Foundations and Trends® in Databases

Data Provenance

Origins, Applications, Algorithms, and Models

Suggested Citation: Boris Glavic (2021), “Data Provenance”, Foundations and Trends® in Databases: Vol. 9, No. 3-4, pp 1–232. DOI: 10.1561/1900000068.

Boris Glavic
Illinois Institute of Technology
bglavic@iit.edu

now
the essence of knowledge
Boston — Delft

Contents

Data Provenance

Boris Glavic[1]

[1] *Illinois Institute of Technology; bglavic@iit.edu*

ABSTRACT

Data provenance has evolved from a niche topic to a mainstream area of research in databases and other research communities. This article gives a comprehensive introduction to data provenance. The main focus is on provenance in the context of databases. However, it will be insightful to also consider connections to related research in programming languages, software engineering, semantic web, formal logic, and other communities. The target audience are researchers and practitioners that want to gain a solid understanding of data provenance and the state-of-the-art in this research area. The article only assumes that the reader has a basic understanding of database concepts, but not necessarily any prior exposure to provenance.

Boris Glavic (2021), "Data Provenance", Foundations and Trends® in Databases: Vol. 9, No. 3-4, pp 1–232. DOI: 10.1561/1900000068.

1

Introduction

The term **provenance** is used in the art world to describe a record of the history of ownership of a piece of art. This term has been adapted by the database community to describe a record of the origin of a piece of data. Data provenance has emerged as a research topic in database community in the late 1990's with some isolated earlier work, e.g., Stonebraker *et al.*, 1993. However, as we will discuss in detail in Chapter 5 other research communities have studied concepts that are closely related to data provenance much earlier. Data provenance, by explaining how the result of an operation was derived from its inputs, has proven to be a useful tool that is applicable in a wide variety of applications including debugging transformations (queries, updates, transactions, . . .) and data, to assess the trustworthiness of data, to aid users in understanding data-intensive processes, to speed-up incremental maintenance of query results, for explaining surprising outcomes, and for reasoning about hypothetical changes to inputs and results of operations. The purpose of this article is to give a comprehensive introduction to data provenance concepts, algorithms, and methodology developed by the database community in the last few decades. The indented audience are researcher and practitioners unfamiliar with the topic that want to develop a basic understanding of provenance techniques and the state-of-the-art in the field as well as researchers with some prior experience in provenance that want to broaden their horizon. While also providing a collection of relevant literature references, this article's main objective is to introduce the reader to the formalisms, algorithms, and system's developments in this fascinating field. To be able to cover topics in sufficient depth, we will focus on work on provenance in databases and closely related areas. Provenance for workflow systems, operating systems, general purpose programming languages, and other areas that are not related to databases will not be discussed in depth. That being said, we will point the reader to important work from outside of the database community where appropriate.

1.1 What is Data Provenance?

Following common terminology we will use the term **data item** to refer to a piece of data, e.g., a relation, a tuple, or JSON document. Data items may be inputs and/or outputs of **transformations**, e.g., queries, updates, transactions, application programs. The provenance of a data item provides a record of how the data item was derived from other data items by a set of transformations. We distinguish between **data dependencies** which record that a data item was produced from / depends on another data item, e.g., a tuple in the result of a query was derived from an input tuple, and **transformation dependencies** which record that a data item was directly (or indirectly) produced by a particular transformation. Some provenance models are only concerned with data dependencies or transformation dependencies while others support both types of dependencies. Additional metadata about the execution of a transformation, e.g., the user that executed a transformation or the execution environment of a transformation, are sometimes also considered to be data provenance. However, in this article the main focus is on data and transformation dependencies.

Data and transformation dependencies can be modeled at different levels of **granularity**. For instance, we may track the data dependencies for a query result at the level of attribute values, tuples, or whole relations. The same applies to transformation dependencies, e.g., we

Query result

name	id
Aishe	r_1
Peter	r_2
Astrid	r_3

$Q_{coffee-drinker}$

```
SELECT name FROM student WHERE daily-coffee > 1
UNION
SELECT name FROM teacher WHERE daily-coffee > 1
```

student

name	GPA	daily-coffee	id
Aishe	3.5	2	s_1
James	2.4	0	s_2
Peter	3.6	3	s_3

teacher

name	salary	daily-coffee	id
Alice	30,000	1	t_1
Peter	131,000	2	t_2
Astrid	140,000	3	t_3

Figure 1.1: Example database

may track them at the level of queries or individual operators within a query.

1.1.1 An Example

Consider the database shown in Figure 1.1. The query shown in this figure returns the names of students and teachers (relations student and teacher) who consume more than one cup of coffee per day. Note that this query does not return duplicates (it uses SQL's `UNION` which eliminates duplicates). Let us reason about data dependencies at tuple granularity. That is, we want to know which of the input tuples were used to derive an output tuple. Consider the result $r_1 = (Aishe)$. Student Aishe is in the result, because she drinks more than one cup of coffee per day. The input tuple that justifies the existence of r_1 in the query result is $s_1 = (Aishe, 3.5, 2)$. All other input tuples have no bearing on whether Aishe is in the result or not. Thus, r_1 is only data-dependent on s_1, but no other input tuple. Since Aishe is a student and there is no teacher called Aishe (let alone a teacher drinking a sufficient amount of coffee), the second part of the query that accesses relation teacher is not needed for producing result tuple r_1. Thus, r_1 is only transformation-dependent on the part of the query highlighted in red.[1] As another example, consider the second result tuple ($r_2 = (Peter)$). There exist two coffee drinkers named Peter. One is a student (tuple s_3) and one is a teacher (tuple t_2). The result tuple r_2 is data-dependent on both of these inputs. Modeling provenance as data dependencies only may not provide us with enough information for all use-cases of data provenance. For example, provenance can help us to determine the effect of deleting a tuple, say s_3, from the input. We can use data dependencies to determine the subset of the output tuples that may be affected by the deletion. Only tuples that are data-dependent on deleted inputs may be affected. However, we need additional information about *how* input tuples were combined by the query to know with certainty whether an output will be deleted or not. For instance, to know whether the deletion of s_3 will cause r_2 to be deleted from the query result, it is not enough to just know the data dependencies of r_2 (which are s_3 and t_2). Additionally, we need to know that as long as one of s_3 or t_3 is in the input, then r_2 will be in the query result. So far we have reasoned intuitively about dependencies. To be able to determine dependencies automatically, we need a formal model of provenance. In Chapter 2 we will review such models and discuss which models provide sufficient information to support particular use-cases in Chapter 3.

1.2 Why Should I Care?

Data provenance has been applied for a diverse set of use cases, many of which we will discuss in detail in Chapter 3. Here we just provide a brief overview of some common use cases.

1.2.1 Error Diagnosis and Debugging

By tracking which input data and parts of a transformation are responsible for producing a suspicious output, provenance information

[1] We may argue about whether the union operation should be considered as relevant or not. For now let us just assume that it is relevant. We will discuss transformation dependencies in detail in Section 2.6.

can be used to narrow down the location of an error. Intuitively, data that was not needed to produce a query result (on which the result is not data dependent on) cannot effect the correctness of the query result. The same applies for transformation dependencies: any part of the transformation that is not a transformation dependency of the result cannot have affected the result.

Example 1 (Debugging). For instance, continuing with the example from Figure 1.1, if the user issuing the query knows that Aishe (tuple r_1) is not a coffee drinker and, thus, should not be in the result of query $Q_{coffee-drinker}$, then the user can restrict their search for errors in the input data to the data dependencies of r_1. In this case the user only needs to inspect s_1 to determine that Aishe's daily-coffee value should 0 instead of having to inspect all six input tuples. In addition to errors in the input, errors in the query result may be caused by bugs in the query. Analog to how data dependencies are used to trace errors to the input data, transformation dependencies can be used to focus attention on the parts of the query (transformation) that could have caused the error. For instance, in our example, only the highlighted part of $Q_{coffee-drinker}$ is responsible for producing result tuple r_1. Obviously, the use of provenance is overkill for this toy example. However, for realistically sized data sets and more complex queries, data and transformation dependencies can significantly improve a user's productivity when debugging data.

1.2.2 Explaining Outcomes

To interpret the result of a complex query, a user may have to understand why and how the result was proved. Data provenance provides such an explanation. However, for large datasets, the full provenance of a query result may be too large to be of any use to a human. Summarization techniques for provenance that produce compact, but semantically meaningful summaries of provenance can be used to address this problem.

Example 2 (Explanations). Continuing with our running example, let us compute the number of non-casual coffee drinkers (more than one cup per day) using the SQL query shown below.

```
SELECT count(*) as num-drinker
FROM (SELECT name, daily-coffee FROM student
      UNION ALL
      SELECT name, daily-coffee FROM teacher) drinkers
WHERE daily-coffee > 1
```

For our example database instance, we get back $r = (4)$, i.e., there are 4 non-casual coffee drinkers. We may explain this result by listing all data dependencies of r: s_1, s_3, t_2, and t_3. However, for larger datasets, the set of data dependencies may be too large to be of any immediate use. Instead, we can employ summarization techniques to compactly describe the set of inputs that are data dependencies. A common summarization technique uses declarative patterns (a limited form of queries) to describe sets of tuples. For instance, instead of listing all data dependencies we may describe them as follows: all students with a GPA higher than 3.4 and all teachers earning more than $100,000$ are non-casual coffee drinkers.

