Definition 3.1 ([43, Definition 3.2]). Let $(K, 0, 1, +, \cdot)$ be an algebraic structure with two binary operations $+$ and $\cdot$ and two distinguished elements 0 and 1. The operations of the positive K-relational algebra are defined as follows.

Empty relation. For any set of attributes U, there exists $\emptyset$: *U-Tuples* $\to K$ such that $\emptyset(t) = 0$.

Selection. Let R: *U-Tuples* $\to K$ and θ be a selection predicate that maps each *U-Tuple* to either 0 or 1. Then $\sigma_\theta(R)$: *U-Tuples* $\to K$ is defined by $(\sigma_\theta(R))(t) = R(t) \cdot \theta(t)$. That is, $(\sigma_\theta(R))(t)$ is $R(t)$ if θ holds on t and 0 otherwise.

Projection. Let R: *U-Tuples* $\to K$ and $V \subseteq U$. Then $\pi_V(R)$: *V-Tuples* $\to K$ is defined by $(\pi_V(R))(t) = \sum_{t = t'[V] \wedge R(t') \neq 0} R(t')$.

Union. Let R_1, R_2: *U-Tuples* $\to K$. Then $R_1 \cup R_2$: *U-Tuples* $\to K$ is defined by $(R_1 \cup R_2)(t) = R_1(t) + R_2(t)$.

Natural join. Let R_1: U_1*-Tuples* $\to K$ and R_2: U_2*-Tuples* $\to K$. Then $R_1 \bowtie R_2$: $U_1 \cup U_2$*-Tuples* $\to K$ is defined by $(R_1 \bowtie R_2)(t) = R_1(t_1) \cdot R_2(t_2)$, where $t_1 = t[U_1]$ and $t_2 = t[U_2]$.

If we assume that K-relational semantics satisfies the same equivalence laws as positive relational algebra operators over bags (i.e., union $(+)$ is associative, commutative and has identity $\emptyset$, join $(\cdot)$ is associative, commutative and distributive over union, and projection and selection commute with each other, as well as with union and join), Green et al. conclude that $(K, 0, 1, +, \cdot)$ must be a commutative semiring. Recall that an algebraic structure $(K, 0, 1, +, \cdot)$ is a commutative semiring if $(K, 0, +)$ and $(K, 1, \cdot)$ are commutative monoids, $\cdot$ distributes over $+$ and $0 \cdot a = a \cdot 0 = 0$, $\forall a \in K$. Here, a commutative monoid is an

algebraic structure (K, id, op), where *op* is associative and commutative and has identity element *id*.

The semiring operations essentially document *how* each output tuple is produced from source tuples. Intuitively, if each source tuple in a database D is tagged with a distinct tuple id, the semiring gives us the how-provenance for each output tuple in the form of a polynomial with coefficients from the set $\mathbb{N}$ of natural numbers and indeterminates (or variables) from the set of source tuple ids.

For clarity, we restate the above definition (which was taken literally from [43]) as a set of equations. In these equations, we write $Q^K(I)$ for the result of evaluating a query Q on an instance I with respect to a particular K. This notation is useful since we sometimes need to discuss more than one K simultaneously elsewhere in the paper.

$$
\begin{aligned}
(\{u\})^K(I)t &= \begin{cases} 1, & t = u, \\ 0, & \text{otherwise.} \end{cases} \\
R^K(I)t &= I(R)(t) \\
(\sigma_\theta(Q))^K(I)t &= \theta(t) \cdot Q^K(I)t \\
(\rho_{A \mapsto B}(Q))^K(I)t &= Q^K(I)(t[B \mapsto A]) \\
(\pi_V(Q))^K(I)t &= \sum_{u \in \mathrm{supp}(Q^K(I)), u[V]=t} Q^K(I)u \\
(Q_1 \bowtie Q_2)^K(I)t &= {Q_1}^K(I)(t[U_1]) \cdot {Q_2}^K(I)(t[U_2]) \\
(Q_1 \cup Q_2)^K(I)t &= {Q_1}^K(I)t + {Q_2}^K(I)t
\end{aligned}
$$

Observe that in the expression $(\{u\})^K(I)t$, the expression $\{u\}$ is a *query expression* describing a constant, singleton relation, not a relation value per se. We interpret such constants as K-relations that assign 1 to u and 0 to all other tuples; this is equivalent to the singleton set $\{u\}$ in relational algebra or the singleton multiset $\{u\}$ in bag relational algebra. The summation in the case for projection is finite since the support of a K-relation is assumed to be finite. Also, in the rule for selection, we view a test θ as a function θ: *U-Tuples* $\to \{0_K, 1_K\}$.

Definition 3.2 (Positive algebra provenance semiring [43, Definition 4.1]). Let X be the set of all tuple ids of a database

instance D. The *positive algebra provenance semiring* for D is defined as the semiring of polynomials $(\mathbb{N}[X], 0, 1, +, \cdot)$, where $\mathbb{N}[X]$ denotes the set of polynomials with coefficients from $\mathbb{N}$ and variables from X, and $+$ and $\cdot$ have the usual definitions from algebra.

Concretely, if *TupleLoc* is the set of all tagged tuples from I, then we define K_{How} as $\mathbb{N}[\mathit{TupleLoc}]$, and we define the K_{How}-instance I_{How} as follows:

$$I_{\mathsf{How}}(R)(t) \quad = \quad \begin{cases} (R,t) & t \in I(R) \\ 0 & t \notin I(R) \end{cases}$$

Finally, we define the how-provenance of a tuple t with respect to $Q(I)$ as $\mathsf{How}(Q,I,t) = Q^{K_{\mathsf{How}}}(I_{\mathsf{How}})t$. That is, we take how-provenance to be a polynomial expression over tuples (R,t).

To illustrate, our source database from Figure 3.1(a) can be seen as consisting of two $\mathbb{N}[t_1,\ldots,t_6]$ relations. Applying the query Q to these relations and performing the calculations in the provenance semiring results in the $\mathbb{N}[t_1,\ldots,t_6]$ relation shown in Figure 3.1(b). As an example, the how-provenance of the first two output tuples is given as the polynomials $t_1 \cdot (t_1 + t_3)$ and respectively, t_2^2. (Note that $t_1 \cdot (t_1 + t_3)$ is the same as $t_1^2 + t_1 t_3$, since $\cdot$ distributes over $+$.)

3.2 Trio Lineage

Trio [5, 52] is a system for managing relational data along with uncertainty and lineage. In this paper, we discuss a simplified version of Trio's *ULDB (Uncertainty-Lineage Databases)* data model which is described in [52]. This simplified model is similar to the relational model, except that each tuple in a relation is associated with a numerical *confidence value* in the interval $[0,1]$, indicating the probability that the tuple belongs to the database, as well as with an expression describing its *lineage*. Trio's lineage is different from the lineage notion of Cui et al. [27], which is a form of why-provenance, as explained in Section 2. In [5, 52], Trio's lineage is described as capturing "*where* data comes from". However, as we shall explain in this section, Trio's lineage can also be characterized as a form of *how-provenance*.

Agencies

	name	based_in	phone		
t_1:	BayTours	San Francisco	415-1200	0.7	$\lambda(t_1) = t_1$
t_2:	HarborCruz	Santa Cruz	831-3000	1.0	$\lambda(t_2) = t_2$

ExternalTours

	name	destination	type	price		
t_3:	BayTours	San Francisco	cable car	$50	0.9	$\lambda(t_3) = t_3$
t_4:	BayTours	Santa Cruz	bus	$100	0.6	$\lambda(t_4) = t_4$
t_5:	BayTours	Santa Cruz	boat	$250	0.5	$\lambda(t_5) = t_5$

(a)

Trips

	destination	phone		
u_1:	San Francisco	415-1200	0.63	$\lambda(u_1) = t_1 \wedge t_3$
u_2:	Santa Cruz	415-1200	0.56	$\lambda(u_2) = (t_1 \wedge t_4) \vee (t_1 \wedge t_5)$

(b)

Fig. 3.2 An example ULDB database with two base relations (a) and one derived relation (b).

Consider the ULDB database shown in Figure 3.2(a). This is a probabilistic version of our online travel agency portal database, which may have been obtained, for example, by scraping and integrating data from various websites. Consequently, each tuple may exist in the database with some confidence, and the confidence values are shown to the right of each tuple. Each tuple has a unique id, denoted as t_1–t_5 in the figure, and is annotated with lineage. Lineage is represented as a function λ that associates to each tuple id a Boolean formula whose symbols are other tuple ids in the database. Since Agencies and ExternalTours are base relations, the lineage of each of their tuples is the tuple itself (i.e., $\lambda(t_i) = t_i$, $i \in [1,5]$).

To illustrate lineage for derived data, suppose we execute the query $\pi_{\text{destination,phone}}(\text{Agencies} \bowtie \text{ExternalTours})$ on the database instance from Figure 3.2(a) and we store its result in a relation called Trips (see Figure 3.2(b)). In Trio, joins produce conjunctive lineage indicating that an output tuple exists due to the tuples in the input relations that joined to produce that output tuple. For example, the lineage of the first tuple in the relation Trips is given by $\lambda(u_1) = t_1 \wedge t_3$, indicating that its existence is due to the existence of both tuples t_1 and t_3 in relations

Agencies and ExternalTours, respectively. In turn, operations such as projection or union, produce disjunctive lineage, if they are duplicate-eliminating. By default, Trio operates with bags of tuples carrying tuple identifiers, and an additional duplicate elimination operator is applied, whenever set semantics are desired. For illustration purposes, let us assume that projection is duplicate-eliminating. The lineage of the second Trips tuple is given by $\lambda(u_2) = (t_1 \wedge t_4) \vee (t_1 \wedge t_5)$. This essentially indicates that the existence of u_2 in Trips is due to the existence of both t_1 and t_4, or both t_1 and t_5.

In a companion technical report, Das Sarma et al. [51] give algorithms for computing lineage over various relational operators in the context of the general ULDB model of Trio. In this general ULDB model, discussed by Benjelloun et al. [5] and Das Sarma et al. [52, Section 6], each tuple may have one or more *alternatives*. Each alternative a of a tuple t is associated with a confidence value which indicates the probability that t takes the value a. Moreover, each alternative is associated with its own distinct lineage, described in terms of alternatives of other tuples in the database. For the purpose of drawing a meaningful comparison between Trio's lineage and other versions of how-provenance discussed in this section, we prefer to base our discussion on the simplified ULDB model discussed for the most part by Das Sarma et al. [52] and ignore the multi-alternative aspect of ULDBs.

In the following, we present the algorithms for computing lineage given in Das Sarma et al. [52, Appendix A], which we restrict to the special case of simplified ULDB relations (i.e., without tuple alternatives). These algorithms can essentially be viewed as a procedural definition for lineage of ULDB data over relational operators.

Definition 3.3 (Trio lineage adapted from [51, Appendix A]). Consider input relations R, R_1 and R_2 with lineage λ, λ_1 and λ_2. Let λ' denote the lineage of the output obtained by applying a relational operator *op* on R (if *op* is unary), or R_1 and R_2 (if *op* is binary).

Selection. For each tuple $t \in R$ satisfying the selection predicate θ, add a new tuple t' in $\sigma_\theta(R)$ such that t' is identical to t, and $\lambda'(t') = \lambda(t)$.

Projection. For each tuple $t \in R$, add a new tuple t' in $\pi_U(R)$ such that $t' = t[U]$, and $\lambda'(t') = \lambda(t)$.

Join. For each pair of tuples $t_1 \in R_1$ and $t_2 \in R_2$ satisfying the join condition, add a new tuple t' in $R_1 \bowtie R_2$ such that $t'[U_1] = t_1$, $t'[U_2] = t_2$, and $\lambda'(t') = \lambda(t_1) \wedge \lambda(t_2)$, where U_1 and U_2 are the attributes of R_1 and respectively, R_2.

Union. For each tuple $t_1 \in R_1$ add a new tuple t' in $R_1 \cup R_2$ such that t' is identical to t_1, and $\lambda'(t') = \lambda(t_1)$. Similarly, for each tuple $t_2 \in R_2$ add a new tuple t' in $R_1 \cup R_2$ such that t' is identical to t_2, and $\lambda'(t') = \lambda(t_2)$.

Intersection. For each tuple $t_1 \in R_1$ such that there exists at least one tuple $t_2 \in R_2$ that is identical to t_1, add a new tuple t' in $R_1 \cap R_2$ such that t' is identical to t_1, and $\lambda'(t') = \lambda(t_1) \wedge (\lambda_2(t_2^1) \vee \cdots \vee \lambda_2(t_2^k))$, where $t_2^1, \ldots, t_2^k$ are all the tuples in R_2 that are identical to t_1.

Difference. For each tuple $t_1 \in R_1$ such that there does not exist a tuple $t_2 \in R_2$ that is identical to t_1, add a new tuple t' in $R_1 - R_2$ such that t' is identical to t_1, and $\lambda'(t') = \lambda(t_1)$.

Duplicate elimination. For each *distinct* tuple $t \in R$, add a new tuple t' in $\delta(R)$ such that $t' = t$, and $\lambda'(t') = \bigvee_{t'' \in R \wedge t''=t} \lambda(t'')$.

Grouping with aggregation. Let G be the attributes of relation R that we group by, and let A be the attribute of R that we aggregate over. For each group of R tuples $t_1, \ldots, t_k$ that agree on the values of attributes G, add a new tuple t' in $\gamma_{G,\text{aggr}(A)\ AS\ A'}(R)$ such that $t'[G] = t_1[G](= t_2[G] = \cdots = t_k[G])$, $t \bullet A'$ is the aggregated value computed over the A values of tuples in the group and $\lambda'(t') = \lambda(t_1) \wedge \cdots \wedge \lambda(t_k)$.