We will discuss summarization techniques for provenance in Section 4.1.

1.2.3 Security and Auditing

Provenance has also been studied extensively by the security community. The record of the operations of a system provided by provenance can be used during the forensic analysis of an attack to understand how a system was breached (Bates and Hassan, 2019). For example, assuming that we collect provenance for every SQL operation executed by a database, then when a user account is compromised this enables us to answer important questions such as *"Which data was accessed or modified by the compromised account?"*. A significant amount of work from the system's security community has investigated how to collect provenance information at the operating system level (Bates *et al.*, 2015; Pasquier *et al.*, 2017). Another security application of data provenance is detection of advanced persistent threads (APTs). It was conjectured it is possible to detect APTs by mining unusual patterns from the provenance of a system. Another security application of provenance is auditing. Many organization have to comply with laws that require them to report how they processed their data. To support auditing, database systems maintain a record of SQL operations executed in the past available as a so-called *audit-log* (Kaushik *et al.*, 2013; Fabbri and LeFevre, 2011; Kaushik and Ramamurthy, 2011; Agrawal *et al.*, 2004). One disadvantage of audit logs is that they do not record which data was affected / accessed by which operation in the audit log. Provenance information can complement audit logs with data and transformation dependencies to provide this information. Provenance has also been used for access control, e.g., to restrict access to data based on which other data it was derived from (Park *et al.*, 2012).

Example 3 (Enhancing Auditing with Provenance)**.** Consider the scenario shown in Figure 1.2. Bob, a DBA for a bank, has abused his privileges to fix the negative balance of the accounts he has with his employer. Bob has issued two statements (with ids 1432 and 1433) which add $10,000 to both his checkings and savings accounts. The bank identifies account owners by their SSN. Bob's SSN is 333-233-4534. Bob's second statement then reduces the balance of all accounts to compensate for the $20,000 he added to his accounts. To obfuscate his activity, Bob did not select his accounts using his social security number, but instead identified an alternative way to uniquely identify his accounts using their balance, his state, and his age.

The bank maintains an audit log to have a record of all data modifications in case of a security breach or to investigate illegal data access by employees. Bob's illegal operations are recorded in the audit log. However, without knowing the data dependencies and transformation dependencies (the provenance) of these operations, it is not obvious what the purpose and effect of the illegal operations were. Transformation dependencies would unearth that the first statement only affected Bob's account. Data dependencies between tuples in the database state before the updates and after the updates can be used to determine how Bob's changes can be undone.

Of course, Example 3 is too simple to be realistic. Nonetheless, it illustrates how provenance can complement audit logging. In Section 2.7 we will introduce provenance models for update operations and transactions. One major challenges with supporting provenance for updates

Database state before statement 1432

Owner	State	Age	Balance	Type
333-233-4534	IL	36	-3,030	Checking
333-233-4534	IL	36	-1,000	Savings
111-232-2323	IL	34	100,004	Checking
...	...	...	...	...

Database state after statement 1432

Owner	State	Age	Balance	Type
333-233-4534	IL	36	6,970	Checking
333-233-4534	IL	36	9,000	Savings
111-232-2323	IL	34	100,004	Checking
...	...	...	...	...

Database state after statement 1433

Owner	State	Age	Balance	Type
333-233-4534	IL	36	6,969.99	Checking
333-233-4534	IL	36	8,999.99	Savings
111-232-2323	IL	34	100,003.99	Checking
...	...	...	...	...

Audit log

id	acc	timestamp	statement
...	...	...	...

1432	Bob	01-01 8:34	`UPDATE accounts`
			`SET Balance = Balance + 10,000`
			`WHERE state = 'IL'AND Age = 36`
			`  AND balance IN (-3,030, -1,000)`

1433	Bob	01-01 8:35	`UPDATE accounts`
			`SET Balance = Balance`
			`     - (20000.0 / (SELECT count(*))`
			`                  FROM accounts))`

...	...	...	...

Figure 1.2: Example audit log and database states

is that it requires access to past database versions, i.e., tuples in the database state after the update are data-dependent on tuples in the database state before the update. We will discuss provenance models for updates in Section 2.7 and methods for capturing update provenance in Section 4.3.

1.2.4 View Maintenance and Provisioning

Many provenance models use a symbolic representation of a computation to record how inputs have been combined by a transformation. Such representations are often invariant under certain changes to the input (and/or the transformation). That is, it is possible to use the provenance of a query result to determine how this result would be affected when the input or transformation is changed in a certain way. A reader familiar with the literature may recognize this is as the well-known **view maintenance** problem (Gupta, Mumick, *et al.*, 1995): given a database D, a query Q, the query's result $Q(D)$ and an update ΔD, compute $Q(D \cup \Delta D)$. Let use consider deletion propagation for a simple projection query as an example of how to exploit provenance information for view maintenance. For sake of the example, we will use a simple provenance model that records the data dependencies for a query result tuple at tuple-granularity. That is, the provenance for each output tuple of the query is the set of input tuples it depends on.

Example 4 (View Maintenance)**.** Consider the SQL query shown below which returns customers which have ordered at least one item. This query returns two tuples: $(Peter)$ whose provenance consists of the three tuples $\{t_1, t_2, t_3\}$ (all the orders from Peter) and $(Alice)$ whose provenance is $\{t_4\}$ (all of her orders). To determine the effect of deleting some of the input tuples, we can simply remove them from the provenance of the result tuples. Any tuple whose provenance is empty will no longer be in the result. For example, if we delete t_1 and t_2 then tuple $(Peter)$ is still in result, because its provenance is not empty ($\{t_3\}$), because one of Peter's order is still present in the input.

```
SELECT DISTINCT customer FROM orders
```

customer	**item**	id
Peter	Umbrella	t_1
Peter	Raincoat	t_2
Peter	Gumboots	t_3
Alice	Umbrella	t_4

We will see in Chapter 2 and Section 3.1 that in general we will need provenance models that record how input data was combined by a computation to be able to use provenance effectively for view maintenance. A specific type of view maintenance is what-if analysis where a user wants to evaluate the effect a hypothetical change to their data has on a query result. Provenance information can be used to provision for what-if analysis if the what-if analysis is restricted to set of scenarios that are known upfront (Assadi *et al.*, 2016).

1.2.5 View Update and How-to Analysis

In the **view update** problem we are given a query Q and database D and an update $\Delta Q(D)$ to the query's result $Q(D)$ as input and have to translated this update into an update ΔD of the database such that $Q(D \cup \Delta D) = Q(D) \cup \Delta Q(D)$. Since such a ΔD may not exist for all inputs, the problem is often relaxed to allow for side-effects, i.e., the query result over $D \cup \Delta D$ is not exactly $Q(D) \cup \Delta Q(D)$. The view update problem is typically stated as an optimization problem where the goal is to find a delta ΔD that minimizes side-effects on the input database and/or query result. Provenance information aids in view update by identifying which inputs needs to be modified to achieve a desired update to a query's result.

Example 5 (View Update)**.** Continuing with Example 4, assume we would like to delete Peter from the query result. This can be achieved by deleting all of the tuples from the input database that the result tuple (*Peter*) depends on (his orders). In this example, these are input tuples t_1, t_2, and t_3.

Closely related to the view update problem are how-to queries (Meliou and Suciu, 2012) where constraints on what is a desired query result are specified declaratively and the goal is to produce an update

Loan applications

name	loanamount	income	assets	criminalrecord
Peter	10,000	15,000	500	1
Alice	50,000	135,000	10,000	0
Bob	35,000	20,000	200,000	0

Normalized data

name	loanamount	income	assets	criminalrecord
Peter	0.1	0.05	0.002	1.0
Alice	0.5	0.45	0.04	0.0
Bob	0.35	0.06	0.8	0.0

Linear classifier

$$\mathcal{C}(t) = -0.2 \cdot t.loanamount + 0.6 \cdot t.assets - 0.3 \cdot t.criminalrecord$$

Figure 1.3: Explaining classification with provenance

to the input database such that the user's constraints are fulfilled and a used-specified optimization goal is met. We will discuss applications of provenance to view update problems in Section 3.2.

1.2.6 Explaining Machine Learning Models and Outcomes

Transparency, fairness, and explainability in machine learning are of immense importance, because decisions that have significant real world impact, e.g., whether to accept or reject loan applications or whether to hire an applicant, are often delegated to machine learning models. Provenance information, while not a magical solution for explainability, is certainly relevant for explaining the outcome of applying an ML model to classify an input as well as for explaining how the training data and algorithm used to train the model affected the outcome indirectly by determining the model.

Example 6 (Explaining ML outcomes)**.** Consider a bank that has uses a linear classifier to decide loan applications. An example relation and the model are shown in Figure 1.3. Loan applications for which the model $\mathcal{C}$ returns a negative number are denied and loan applications where $\mathcal{C}$ is positive are granted. Note that the income of a person is not considered when making loan decisions. Data dependencies at the granularity of attribute values can help us identify which features of a loan application are considered in the loan application decision process. However, the the degree to which a feature affects the result depends on the feature. Note that this is common for machine learning models to have all (or most) features affect the classification outcome, but not all to the same degree. Data dependencies only record the existence of an a dependency, but do not measure the amount of influence. For instance, in our example linear model, features assets and criminal-record have large weights, i.e., have larger impact on the result in general than loan-amount and income. However, whether a feature can be held responsible for the classification of an input o also depends on o's features. This is less of a concern with simple models (like our linear classifier), but can be significant for models where features are not treated independently.

In Section 2.1.5 we will review the notion of responsibility which measures the degree of impact an input has on the output of a query. We will discuss the relationship of provenance and other types of explanations for ML models and outcomes in Section 1.2.6.

1.3 Background and Notation

We now introduce notational conventions used in this article and briefly review the relational data model and some relational query languages. Readers familiar with these concepts may skip this section.

1.3.1 The Relational Model

A relation schema $\mathbf{R}$ is a list of attribute names $(A_1, \ldots, A_n)$. The arity $arity(\mathbf{R})$ of a relation schema $\mathbf{R}$ is the number of attributes in the schema. Consider a universal domain $\mathcal{U}$ of values. Under the *named perspective* a tuple over a relation schema $\mathbf{R}$ is a function that maps attributes from $\mathbf{R}$ to values from $\mathcal{U}$. Under the *unnamed perspective* a tuple with arity n is an element from $\mathcal{U}^n$. We will opportunistically switch between these perspectives to simplify the exposition. A relation R of schema $\mathbf{R} = (A_1, \ldots, A_n)$ under set semantics is a set of tuples over schema $\mathbf{R}$. A database schema $\mathbf{D}$ is a set of relation schema $\{\mathbf{R}_1, \ldots, \mathbf{R}_m\}$. A database D over a schema $\mathbf{D}$ is set of relations $\{R_1, \ldots, R_n\}$, one for each relation schema in $\mathbf{D}$. If R is a relation (D is a database) then we use $\mathbf{R}$ ($\mathbf{D}$) to denote its schema. We will sometimes consider a slightly different definition of relations where each attribute A_i is associated with a domain $\mathcal{D}_i$. Under this model, Relation schemas are lists $(A_1 : \mathcal{D}_1, \ldots, A_n : \mathcal{D}_n)$ and a set relation of schema $(A_1 : \mathcal{D}_1, \ldots, A_n : \mathcal{D}_n)$ is a subset of $\mathcal{D}_1 \times \ldots \times \mathcal{D}_n$.

A bag semantics (or multiset) relation R of arity n is a multiset of tuples from $\mathcal{U}^n$. That is, in R each tuple is associated with a multiplicity (the number of duplicates of the tuple that are in the relation). Formally, a bag relation is a function $R : \mathcal{U}^n \to \mathbb{N}$ that associates each tuple $t \in \mathcal{U}^n$ with natural number $R(t)$ (its multiplicity). Note that here we consider bag relations to be total functions. Tuples that do not exist in the relation are assigned multiplicity 0. If $\mathcal{U}$ is infinite, then we require that there exist only finitely many t such that $R(t) \neq 0$. That is, we only consider finite relations. The use of total functions may seem to complicate matters unnecessarily, but will be beneficial when we discuss provenance models in Chapter 2.

As we use the two query languages, relational algebra and Datalog, extensively throughout this work, we briefly review them below.

1.3.2 Relational Algebra

Given a query Q, we use $\textsc{Sch}(Q)$ to denote the schema of the result of Q. We use $[\![Q]\!]_D$ to denote the result of evaluating query Q over database D. The arity $arity(Q)$ of a query Q is the arity of $\textsc{Sch}(Q)$. Sometimes we will use $Q(D)$ instead of $[\![Q]\!]_D$. For set semantics relations we will use the relational algebra shown in Figure 1.4.

As is customary, we define the semantics of algebra operators using set compressions. We use $\circ$ to denot concatenation of tuples (and other types of sequences). For a tuple t, $t.A$ denotes the projection of the tuple onto a list of expressions. A relation access R returns the instance of this relation in database D. Selection returns all input tuples t that fulfill a condition θ, written as $t \models \theta$. Projection projects all input tuples onto a list of expressions A. We typically will assume that

$$\llbracket R \rrbracket_D = R \qquad \textbf{(Relation access)}$$
$$\llbracket \sigma_\theta(Q) \rrbracket_D = \{t \mid t \in \llbracket Q \rrbracket_D \wedge t \models \theta\} \qquad \textbf{(Selection)}$$
$$\llbracket \Pi_A(Q) \rrbracket_D = \{t.A \mid t \in \llbracket Q \rrbracket_D\} \qquad \textbf{(Projection)}$$
$$\llbracket \rho_{A \to B}(Q) \rrbracket_D = \{t[A \to B] \mid t \in \llbracket Q \rrbracket_D\} \qquad \textbf{(Renaming)}$$
$$\llbracket Q_1 \times Q_2 \rrbracket_D = \{t_1 \circ t_2 \mid t_1 \in \llbracket Q_1 \rrbracket_D \wedge t_2 \in \llbracket Q_2 \rrbracket_D\} \qquad \textbf{(Crossproduct)}$$

$$\llbracket Q_1 \cup Q_2 \rrbracket_D = \llbracket Q_1 \rrbracket_D \cup \llbracket Q_2 \rrbracket_D \qquad \textbf{(Union)}$$
$$\llbracket Q_1 \cap Q_2 \rrbracket_D = \llbracket Q_1 \rrbracket_D \cap \llbracket Q_2 \rrbracket_D \qquad \textbf{(Intersection)}$$
$$\llbracket Q_1 - Q_2 \rrbracket_D = \llbracket Q_1 \rrbracket_D - \llbracket Q_2 \rrbracket_D \qquad \textbf{(Difference)}$$
$$\llbracket \gamma_{f(A) \to B}(Q) \rrbracket_D = \{(f(\llbracket \Pi_A(Q) \rrbracket_D)\} \qquad \textbf{(Aggregation)}$$
$$\llbracket \gamma_{f(A) \to B; G}(Q) \rrbracket_D = \{f(\llbracket \Pi_A(\sigma_{G=t.G}(Q)) \rrbracket_D) \circ g \mid g \in \llbracket \Pi_G(Q) \rrbracket_D\}$$

Figure 1.4: Set semantics relational algebra

such expressions can consist of references to attributes, constants, and arithmetic operations, e.g., $(A+3)*C$. This variant of projection if often referred to as generalized projection. In contexts where A is subject to restrictions, we will explicitly mention that. For convenience, we will also allow projection to rename the results of expression, e.g. ,$\Pi_{A+B \to C}$ projects the input on $A + B$ and the attribute storing the result of this expression is named C. Renaming $\rho_{A_1 \to B_1, \ldots, A_n \to B_n}$ renames attribute A_i as B_i. As a notational shortcut we will write $A \to B$ were both $A = (A_1, \ldots A_n)$ and $B = (B_1, \ldots, B_n)$ are lists of attributes to denote $A_1 \to B_1, \ldots, A_n \to B_n$. Here, $t[A \to B]$ denotes renaming attributes in the schema of tuple t (assuming the named perspective). Cross product is the set-theoretical cross product of the two input relations. Join $Q_1 \bowtie Q_2$ (not shown in the figure) is syntatic sugar for a cross product followed by a selection. Set operations (union, intersection, and difference) are defined as in set theory. These operations are only defined for inputs of the same arity (with the same data types if we consider typed relations). We consider two variants of aggregation: aggregation with group-by and aggregration without group-by. Aggregation without group-by applies an aggregation function $f : \{\mathcal{U}\} \to \mathcal{U}$ to all values from an attribute A. The result is a relation with a single tuple and single attribute named B. Note that this operator returns a single result tuple, even if the input is empty. Aggregation with group-by, partitions the input relation into groups (subsets) such that each group consists of precisely the set of tuples that have a particular value in the group-by attributes G. The aggregation function f is then applied to each group. The operator returns a tuple for each group consisting of the aggregation function result for that group and the group-by attribute values. Throughout this paper we will also allow aggregation to apply a (possibly empty) list of aggregation functions instead of a single aggregation, e.g., $\gamma_{count(*),sum(salary);dept}(\mathsf{employee})$.