As mentioned earlier, all Trio operators are designed to work with bags of tuples. Set semantics for each operator can be obtained by explicitly applying the duplicate-elimination operator as an additional step. In addition to the basic operators, Trio also defines lineage for three other relational operators: intersection, difference and grouping with aggregation. For intersection, the lineage of a tuple t in the result of $R_1 \cap R_2$ is the conjunction between the lineage of the tuple t_1 in R_1 that t was created from, and a disjunction between the lineages of all tuples $t_2^1, \ldots, t_2^k$ in R_2 that are identical to t. Hence, the lineage indicates

that t was created due to the existence of t_1 in R_1, *together with* the existence of at least one identical tuple in R_2, and pinpoints exactly those identical tuples in R_2. Note that the procedure for computing lineage is not symmetric, in that $R_1 \cap R_2$ and $R_2 \cap R_1$ have different lineages.[1] However, it becomes symmetric if we take intersection to be duplicate-eliminating. In this case, the lineage of a tuple t in $R_1 \cap R_2$ is identical to the lineage of the same tuple in $R_2 \cap R_1$ and it is given by $\lambda'(t) = (\lambda_1(t_1^1) \vee \cdots \vee \lambda_1(t_1^m)) \wedge (\lambda_2(t_2^1) \vee \cdots \vee \lambda_2(t_2^k))$, where $t_1^1, \dots, t_1^m$ and respectively, $t_2^1, \dots, t_2^k$ are all the tuples identical to t in R_1, and respectively R_2. For difference, the lineage of a tuple t in the result of $R_1 - R_2$ is the lineage of the tuple t_1 in R_1 that t was created from. Finally, for grouping with aggregation, the lineage of a tuple t in the result is the conjunction of the lineages of all tuples in the group that corresponds to t.

If we consider only the basic relational operators consisting of selection, projection, union and natural join (as a special case of the join operator), and assume they are all duplicate-eliminating, then conjunction and disjunction in the lineage of an output tuple indicate *how* input tuples have been joined, and respectively, merged together to produce that output tuple. This is essentially the reason why we consider Trio's lineage to be a form of *how-provenance*, as opposed to having a flavor of *where-provenance* as described in [5, 52]. In fact, if we consider the simplified ULDB model where each tuple has a single alternative, then the Trio model is similar to the $\mathbb{N}[X]/_{\equiv}$-relational model where $\mathbb{N}[X]/_{\equiv}$ is the provenance semiring quotiented by the least congruence satisfying $x \cdot x \equiv x$. Thus, multiplication is idempotent (reflecting Boolean identity $x \wedge x = x$) but addition is not (reflecting the fact that tuples in Trio may appear multiple times). In particular, for the example query discussed above (with the input and output shown in Figure 3.2), evaluation in this semiring yields lineage $t_1 \cdot t_3$ for tuple u_1 and $t_1 \cdot t_4 + t_1 \cdot t_5$ for tuple u_2, which is the same as the results of Trio (reading $+$ as $\vee$ and $\cdot$ as $\wedge$). This characterization was suggested by Green [40]; note that this approach does not take node identities

[1] However, the possible instances for $R_1 \cap R_2$ and $R_2 \cap R_1$ are the same.

or tuple alternatives into account, so it is not clear that the full Trio model is an instance of the semiring model.

However, Trio's design is still under active development and so further comparison with other models is beyond the scope of this paper.

3.2.1 An Application: Computing Confidence Values of Derived Data

One of the main applications of lineage in Trio is in computing the confidence values for derived data. For example, since $\lambda(u_1) = t_1 \wedge t_3$, the confidence value associated with u_1 is computed as the probability that both t_1 and t_3 are in the database: $Pr(u_1) = Pr(t_1 \wedge t_3) = Pr(t_1) \cdot Pr(t_3) = 0.7 \cdot 0.9 = 0.63$, where $Pr(t)$ denotes the confidence value associated with tuple t. The lineage of u_2 can be reduced to $\lambda(u_2) = t_1 \wedge (t_4 \vee t_5)$. Hence, the confidence of u_2 is computed as $Pr(u_2) = Pr(t_1 \wedge (t_4 \vee t_5)) = Pr(t_1) \cdot (Pr(t_4 \vee t_5))$. Since t_4 and t_5 are independent of each other, we have $Pr(t_4 \vee t_5) = Pr(t_4) + Pr(t_5) - Pr(t_4) \cdot Pr(t_5) = 0.6 + 0.5 - 0.6 \cdot 0.5 = 0.8$. Hence, we have $Pr(u_2) = 0.7 \cdot 0.8 = 0.56$.

Trio adopts the eager approach for computing lineage. That is, the lineage of data that is the result of a query is computed at query execution time. Furthermore, whenever the derived data is stored in the database, its lineage is also recorded along with the data. The recorded lineage is subsequently used to ensure that confidence values of derived data are computed correctly, regardless of the query plan chosen by the query optimizer. To exemplify, let us consider the following two equivalent query plans for executing the query that produces the relation Trips.

Plan 1: $\pi_{\text{destination,phone}}(\text{Agencies} \bowtie \pi_{\text{name,destination}}(\text{ExternalTours}))$
Plan 2: $\pi_{\text{destination,phone}}(\text{Agencies} \bowtie \text{ExternalTours})$

Suppose confidence values are not computed based on the lineage associated with result tuples. Instead, let us assume that confidences and lineage are computed within each operator in the query plan, propagated along with data to the next operator, and finally to the result. For simplicity, assume the base relations consist of only tuples t_1, t_4 and t_5. According to Plan 1, the projection on External-

Tours produces one intermediate tuple (BayTours, SantaCruz), which we denote as w. Its lineage is $\lambda(w) = t_4 \vee t_5$ and its confidence is computed as $Pr(w) = Pr(t_4 \vee t_5) = 0.8$. The join results in one intermediate tuple with lineage $t_1 \wedge w$ whose confidence is computed as $Pr(t_1) \cdot Pr(w) = 0.7 \cdot 0.8 = 0.56$. The final projection produces the output tuple u_2 with final confidence 0.56.

In Plan 2, the join produces two intermediate tuples, which we denote as w_1 and w_2, where $\lambda(w_1) = t_1 \wedge t_4$ and $\lambda(w_2) = t_1 \wedge t_5$. Hence, we have $Pr(w_1) = Pr(t_1) \cdot Pr(t_4) = 0.42$ and $Pr(w_2) = Pr(t_1) \cdot Pr(t_5) = 0.35$. After projection, we obtain the output tuple u_2 with lineage $\lambda(u_2) = w_1 \vee w_2$. Hence, the confidence for u_2 is computed as $Pr(u_2) = Pr(w_1) + Pr(w_2) - Pr(w_1) \cdot Pr(w_2) = 0.623$, which is an incorrect result. This is because the computation assumed independence between the two intermediate tuples w_1 and w_2, which does not hold, since they both have "contributions" from the input tuple t_1. Intuitively, Plan 2 is "unsafe" in that computing confidences operator by operator is not guaranteed to produce the correct result. In contrast, computing confidences based on the lineage associated with the final output result, as done in Trio, is guaranteed to produce the correct confidences, even when the query plan is unsafe. Observe that expanding, followed by simplifying, the lineage expression of u_2 obtained with Plan 2 leads to $\lambda(u_2) = \lambda(w_1) \wedge \lambda(w_2) = (t_1 \wedge t_4) \vee (t_1 \wedge t_5) = t_1 \wedge (t_4 \vee t_5)$ and results in the correct confidence value as explained above.

Dalvi and Suciu [28] gave a sound and complete algorithm that will rewrite a select–distinct–project–join query into an equivalent safe query if and only if there is a safe plan for the query. This work complements the above work on calculating confidence using lineage in Trio.

3.3 Provenance Semirings and Recursion

Green et al. [43] generalized the provenance semirings framework to consider "pure" datalog programs, where the body of each datalog rule consists of only relational atoms. A difficulty that arises in extending the semiring model in this context has to do with handling recursion. We use the recursive transitive closure datalog program from Figure 3.3 to briefly illustrate the extension.

Source instance I:

Agencies

	name	based_in	phone
t_1:	BayTours	San Francisco	415-1200
t_2:	HarborCruz	Santa Cruz	831-3001

ExternalTours

	name	destination	type
t_3:	BayTours	San Francisco	cable car
t_4:	BayTours	Marine County	bus
t_5:	HarborCruz	Monterey	boat

Datalog program P:

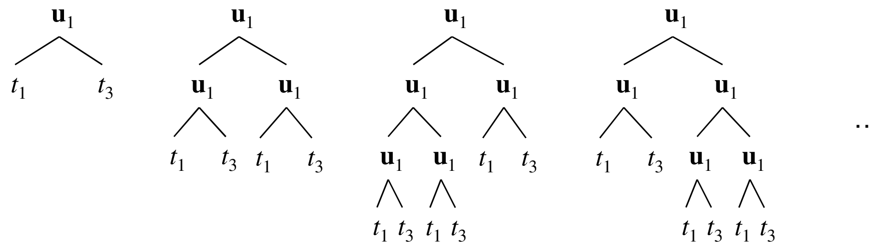
m_1: Tours(c_1,c_2) :– Agencies(n,c_1,p), ExternalTours(n,c_2,t).
m_2: Tours(c_1,c_2) :– Tours(c_1,c_3), Tours(c_3,c_2).

Instance J (result of P on I):

$\mathbf{u}_1$:	San Francisco	San Francisco	$\mathbf{u}_1 = t_1t_3 + \mathbf{u}_1^2$
$\mathbf{u}_2$:	San Francisco	Marine County	$\mathbf{u}_2 = t_1t_4 + \mathbf{u}_1\mathbf{u}_2$
$\mathbf{u}_3$:	Santa Cruz	Monterey	$\mathbf{u}_3 = t_2t_5$

Fig. 3.3 An example illustrating provenance semirings for datalog.

Green et al. [43] developed two interpretations for the semantics of datalog queries on K-relations, corresponding to the proof-theoretic and, respectively, fixpoint interpretations of datalog, and prove that they are equivalent. Under the proof-theoretic interpretation, the tag of an output tuple is the sum over all its derivation trees of the product of the tags at the leaves of each tree. Due to recursion, a tuple may have infinitely many derivation trees which leads to infinite sums. In our example from Figure 3.3, both output tuples (e.g., denoted $\mathbf{u}_1$ and $\mathbf{u}_2$) have infinitely many derivation trees. For example, some of the derivation trees for $\mathbf{u}_1$ are shown below.



Under the alternative, but equivalent, fixpoint interpretation, the possibly infinite sums of products are represented by means of an algebraic system of equations which reflect all possible ways of producing an output tuple as an effect of applying the immediate consequence operator of the datalog query. The algebraic system obtained for our example is shown in Figure 3.3. For example, the equation $\mathbf{u}_1 = t_1 t_3 + \mathbf{u}_1^2$ indicates that the output tuple $\mathbf{u}_1$ can be obtained in two ways: by joining the source tuples t_1 and t_3, or by joining with itself. Note that since the immediate consequence operator may involve output tuples, in addition to input tuples, the algebraic system involves two sets of variables, corresponding to the set of input, and respectively, output tuple tags.

To capture the how-provenance of tuples in the result of a datalog query on $\mathbb{N}[X]$-relations, where X is the set of source tuple ids, Green et al. identify the semiring $\mathbb{N}^\infty[[X]]$ of formal power series as appropriate. A formal power series with variables from X and coefficients from $\mathbb{N}^\infty$ is a mapping that associates to each monomial with variables from X a coefficient in $\mathbb{N}^\infty$. Note that polynomials in $\mathbb{N}[X]$ are no longer sufficient, since infinite sums need to be supported. Also, the coefficients are from $\mathbb{N}^\infty$, as opposed to $\mathbb{N}$, since an output tuple may be produced in infinitely many ways from the same monomial. To illustrate, the provenance semiring for our example in Figure 3.3 is $\mathbb{N}^\infty[[\{t_1,\ldots,t_6\}]]$.

3.3.1 Computing How-Provenance for Datalog Programs

Intuitively, the how-provenance of tuples in the output of a datalog program can be computed by solving the corresponding fixpoint system of equations. For example, the how-provenance of the output tuple $\mathbf{u}_1$ in Figure 3.3 is the infinite sum $t_1 t_3 + t_1^2 t_3^2 + 2t_1^3 t_3^3 + 5t_1^4 t_3^4 + \cdots$. Green et al. [43] show that it is decidable whether the how-provenance of a tuple is a polynomial in $\mathbb{N}[X]$ and give an algorithm for computing it. The algorithm proceeds by repeatedly applying the immediate consequence operator starting from source tuples and keeping track of the derivation trees produced in the process until a fixpoint is reached. At every step, the algorithm detects if an output tuple has infinitely many derivations and places it in a separate pool, so that no other derivation

trees are produced for it, thus ensuring termination. For each output tuple t with finitely many derivation trees, the algorithm outputs a polynomial obtained by taking the sum of products of tuple ids at the leaves of all t's derivation trees. If t has infinitely many derivation trees, Green et al. show that it is decidable whether the how-provenance of t is in $\mathbb{N}[[X]]$, or in $\mathbb{N}^\infty[[X]]$, and give an algorithm for computing the coefficient for a particular monomial in the provenance series of t, even when this coefficient is ∞.

3.4 How-Provenance for Schema Mappings

In this section we turn our attention to *schema mappings*, which have been widely used to specify relationships between schemas in applications requiring interoperability between heterogeneous data sources (e.g., data exchange [47], data integration [48], and dataspaces [45]). The ORCHESTRA data sharing system [42, 44] proposed an extension of provenance semirings to express how-provenance over schema mappings. A year earlier, the notion of *route* was introduced in the context of the SPIDER system [3, 21] for debugging schema mappings. Routes are also a form of how-provenance over schema mappings. In this section, we discuss the two systems and compare their corresponding notions of how-provenance.

Schema Mappings. Intuitively, schema mappings are high-level abstractions for expressing the relationships between two database schemas. Formally, a schema mapping is a quadruple $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{\mathrm{st}}, \Sigma_{\mathrm{t}})$ [47], where $\mathbf{S}$ is a source schema, $\mathbf{T}$ is a target schema, and Σ_{st} is a set of *source-to-target* (s–t) tgds and Σ_{t} is the union of a finite set of *target* tgds with a finite set of *target equality generating dependencies (egds)*. An s–t tgd is a tgd of the form $\forall \mathbf{x}\, \phi(\mathbf{x}) \rightarrow \exists \mathbf{y} \psi(\mathbf{x}, \mathbf{y})$, where $\phi(\mathbf{x})$ is a conjunction of atomic formulas over $\mathbf{S}$ and $\psi(\mathbf{x}, \mathbf{y})$ is a conjunction of atomic formulas over $\mathbf{T}$. A *target tgd* has a similar form, except that $\phi(\mathbf{x})$ is a conjunction of atomic formulas over $\mathbf{T}$. A *target egd* is of the form $\forall \mathbf{x}\, \phi(\mathbf{x}) \rightarrow x_1 = x_2$, where $\phi(\mathbf{x})$ is a conjunction of atomic formulas over $\mathbf{T}$, and x_1 and x_2 are variables among $\mathbf{x}$.