Example 7 (Relational algebra (set semantics))**.** The SQL queries from Figure 1.1 and Example 2 can be written relational algebra as:

$$\Pi_{name}(\mathsf{student}) \cup \Pi_{name}(\mathsf{teaching}) \qquad \text{(Figure 1.1)}$$

$$\gamma_{count(*)}(\sigma_{daily-coffee>1}(\Pi_{name,daily-coffee}(\mathsf{student})$$
$$\cup\, \Pi_{name,daily-coffee}(\mathsf{teaching}))) \qquad \text{(Example 2)}$$

To be able to reason about SQL databases which use the bag semantics version of the relational model, we also introduce a bag semantics version of relational algebra. The semantics of the operators of this algebra is defined in Figure 1.5. Recall that we model bag relations as functions from tuples to the set of natural numbers $\mathbb{N}$. Thus, a query result is a function that maps result tuples to their multiplicity. In Figure 1.5 we define these functions pointwise, i.e., we define how to calculate the multiplicity of a query result tuple t (the result of applying function $[\![Q]\!]_D$ to t) based on the multiplicities of input tuples.[2] For instance, the number of duplicates of a tuple t in the result of $Q_1 \cup Q_2$ is the sum of the number of duplicates of t in the result of Q_1 and in the result of Q_2. Projection sums up the multiplicities of all input tuples that are projected onto the result tuple t. For a cross product we have to multiply the multiplities of input tuples. The set

[2] A reader familiar with $\mathcal{K}$-relations may recognize that for positive relational algbra we have defined the semantics of operators as is done for $\mathcal{K}$-relations. This is deliberate and will come in handy when we discuss $\mathcal{K}$-relations in Chapter 2.

$$\llbracket R \rrbracket_D(t) = R(t) \qquad (\textbf{Relation access})$$

$$\llbracket \sigma_\theta(Q) \rrbracket_D(t) = \begin{cases} \llbracket Q \rrbracket_D(t) & \text{if } t \models \theta \\ \\ 0 & \text{otherwise} \end{cases} \qquad (\textbf{Selection})$$

$$\llbracket \Pi_A(Q) \rrbracket_D(t) = \sum_{t'.A=t} \llbracket Q \rrbracket_D(t') \qquad (\textbf{Projection})$$

$$\llbracket \rho_{A \to B}(Q) \rrbracket_D(t) = \llbracket Q \rrbracket_D(t[B \to A]) \qquad (\textbf{Renaming})$$

$$\llbracket Q_1 \times Q_2 \rrbracket_D(t) = \llbracket Q_1 \rrbracket_D(t[\textsc{Sch}(Q_1)]) \cdot \llbracket Q_2 \rrbracket_D(t[\textsc{Sch}(Q_2)]) \qquad (\textbf{Crossproduct})$$

$$\llbracket Q_1 \cup Q_2 \rrbracket_D(t) = \llbracket Q_1 \rrbracket_D(t) + \llbracket Q_2 \rrbracket_D(t) \qquad (\textbf{Union})$$

$$\llbracket Q_1 \cap Q_2 \rrbracket_D(t) = \min(\llbracket Q_1 \rrbracket_D(t), \llbracket Q_2 \rrbracket_D(t)) \qquad (\textbf{Intersection})$$

$$\llbracket Q_1 - Q_2 \rrbracket_D(t) = \max(0, \llbracket Q_1 \rrbracket_D(t) - \llbracket Q_2 \rrbracket_D(t)) \qquad (\textbf{Difference})$$

$$\llbracket \gamma_{f(A) \to B}(Q) \rrbracket_D(t) = \begin{cases} 1 & \textbf{if } t = (f(\llbracket \Pi_A(Q) \rrbracket_D) \\ 0 & \textbf{otherwise} \end{cases} \qquad (\textbf{Aggregation})$$

$$\llbracket \gamma_{f(A) \to B; G}(Q) \rrbracket_D(t) = \begin{cases} 1 & \textbf{if } t.G \in \{t.G \mid \llbracket Q \rrbracket_D(t) > 0\} \\ & \quad \wedge t.B = f(\llbracket \Pi_A(\sigma_{G=t.G}(Q)) \rrbracket_D) \\ 0 & \textbf{otherwise} \end{cases}$$

Figure 1.5: Bag semantics relational algebra

operations, as expected, correspond to SQL's `UNION ALL`, `INTERSECT ALL`, and `EXCEPT ALL`. As syntactic sugar, we will use δ to denote duplicate elimination which can be expressed as group-by aggregation without an aggregation function.

Example 8 (Relational algebra (bag semantics))**.** For instance, below we show the query from Example 4 expressed in bag semantics relational algebra using duplicate elimination and using group-by aggregation.

$$\delta(\Pi_{customer}(\mathsf{orders}))$$

$$\gamma_{customer}(\mathsf{orders})$$

1.3.3 Datalog

Datalog is a query language based on formal logic. A Datalog program consists of a set of rules of the form

$$r : \underbrace{Q(\overline{X})}_{\text{head}} :- \underbrace{\mathsf{R}_1(\overline{X_1}), \dots, \mathsf{R}_n(\overline{X_n})}_{\text{body}}$$

where Q is a predicate (q relation) and all R_i are predicates or their negation and each $\overline{X_i}$ is a list of variables from an infinite set of variable symbols $\mathcal{V}$ and constants from a domain $\mathcal{U}$. We use $vars(r)$ to denote the set variables that occur in rule r. Each $\mathsf{R}_i(\overline{X_i})$ is called a goal. The left-hand side (LHS) of a Datalog rule r is called its head $head(r)$ and the right-hand side (RHS) is its body $body(r)$. A Datalog rule represents a logical implication:

$$\mathsf{R}_1(\overline{X_1}) \wedge \ldots \wedge \mathsf{R}_n(\overline{X_n}) \rightarrow Q(\overline{X})$$

A Datalog rule is safe if all the variables in the head occur in at least one positive (non-negated) body goal. A Datalog program P consists of a set of Datalog rules. The relations occurring in a Datalog are partitioned into two sets. The extensional database (or *edb*) are relations that do not occur in the head of rules in P, i.e., these are the relations in the database. The intentional database (or *idb*) are relations that occur in the head of rules (the relations computed by the program). The set of *edb* and *idb* relations are required to be disjoint. Note that in contrast to a relational algebra expression, Datalog programs may compute multiple result relations. A Datalog query Q is a Datalog program with a distinguished *idb* relation Q called the answer relation. Note that Datalog programs can be recursive, e.g., the head predicate of a rule may appear in the rule's body. A canonical example of recursion is the computation of the transitive closure of the edge relation of a graph.

The semantics of a Datalog program can be defined in several equivalent ways. Discussing these different semantics in detail is beyond the scope of this paper. We refer the interested reader to Abiteboul *et al.* (1995) and Ceri *et al.* (1989). For example, the model-theoretic semantics of Datalog treats the Datalog program and extensional database as a set of sentences in first-order logic and defines the result of the program to be the smallest model for this set of sentences.

Here we will use the fixed points semantics of Datalog. A valuation $\varphi : vars(r) \to \textsc{adom}(D)$ assigns variables from a rule r to constants from the active domain $\textsc{adom}(D)$ of a database D (which we refer to as an *edb* instance in the context of Datalog). The active domain of a database is the set of constants that occur in D. A valuation is applied to a Datalog rule r by replacing each variable X in r with $\varphi(X)$. We refer to $\varphi(r)$ as a *grounded rule*. The semantics of evaluating program P over an *edb* database instance D is the least fix point, denoted as $T_P^*(D)$, of the immediate consequences operator T_P over D. Intuitively, the immediate consequence operator takes as input an instance I and returns all new facts that can be derived based on the facts in I using the rules of P, i.e., that are the heads of grounded rules where the body evaluates too true in I. Note that as mentioned above the body of a Datalog rule is interpreted as a conjunction of its goal. A positive grounded goal $\mathsf{R}(\overline{c})$ evaluates to true over an instance I if $\mathsf{R}(\overline{c})$ exists in I. A negated goal $\neg\,\mathsf{R}(\overline{c})$ evaluates to true over I if $\mathsf{R}(\overline{c}) \not\in I$. The immediate consequence operator and its fixed point T_P^* are defined below.

$$\begin{aligned} T_P(I) &= \{\varphi(head(r)) \mid r \in P \wedge I \models \varphi(body(r))\} \\ T_P^0(D) &= D \\ T_P^{n+1}(D) &= T_P^n(D) \cup T_P(T_P^n(D)) \\ T_P^*(D) &= T_P^m(D) \textbf{ where } m = \operatorname*{argmin}_{i \in \mathbb{N}}(T_P^i(D) = T_P^{i+1}(D)) \end{aligned}$$

Note that the least fixed point $T_P^*(D)$ is guaranteed to exist and to be unique for all positive Datalog programs.

Example 9 (Transitive Closure)**.** The transitive closure of the edge relation of a directed graph contains all pairs of nodes (a, b) such that there exists a path from a to b. Below we show a recursive Datalog program P_{tc} that computes the transitive closure over a relation $\mathsf{edge}(in, out)$ storing the edges of the input graph. Rule r_1 initializes the transitive closure with the end points of all paths of length 1 (the edges of the graph). Rule r_2 takes the end points of a path of length n (a pair of nodes that we already have established to be in the transitive closure) and returns the end points of an extension of such a path by one additional edge. Figure 1.6 shows an example graph and the evaluation of P_{tc} over this graph using the immediate consequence operator.

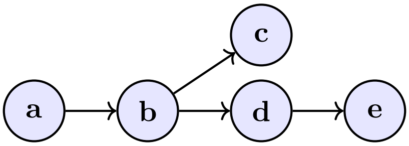
$$\begin{aligned} r_1 &: \mathsf{tc}(X, Y) :- \mathsf{edge}(X, Y) \\ r_2 &: \mathsf{tc}(X, Y) :- \mathsf{tc}(X, Z), \mathsf{edge}(Z, Y) \end{aligned}$$

1.3.4 Query Classes

Many provenance models are limited to certain classes of queries, e.g., positive relational algebra. Here we review commonly used classes of queries.

Relational Algebra

The full relational algebra $\mathcal{RA}$ consists of operators projection, union, selection, cross product, and difference. Positive relational algebra $\mathcal{RA}^+$



edge

in	out
a	b
b	c
b	d
d	e

I^0

in	out
a	b
b	c
b	d
d	e

I^1

X	Y
a	b
b	c
b	d
d	e
a	c
a	d
b	e

I^2

X	Y
a	b
b	c
b	d
d	e
a	c
a	d
b	e
a	e

Figure 1.6: Computing the transitive closure of a graph with Datalog

consists of all monotone operators of relational algebra, i.e., all operators of $\mathcal{RA}$ expect difference. The name stems from the fact that if expressed in formal logic, queries of $\mathcal{RA}^+$ do not contain negation.[3]. Adding aggregation to full relational algebra, we get the class $\mathcal{RA}^{agg}$.

Datalog and First-Order Logic

Many fundamental classes of queries have a natural representation in Datalog. Conjunctive queries ($\mathcal{CQ}$) are queries that consist of a single Datalog rule without negation and comparison predicates and where all body atoms edb relations. By allowing certain comparison predicates in conjunctive queries we get the classes of conjunctive queries with inequalities $\mathcal{CQ}^{\neq}$ and conjunctive queries with ordering $\mathcal{CQ}^{<}$. A union of conjunctive queries ($\mathcal{UCQ}$) is a Datalog query with one or more rules that are each conjunctive queries. Non-recursive positive Datalog programs may reference *idb* relations in rule bodys,

[3] Assuming that comparison operators such as $\neq$ are "build-in" and not considered as negation, e.g., $a \neq b$ instead of $\neg(a = b)$.

but no direct or indirect recursion or negated atoms are allowed. This class of queries is equivalent in terms of expressive power to the class $\mathcal{UCQ}$, but queries in this class can be exponentially more concise than the corresponding queries from $\mathcal{UCQ}$. First-order queries $\mathcal{FO}$ are non-recursive Datalog programs with negated atoms. The class Datalog consists of, possibly recursive, rules without negated atoms. Datalog$^{\neg}$ is the class of Datalog programs with recursion and negation. Note that the fixed point semantics for Datalog we have introduced is no longer sufficient for dealing with programs that contain both recursion and negation, because such programs may not have a unique smallest model. Several alternative semantics have been proposed in the literature to deal with this, e.g., the well-founded semantics (Van Gelder *et al.*, 1991), the stratified semantics (Chandra and Harel, 1985) which is only applicable to stratified programs (a subclass of Datalog$^{\neg}$), and the inflationary semantics (Kolaitis and Papadimitriou, 1988). However, the discussion of these semantics is beyond the scope of this paper.

1.3.5 Query Equivalence and Containment

Semantically, queries can be viewed as functions that map databases (their input) to relations (the query result). It is often important to be able to identify two queries which differ in syntax, but have the same semantics, i.e., produce the same result over all databases. Such queries are called *equivalent*. For certain applications it is useful to generalize this notion and reason about whether one query Q provides strictly more information than another query Q', i.e., for any database query Q' returns a subset of Q's result.

Definition 1 (Query Equivalence and Containment)**.** Given two queries Q and Q' over the same database schema, we say that Q is *equivalent* to Q', written as $Q \equiv Q'$ if:

$$\forall D : Q(D) = Q'(D)$$

Query Q is *contained* in query Q', written as $Q \sqsubseteq Q'$ if:

$$\forall D : Q(D) \subseteq Q'(D)$$

Note that $Q \equiv Q'$ if and only if $Q \sqsubseteq Q' \wedge Q' \sqsubseteq Q$.

1.4 Organization of this Monograph

The chapters of this article were written to be mostly self-contained. That being said, a basic understanding of provenance models is necessary for following the discussion in Chapter 3 and Chapter 4. Thus, we recommend readers without background in formal provenance models to read the beginning of Chapter 2 first before moving on to later chapters.

Chapter 2 introduces the reader to models that define a formal semantics for provenance. We will introduce well-established models and will compare them with respect to their expressive power, correctness guarantees, and supported transformation languages. Furthermore, we will shine light on the relationship between provenance for non-monotone queries and why-not provenance which explains missing answers.

In Chapter 3 we will discuss several applications that benefit from or are enabled by data provenance. As we already hinted at in Section 1.2, provenance can aide in a variety of view maintenance and update problems, is used to debug transformations and data, can serve as the foundation for explanations of outcomes, and is applied to explain predictions and models in machine learning.

In Chapter 4 we will discuss algorithms, techniques, and systems that manage provenance information. Our main focus will be on how to *represent and store* provenance information, how to automatically *capture* provenance information, and how to *query* data provenance.

We will cover research from other communities that is closely to related to data provenance in Chapter 5. These include data- and controlflow analysis, program slicing, and other related program analysis techniques that have been developed by the software engineering, programming languages, and compiler communities; taint analysis that has been used extensively by the security community; justifications and debugging for logic programming; symbolic program execution; and explainability in machine learning.

2

Provenance Models - Formalizing Provenance Semantics

In the introduction, we have discussed the intuitive meaning of provenance concepts such as *data dependencies* and *transformation dependencies*. For example, there is a data dependency between an output of a transformation and one of its inputs if the input was *"used to derive"* the output. However, a solid formal semantics is needed to ensure that the provenance we are tracking for a transformation is sensible. In the following we discuss such formal provenance semantics. We start by introducing intuitive requirements for data provenance semantics and discuss how they relate to the applications discussed in Chapter 3. One important observation in this regard is that applications differ in their expressiveness requirements for provenance models. This leads to interesting trade-offs, because expressiveness often comes at the cost of computational complexity and results in limitations in terms of the class of supported transformations. In the beginning of this section we focus on transformations that are queries. Other types of transformations such as updates will be covered in Section 2.7. Note that models that standardize the representation of provenance information such as the *open provenance model* (Moreau *et al.*, 2011; Moreau *et al.*, 2007; Moreau *et al.*, 2008) and the PROV model (Moreau and Missier, 2013b; Moreau and Missier, 2013a; Moreau and Groth, 2013) will not be discussed here, because their goal is to provide a standard for representing provenance rather then defining what the data and transformation dependencies of a query (transformation) are. We will briefly discuss these models in Section 4.1.

2.1 Requirements for Provenance Semantics

Consider a single query Q evaluated over a relational database D and let us assume that we want to define the provenance for a query result tuple $t \in Q(D)$. For now we will consider only data dependencies and want to compute the set $\mathcal{DD}(Q, D, t)$ of tuples from the input database that t depends on. That is, we are searching for a set $\mathcal{DD}(Q, D, t) \subseteq D$. How do we know whether there exists a data dependency between t and a tuple $t' \in D$?

Example 10 (Data dependencies)**.** Figure 2.1 shows an example online grocery orders database. To be able to refer to individual tuples, we assign a unique identifier to each tuple (shown to the right of a tuple). Consider a query that returns the names of customers from relation Customers. Figure 2.2 shows this query ($Q_{custnames}$) expressed in SQL and relational algebra and the result of evaluating this query under set semantics over the database instance from Figure 2.1. Intuitively, each result tuple depends only on a single input tuple: the customer with that particular name. For instance, tuple n_1 was generated from tuple c_1. Obviously, tuples from relations not accessed by the query, e.g., Items can not impact the query's result and, thus, cannot possibly belong to the set of data dependencies of any of the query's results. As another example, consider query $Q_{custWithOrders}$ which returns the names of customers that have ordered from our store at least once. Note that some customers have placed more than one order. The existence of any of these orders is sufficient for the customer to be included in the result of the query.

Based on this example, it seems that we should require data dependencies to (i) not contain tuples that are irrelevant for producing the

Name	Age	Card	id
Peter	39	Visa	c_1
Alice	25	AE	c_2
Bob	25	Visa	c_3
Astrid	26	Master	c_4

(a) Customers

Item	Price	Weight (lb)	id
Lettuce	0.99	2.5	i_1
Oranges	2.49	3	i_2
Apples	3.99	6	i_3
Bok choy	1.99	1.5	i_4
Peanuts	3.99	2	i_5

(b) Items

Customer	Item	NumItems	Date	id
Peter	Lettuce	3	2020-01-03	o_1
Peter	Oranges	1	2020-01-03	o_2
Peter	Lettuce	3	2020-01-04	o_3
Bob	Oranges	2	2020-01-04	o_4
Alice	Peanuts	3	2020-01-04	o_5

(c) Orders

Figure 2.1: Example online grocery store database

$Q_{custnames}$: `SELECT Name FROM Customers;`
$Q_{custWithOrders}$: `SELECT DISTINCT Name FROM Orders;`

$$Q_{custnames} \coloneqq \Pi_{Name}(\mathsf{Customers})$$
$$Q_{custWithOrders} \coloneqq \Pi_{Customer}(\mathsf{Orders})$$

Name	id
Peter	n_1
Alice	n_2
Bob	n_3
Astrid	n_4

(a) $Q_{custnames}$

Customer	id
Peter	w_1
Alice	w_2
Bob	w_3

(b) $Q_{custWithOrders}$

Figure 2.2: Online grovery store queries and results

result of interest and (ii) to contain all tuples that are necessary for producing the result of interest. Next, we will formalize these intuitions.