A simple schema mapping is shown in the top of Figure 3.4. The source schema is our travel portal database schema, the target schema

Source-to-target dependencies (Σ_{st}):

m_1 : Agencies(n,b,p) $\rightarrow \exists I$ Trips(I,n,p)

m_2 : ExternalTours(n,d,t) $\rightarrow \exists I$ Transportation(I,t,p)

Target dependencies (Σ_t):

m_3 : Transportation(i,t,p) $\rightarrow \exists N \exists P$ Trips(i,N,P)

Source instance I (of peer S):

Agencies

	name	based_in	phone
t_1:	BayTours	San Francisco	415-1200
t_2:	HarborCruz	Santa Cruz	831-3000

ExternalTours

	name	destination	type
t_3:	BayTours	San Francisco	cable car

Target instance J (of peer T):

Trips

	id	agency	phone	
$\mathbf{u}_1$:	I_1	BayTours	415-1200	$Pv(\mathbf{u}_1) = m_1(t_1)$
$\mathbf{u}_2$:	I_2	HarborCruz	831-3000	$Pv(\mathbf{u}_2) = m_1(t_2)$
$\mathbf{u}_3$:	I_3	N_3	P_3	$Pv(\mathbf{u}_3) = m_3(\mathbf{u}_4)$

Transportation

	id	type
$\mathbf{u}_4$:	I_3	cable car

$Pv(\mathbf{u}_4) = m_2(t_3)$

Fig. 3.4 Example schema mapping, and source and target instances satisfying the mapping.

consists of relations Trips(`id`, `agency`, `phone`) and Transportation(`id`, `type`), and the relationships between the two schemas are given by the s–t tgds m_1 and m_2, and a target tgd m_3. Intuitively, m_1 specifies that for each Agencies tuple in the source, there must exist a tuple in the Trips target relation, with `agency_name` and `phone` values extracted from the `name` and `phone` values of the Agencies source tuple. Similarly, according to m_2, for every ExternalTours source tuple, there must exist a Transportation tuple in the target with `type` and `price` values extracted from the corresponding values of the ExternalTours tuple. Finally, the target tgd m_3 expresses a referential constraint on the target data: for every Transportation tuple there must exist a corresponding Trips tuple with the same `trip_id` value.

3.4.1 ORCHESTRA

ORCHESTRA [42, 44] is a collaborative data sharing system that supports a network of interconnected peers wishing to exchange information with each other. In this context, provenance semirings are used to filter data based on trust conditions, as well as perform

updates *incrementally*, without recomputing a peer's data instance from scratch.

We illustrate the provenance component of ORCHESTRA using the example schema mapping in Figure 3.4. Assume we have two peers **S** and **T**, whose database schemas are the source and respectively, the target schemas used in our example. The tgds m_1–m_3 specify the relationships between the relational schemas of the two peers.[2] Initially, the instances of both peers are empty. Suppose tuples t_1–t_3 shown in Figure 3.4 are inserted into peer's **S** instance, and **S** decides to share this newly added information with **T**. Hence, the instance of **T** is updated as follows. Tuples $\mathbf{u}_1$ and $\mathbf{u}_2$ are added to the Trips relation due to the tgd m_1. At the same time, the how-provenance of these tuples is also computed and stored in additional data structures at peer **T**. Conceptually, the provenance is recorded as shown in Figure 3.4, next to each tuple of peer **T**. For example, the provenance of $\mathbf{u}_1$ is recorded as $Pv(\mathbf{u}_1) = m_1(t_1)$, indicating that u_1 can been derived from tuple t_1, and mapping m_1 is involved in the derivation. Furthermore, due to m_2, the tuple $\mathbf{u}_4$ is added in Trips, with $Pv(\mathbf{u}_4) = m_2(t_3)$. Finally, due to m_3, the tuple $\mathbf{u}_3$ is added in Trips, with $Pv(\mathbf{u}_3) = m_3(\mathbf{u}_4)$. Note that "unknown" new values not present in the instance of **S** (e.g., I_1–I_3) are generated in the process. Such values are called *labeled nulls* [32] and they are automatically generated during the transformation as needed, to enforce the semantics of the tgds.

If we ignore the provenance aspect, the process of computing a peer's instance described above is known as the classical *chase* procedure [1]. Chase with tgds has been used before, in a closely related context, to compute canonical instances in data exchange [32]. Given a schema mapping $\mathcal{M}$ and a source instance I, a target instance that together with I satisfies $\mathcal{M}$ is called a *solution* for I under $\mathcal{M}$ in the terminology of [32]. For example, the target instance J shown in Figure 3.4 is a solution for the source instance I under the schema mapping shown in the same figure.

An important aspect of ORCHESTRA is that it *eagerly* computes and stores data provenance as new data is derived from one peer to

[2] In ORCHESTRA, egds are not considered.

another. ORCHESTRA's representation of provenance is based on the datalog provenance semirings framework of Green et al. [43]: provenance semiring expressions are essentially recorded as a system of fixpoint equations (one equation for each derived tuple). Note that recording mapping information along with the tuple ids in the provenance representation is in fact an extension of the original semiring model. As such, this provenance representation is used to support two important aspects of ORCHESTRA: (1) filtering updates based on trust conditions, and (2) *incrementally* maintaining a peer's instance, when deletions occur in the system.

Filtering updates based on trust. Each peer may specify trust conditions on the data derived from other peers. For example, a peer may distrust a subset of data of a specific peer, or data derived through a specific mapping. Trustworthiness of derived data is formalized in terms of provenance: data that cannot be derived from trusted tuples and via trusted mappings is considered untrustworthy and is not used in updating the peer's local instance. For example, suppose peer **T** does not trust data derived through m_2. The provenance expression for $\mathbf{u}_4$ is $Pv(\mathbf{u}_4) = m_2(t_3)$ and shows that $\mathbf{u}_4$ is derived via the distrusted mapping m_2. Hence, $\mathbf{u}_4$ will not be added to **T**'s instance. On the other hand, if the provenance of $\mathbf{u}_4$ would be $Pv(\mathbf{u}_4)) = m_2(t_3) + m^*(t^*)$, where m^* is a trusted mapping and t^* is a trusted tuple, then **T** would accept the insertion of $\mathbf{u}_4$, since it can be derived from trusted mappings and tuples.

Incremental update exchange. An algorithm for propagating deletions of tuples that occur in a peer's instance to other peers in the system is given in [42]. The algorithm leverages provenance information to compute the set of tuples which can no longer be derived from data still present in the system, and hence should be deleted. To illustrate, suppose t_3 is deleted from peer **S**. Using the provenance recorded for $\mathbf{u}_3$, it can be easily tested that this tuple is no longer derivable. Hence, $\mathbf{u}_3$ can be deleted from **T**'s instance and the effect is propagated recursively in the system. Provenance information indicates that the only tuple that could be affected by the additional deletion is $\mathbf{u}_4$, hence the algorithm checks if $\mathbf{u}_4$ is still derivable, and the process continues until no other tuples need to be deleted.

3.4.2 Routes

We illustrate the notion of routes and overview the SPIDER schema mapping debugging system [3, 21] with the example from Figure 3.4. The idea behind SPIDER is to use source and target data to assist a human mapping designer in understanding, refining and debugging a schema mapping, and this is similar in spirit to debuggers for standard programming languages which allow one to understand a program by analyzing its behavior on input test data. A mapping designer is allowed to browse through and select on source and target data, and SPIDER displays *routes* which illustrate the behavior of the schema mapping with the selected data. For example, R_1 shown below is a route for the target tuple $\mathbf{u}_3$:

$$R_1 : \langle I, \emptyset \rangle \xrightarrow{(m_2,h)} \langle I, \{\mathbf{u}_4\} \rangle \xrightarrow{(m_3,h')} \langle I, \{\mathbf{u}_3, \mathbf{u}_4\} \rangle$$

where $h = \{n \mapsto$ "BayTours", $d \mapsto$ "San Francisco", $t \mapsto$ "cable car", $I \mapsto I_3\}$ and $h' = \{i \mapsto I_3$, $t \mapsto$ "cable car", $N \mapsto N_3$, $P \mapsto P_3\}$. For simplicity, the route can be schematically represented as $t_3 \xrightarrow{m_2} \mathbf{u}_4 \xrightarrow{m_3} \mathbf{u}_3$.

Intuitively, the route demonstrates the existence of the selected target tuple $\mathbf{u}_3$ with the schema mapping, the source data and some intermediate target data (i.e., the target tuple $\mathbf{u}_4$). The route consists of two *satisfaction steps* with (m_2, h) and (m_3, h'), respectively. Note that under h, the LHS of m_2 is t_3, while the RHS of m_2 is $\mathbf{u}_4$. Hence, the first satisfaction step (m_2, h) demonstrates the existence of the target tuple $\mathbf{u}_4$ with the mapping m_2 and the source tuple t_3. Furthermore, the second satisfaction step (m_3, h') demonstrates the existence of $\mathbf{u}_3$ with m_3 and the target tuple $\mathbf{u}_4$. This route may help a mapping designer in discovering a "bug" in the schema mapping: the association between the values "BayTours" and "cable car" in the External-Tours tuple t_3 is somehow lost when this data was transferred from the source to the target. Indeed, the mapping m_2 does not migrate the `name` values of ExternalTours tuples to the target. Furthermore, the route indicates that $\mathbf{u}_3$ is created to ensure the satisfaction of the target referential constraint m_3. With this information, the designer may now proceed to refine the schema mapping so as to eliminate the bug. She may then repeat the process over several iterations, until she is satisfied with the resulting schema mapping.

From the example above it is easy to see that routes are a form of how-provenance. They demonstrate *how* the selected target tuple is witnessed with data from the source and the target instances, and the schema mapping. Formally, a route for a set of selected target tuples J_s is a sequence of satisfaction steps defined as follows.

Definition 3.4 (Satisfaction step [21, Definition 3.1]). Let m be a tgd $\forall \mathbf{x}\, \phi(\mathbf{x}) \rightarrow \exists \mathbf{y} \psi(\mathbf{x}, \mathbf{y})$. Let K_1 and K be instances such that $K_1 \subseteq K$ and $K \models m$. Furthermore, let h be a homomorphism from $\phi(\mathbf{x}) \wedge \psi(\mathbf{x}, \mathbf{y})$ to K such that h is also a homomorphism from $\phi(\mathbf{x})$ to K_1. We say that m can be satisfied on K_1 with homomorphism h and solution K, or simply *m can be satisfied on K_1 with homomorphism h*, if K is understood from the context. The *result of satisfying m on K_1 with homomorphism h is K_2*, where $K_2 = K_1 \cup h(\psi(\mathbf{x}, \mathbf{y}))$ and $h(\psi(\mathbf{x}, \mathbf{y})) = \{R(h(\mathbf{z})) | R(\mathbf{z})$ is a relation atom in $\psi(\mathbf{x}, \mathbf{y})\}$. We denote this step as $K_1 \xrightarrow{m,h} K_2$.

Definition 3.5 (Route [21, Definition 3.3]). Let $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ be a schema mapping, I be a source instance and J be a solution of I under $\mathcal{M}$. Let $J_s \subseteq J$. A *route for J_s with $\mathcal{M}$, I and J* (in short, *a route for J_s*) is a finite non-empty sequence of satisfaction steps $(I, \emptyset) \xrightarrow{m_1,h_1} (I, J_1), \ldots, (I, J_{n-1}) \xrightarrow{m_n,h_n} (I, J_n)$, where (a) $J_i \subseteq J, 1 \leq i \leq n$, (b) m_i, $1 \leq i \leq n$, are among $\Sigma_{st} \cup \Sigma_t$, and (c) $J_s \subseteq J_n$.

3.4.2.1 Computing Routes

As Definition 3.5 shows, routes have declarative semantics, based on the *logical satisfaction* of dependencies, and hence independent of the implementation of the schema mapping. In [21], Chiticariu and Tan describe algorithms for *lazily* computing routes by examining the source instance, the target instance, and the schema mapping. Thus, their debugging technique based on routes may be easily deployed on any mapping-based data integration, data exchange or peer-to-peer data sharing system *without* changes to the underlying mapping execution engine.

One of the algorithms of [21] computes a finite, concise representation of *all routes* for a set of target tuples. Given a source instance I, a target instance J that is a solution for I under the schema mapping $\mathcal{M}$, and a selected set of target tuples J_s from J, their algorithm outputs a concise representation of *all routes* for J_s with I, J and $\mathcal{M}$, called a *route forest*. In the terminology of [21], *all routes* for J_s are characterized by the set of all *minimal* routes for J_s. Intuitively, a route for J_s is *minimal* if none of its satisfaction steps can be removed and the result still remains a route for J_s. For example, the earlier route R_1 for $\mathbf{u}_3$ is minimal, while the route R_2 shown below is not a minimal route for $\mathbf{u}_3$. (Below, h'' is such that it maps the LHS of m_1 to t_1 and the RHS of m_1 to $\mathbf{u}_1$.)

$$R_2 : \langle I, \emptyset \rangle \xrightarrow{(m_2,h)} \langle I, \{\mathbf{u}_4\} \rangle \xrightarrow{(m_3,h')} \langle I, \{\mathbf{u}_3, \mathbf{u}_4\} \rangle \xrightarrow{(m_1,h'')} \langle I, \{\mathbf{u}_1, \mathbf{u}_3, \mathbf{u}_4\} \rangle$$

The challenge in developing an algorithm for computing the route forest was reasoning about recursive mappings. The algorithm given in [21] proceeds by finding all pairs (m,h), where m is an s–t tgd, that can witness the selected tuple u with source data. Next, it finds all pairs (m,h), where m is a target tgd, that can witness u with target tuples. Finally, the algorithm recursively proceeds to find all possible pairs (m,h) for witnessing each additional target tuple encountered in the process. Each such tuple once explored, is never explored again. This ensures termination, since the computation is not trapped in infinite loops, and furthermore, a polynomial running time for the algorithm, since the total number of possible (m,h) pairs is polynomial in the sizes of I and J. Indeed, one of the main results of [21] shows that although there may be exponentially many minimal routes, the route forest is a complete and polynomial size representation of all such minimal routes for the selected tuples, and moreover, it is computed in polynomial time (in the sizes of I and J).