2.1.1 Data Dependencies and Inverse Functions

Since queries are functions mapping databases to relations, it is tempting to define provenance as the inverses of such functions which take a query result as input and return the database that was used to generate this result. However, there are several problems which this approach. First off, most queries are not invertible in the mathematical sense, because they are not injective. That is, queries may produce the same result for two different input databases. Even for queries that are invertible, the inverse query would simply return the input database. However, for data dependency tracking we are typically interested in tracking data at a finer granularity, e.g., which input tuples are responsible for producing a single result tuple. Thus, the inversion of a query in a mathematical sense does not help us to track data dependencies.

2.1.2 Sufficiency

Let us first adopt a conservative standpoint and ensure that $\mathcal{DD}(Q, D, t)$ is not missing any inputs that were needed to compute t. That is, we want to ensure that $\mathcal{DD}(Q, D, t)$ is **sufficient** for producing the output t. We can formalize this as requiring that evaluating Q over $\mathcal{DD}(Q, D, t)$ should at least return t.

Definition 2 (Sufficiency). Let Q be a query, D a database, and $t \in Q(D)$. A set $D_{sub} \subseteq D$ is called sufficient for producing t through Q if:

$$t \in Q(D_{sub}) \tag{2.1}$$

Sufficiency has been considered as a requirement for provenance in early work on database provenance. For instance, in their seminal paper, Buneman *et al.* (2001) introduced Why-provenance which is based on sufficiency (and the criterion of minimality that we discuss next).

Example 11 (Sufficiency). Let us apply our definition of sufficiency to the example from Figure 2.2. Consider the result tuple with id n_1

(Name=Peter) of query $Q_{custnames}$. The tuple corresponding to the customer named Peter (c_1) is sufficient for producing tuple n_1:

$$Q_{custnames}(\{c_1\}) = \{n_1\}$$

Observe that no subset of the input database that does not contain c_1 can be sufficient. Furthermore, any subset containing $\{c_1\}$, e.g., $\{c_1, c_2, c_3\}$ is sufficient. Now consider the result tuple w_1 for query $Q_{custWithOrders}$. Any one of the orders of the customer named Peter (tuples o_1, o_2, and o_3) is sufficient for producing this result tuple, i.e., Peter is in the result if at least one of his orders exists in the input.

Based on the example above, we may conjecture that any superset of a sufficient subset of an input database is also sufficient. This conjecture holds, but only for queries that are monotone. For non-monotone queries, adding inputs to a sufficient subset of the database may result in the removal of the result tuple of interest.

Lemma 1 (Supersets of Sufficient Subsets are Sufficient for Monotone Queries)**.** Let Q be a monotone query, D be a database, and t a tuple in $Q(D)$. If $D' \subseteq D$ is sufficient for t then so is any set D'' for which $D' \subseteq D'' \subseteq D$.

Proof. The Lemma follows directly from the monotonicity of the query. Since D' is sufficient we have $t \in Q(D')$. Recall the definition of monotonicity for queries: $D_1 \subseteq D_2 \Rightarrow Q(D_1) \subseteq Q(D_2)$. Thus, $Q(D') \subseteq Q(D'')$ which implies $t \in Q(D'')$. □

2.1.3 Necessity and Minimality

Defining provenance based on sufficiency alone has two major drawbacks: (i) there may exist more than one sufficient subset of the input database and it is unclear which subset should be designated to be the provenance and (ii) sufficiency does not prevent us from including inputs in the provenance that are irrelevant for deriving the output of interest. This observations are confirmed by Example 11: we can add tuples from tables not accessed by the query to a sufficient subset for $Q_{custWithOrders}$ and the results is still sufficient. One may be tempted to address these two drawbacks by requiring that only inputs that are **necessary** for producing the output are included in the provenance.

Definition 3 (Necessity). Let Q be a query and D a database. A tuple $t_{in} \in D$ is *necessary* for producing a tuple $t \in Q(D)$ if:

$$\forall D' \subseteq D : t \in Q(D') \rightarrow t_{in} \in D'$$

We use $D_{necessary}$, called the necessary core of D with respect to Q, to denote the subset of D that consists of all necessary input tuples.

Reconsidering our example query $Q_{custnames}$, necessity solves both (i) and (ii): the necessary core of D is sufficient. However, necessity fails in the presence of disjunction. For instance, for $Q_{custWithOrders}$, none of the inputs is necessary since subsets of the input are sufficient as long as they contain at least one tuple from the set $\{o_1, o_2, o_3\}$ (one of Peter's orders). So why do we refer to this as disjunction? The reason becomes clear if we write the requirement on any sufficient subset of database D as a logical condition:

$$\forall D' \subset D : o_1 \in D' \vee o_2 \in D' \vee o_3 \in D' \Rightarrow D' \textbf{ is sufficient}$$

An alternative way to ensure that irrelevant inputs are not included that does not fail in the presence of disjunctions is to require that in addition to being sufficient, the provenance of an output should be **minimal**.

Definition 4 (Minimality). Let Q be a query, D a database, and $t \in Q(D)$. A set $D_{sub} \subseteq D$ that is sufficient for producing t through Q is called **minimal** if no proper subset of D_{sub} is sufficient:

$$\nexists D' \subset D_{sub} : t' \in Q(D')$$

However, this only resolves issue (ii) and not (i), because for queries that are disjunctive in nature, i.e., use disjunctive operators such as union or projection in relational algebra, there may be multiple alternative ways of deriving t from the inputs.

Example 12 (Minimality)**.** We invite the reader to confirm that there are 3 subsets of the example database that are minimal and sufficient for producing result tuple w_1 of query $Q_{custWithOrders}$:

$$\{o_1\} \qquad \{o_2\} \qquad \{o_3\}$$

Note that any subset of the input database that does not contain at least one of the orders from customer Peter (o_1, o_2, and o_3) cannot produce w_1 as an output and, thus, cannot be sufficient. Furthermore, any superset of one of the sufficient subsets shown above would be sufficient, but not minimal.

To summarize, sufficiency ensures that provenance contains enough information for producing the result through the query, necessity excludes irrelevant inputs while failing in disjunctive contexts, and minimality ensure that irrelevant inputs will not be included in the provenance. For queries that are disjunctive in nature, e.g., queries with union or projection, there may exist more than one minimal and sufficient subset of the input database.

2.1.4 Intervention-based Causality

An alternative to defining provenance through sufficiency and minimality is to rely on intervention-based notions of causality. The causality notion we review here was introduced by Halpern and Pearl in Chockler *et al.* (2008), Halpern and Pearl (2005), Chockler and Halpern (2004), Halpern (2000), and Pearl (2000). This notion of causality uses interventions which are changes to the input of an operation to determine which inputs caused an output. Here we discuss an adoption of this definition for database queries from Meliou *et al.* (2010). Interventions in this context correspond to changes to the input database. Specifically, to test whether an input tuple t_{cause} is a **cause** for a result tuple t of a query Q, we remove this tuple from the input (this is the intervention) and check whether this causes t to be removed from the output of the query. If that is then case, then t_{cause} has to exist for t to be in the result.

Definition 5 (Counterfactual Causes)**.** A tuple $t_{cause} \in D$ is a counterfactual cause for a tuple $t \in Q(D)$ if

$$t \notin Q(D - \{t_{cause}\})$$

Observe the relationship between **counterfactual causes** and the notions of sufficiency and minimality (necessity) we have discussed previously. Any counterfactual cause is strictly necessary for producing the result tuple of interest: removing the cause from the input results in a the tuple of interest to be removed from the query's result. The relationship between causality and necessity for monotone queries become even more clear if we express them as logical implications. A tuple $t_{in} \in D$ is necessary for producing a result tuple t if every sufficient subset of D contains t_{in}. Analog, t_{in} is a counterfactual cause for a monotone query, if any subset D' of D that does not contain t_{in} is not sufficient. While Definition 5 only requires that t is not in the result in a specific such D', namely $(D - \{t_{in}\})$, for monotone queries this two statements are equivalent. To see why this is the case recall the definition of a monotone query: for any $D' \subseteq D$ it is the case that $Q(D') \subseteq Q(D)$. Thus, if $t \notin (D - \{t_{in}\})$ then the same has to be true for any subset of $D - \{t_{in}\}$. Below we show the conditions for necessity and counterfactual causality as logical implications.

$$\forall D' \subset D : t \in Q(D') \rightarrow t_{in} \in D' \qquad (\textbf{Necessity})$$

$$\forall D' \subset D : t_{in} \notin D' \rightarrow t \notin Q(D') \qquad (\textbf{Counterfactual Causality})$$

Using the standard logical equivalence $a \rightarrow b \Leftrightarrow \neg b \rightarrow \neg a$, it is immediately obvious that these two formulas are equivalent. That is, counterfactual causes and necessity are the same for monotone queries! Thus, counterfactual causes also suffer from the same limitation as necessity when dealing with disjunctive derivations. While counterfactual causes cannot detect causes when there are multiple alternative derivations of a result, it is possible to extend this idea to deal with alternatives. Intuitively, for a tuple to be a cause when multiple alternative derivations of a result may exist, it has to become counterfactual under the assumption that the alternative derivations which do not contain this tuple have failed. This can be modeled as creating a hypothetical database by removing tuples until only derivations containing the tuple whose causal effect on the result we would like to test remain. This generalized types of causes are referred to as **actual causes**. An alternative way to interpret actual causes is that these are counterfactual causes under some intervention that deletes from tuples from the input database with causing the result of interest to be removed from query's result.

Definition 6 (Actual Causes)**.** A tuple t_{cause} is an actual cause for a tuple $t \in Q(D)$, if there exists $\Gamma \subset D - \{t_{cause}\}$, called a **contingency**, such that the following conditions hold:

$$t \in (D - \Gamma)$$
$$t \notin (D - \Gamma - \{t_{cause}\})$$

Actual causes encode a form of conditional necessity. Under a change to the input that does not result in t to be deleted, t_{cause} becomes necessary.

Example 13 (Actual Causes)**.** Consider query $Q_{cheapOrders}$ shown in Figure 2.3. This query returns customers which have ordered "cheap" items (the item's price is less than \$1.00). Peter, having ordered Lettuce twice, is the only customer fulfilling this condition. The actual causes for the result tuple c_1 and contingencies for each of these causes are shown below:

- o_1 with contingency $\{o_3\}$
- o_3 with contingency $\{o_1\}$
- i_1 with contingency $\emptyset$ (a counterfactual cause)

Note that actual causes strictly generalize counterfactual causes: every counterfactual cause is an actual cause using the empty set as a contingency. Note that in Meliou *et al.* (2010) the user can partition the input database into tuples that should be considered as causes and those which are not. For now we will ignore this extension and assume that no input tuples are excluded from causal analysis.

$$Q_{cheapOrders} := \Pi_{Customer}(\mathsf{Orders} \bowtie \sigma_{Price<1.00}(\mathsf{Items}))j$$

Customer	id
Peter	c_1

Figure 2.3: Example illustrating the degree of responsibility of data dependencies.

2.1.5 Responsibility

The concepts we have defined so far are declarative ways of specifying what properties the provenance (data dependencies) should fulfill. Thus, they can be used to determine a subset of the input database that contributes to a result tuple. However, there is no way to measure the degree of contribution of an input. To illustrate what we mean by degree of contribution, consider the following example.

Example 14. Reconsider the causes for the result tuple c_1 of query $Q_{cheapOrders}$. For convenience we repeat the contingencies from Example 13 for these causes below. Note that tuple i_1 is necessary for producing the result while one of o_1 and o_2 can be removed without affecting the result tuple c_1. Thus, we could argue that i_1 is more responsible for producing c_1 than o_1 and o_2.

- o_1 with contingency $\{o_2\}$
- o_2 with contingency $\{o_1\}$
- i_1 with contingency $\emptyset$ (a counterfactual cause)

As shown in this example, not all inputs that belong to the provenance (are causes) have the same impact on the result if removed. An important take-away message of this example is that the degree of contribution of a cause seems to be correlated with the size of contingencies for the cause. Meliou *et al.* (2010) introduced the notion of **responsibility** for causes shown below that is based on the size of contingencies. This is an adaption of the definition from Chockler and Halpern (2004) to the database context.

Definition 7 (Responsibility)**.** Let Q be a query, D a database, and $t \in Q(D)$. Let $\mathbf{\Gamma_{cause}}$ be the set of all contingencies of an actual cause t_{cause}. The responsibility $\rho_{t_{cause}}$ of t_{cause} for t is:

$$\frac{1}{1 + \min_{\Gamma \in \mathbf{\Gamma_{cause}}} |\Gamma|}$$

The degree of responsibility as defined above is a number in the interval $(0, 1]$. The responsibility of counterfactual causes, and only that of counterfactual causes, is 1. Let us revisit the example above to apply this definition.

Example 15 (Responsibility)**.** Note that the contingencies for the actual causes of result tuple c_1 (Figure 2.3) shown in Example 14 are minimal in size. Based on these contingencies we can compute the responsibility of the causes for c_1 as shown below:

$$\rho_{o_1} = \frac{1}{1+1} = \frac{1}{2} \qquad \rho_{o_2} = \frac{1}{1+1} = \frac{1}{2} \qquad \rho_{i_1} = \frac{1}{1+0} = 1$$

Both o_1 and o_2 are assigned a responsibility of $\frac{1}{2}$ since only one of these two tuples is needed to produce result tuple c_1. Tuple i_1 has the maximum responsibility of 1. This tuple is strictly necessary for computing tuple c_1.

Tracking responsibility based on existence of tuples in the query result is not enough for all types of queries. For example, consider an aggregation query without group-by. Such a query returns exactly one result tuple over any database. Removal of a particular input does not cause the aggregation result tuple to be removed, but instead affects the aggregated value. For instance, consider a query summing up the price of items from relation Items (Figure 2.1b). In SQL, this query can be written as:

```
SELECT sum(Price) AS totalPrice FROM Items
```

The total price computed over the example database is \$13.45. A reasonable way to define intervention-based responsibility for such queries is to measure an input's relative contribution to the aggregation function result. For instance, tuple i_1 with a price of \$0.99 has a relative contribution of $\frac{0.99}{13.45} \approx 7\%$ while tuple i_3 with a price of \$3.99 has a relative contribution of $\frac{3.99}{13.45} \approx 30\%$. For example, Roy and Suciu (2014) generates explanations for aggregation results as patterns that compactly encode sets of inputs that have a large effect on the aggregation function result based on this definition of responsibility.

2.1.6 Syntax Independence

So far we have been concerned with the provenance of a single query, defining requirements that are not specific to a particular query language. In fact, we have treated queries as black box functions that take a database as an input and returns relations. When dealing with a concrete query language, say Datalog, we face the potential problem that there may be multiple equivalent ways for how to specify a particular such function. That is, two syntactically different queries can have the same semantics, i.e., specify the same function from databases to relations. Formally, this is the notion of query equivalence we have introduced in Section 1.3.4. Buneman *et al.* (2001) argued that the provenance of a query should only depend on the semantics and not on the syntax of a query which means that equivalent queries should have the same provenance. In Buneman *et al.* (2001) this property was called "invariance under query rewriting". We will refer to this property as **syntax independence**. In contrast to the other properties we have discussed so far, syntax independence is specific to a particular provenance model $\mathbb{P}$ and class of queries $\mathcal{C}$ which determines what is the provenance for a query, database, and result tuple.

Definition 8 (Syntax Independence)**.** Let $\mathbb{P}$ be a provenance model, $\mathcal{C}$ a class of queries, and let $\mathcal{P}(\mathbb{P}, Q, D, t)$ for a query $Q \in \mathcal{C}$, database D, and tuple $t \in Q(D)$ denote the provenance of tuple t according to the model $\mathbb{P}$. The model $\mathbb{P}$ is *syntax independent* if the condition shown below holds for any pair of queries $Q \in \mathcal{C}$ and $Q' \in \mathcal{C}$.

$$Q \equiv Q' \Rightarrow \forall D, t \in Q(D) : \mathcal{P}(\mathbb{P}, Q, D, t) = \mathcal{P}(\mathbb{P}, Q', D, t)$$

Note that syntax independence can be problematic when applied to transformation dependencies, i.e., when the provenance describes which parts of the query are responsible for producing a result. For instance, consider the use of provenance for debugging a query: transformation dependencies will be used to identify which parts of the query are responsible for producing query results that the user has identified of being incorrect. The user can then focus their debugging efforts on these parts of the query. This, of course, is only sensible if the provenance is specific to the syntax of the user's query. That being said, syntax independence is certainly a sensible notion for data dependencies and for use cases where the semantics of a query is more important than the way it is written. Note that the definitions of sufficiency, necessity, and causality we have presented are syntax independent, because they treat queries as black-box functions.

Example 16 (Syntax Independence)**.** Consider the two queries Q_1 and Q_2 shown below. Note that these queries are equivalent under set semantics. Note that query Q_1 is the same as query $Q_{custnames}$ whose result is shown in Figure 2.2a. We leave it to the reader to validate that the actual causes and necessary tuples for both queries are the same.

$$Q_1 \coloneqq \Pi_{Name}(\mathsf{Customers}) \quad Q_2 \coloneqq \Pi_{Name}(\mathsf{Customers} \bowtie \mathsf{Customers})$$

A potential solution for the disadvantage of syntax independence is to allow provenance to be syntax dependent to enable debugging of transformations, but define equivalence laws that holds over the provenance to ensure that equivalent queries have equivalent, but not equal, provenance.

2.1.7 Computability

Sufficiency, necessity, and causality are only concerned with data dependencies, i.e., which part of the input data is relevant for producing a result. The next property that we discuss also takes into account *how* inputs were combined by the query to produce a result. That is provenance models that enjoy this property encode both data and transformation dependencies.

Example 17 (Data and Transformation Dependencies)**.** Reconsider the query $Q_{cheapOrders}$ from Figure 2.3. Recall that three tuples are causes for the result tuple c_1: $\{o_1, o_2, i_1\}$. However, the set of causes or equivalently the set of necessary tuples does not explain *how* these tuples were combined by the query to produce the result: tuple o_1 was joined with i_1 and tuple o_2 was joined with i_1.