3.4.3 Provenance Semirings and ORCHESTRA vs Routes and SPIDER

Let us now illustrate some of the similarities and differences between routes and provenance semirings. A first observation is that the two

models have been originally designed for different languages: the provenance semiring model [43] applies to datalog programs, while routes apply to schema mappings (expressed using tgds and egds), which are more expressive than datalog. (Essentially, datalog rules are *full* tgds, i.e., without existential quantifiers, with a single atom in the RHS.) However, the original provenance semiring model has been extended in ORCHESTRA to handle a version of datalog with skolem functions, which, in addition to full tgds, also captures non-full tgds by appropriately skolemizing the existential variables. Hence, ORCHESTRA's provenance semirings model handles schema mappings expressed using tgds. It does not, however, handle schema mappings with egds.

In order to draw a meaningful comparison between routes and provenance semirings we use the schema mapping shown below, obtained by expressing the two datalog rules from Figure 3.3 as tgds. We also use I and J from Figure 3.3 as source and target instances. Clearly, I and J satisfy the schema mapping below.

Source-to-target dependencies (Σ_{st}):
m_1 : Agencies(n,c_1,ph) $\wedge$ ExternalTours(n,c_2,t,p) $\rightarrow$ Tours(c_1,c_2)
Target dependencies (Σ_t):
m_2 : Tours(c_1,c_3) $\wedge$ Tours(c_3,c_2) $\rightarrow$ Tours(c_1,c_2)

Consider the tuple $\mathbf{u}_3$ in J. In the original semirings framework of Green et al. [43] the provenance polynomial of $\mathbf{u}_3$ is $t_2 t_5$. A route for $\mathbf{u}_3$ is $\langle I, \emptyset\rangle \xrightarrow{(m_1,h)} \langle \mathrm{I}, \{\mathbf{u}_3\}\rangle$, where $h =\{n \mapsto$ "HarborCruz", $c_1 \mapsto$ "Santa Cruz", $ph \mapsto$ "831-3000", $c_2 \mapsto$ "Monterey", $t \mapsto$ "boat"$\}$. Under h, the LHS of m_1 consists of tuples t_2 and t_5 and hence, the route can be schematically represented as $\{t_2, t_5\} \xrightarrow{m_1} \{\mathbf{u}_3\}$. A minor difference between the route and the provenance polynomial is that the former tells us more about $\mathbf{u}_3$, in addition to the fact that it can be obtained by joining t_2 and t_5. In particular, the route reveals that the tgd m_1 is involved, and demonstrates *how* it is involved, through the assignment h. However, the mapping is present in the provenance semiring representation used in ORCHESTRA (i.e., in the form $m_1(t_2 \cdot t_5)$), and the homomorphism h can be easily derived from this representation.

Another difference between routes and provenance semirings lies in the representation of provenance for tuples that can be derived in

infinitely many ways. To illustrate, consider the target tuple $\mathbf{u}_1$ whose provenance is given by the polynomial with infinitely many summands

$$t_1t_2 + t_1^2t_2^2 + 2t_1^3t_2^3 + 5t_1^4t_2^4 + \cdots .$$

In the work of Green et al. [43], as well as in ORCHESTRA, infinite provenance is represented as part of the solution for a finite system of equations, and gave an algorithm that computes the coefficient of a given monomial in a provenance series, even if it is ∞. As expected, there are also infinitely many routes for $\mathbf{u}_1$, some of which are shown below.

$$\langle I, \emptyset\rangle \xrightarrow{(m_1,h_1)} \langle I, \{\mathbf{u}_1\}\rangle$$

$$\langle I, \emptyset\rangle \xrightarrow{(m_1,h_1)} \langle I, \{\mathbf{u}_1\}\rangle \xrightarrow{(m_2,h_2)} \langle I, \{\mathbf{u}_1\}\rangle$$

$$\langle I, \emptyset\rangle \xrightarrow{(m_1,h_1)} \langle I, \{\mathbf{u}_1\}\rangle \xrightarrow{(m_2,h_2)} \langle I, \{\mathbf{u}_1\}\rangle \xrightarrow{(m_2,h_2)} \langle I, \{\mathbf{u}_1\}\rangle$$

$$\cdots$$

$$\langle I, \{\mathbf{u}_1\}\rangle \xrightarrow{(m_2,h_2)} \langle I, \{\mathbf{u}_1\}\rangle \xrightarrow{(m_2,h_2)} \langle I, \{\mathbf{u}_1\}\rangle \xrightarrow{(m_2,h_2)} \ldots \langle I, \{\mathbf{u}_1\}\rangle \xrightarrow{(m_2,h_2)} \ldots$$

Here $h_1 = \{n \mapsto$ "BayTours", $c_1 \mapsto$ "San Francisco", $ph \mapsto$ "415-1200", $c_2 \mapsto$ "San Francisco", $t \mapsto$ "cable car"$\}$ and $h_2 = \{c_1 \mapsto$ "San Francisco", $c_2 \mapsto$ "San Francisco", $c_3 \mapsto$ "San Francisco"$\}$. Only two distinct satisfaction steps are essentially involved in the routes above, and they can be schematically depicted as $\{t_1, t_3\} \xrightarrow{m_1} \{\mathbf{u}_1\}$ and respectively, $\{\mathbf{u}_1\} \xrightarrow{m_2} \{\mathbf{u}_1\}$. In [21], Chiticariu and Tan characterize the (possibly infinite) set of all routes in terms of the route forest which embeds all minimal routes and give an algorithm for extracting routes from the route forest. However, their algorithm does not capture the number of distinct derivations as provenance semirings do.

Apart from the representation of provenance, a difference between SPIDER and ORCHESTRA consists in the method by which they compute provenance. ORCHESTRA adopts the eager approach, where provenance is computed and stored along with the data as data is exchanged between peers. In contrast, SPIDER adopts the lazy approach for computing provenance: no extra information is stored during exchange and provenance is computed only when needed by analyzing the source and target instances, along with the schema mapping. Also recall that the routes definition is based on the logical

satisfaction of the mapping and applies to any target instance that is a *solution*, with no additional properties. Hence, SPIDER is independent of the mapping execution engine and can be deployed on any mapping-based data exchange, integration or sharing system without making any changes to the system. In contrast, ORCHESTRA's approach is tied to the mapping execution engine: it can only work with instances computed within ORCHESTRA, since it requires storing provenance information during exchange.

Another difference between SPIDER and ORCHESTRA (and provenance semirings for that matter) is that ORCHESTRA does not handle schema mappings with target egds. SPIDER works in the presence of target egds, however, it still does not provide the level of support one might desire. To illustrate, consider a schema mappings with two source-to-target tgds $m_1\colon S(x,y,z) \to \exists N\ T(x,y,N)$ and $m_2\colon S(x,y,z) \to \exists N\ T(x,N,z)$, and a target egd $m_3\colon T(x,y_1,z_1) \wedge T(x,y_2,z_2) \to y_1 = y_2 \wedge z_1 = z_2$, expressing that the first attribute of the target relation T is a key. Consider a source instance $I = \{(1,2,3)\}$. Applying the tgds m_1 and m_2 results in $J' = \{T(1,2,N_1), T(1,N_2,3)\}$, where N_1 and N_2 are labeled nulls. However, to satisfy m_3, we need to equate N_1 with the value 3 and N_2 with the value 2, and thus we obtain the solution $J = \{T(1,2,3)\}$. (If a labeled null and a source value need to be equated during the chase procedure used for data exchange [32], the null is always replaced by the source value.) SPIDER computes two routes for the target tuple: $S(1,2,3) \xrightarrow{m_1,h_1} T(1,2,3)$ and $S(1,2,3) \xrightarrow{m_2,h_2} T(1,2,3)$, where $h_1 = \{x \mapsto 1, y \mapsto 2, z \mapsto 3, N \mapsto 3\}$ and $h_2 = \{x \mapsto 1, y \mapsto 2, z \mapsto 3, N \mapsto 2\}$. Note that none of these routes demonstrate how the egd m_3 is involved in witnessing the target tuple. Certainly, it would be useful to extend SPIDER and ORCHESTRA to demonstrate the contribution of egds in the provenance of target tuples.

Finally, since the definition of routes is based on the logical satisfaction of tgds, SPIDER may arguably compute "spurious" routes, if one takes an operational view of the exchange. To illustrate, consider a schema mapping consisting of tgds $m_1\colon R(x,y) \to T(x,y)$ and $m_2\colon S(x,y) \to \exists N\ T(x,N)$. Consider a source instance I consisting of

tuples $R(1,2)$ and $S(1,3)$. A possible solution for I consists of the tuple $T(1,2)$. SPIDER computes two routes for this tuple: $R(1,2) \xrightarrow{m_1} T(1,2)$ and $S(1,3) \xrightarrow{m_2} T(1,2)$. Arguably, the second route is "spurious" because $T(1,2)$ could not be created by executing m_2. In ORCHESTRA, only the first is a derivation computed for $T(1,2)$.

4

Where-Provenance

In this section, we discuss the notion of *where-provenance* introduced by Buneman et al. [13] and describe subsequent work [14] in which where-provenance has been used to study the annotation placement problem. We also discuss an alternative where-provenance semantics implemented in the DBNotes annotation management system [7, 22] that is invariant under query rewriting. Finally, we give an overview of DBNotes, which provides a mechanism for systematically tracing where-provenance.

4.1 Where-Provenance

The notion of *where-provenance* has been introduced by Buneman et al. [13]. In contrast to why- and how-provenance, which indicate the tuples in the input that witness the existence of an output tuple according to a query Q, *where-provenance* tells us precisely *from where* an attribute value v in the output was copied according to Q. Hence, while why- and how-provenance describe the relationships between input and output tuples, where-provenance describes the relationships between input and output *locations*. Recall that a *location* (R, t, A) refers to the

Agencies

	name	based_in	phone
t_1:	BayTours	San Francisco	415-1200
t_2:	HarborCruz	Santa Cruz	831-3000

ExternalTours

	name	destination	type	price
t_3:	BayTours	San Francisco	cable car	$50
t_4:	BayTours	Santa Cruz	bus	$100
t_5:	BayTours	Santa Cruz	boat	$250
t_6:	BayTours	Monterey	boat	$400
t_7:	HarborCruz	Monterey	boat	$200
t_8:	HarborCruz	Carmel	train	$90

Fig. 4.1 Our example database: an online travel portal.

Q_1:
SELECT a.name, a.phone
FROM Agencies a, ExternalTours e
WHERE a.name = e.name AND e.type='boat'

Result of Q_1:

name	phone
BayTours	415-1200
HarborCruz	831-3000

Fig. 4.2 Example query.

field A of a tuple t of a relation R. Intuitively, the where-provenance of a value v found at location l in the output of Q is the set of all locations in the input database from which v was copied according to Q.

To illustrate, consider our example travel portal database reproduced for convenience in Figure 4.1. Consider the query Q_1 asking for travel agencies offering boat tours shown in Figure 4.2. The where-provenance of the value "BayTours" in the first tuple in the result of Q_1 is the location (Agencies, t_1, name) in the input database, since "BayTours" was copied from the name attribute of the tuple t_1 in the Agencies relation, according to Q_1. Similarly, the where-provenance of the value "HarborCruz" in the second output tuple is the input location (Agencies, t_2, name).

Buneman et al. [13] defined where-provenance for a deterministic semi-structured data model and an associated query language. In a follow-up work [14], they adapted the definition to the relational model with SPJRU queries, and studied the connection between

where-provenance and annotation propagation through queries. Intuitively, the where-provenance of a location in the result of a query determines the set of all annotations in the input database to be associated with that output location. Subsequently, Buneman et al. [14] defined the semantics of where-provenance by means of a set of *propagation rules* for annotations, which specify, for each individual relational algebra operator, how annotations associated with input locations propagate to output locations according to the operator.

We next explain the annotation propagation rules of [14], and formally define where-provenance after that. In what follows, we consider annotated relations in which each location carries a set of zero or more annotations. To illustrate, consider the database instance shown in Figure 4.3(a). Each location in R_1 and R_2 is associated with exactly one annotation. For example, the locations (R_1, t_1, A) and (R_1, t_1, B) have annotations $\{a_1\}$, and respectively, $\{a_2\}$.

Definition 4.1 (Annotation Propagation Rules [14, Section 3]).

(1) If $t \in \sigma_\theta(R)$ then an annotation on (R, t', A) propagates to $(\sigma_\theta(R), t, A)$ if $t = t'$.
(2) If $t \in \pi_U(R)$, where U is a set of attributes, then an annotation on (R, t', A) propagates to $(\pi_U(R), t, A)$ if $A \in U$ and $t = t'[U]$.
(3) If $t \in \rho_{[A \mapsto B]}(R)$, then an annotation on (R, t', C) propagates to $(\rho_{[A \mapsto B]}(R), t, C[A \mapsto B]))$ if $t = t'[A \mapsto B]$.
(4) If $t \in R_1 \bowtie R_2$ then an annotation on (R_1, t_1, A) (or (R_2, t_2, A)) propagates to $(R_1 \bowtie R_2, t, A)$ if $t[U_1] = t_1$ (or $t[U_2] = t_2$), where U_1 and U_2 are the attributes of R_1 and R_2, respectively.
(5) If $t \in R_1 \cup R_2$ then an annotation on (R_1, t_1, A) (or (R_2, t_2, A)) propagates to $(R_1 \cup R_2, t, A)$ if $t = t_1$ (or $t = t_2$).

It is easy to see that the annotation propagation behavior defined by the above rules is indeed based on where data is copied from. Figure 4.3(b) illustrates the rules for the select, project, renaming and join

R_1

	A	B
t_1:	1^{a_1}	2^{a_2}
t_2:	1^{a_3}	3^{a_4}

R_2

	A	C
t_3:	1^{a_5}	2^{a_6}
t_4:	4^{a_7}	5^{a_8}

(a)

$\sigma_{B=2}(R_1)$

A	B
1^{a_1}	2^{a_2}

$\pi_A(R_1)$

A
$1^{a_1,a_3}$

$\rho_{B\mapsto C}(R_1)$

A	C
1^{a_1}	2^{a_2}
1^{a_3}	3^{a_4}

$R_1 \bowtie R_2$

A	B	C
$1^{a_1,a_5}$	2^{a_2}	2^{a_6}
$1^{a_3,a_5}$	3^{a_4}	2^{a_6}

(b)

Fig. 4.3 An input database (a); annotation propagation for relational algebra operators based on where-provenance (b).

operators with the input database from Figure 4.3(a). In the result of $\sigma_{B=2}(R_1)$ the values "1" and "2" are associated with annotations $\{a_1\}$, and respectively, $\{a_2\}$, since they were copied from the values at input locations (R_1,t_1,A) and (R_1,t_1,B), respectively. In the result of $\pi_A(R_1)$, the value "1" has two annotations, since it was copied from the "1" values at locations (R_1,t_1,A) and (R_1,t_2,A), according to the semantics of projection. Note that by projecting on the A attribute of R_1 we obtain $\{(1^{a_1}),(1^{a_3})\}$. However, since set semantics are assumed, identical tuples are merged and their annotations are unioned together, thus obtaining a single tuple $(1^{a_1,a_3})$. The above operation is called *annotation-union* in [7, 56].

The rule for renaming is straightforward. For $R_1 \bowtie R_2$, an output tuple is formed by combining two input tuples (one from R_1 and one from R_2) Thus, the values corresponding to the join attributes in the output always collect annotations from the corresponding locations in both these input tuples. For example, the value "1" of the first tuple in $R_1 \bowtie R_2$ has annotations $\{a_1,a_3\}$, since "1" was copied from both the values at locations (R_1,t_1,A) and (R_2,t_3,A) according to the semantics of join. For the case of union, the result of $R_1 \cup R_2$ is $\{(1^{a_1,a_5},2^{a_2,a_6}),(1^{a_3},3^{a_4}),(4^{a_7},5^{a_8})\}$. Note that the values "1" and "2"

of the first tuple have two annotations each, obtained by unioning together the annotations from the corresponding values of the first tuple in R_1 and respectively, the first tuple in R_2.

4.1.1 An Alternative Definition of Where-Provenance

In this section, we shall give an alternative, more direct, definition of where-provenance for SPJRU queries, which will allow us to relate it with other notions of provenance discussed in previous sections. Recall that we defined *FieldLoc* as the set of locations, or triples of the form (R,t,A); we write $\mathcal{P}(\mathit{FieldLoc})$ for the set of sets of locations.

We define where-provenance as a partial function $\mathsf{Where}(Q,I,t)$ that takes an instance $I \in \mathbf{R}\text{-}\mathit{Inst}$, an SPJRU query $Q\colon \mathbf{R} \to U$, and a tuple $t \in U\text{-}\mathit{Tuple}$ and yields either a record of sets of locations, or $\perp$, meaning "undefined". As with lineage and why-provenance, we return $\perp$ to indicate that $t \notin Q(I)$. Otherwise, we obtain a record u where each of the fields is associated with a set of annotations. For example, if $u = \mathsf{Where}(Q,I,t)$ then $u \cdot A$ will be the where-provenance associated with $t \bullet A$ with respect to Q and I.

Our definition of $\mathsf{Where}(Q,I,t)$ makes use of a binary *merging* operation $\sqcup$ on records of sets, which is defined as follows. Suppose t_1 and t_2 are tuples of types $U \to \mathcal{P}(\mathit{FieldLoc})$ and $V \to \mathcal{P}(\mathit{FieldLoc})$, respectively. Then,

$$(t_1 \sqcup t_2) \cdot A = \begin{cases} t_1 \cdot A, & A \in U - V \\ t_1 \cdot A \cup t_2 \cdot A, & A \in U \cap V. \\ t_2 \cdot A, & A \in V - U \end{cases}$$

This means that $\sqcup$ is a partial function which returns an element of $(U \cup V) \to \mathcal{P}(\mathit{FieldLoc})$, where the set of annotations associated with a field A is the union of the respective sets of annotations in field A of t_1 and t_2, when $A \in U \cap V$, and for other fields, we just inherit the annotations from t_1 or t_2. Note that $\sqcup$ generalizes the annotation–union operation of [7, 56], which only takes as input two tuples of the same type. In order to deal with $\perp$, we define strict merge $\sqcup_{\mathrm{S}}$ and lazy merge $\sqcup_{\mathrm{L}}$ and $\bigsqcup_{\mathrm{L}}$ operations, similar to analogous operations defined

in Section 2.1.1:

$$
\begin{aligned}
X \sqcup_{\mathrm{S}} \bot = \bot \sqcup_{\mathrm{S}} X &= \bot \\
X \sqcup_{\mathrm{S}} Y &= X \sqcup Y \qquad (X \neq \bot \neq Y) \\
X \sqcup_{\mathrm{L}} \bot = \bot \sqcup_{\mathrm{L}} X &= X \\
X \sqcup_{\mathrm{L}} Y &= X \sqcup Y \qquad (X \neq \bot \neq Y) \\
\bigsqcup\nolimits_L S &= \begin{cases} t_1 \sqcup_{\mathrm{L}} \cdots \sqcup_{\mathrm{L}} t_n, & S = \{t_1, \ldots, t_n\} \\ \bot, & S = \emptyset \end{cases}
\end{aligned}
$$

Finally, we define where-provenance for SPJRU queries as follows.

Definition 4.2 (Where-provenance). Let $I \in \mathbf{R}\text{-}Inst$ be an instance, $Q : \mathbf{R} \to U$ be an SPJRU query and $t : U$ be a tuple. The where-provenance of t with respect to Q and I, denoted as $\mathsf{Where}(R, I, t)$ is as follows.

$$
\begin{aligned}
\mathsf{Where}(\{u\}, I, t) &= \begin{cases} (A : \emptyset)_{A \in U}, & \text{if } t = u \\ \bot, & \text{otherwise} \end{cases} \\
\mathsf{Where}(R, I, t) &= \begin{cases} (A : \{(R, t, A)\})_{A \in U}, & \text{if } t \in I(R) \\ \bot, & \text{otherwise} \end{cases} \\
\mathsf{Where}(\sigma_\theta(Q), I, t) &= \begin{cases} \mathsf{Where}(Q, I, t), & \text{if } \theta(t) \\ \bot, & \text{otherwise} \end{cases} \\
\mathsf{Where}(\pi_U(Q), I, t) &= \bigsqcup\nolimits_L \{\mathsf{Where}(Q, I, u)[U] \mid u[U] = t\} \\
\mathsf{Where}(\rho_{B \mapsto C}(Q), I, t) &= (A : \mathsf{Where}(Q, I, t[C \mapsto B]) \cdot (A[C \mapsto B]))_{A \in U} \\
\mathsf{Where}(Q_1 \bowtie Q_2, I, t) &= \mathsf{Where}(Q_1, I, t[U_1]) \sqcup_S \mathsf{Where}(Q_2, I, t[U_2]) \\
\mathsf{Where}(Q_1 \cup Q_2, I, t) &= \mathsf{Where}(Q_1, I, t) \sqcup_L \mathsf{Where}(Q_2, I, t)
\end{aligned}
$$

To illustrate, consider again the query Q_1 from Figure 4.2, which can be expressed in relational algebra as $\pi_{\text{name,phone}}(\sigma_{\text{type='boat'}}(\text{Agencies} \bowtie \text{ExternalTours}))$. According to Definition 4.2, the where-provenance of the second tuple (*name*: HarborCruz, *phone*: 831-3000) in the result of Q_1 is the record (name: {(Agencies, t_2, name), (ExternalTours, t_7, name)}, phone: {(ExternalTours, t_2, phone)}), where the first component tells us that the value “HarborCruz” was copied from the locations

corresponding to the name fields of input tuples t_2 and t_7, whereas the second component tells us that "831-3000" was copied from the phone field of t_2. Note a slight difference in the semantics of where-provenance in case of joins, if the query is expressed in relational algebra as opposed to SQL. In the latter case (refer to Figure 4.2) the where-provenance of "HarborCruz" is (Agencies, t_2, name), since according to the select clause of Q_1 "HarborCruz" is copied from the name value of t_2 (and not the name values in both t_2 and t_7).

Note that in contrast to lineage and why-provenance, the rule for renaming is more involved: we need to invert the renaming applied to t and ask for the provenance of the field B that was renamed to C. For example, consider an instance $R = \{t\}$ where $t = (A{:}1, B{:}2)$. The result of the query $\rho_{A \mapsto C}(R)$ is $\{t'\}$, where $t' = (C{:}1, B{:}2)$. Then:

$$\begin{aligned}\mathsf{Where}(\rho_{A\mapsto C}(R), I, t') \cdot C &= \mathsf{Where}(R, I, t'[C \mapsto A]) \cdot (C[C \mapsto A]) \\ &= \mathsf{Where}(R, I, t) \cdot A = \{(R, t, A)\}\end{aligned}$$

Another interesting observation that there is a slight mismatch between our definition of where-provenance, which is based on relational algebra, and where-provenance with SQL queries in general. The mismatch is due to the fact that there are multiple ways to define a natural join operator in SQL. For example, consider the relational algebra query

$$\pi_{\texttt{name,phone}}(\sigma_{\texttt{name='boat'}}(\texttt{Agencies} \bowtie \texttt{ExternalTours}) \,.$$

In our definition of where-provenance, the `name` field in the output will receive annotations propagated from both `Agencies.name` and `ExternalTours.name`. In SQL, it is natural to expect that this behavior could be expressed as:

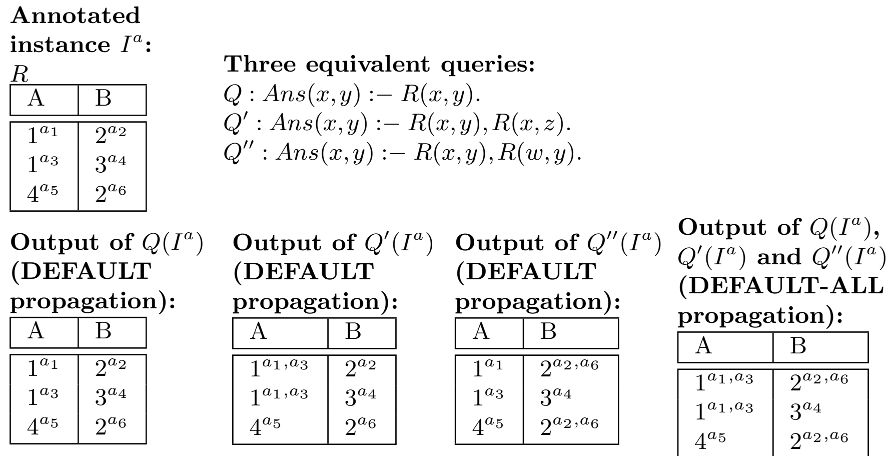
```
SELECT name, phone
FROM Agencies JOIN ExternalTours
WHERE type ='boat'
```

However, this is not valid SQL; the SQL-92 standard requires that column name references be unambiguous. Conforming SQL processors should reject queries with ambiguous field references such as `name`

above, although implementations could make a best-effort to disambiguate it as one of two different relational algebra queries, one in which the `name` field is copied from `Agencies.name` (as in Q_1 in Figure 4.2), and the other in which the field is copied from `ExternalTours.name`. These two queries are equivalent with respect to the ordinary semantics of queries, but not with respect to their where-provenance semantics. Nevertheless, a knowledgeable programmer can always avoid this ambiguity by writing a more specific query expression; in addition, the techniques described in the next section avoid this problem entirely by modifying the where-provenance semantics so that equivalent queries have equivalent where-provenance.

4.1.2 Semantics-Invariant Where-Provenance

The definition of where-provenance and the annotation propagation behavior discussed in the previous section are closely tied to the syntactic form of the query, and hence sensitive to query rewriting. The abstract example shown in Figure 4.4 (and also discussed in Section 1.1.3) illustrates three queries Q, Q' and Q'' that are equivalent, but not *annotation-equivalent*: given the same input, they may produce outputs annotated differently. For example, observe that in the output



Annotated instance I^a:

R

A	B
1^{a_1}	2^{a_2}
1^{a_3}	3^{a_4}
4^{a_5}	2^{a_6}

Three equivalent queries:

$Q : Ans(x,y) :- R(x,y).$

$Q' : Ans(x,y) :- R(x,y), R(x,z).$

$Q'' : Ans(x,y) :- R(x,y), R(w,y).$

Output of $Q(I^a)$ (DEFAULT propagation):

A	B
1^{a_1}	2^{a_2}
1^{a_3}	3^{a_4}
4^{a_5}	2^{a_6}

Output of $Q'(I^a)$ (DEFAULT propagation):

A	B
$1^{a_1,a_3}$	2^{a_2}
$1^{a_1,a_3}$	3^{a_4}
4^{a_5}	2^{a_6}

Output of $Q''(I^a)$ (DEFAULT propagation):

A	B
1^{a_1}	$2^{a_2,a_6}$
1^{a_3}	3^{a_4}
4^{a_5}	$2^{a_2,a_6}$

Output of $Q(I^a)$, $Q'(I^a)$ and $Q''(I^a)$ (DEFAULT-ALL propagation):

A	B
$1^{a_1,a_3}$	$2^{a_2,a_6}$
$1^{a_1,a_3}$	3^{a_4}
4^{a_5}	$2^{a_2,a_6}$

Fig. 4.4 Example illustrating the semantics-invariant where-provenance.

of the three queries, the locations of the output tuple $(1,2)$ are associated with different sets of annotations. Formally, two queries Q and Q' are *annotation-equivalent* if they are *annotation-contained* in each other. We say that Q is *annotation-contained* in Q' if Q is contained in Q' (under the classical interpretation of query containment) and, furthermore, for every source instance I and for every location l in $Q(I)$, we have that the set of annotations associated with location l is contained in the set of annotations associated with the corresponding location in $Q'(I)$. The problem of query containment with annotation propagation has been studied by Tan [56], who gives a necessary and sufficient condition for deciding annotation-containment [56, Theorem 3.6] and shows that the problem has the same complexity as classical query containment, which is NP-complete [56, Proposition 3.1].

The definition of where-provenance discussed in Section 4.1 corresponds to the *default* annotation propagation scheme in the DBNotes annotation management system [7, 22]. An overview of the DBNotes system is given in Section 4.2.2. In this section, we shall discuss an alternative annotation propagation scheme implemented in DBNotes, called the *default-all* scheme, that is insensitive to query rewriting.

Under the *default-all* scheme of DBNotes, a query Q propagates annotations based on where data is copied from according to *all* equivalent queries of Q. Hence, equivalent queries will always have identical annotation propagation behavior under the default-all scheme, and so this scheme can be seen as a "better" method for propagating annotations for Q. The essence of the default-all propagation scheme is to consider all rewritings of the query, some of which possibly having different where-provenance behaviors, and combine the resulting annotations:

$$\mathsf{Default_all}(Q,I,t) = \bigsqcup\nolimits_{\mathrm{L}} \{\mathsf{Where}(Q',I,t) \mid Q \equiv Q'\}$$

It was shown in [7] that although there are infinitely many rewritings Q' for a given query Q, only finitely many rewritings have different where-provenance behavior, and thus it suffices to consider only a finite set of such rewritings [7, Theorems 1, 2]. To illustrate, consider again the annotated instance I and the query Q from Figure 4.4. The result of Q on I under the default-all propagation behavior is shown at the bottom right of Figure 4.4.

4.2 Applications

4.2.1 Where-Provenance and the Annotation Placement Problem

A closely related issue to annotation propagation studied by Buneman et al. [14] is that of propagating annotations *backwards*, from the output of a query back to the source database. Specifically, they study the following *annotation placement* problem: *given a source database I, a query Q, the view $V = Q(I)$ and an annotation a placed in some location l in V, decide whether there exists a source location in I to place the annotation "a" such that "a" propagates to the minimal number of output locations, including the specified location l in V.* The annotation a is called *side-effect free* if it propagates only to the desired location l and nowhere else in V.

Buneman et al. showed a dichotomy in the complexity of the annotation placement problem. They showed that for queries involving projection and join it is NP-hard to decide whether there exists a side-effect-free annotation [14, Theorem 3.2]. Subsequently, Tan showed in [56] that this problem is in fact DP-hard for this type of queries [56, Theorem 4.1], and conjectured that the exact complexity lies in a class that is slightly above DP.[1] When the query is of bounded size, the problem of determining a minimum side-effect annotation for queries involving only project and join can be solved in polynomial time [55, Theorem 2.4.3]. On the other hand, Buneman et al. [14] showed that the annotation placement problem can be solved in polynomial time for the subclass of queries that do not simultaneously involve projection and join (i.e., SPU and SJU queries) [14, Theorem 3.4]. In the case of key-preserving SPJ queries (i.e., where a key is retained for every input relation involved in the query), Cong et al. [23] subsequently showed that the annotation placement problem coincides with the view side-effect problem and becomes tractable for this type of queries [23, Theorem 3.1].

[1] DP is the class of decision problems that are intersections of a problem in NP and a problem in co-NP.

4.2.2 Tracing Where-Provenance in DBNotes

DBNotes [7, 22] is an annotation management system for relational databases built upon the ideas in [13, 58]. In DBNotes, every attribute value of a tuple in a database (i.e., a location) may be associated with a set of zero or more annotations, and these annotations propagate along with the data, as data is being transformed through SQL queries. The query language of DBNotes, called pSQL, is essentially the fragment of SQL corresponding to conjunctive queries with union enriched with a PROPAGATE clause that specifies how annotations propagate. DBNotes provides three different schemes for propagating annotations: the *default*, the *default-all* and the *custom* propagation schemes. A pSQL query Q with *default* scheme propagates annotations based on where-provenance. Under the *default-all* scheme, Q propagates annotations based on the combined where-provenance behavior of all equivalent queries of Q, as discussed in Section 4.1.2. The *custom* scheme provides a user with the means to specify how annotations should be propagated. This scheme is useful, for example, when one is only interested in annotations provided by a certain trusted data source.

As described by Chiticariu et al. [22], pSQL also provides limited facilities for querying the annotations associated with the data, in addition to the data itself. With this feature, a user can pose queries to retrieve, for example, all tuples that are derived from a particular data source, provided that the annotations carry information about the sources. Additionally, pSQL can also be used to count the number of annotations. For example, one may ask to retrieve all tuples having more than x number of annotations on a particular field.

DBNotes makes use of the default propagation mechanism to systematically and eagerly trace the where-provenance of data. Each attribute value in a database is automatically assigned a special annotation indicating its exact location in the database. As data is being transformed through a pSQL query, these special "address" annotations automatically propagate along according to the default propagation behavior, and are accumulated in the output. When the output is materialized, each data value receives a new address annotation, in

addition to the accumulated ones, indicating its location in the newly materialized database. DBNotes can therefore *eagerly* compute the where-provenance of a data value, through various databases and chains of transformation steps, by only examining the collection of "address" annotations associated with that data value.

5

Comparing Models of Provenance

In the previous sections, we have reviewed the where-, why- and how-provenance models, as well as related techniques such as lineage and Trio. Along the way, we have developed uniform definitions of most of these models in terms of monotone, relational (SPJRU) queries. Moreover, as we have surveyed these models, we have also raised questions about relationships among them. In this section, we clarify precisely some of the relationships between various notions of provenance. Questions about some of the relationships have also been raised or stated without further elaboration in some previous studies, such as whether semiring-based how-provenance is more general than why-provenance and lineage [43], whether how-provenance is "the most general form of provenance" [43], and whether how-provenance is related to where-provenance [34]. In addition, we clarify precisely the similarities and differences between the two notions, lineage and why-provenance, which are terms that have been used interchangeably in many previous studies.

As we shall show, part of the benefit of formalizing the different models of provenance in a uniform framework is that we now have

direct proofs to many results. More specifically, we show:

(1) Lineage, why-provenance, and minimal why-provenance are instances of the semiring model, and can be derived from how-provenance (Section 5.1).
(2) Lineage can be derived from why-provenance (and hence also how-provenance), but not from minimal why-provenance (Section 5.2).
(3) Where-provenance is "contained in" lineage, why, and how-provenance in a certain sense (Section 5.3).
(4) However, the semiring model cannot exactly express where-provenance (Section 5.4).

Hence, in particular, how-provenance indeed generalizes lineage and why-provenance, and how-provenance contains where-provenance in a certain sense.

5.1 Relating Semiring-Based Techniques

Green et al. [43] stated that how-provenance generalizes why-provenance and lineage, and illustrated this by describing a particular semiring (see K_{Lin} below), but did not elaborate further on the generalization. In this section, we make precise the way how-provenance generalizes these two notions of provenance. In particular, we show that lineage and why-provenance are instances of the semiring-valued relational model. We also show that minimal why-provenance is an instance of the semiring-valued relational model.

To show these results, we recapitulate a key property of K-relational algebra, which we call the *homomorphism invariance property* (Proposition 5.1). Recall that:

Definition 5.1. Let K and K' be two semirings. A *semiring homomorphism* is a function $h\colon K \to K'$ such that:

$$h(0_K) = 0_{K'} \quad h(x +_K y) = h(x) +_{K'} h(y)$$
$$h(1_K) = 1_{K'} \quad h(x \cdot_K y) = h(x) \cdot_{K'} h(y)$$

Moreover, we can lift a homomorphism h to act on K-relations and K-instances in the obvious way, e.g., $h(r) = h \circ r$. In [43], Green et al. established the following useful property:

Proposition 5.1 (Homomorphism Invariance [43, Proposition 3.5]). Let Q be a query, K and K' be two semirings, I be a K-instance, and $h\colon K \to K'$ be a semiring homomorphism. Then $h(Q^K(I)) = Q^{K'}(h(I))$.

Finally, we recall the definition of free semiring from abstract algebra:

Definition 5.2. A semiring K is called a *free semiring on generators* X provided $X \subseteq K$ and for any semiring K' and function $g\colon X \to K'$, there exists a unique homomorphism $h\colon K \to K'$ such that $g(x) = h(x)$ for all $x \in X$.

It is a standard result in abstract algebra that $\mathbb{N}[X]$ is the free semiring on generators X. It is not difficult to show that:

Lemma 5.2. Suppose K is a semiring and I a K-instance. Suppose $\mathrm{dom}(I) \subseteq \mathrm{dom}(I_{\mathsf{How}})$. Then there is a unique homomorphism $h_K\colon K_{\mathsf{How}} \to K$ such that $h_K(I_{\mathsf{How}}) = I$.

To show that lineage is an instance of the semiring-valued relational model, consider the structure:

$$K_{\mathsf{Lin}} = (\mathcal{P}(\mathit{TupleLoc})_{\perp}, \perp, \emptyset, \cup_{\mathrm{L}}, \cup_{\mathrm{S}})$$

which is easily verified to be a semiring. Moreover, suppose we define $I_{\mathsf{Lin}} = \{\{(R,t)\} \mid R(t) \in I\}$. That is, for an ordinary instance I, we define I_{Lin} as the K_{Lin}-instance in which each tuple $R(t)$ is annotated with $\{(R,t)\}$. Using this annotated database instance, we can characterize lineage in terms of semirings:

Proposition 5.3. Let Q be a SPJRU query, I be an instance and t be a tuple. Then $\mathsf{Lin}(Q,I,t) = Q^{K_{\mathsf{Lin}}}(I_{\mathsf{Lin}})t$.

Proof. Straightforward induction on Q. $\square$

Since K_{How} is a free semiring, we know (by Lemma 5.2) that there exists a unique homomorphism $h_{\mathsf{Lin}}\colon K_{\mathsf{How}} \to K_{\mathsf{Lin}}$. Specifically:

$$h_{\mathsf{Lin}}(0) = \bot, \qquad h_{\mathsf{Lin}}(x+y) = x \cup_{\mathrm{L}} y,$$
$$h_{\mathsf{Lin}}(1) = \emptyset, \qquad h_{\mathsf{Lin}}(x \cdot y) = x \cup_{\mathrm{S}} y.$$

Hence, lineage can be directly obtained from how-provenance:

Corollary 5.4. Let Q be an SPJRU query, I be an instance and t be a tuple. Then $\mathsf{Lin}(Q,I,t) = h_{\mathsf{Lin}}(\mathsf{How}(Q,I,t))$.

Proof. Observe that $h_{\mathsf{Lin}}(I_{\mathsf{How}}) = I_{\mathsf{Lin}}$. Hence, using the homomorphism invariance property, we have:

$$\begin{aligned}\mathsf{Lin}(Q,I,t) &= Q^{K_{\mathsf{Lin}}}(I_{\mathsf{Lin}})t \\ &= Q^{K_{\mathsf{Lin}}}(h_{\mathsf{Lin}}(I_{\mathsf{How}}))t \\ &= h_{\mathsf{Lin}}(Q^{K_{\mathsf{How}}}(I_{\mathsf{How}})t) \\ &= h_{\mathsf{Lin}}(\mathsf{How}(Q,I,t)).\end{aligned}$$ $\square$

Next, for why-provenance, consider the structure:

$$K_{\mathsf{Why}} = (\mathcal{P}(\mathcal{P}(\mathit{TupleLoc})), \emptyset, \{\emptyset\}, \cup, \Cup)$$

which is again a semiring. Suppose we define $I_{\mathsf{Why}} = \{\{\{(R,t)\}\} \mid R(t) \in I\}$. That is, I_{Why} is a K_{Why}-instance obtained by annotating each tuple with the collection $\{\{(R,t)\}\}$. Using this instance, we can also characterize why-provenance in terms of semirings:

Proposition 5.5. Let Q be an SPJRU query, I an instance and t a tuple. Then, $\mathsf{Why}(Q,I,t) = Q^{K_{\mathsf{Why}}}(I_{\mathsf{Why}})t$.

Proof. Straightforward induction on Q. $\square$

Again, since $\mathsf{How}(Q,I,t) = Q^{K_{\mathsf{How}}}(I_{\mathsf{How}})(t)$ where K_{How} is a free semiring, we know (by Lemma 5.2) that there exists a unique

homomorphism $h_{\mathsf{Why}}\colon K_{\mathsf{How}} \to K_{\mathsf{Why}}$ such that $I_{\mathsf{Why}} = h(I_{\mathsf{How}})$. Specifically:

$$h_{\mathsf{Why}}(0) = \emptyset, \qquad h_{\mathsf{Why}}(x+y) = x \cup y,$$
$$h_{\mathsf{Why}}(1) = \{\emptyset\}, \qquad h_{\mathsf{Why}}(x \cdot y) = x \Cup y.$$

(Recall that $\Cup$ is an operator that takes the pairwise unions of all sets in two collections; see Definition 2.4.) Hence, why-provenance can be directly obtained from how-provenance:

Corollary 5.6. Let Q be an SPJRU query, I an instance and t a tuple. Then, $\mathsf{Why}(Q,I,t) = h_{\mathsf{Why}}(\mathsf{How}(Q,I,t))$

Proof. Observe that $h_{\mathsf{Why}}(I_{\mathsf{How}}) = I_{\mathsf{Why}}$. Hence, using the homomorphism invariance property, we have:

$$\begin{aligned}\mathsf{Why}(Q,I,t) &= Q^{K_{\mathsf{Why}}}(I_{\mathsf{Why}})t \\ &= Q^{K_{\mathsf{Why}}}(h_{\mathsf{Why}}(I_{\mathsf{How}}))t \\ &= h_{\mathsf{Why}}(Q^{K_{\mathsf{How}}}(I_{\mathsf{How}})t) \\ &= h_{\mathsf{Why}}(\mathsf{How}(Q,I,t)).\end{aligned}$$ □

Finally, the minimal witness variant of why-provenance also has a semiring characterization. The easiest way to see this is to observe that there is a homomorphism from K_{Why} to another semiring which happens to correspond to minimal why-provenance.

Let

$$\mathsf{Min}(S) = \{x \in S \mid \forall y \in S. y \subseteq x \text{ implies } y = x\}$$

That is, $\mathsf{Min}(S)$ consists of all of the minimal elements of S (with respect to the subset ordering). By definition, we know that $\mathsf{MWhy}(Q,I,t)$ is the set of all minimal elements of $\mathsf{Why}(Q,I,t)$; that is, $\mathsf{MWhy}(Q,I,t) = \mathsf{Min}(\mathsf{Why}(Q,I,t))$. Consider the semiring $\mathsf{Irr}(K_{\mathsf{Why}})$ of "irreducible" elements of K_{Why}, with $0 = 0_{\mathsf{Why}}$, $1 = 1_{\mathsf{Why}}$, and operations $x + y = \mathsf{Min}(x \cup y)$ and $x \cdot y = \mathsf{Min}(x \Cup y)$. It is not difficult to show that $\mathsf{Min}\colon K_{\mathsf{Why}} \to \mathsf{Irr}(K_{\mathsf{Why}})$ is a semiring homomorphism. Now, we can show that minimal why-provenance MWhy (which is the same as MWit according to Theorem 2.10) is also an instance of the semiring model.

Proposition 5.7. Let Q be an SPJRU query, I an instance and t a tuple. Then, $\mathsf{MWhy}(Q,I,t) = Q^{\mathsf{Irr}(K_{\mathsf{Why}})}(I_{\mathsf{Why}})t$.

Proof. Using the homomorphism invariance property, and since $\mathsf{Min}(I_{\mathsf{Why}}) = I_{\mathsf{Why}}$, we have:

$$\begin{aligned} \mathsf{MWhy}(Q,I,t) &= \mathsf{Min}(\mathsf{Why}(Q,I,t)) \\ &= \mathsf{Min}(Q^{K_{\mathsf{Why}}}(I_{\mathsf{Why}})t) \\ &= Q^{\mathsf{Irr}(K_{\mathsf{Why}})}(\mathsf{Min}(I_{\mathsf{Why}}))t \\ &= Q^{\mathsf{Irr}(K_{\mathsf{Why}})}(I_{\mathsf{Why}})t. \end{aligned}$$ □

To conclude, we can obtain minimal why-provenance directly from how-provenance by composing h_{Why} and Min:

Corollary 5.8. Let Q be an SPJRU query, I an instance and t a tuple. Then, $\mathsf{MWhy}(Q,I,t) = \mathsf{Min}(h_{\mathsf{Why}}(\mathsf{How}(Q,I,t)))$

Proof. By the definition of $\mathsf{MWhy}(Q,I,t)$ and Corollary 5.6 we have:

$$\mathsf{MWhy}(Q,I,t) = \mathsf{Min}(\mathsf{Why}(Q,I,t)) = \mathsf{Min}(h_{\mathsf{Why}}(\mathsf{How}(Q,I,t))).$$ □

5.2 Relating Lineage, Why-Provenance and Minimal Why-Provenance

We now consider the relationships between lineage, why-provenance, and minimal why-provenance. We already showed above (Proposition 5.7) that minimal why-provenance can be obtained from why-provenance (and hence how-provenance) via a semiring homomorphism. We shall now show that lineage can be obtained from why-provenance, but not from minimal why-provenance.

At first sight, the lineage and why-provenance of a tuple are two different kinds of objects: one is a set of tuples, while the other is a set of sets of tuples. However, it turns out that they are related in a natural way: lineage "summarizes" why-provenance in the sense that the lineage collects all tuple locations mentioned in any witness

in the why-provenance. The argument makes use of the semiring-based characterizations.

Proposition 5.9. Let Q be an SPJRU query, I an instance and t a tuple. Then, $\mathsf{Lin}(Q,I,t) = \bigcup_L \mathsf{Why}(Q,I,t)$.

Proof. It is straightforward to verify that $\bigcup_{\mathrm{L}}$ is a semiring homomorphism (recall Definition 5.1) from K_{Why} to K_{Lin}, and that $\bigcup_{\mathrm{L}} I_{\mathsf{Why}} = I_{\mathsf{Lin}}$. Hence (using the homomorphism invariance property) we conclude:

$$\begin{aligned}\mathsf{Lin}(Q,I,t) &= Q^{K_{\mathsf{Lin}}}(I_{\mathsf{Lin}})t \\ &= Q^{K_{\mathsf{Lin}}}(\textstyle\bigcup_L (I_{\mathsf{Why}}))t \\ &= \textstyle\bigcup_L (Q^{K_{\mathsf{Why}}}(I_{\mathsf{Why}})t) = \bigcup_L \mathsf{Why}(Q,I,t).\end{aligned}$$ □

However, lineage *cannot* be computed from the minimal witness variant of why-provenance. The reason is that two equivalent queries may have different lineage. On the other hand, minimal why-provenance is invariant under query equivalence (Corollary 2.11).

Corollary 5.10. There is no function $h\colon \mathsf{Irr}(K_{\mathsf{Why}}) \to K_{\mathsf{Lin}}$ such that $\mathsf{Lin}(Q,I,t) = h(\mathsf{MWhy}(Q,I,t)))$ for every query Q, instance I, and tuple t.

5.3 Where-Provenance is "Contained" in Why-Provenance

Another natural question is how where-provenance is related to the other techniques. It seems reasonable to expect that if the output data in tuple t is "copied from" some tuple (S,u) in the input, then (S,u) is present in the lineage of t (which implies that (S,u) is in one of the elements of $\mathsf{Why}(Q,I,t)$ by Proposition 5.9). In fact, we can show:

Proposition 5.11. Let Q be an SPJRU query, I be an instance, and t be a U-tuple in $Q(I)$. Let $A \in U$, and suppose $(S,u,B) \in \mathsf{Where}(Q,I,t) \cdot A$. Then $(S,u) \in \mathsf{Lin}(Q,I,t)$.

Proof. We prove that $(S,u) \in \mathsf{Lin}(Q,I,t)$ by induction on Q. The case of constant queries $\{t\}$ is vacuous, so immediate.

Case 1. Suppose $Q = R$. Then $\mathsf{Where}(R,I,t) = (A\colon \{(R,t,A)\})_{A \in U}$. Hence $(S,u,B) = (R,t,A)$ so $(S,u) = (R,t) \in \{(R,t)\} = \mathsf{Lin}(R,I,t)$, as desired.

Case 2. Suppose $Q = \sigma_\theta(Q')$. Then $(S,u,B) \in \mathsf{Where}(Q',I,t) \cdot A$ and $\theta(t)$ holds, so by induction we have $(S,u) \in \mathsf{Lin}(Q',I,t) = \mathsf{Lin}(\sigma_\theta(Q'),I,t)$.

Case 3. Suppose $Q = \pi_U(Q')$. Then $(S,u,B) \in \mathsf{Where}(Q',I,u) \cdot A$ for some u such that $u[U] = t$. Consequently, by induction we have $(S,u) \in \mathsf{Lin}(Q',I,u) \subseteq \mathsf{Lin}(\pi_U(Q'),I,t)$.

Case 4. Suppose $Q = \rho_{C \mapsto D}(Q')$. Then $(S,u,B) \in \mathsf{Where}(Q',I, t[D \mapsto C]) \cdot (A[D \mapsto C])$. Consequently, by induction we have $(S,u) \in \mathsf{Lin}(Q',I,t[D \mapsto C]) \subseteq \mathsf{Lin}(\rho_{C \mapsto D}(Q'),I,t)$.

Case 5. Suppose $Q = Q_1 \bowtie Q_2$. Then $\mathsf{Where}(Q_1 \bowtie Q_2,I,t) = \mathsf{Where}(Q_1,I,t[U_1]) \sqcup_{\mathrm{S}} \mathsf{Where}(Q_2,I,t[U_2])$. Since we assume $t \in (Q_1 \bowtie Q_2)(I)$, both sides must be defined. Suppose $(S,u,B) \in \mathsf{Where}(Q_1,I,t) \cdot A$; the case where $(S,u,B) \in \mathsf{Where}(Q_2,I,t) \cdot A$ is symmetric. Then by induction we have $(S,u) \in \mathsf{Lin}(Q_1,I,t[U_1]) \subseteq \mathsf{Lin}(Q_1,I,t[U_1]) \cup_S \mathsf{Lin}(Q_2,I,t[U_2]) = \mathsf{Lin}(Q_1 \bowtie Q_2,I,t)$, where the latter equality again relies on the fact that $t \in (Q_1 \bowtie Q_2)(I)$.

Case 6. Suppose $Q = Q_1 \cup Q_2$. Then $\mathsf{Where}(Q_1 \cup Q_2,I,t) = \mathsf{Where}(Q_1,I,t) \sqcup_{\mathrm{L}} \mathsf{Where}(Q_2,I,t)$. Suppose $(S,u,B) \in \mathsf{Where}(Q_1,I,t) \cdot A$; the case where $(S,u,B) \in \mathsf{Where}(Q_2,I,t) \cdot A$ is symmetric. Then by induction we have $(S,u) \in \mathsf{Lin}(Q_1, I,t) \subseteq \mathsf{Lin}(Q_1,I,t) \cup_{\mathrm{L}} \mathsf{Lin}(Q_2,I,t) = \mathsf{Lin}(Q_1 \cup Q_2,I,t)$. □

Since lineage contains all of the tuples mentioned in either the why- or how-provenance, as discussed earlier, it should be clear that all of the locations mentioned in the where-provenance of a tuple (S,u) are also present in the why- or how-provenance of (S,u).

5.4 Is Where-Provenance an Instance of How-Provenance?

We just showed that the lineage (or why- or how-provenance) of a tuple contains all of the input sources that can appear in its

where-provenance. But, can where-provenance be defined as an instance of the semiring model?

We show that the answer is no. Intuitively, the reason that where-provenance cannot be cast straightforwardly into the semiring-valued model is because semiring valued models are tuple-based, whereas annotations on where-provenance are of a finer granularity (i.e., attribute based). This intuition is a little naive since our approach to defining where-provenance is also "tuple-based" in a sense: we view the field annotations as being collected into a tuple-based record. What makes our counterexample work is that renaming (and similarly projection) need to manipulate the "field-level" parts of an annotation, to which the semiring model does not have access.

Proposition 5.12. There exist SPJRU queries Q_1 and Q_2, instance I and tuple t such that for any K and K-instance J, we have ${Q_1}^K(J)t = {Q_2}^K(J)t$, yet $\mathsf{Where}(Q_1, I, t) \neq \mathsf{Where}(Q_2, I, t)$.

Proof. A simple example is shown below, where R is a binary relation with attributes A and B.

$$Q_1 = R, \quad Q_2 = \rho_{C \mapsto A}(\rho_{A \mapsto B}(\rho_{B \mapsto C}(R)))$$

which when run on input $I = \{R(A{:}1, B{:}1)\}$ yields $Q_1(I) = \{(A{:}1, \mathrm{B}{:}1)\} = Q_2(I)$. Take I as above and $t = (A{:}1, B{:}1)$. Then for any K and K-instance J, we have

$$\begin{aligned} {Q_1}^K(J)t &= R^K(J)t \\ &= R^K(J)(t[A \mapsto C][B \mapsto A][C \mapsto B]) \\ &= (\rho_{C \mapsto A}(\rho_{A \mapsto B}(\rho_{B \mapsto C}(R))))^K(J)t \\ &= {Q_2}^K(J)t \end{aligned}$$

because $t[A \mapsto C][B \mapsto A][C \mapsto B] = t$. However,

$$\begin{aligned} \mathsf{Where}(Q_1, I, t) &= (A{:}\{(R, t, A)\}, B{:}\{(R, t, B)\}) \\ \mathsf{Where}(Q_2, I, t) &= (A{:}\{(R, t, B)\}, B{:}\{(R, t, A)\}) . \end{aligned}$$ □

This implies that where-provenance cannot be defined in terms of the semiring model, because in the above example we have

${Q_1}^K(J)t = {Q_2}^K(J)t$ for any K and J, whereas $\mathsf{Where}(Q_1, I, t) \neq \mathsf{Where}(Q_2, I, t)$. Thus, there can exist no K-instance J corresponding to I such that we can extract the where-provenance of t from ${Q_1}^K(J)t$, since another query with different where-provenance produces the same result in the K-valued semantics.

On the other hand, our formulation of where-provenance is similar to the semiring-valued semantics in a number of ways. Thus, there may be a natural generalization of the two approaches. We leave investigating this possibility to future work.

6

Conclusions

We conclude this article by surveying related research on provenance that we have not covered in detail, and discussing some of the remaining research challenges involving provenance and annotation management in databases. Provenance is also an active topic for research in the scientific computation and workflow management system communities. Although there is some overlap with database and data provenance research, we will restrict attention to work that focuses on provenance within databases, and refer readers interested in workflow provenance to recent surveys [8, 53] and a recent tutorial [29]. A tutorial on database provenance (including additional discussion of related work) can be found in [15], and a recent issue of IEEE Data Engineering Bulletin [12] presents high-level overviews of much of the recent work on data provenance we discuss below.

The lineage, where, why and how models. The lineage model introduced by Cui et al. [27] seems to be the first model to be defined for relational queries based on a semantic criterion. Subsequently, the (minimal) why-provenance model of Buneman et al. [13] was developed based on the idea of (minimal) witnesses. The how-provenance and semiring-valued relational models were introduced by Green et al. [43],

along with a high-level discussion of the possibility of extracting lineage and why-provenance from how-provenance. The semiring characterizations for lineage, why- and minimal why-provenance were first presented in (terse) detail in [10], following discussions including the authors, Green, Vansummeren and Tannen. In this article, we have described these characterizations in a more expository fashion, in order to show clearly how the original ideas are captured by the semiring model and make these concepts more broadly accessible to a general audience.

Techniques for computing lineage lazily were also explored by Cui et al. However, query optimization in the presence of eager provenance-propagation is an important problem which has only begun to be studied. Tan [56] investigated query containment in the presence of annotation propagation based on where-provenance. More recently, Green [41] has studied query containment and equivalence for the semiring-valued model, including the semiring characterizations of lineage, how-, why- and minimal why-provenance we used here. Green's results also establish relationships among the containment and equivalence problems for the different models, providing further insight into their relative expressiveness.

Beyond why, how and where. In this article, we have restricted our attention to the well-understood why-, how-, and where-provenance models, a few closely related models such as lineage and Trio, and their main applications. However, there has also been work on provenance models that do not appear to fit the why, how, or where models. Some early approaches to provenance and lineage resist such classification, including Wang and Madnick's Polygen model [58] and Woodruff and Stonebraker's work [61]. Furthermore, the full ULDB model used in Trio [5] does not seem to be a direct instance of why-, how-, or where-provenance due to its use of tuple identifiers and tuple alternatives.

There is also complementary work on *annotation*, whose goal is providing high expressive query languages for annotated databases. The MONDRIAN system of Geerts et al. [37, 38] extends the where-provenance approach used in DBNotes [7, 22] by allowing annotations to be placed on sets of attribute values of a tuple. In Mondrian, annotated databases are abstractly represented using *colors*. Different

colors applied to a set of attribute values signify different annotations. Geerts et al. [38] describe a *color algebra* that allows one to query both data and annotations, and prove that this algebra is both complete (i.e., it can express all possible queries over annotated databases) and minimal (i.e., none of its operators can be simulated using the others). Geerts and Van den Bussche [36] subsequently proved expressive completeness for the color algebra. Srivastava and Velegrakis [54] investigated techniques for associating annotations with arbitrary subsets of the locations in the input, defined using queries. They report improvements in storage overhead of annotations and query execution time with respect to the DBNotes and MONDRIAN approaches. Finally, Eltabakh et al. [31] proposed the BDBMS system where annotations can be placed on rectangular regions consisting of adjacent columns of adjacent rows in a table, and investigated various storage schemes for such annotations.

Another approach, called *dependency provenance*, has been introduced by Cheney et al. [20] and is based on the idea of tracking the parts of the input that parts of the output depend on. We discuss this approach further below. The variety of approaches that go beyond "why, how or where" illustrates that we are probably far from having fully explored the design space of provenance-tracking or annotation-management techniques.

Other data models and query languages. We have also restricted our attention to monotone relational queries. Some work has been done on provenance or lineage for other query languages. Both lineage and Trio lineage have been defined for non-monotone operations such as aggregation and negation operations. However, it is non-obvious how to extend these definitions to why- or how-provenance while preserving their nice properties (e.g., Theorem 2.7 or Corollary 2.11). Recently, Geerts and Poggi [39] studied expressive completeness results for the K-relational model, and along the way extended it with negation and duplicate elimination operations. However, we believe further work is needed to understand the principles of provenance in the presence of negation and aggregation.

Other work has considered provenance in settings besides the flat relational model. For example, as discussed in Sections 2 and 4, the

initial work on why- and where-provenance of Buneman et al. [13] was based on a deterministic tree model. More recent work by Buneman et al. [11] has extended where-provenance to the nested relational calculus (NRC). In this approach, as in Polygen and the where-provenance model of Buneman et al. [14], the where-provenance of an output part is a part of the input from which the output was "copied". Each part of the nested relational value (data value, record or collection) carries an annotation; the annotation semantics can be used to define provenance by observing how the annotations propagate from a distinctly annotated input. The main results of [11] include identifying suitable semantic invariants satisfied by where-provenance and proving an expressive completeness result stating that these invariants *exactly* characterize queries with where-provenance semantics. The model and expressive completeness results are also adapted to an NRC-based update language; in the context of provenance, update languages are distinctly more expressive than query languages.

The *dependency provenance* model proposed by Cheney et al. [20] was defined in terms of NRC as well. In this approach each part of the database carries a set of annotations, and an annotation–propagation semantics is defined such that the annotations on a part of the output highlight parts of the input on which the given output "depends". Cheney et al. develop a formal characterization of this dependence property inspired by techniques in information flow security and program slicing, show that obtaining "minimal" dependence information is undecidable for full NRC, and show that the annotation–propagation semantics is a safe approximation. Cheney [18] further discusses the relationship between this form of provenance and program slicing.

Foster et al. [34] extended the semiring model (and hence also why- and how-provenance) to nested relations. Their approach is based on the observation that semiring-annotated relations can be viewed as a monadic collection type, generalizing sets and bags (but not lists). Furthermore, they add primitives for constructing (unordered) XML trees and structural recursion over trees, and show how to translate XQuery over unordered trees to this language, yielding a provenance semantics for unordered XML.

These NRC-based approaches provide smoother handling of features such as aggregation and grouping, but incorporating techniques based on NRC into standard relational or XML databases may be nontrivial. In a recent invited paper, Cheney [19] outlined an approach to adapting various forms of provenance to XQuery; however, this paper only scratches the surface. Thus, more work is needed to adapt these techniques for use in practice.

Update languages and curated databases. Provenance has also been studied in the context of update languages. An important motivation for modeling provenance in update languages is to support provenance tracking for *curated databases.* Curated databases consist largely of data manually added or corrected by experts, and are especially popular in bioinformatics; the scientific value of such databases relies on provenance records. Buneman et al. [9] introduced a where-provenance model for manual update operations to deterministic trees; Buneman et al. [11] subsequently generalized this idea to an update language for nested relational data. Further work is needed to better understand the design space of provenance techniques for updates: for example, can why- or how-provenance also be adapted to update languages? Also, further work on implementing these techniques within practical systems used by data curators is essential for establishing their feasibility and effectiveness. Buneman et al. [10] give an overview of research issues involving curated databases, including provenance, annotation, and archiving.

Querying provenance. Finally, for many techniques there has been little work on querying the provenance information itself alongside ordinary data, or presenting (often large amounts of) provenance information in a way casual users find useful. A few exceptions include DBNotes [22], Mondrian [37, 38], and Srivastava and Velegrakis [54], which provide capabilities for querying annotations and hence provenance, assuming that annotations carry information about provenance. DBNotes can also display visual diagrams of the where-provenance of data that has been repeatedly copied from one database to another through a sequence of queries. Other systems that provide facilities for visualizing provenance include the WHIPS data warehouse system [25], which

displays visual diagrams of the lineage of view tuples at the granularity of individual relational operators, and the SPIDER debugging system for schema mappings [3], which shows graphical representations of routes. The problem of reducing the amount of provenance information stored, and therefore shown to a user, has been recently tackled. Techniques proposed include compressing [9, 16] and approximating [20, 50] provenance.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. Part of the work was done while Chiticariu was a Ph.D. candidate at University of California, Santa Cruz. Chiticariu and Tan are supported in part by NSF CAREER Award IIS-0347065 and NSF grant IIS-0430994. Cheney has been supported by UK EPSRC grants EP/F028288/1 and GR/S63205/01, by a Royal Society University Research Fellowship, and by the UK eScience Institute Theme Program on "Principles of Provenance."

References

[1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison Wesley Publishing Co, 1995.

[2] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom, "Trio: A system for data, uncertainty, and lineage," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp. 1151–1154, 2006. (Demonstration Track).

[3] B. Alexe, L. Chiticariu, and W.-C. Tan, "Spider: A schema mapping debugger," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp. 1179–1182, 2006. (Demonstration Track).

[4] F. Bancilhon and N. Spyratos, "Update semantics of relational views," *ACM Transactions on Database Systems (TODS)*, vol. 6, pp. 557–575, 1981.

[5] O. Benjelloun, A. D. Sarma, A. Halevy, M. Theobald, and J. Widom, "Databases with uncertainty and lineage," *The VLDB Journal*, vol. 17, pp. 243–264, 2008.

[6] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom, "Uldbs: Databases with uncertainty and lineage," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp. 953–964, 2006.

[7] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, "An annotation management system for relational databases," *The VLDB Journal*, vol. 14, pp. 373–396, 2005. (A preliminary version of this paper appeared in the VLDB 2004 proceedings).

[8] R. Bose and J. Frew, "Lineage retrieval for scientific data processing: A survey," *ACM Computing Survey*, vol. 37, pp. 1–28, 2005.

[9] P. Buneman, A. Chapman, and J. Cheney, "Provenance management in curated databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 539–550, 2006.

[10] P. Buneman, J. Cheney, W.-C. Tan, and S. Vansummeren, "Curated databases," in *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pp. 1–12, 2008.

[11] P. Buneman, J. Cheney, and S. Vansummeren, "On the expressiveness of implicit provenance in query and update languages," *ACM Transactions Database Systems*, vol. 33, pp. 1–47, 2008.

[12] P. Buneman and D. Suciu (eds.), "Special issue on data provenance," *IEEE Data Engineering Bulletin*, vol. 30, 2007.

[13] P. Buneman, S. Khanna, and W.-C. Tan, "Why and where: A characterization of data provenance," in *Proceedings of the International Conference on Database Theory (ICDT)*, pp. 316–330, 2001.

[14] P. Buneman, S. Khanna, and W.-C. Tan, "On propagation of deletions and annotations through views," in *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pp. 150–158, 2002.

[15] P. Buneman and W.-C. Tan, "Provenance in databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 1171–1173, 2007.

[16] A. P. Chapman, H. V. Jagadish, and P. Ramanan, "Efficient provenance storage," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 993–1006, 2008.

[17] S. Chaudhuri and U. Dayal, "Data warehousing and olap for decision support," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 507–508, 1997.

[18] J. Cheney, "Program slicing and data provenance," *IEEE Data Engineering Bulletin*, pp. 22–28, December 2007.

[19] J. Cheney, "Provenance, XML and the Scientific Web," in *ACM SIGPLAN Workshop on Programming Language Technology and XML (PLAN-X 2009)*, 2009. Invited paper.

[20] J. Cheney, A. Ahmed, and U. A. Acar, "Provenance as dependency analysis," in *Proceedings of the International Workshop on Database and Programming Languages (DBPL)*, pp. 138–152, 2007.

[21] L. Chiticariu and W.-C. Tan, "Debugging schema mappings with routes," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp. 79–90, 2006.

[22] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, "Dbnotes: A post-it system for relational databases based on provenance," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 942–944, 2005. (Demonstration Track).

[23] G. Cong, W. Fan, and F. Geerts, "Annotation propagation revisited for key preserving views," in *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, pp. 632–641, 2006.

[24] S. S. Cosmadakis and C. H. Papadimitriou, "Updates of relational views," *Journal of the Association for Computing Machinery (JACM)*, vol. 31, pp. 742–760, 1984.

[25] Y. Cui and J. Widom, "Lineage tracing in a data warehousing system," in *Proceedings of the International Conference on Data Engineering (ICDE)*, pp. 683–684, 2000. (Demonstration Track).
[26] Y. Cui and J. Widom, "Run-time translation of view tuple deletions using data lineage," Technical Report, Stanford University, 2001.
[27] Y. Cui, J. Widom, and J. L. Wiener, "Tracing the lineage of view data in a warehousing environment," *ACM Transactions on Database Systems (TODS)*, vol. 25, pp. 179–227, 2000.
[28] N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," *The VLDB Journal*, vol. 14, pp. 523–544, 2007.
[29] S. B. Davidson and J. Freire, "Provenance and scientific workflows: Challenges and opportunities," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 1345–1350, 2008.
[30] U. Dayal and P. A. Bernstein, "On the correct translation of update operations on relational views," *ACM Transactions on Database Systems (TODS)*, vol. 7, pp. 381–416, 1982.
[31] M. Y. Eltabakh, W. G. Aref, A. K. Elmagarmid, M. Ouzzani, and Y. N. Silva, "Supporting annotations on relations," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pp. 379–390, 2009.
[32] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa, "Data exchange: Semantics and query answering," *Theoretical Computer Science (TCS)*, vol. 336, pp. 89–124, 2005.
[33] First Workshop on Theory and Practice of Provenance (TaPP). http://www.usenix.org/event/tapp09, February 2009.
[34] J. N. Foster, T. J. Green, and V. Tannen, "Annotated XML: Queries and provenance," in *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pp. 271–280, 2008.
[35] J. Freire, D. Koop, and L. Moreau, eds., *Provenance and Annotation of Data and Processes: Second International Provenance and Annotation Workshop (IPAW 2008), Revised Selected Papers*, number 5272 in LNCS. Springer, 2008.
[36] F. Geerts and J. V. den Bussche, "Relational completeness of query languages for annotated databases," in *Proceedings of the International Workshop on Database and Programming Languages (DBPL)*, pp. 127–137, 2007.
[37] F. Geerts, A. Kementsietsidis, and D. Milano, "imondrian: A visual tool to annotate and query scientific databases," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pp. 1168–1171, 2006. (Demonstration Track).
[38] F. Geerts, A. Kementsietsidis, and D. Milano, "Mondrian: Annotating and querying databases through colors and blocks," in *Proceedings of the International Conference on Data Engineering (ICDE)*, p. 82, 2006.
[39] F. Geerts and A. Poggi, "On BP-complete query languages on k-relations," in *Workshop on Logic in Databases (LID)*, 2008. Available online: http://conferenze.dei.polimi.it/lid2008/LID08geerts.pdf.
[40] T. J. Green, Personal communication, August 2008.
[41] T. J. Green, "Containment of conjunctive queries on annotated relations," in *Proceedings of the International Conference on Database Theory (ICDT)*, pp. 296–309, 2009.

[42] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen, "Update exchange with mappings and provenance," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp. 675–686, 2007.

[43] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pp. 675–686, 2007.

[44] T. J. Green, G. Karvounarakis, N. E. Taylor, O. Biton, Z. G. Ives, and V. Tannen, "Orchestra: Facilitating collaborative data sharing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 1131–1133, 2007. (Demonstration Track).

[45] A. Halevy, M. Franklin, and D. Maier, "Principles of dataspace systems," in *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pp. 1–9, 2006.

[46] A. M. Keller, "Choosing a view update translator by dialog at view definition time," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp. 467–474, 1986.

[47] P. G. Kolaitis, "Schema mappings, data exchange and metadata management," in *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pp. 61–75, 2005.

[48] M. Lenzerini, "Data integration: A theoretical perspective," in *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pp. 233–246, 2002.

[49] C. A. Lynch, "When documents deceive: Trust and provenance as new factors for information retrieval in a tangled web," *Journal of the American Society for Information Science and Technology*, vol. 52, pp. 12–17, 2001.

[50] C. Ré and D. Suciu, "Approximate lineage for probabilistic databases," *Proceedings of the VLDB Endowment*, vol. 1, pp. 797–808, 2008.

[51] A. D. Sarma, M. Theobald, and J. Widom, "Exploiting lineage for confidence computation in uncertain and probabilistic databases," Technical Report 2007-15, Stanford InfoLab, March 2007.

[52] A. D. Sarma, M. Theobald, and J. Widom, "Exploiting lineage for confidence computation in uncertain and probabilistic databases," in *Proceedings of the International Conference on Data Engineering (ICDE)*, pp. 1023–1032, 2008.

[53] Y. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *SIGMOD Record*, vol. 34, pp. 31–36, 2005.

[54] D. Srivastava and Y. Velegrakis, "Intensional associations between data and metadata," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 401–412, 2007.

[55] W.-C. Tan, "Data annotations, provenance and archiving," PhD thesis, University of Pennsylvania, 2002.

[56] W.-C. Tan, "Containment of relational queries with annotation propagation," in *Proceedings of the International Workshop on Database and Programming Languages (DBPL)*, pp. 37–53, 2003.

[57] W.-C. Tan, "Provenance in databases: Past, current and future," *IEEE Data Engineering Bulletin*, vol. 30, pp. 3–12, 2007.

[58] Y. R. Wang and S. E. Madnick, "A polygen model for heterogeneous database systems: The source tagging perspective," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp. 519–538, 1990.

[59] J. Widom, "Trio: A system for integrated management of data, accuracy and lineage," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, pp. 262–276, 2005.

[60] J. L. Wiener, H. Gupta, W. Labio, Y. Zhuge, and H. Garcia-Molina, "The whips prototype for data warehouse creation and maintenance," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 557–559, 1997. (Demonstration Track).

[61] A. Woodruff and M. Stonebraker, "Supporting fine-grained data lineage in a database visualization environment," in *Proceedings of the International Conference on Data Engineering (ICDE)*, pp. 91–102, 1997.