A major question is how can we check whether a provenance model correctly models how inputs were combined by a query? For that we define a property called **computability**. Computability requires that the result of the query can be reconstructed from its provenance. Thus, it ensures that provenance captures the "essence" of the query's semantics. Recall that we use $\mathcal{P}(\mathbb{P}, Q, D, t)$ to denote the provenance of a tuple $t \in Q(D)$ according to a provenance model $\mathbb{P}$. In the following we will use $\mathcal{P}(\mathbb{P}, Q, D)$ to denote the union of $\mathcal{P}(\mathbb{P}, Q, D, t)$ over all $t \in Q(D)$.

Definition 9 (Computability)**.** Consider a query Q and database D. The provenance produced by a provenance model $\mathbb{P}$ is computable, if $Q(D)$ can be reconstructed from $\mathcal{P}(\mathbb{P}, Q, D)$.

Note that computability is a stricter form of sufficiency: the provenance does not just have to contain a sufficient subset of the input, but also has to encode how to compute the result using this subset of the input data. While this property may seem mysterious, it will become more clear when we introduce provenance models that enjoy this property.

2.2 Provenance as Annotations on Data

A concept employed by several provenance models is to model provenance as **annotations** on data. Annotations allow metadata of a certain type to be associated with pieces of data. For instance, to record the set of actual causes for each result tuple of a query, we may annotate result tuples with these sets.

Example 18 (Annotating Data with Provenance)**.** Reconsider query $Q_{custWithOrders}$ from Figure 2.2. Below we show the result of this query annotating each tuple with its set of necessary tuples (shown to the right of each tuple). For example, Alice is in the result, because her order (o_4) and the item she ordered (i_5) exist in the input.

Name	Provenance
Peter	$\{o_1, o_2, o_3, i_1, i_2\}$
Alice	$\{o_5, i_5\}$
Bob	$\{o_4, i_2\}$

While it should be obvious that any type of provenance can be modeled as annotations on data, the reader may wonder what the purpose of this exercise is. The main benefit of modeling provenance as annotation is that the provenance of a query can be defined through annotation propagation rules instead of rules that invert operations. By annotation propagation rules, we mean rules that determine the annotation (provenance) of a query result based on the semantics of the query and the annotations of its input (e.g., annotating each tuple with an identifier that serves as its initial provenance).

2.3 Provenance for Monotone Queries

Having introduced necessary background, we are now ready to discuss provenance models, their properties, and their interrelationships. Some provenance models are based on declarative definitions similar to the properties we have discussed in the earlier parts of this chapter, i.e., they define conditions that the provenance has to fulfill rather than defining how to calculate provenance, while others use operational definitions, i.e., they are defined through the rules they use for computing provenance. While declarative definitions are conceptually cleaner, operational definitions lay the foundation for efficient provenance capture techniques that we will discuss in Section 4.3. Thus, it will be insightful to also introduce operational definitions for some declarative provenance models. Some of the models discussed in this section do support non-monotone queries. We will revisit these models when covering non-monotone queries in Section 2.4.

2.3.1 Why-provenance

Why-provenance is a provenance model defined based on sufficiency and minimality. This is one of the first attempts to formalize provenance and was introduced in the seminal paper in Buneman *et al.* (2001).

Buneman *et al.* (2001) introduced this model for a nested data model. Here we discuss the version of this model for relational data described in Cheney *et al.* (2009). Why-provenance is based on sufficient subsets of the input database. In the terminology of Buneman *et al.* (2001), any sufficient subset of the input is called a *witness*. The set of all witnesses for a query result is called the **set of witnesses** of the result.

Definition 10 (Set of Witnesses)**.** Consider a query Q, database D, and tuple $t \in Q(D)$. The *set of witnesses* $\textsc{Wit}(Q, D, t)$ of t with respect to Q and D is the set of all witnesses for t:

$$\textsc{Wit}(Q, D, t) = \{D' \mid D' \subseteq D \wedge t \in Q(D')\}$$

Since the concept of a set of witnesses is defined solely through sufficiency, the set of witnesses of a query result will contain many witnesses that contain non-necessary tuples. Buneman *et al.* (2001) uses minimality (Definition 4) to address this problem. The **set of minimal witnesses** contains all witnesses that are minimal, i.e., which do not contain any other witnesses. That is, the set of minimal witnesses contains all alternative, non-redundant subsets of the input that are sufficient for producing a result of interest.

Definition 11 (Set of Minimial Witnesses)**.** Consider a query Q, database D, and tuple $t \in Q(D)$. The *set of minimal witnesses* $\mathrm{M}\textsc{Wit}(Q, D, t)$ of t with respect to Q and D is the set of all witnesses that do not contain any other witnesses:

$$\begin{aligned}\mathrm{M}\textsc{Wit}(Q, D, t) = \{D' \mid D' \in \textsc{Wit}(Q, D, t) & \\ \wedge \neg\exists D'' \in \textsc{Wit}(Q, D, t) : D'' \subset D\}\end{aligned}$$

Note that both the set of witnesses and the set of minimal witnesses are syntax independent, because they are defined declaratively using sufficiency (and minimality in the case of minimal witnesses).

Example 19 (Witness Sets)**.** Consider the first result tuple $w_!$ of query $Q_{custWithOrders}$. Some witnesses (sufficient subsets of the input) for this tuple are shown below.

$$\{o_1, i_1\} \quad \{o_2, i_2\} \quad \{o_3, i_1\} \quad \{o_1, o_2, o_3, i_1\} \quad \{o_1, o_5, c_1, c_2, i_1\}$$

Only three of these witnesses are minimal, corresponding to the the orders of Peter (o_1, o_2, and o_3) and the item of each of these orders (i_1, i_2):

$$\{o_1, i_1\} \qquad \{o_2, i_2\} \qquad \{o_3, i_1\}$$

In addition to the declarative definitions shown above, Buneman *et al.* (2001) also introduced a syntax-driven definition called **why-provenance**. Here we discuss the adaption of this definition for positive relational algebra from Cheney *et al.* (2009). Recall that positive relational algebra or $\mathcal{RA}^+$ is the query language consisting of the relational algebra operators union, projection, selection, cross product, and renaming.[1] The term why-provenance is sometimes used in the literature to denote this model and sometimes to denote a more general concept which we refer to as data dependencies in this article. To avoid confusion we will exclusively use the term data dependencies when referring to the more general concept and reserve why-provenance for the specific model. Why-provenance is defined as recursive rules which define the why-provenance for the result of an relational algebra operator based on the why-provenance of the operator's inputs. That is, there is one rule for each operator of $\mathcal{RA}^+$. Recall that we use $t.A$ to denote the projection of a tuple t on a list of attributes A, $t[A \to B]$ to denote renaming attributes A from the schema of tuple t to B, and $t[Q]$ to denote projecting the tuple t onto the schema of Q.

Definition 12 (Why-provenance Cheney *et al.*, 2009)**.** Let Q be an $\mathcal{RA}^+$ query, D a database, and $t \in Q(D)$. The why-provenance $\textsc{Why}(Q, D, t)$ is defined as shown below:

$$\textsc{Why}(\mathsf{R}, D, t) = \begin{cases} \{\{t\}\} & \text{if } t \in R \\ \emptyset & \text{otherwise} \end{cases}$$

$$\textsc{Why}(\rho_{A \to B}(Q), D, t) = \textsc{Why}(Q, D, t[A \to B])$$

$$\textsc{Why}(\sigma_\theta(Q), D, t) = \begin{cases} \textsc{Why}(Q, D, t) & \text{if } t \models \theta \\ \emptyset & \text{otherwise} \end{cases}$$

[1] Note that join is also part of positive relational algebra since it it can be expressed using cross product and selection.

$$\begin{aligned}
\textsc{Why}(\Pi_A(Q), D, t) &= \bigcup_{u \in Q(D) : u.A = t} \textsc{Why}(Q, D, u) \\
\textsc{Why}(Q_1 \bowtie Q_2, D, t) &= \{D' \cup D'' \mid D' \in \textsc{Why}(Q, D, t[Q_1]) \\
&\qquad\qquad \wedge D'' \in \textsc{Why}(Q, D, t[Q_2])\} \\
\textsc{Why}(Q_1 \cup Q_2, D, t) &= \textsc{Why}(Q_1, D, t) \cup \textsc{Why}(Q_2, D, t)
\end{aligned}$$

As was shown in Buneman *et al.* (2001), every element in the why-provenance is a witness and $\textsc{Why}(Q, D, t)$ contains all minimal witnesses from $\textsc{MWit}(Q, D, t)$. Thus, an alternative way of computing the set of minimal witnesses is to compute the why-provenance and then removing all non-minimal witnesses. Thus, we will also refer to the set of minimal witnesses as the **minimal why-provenance**, denoted as $\textsc{MWhy}(Q, D, t)$. Since minimal why-provenance is equal to the set of minimal witnesses, it also is syntax independent.

2.3.2 Lineage

Cui and Widom (2000a) introduced a provenance model called **lineage** based on data dependencies that tracks data dependencies separately for each input relation of a query. That is, for a query accessing relations R_1, ..., R_n, the provenance is a list $({R_1}^*, \ldots, {R_n}^*)$ where each ${R_i}^*$ is a subset of R_i. Cui and Widom (2000a) provide a declarative definition for the lineage of a query for single operator and another definition that determines how the lineage of multiple operators has to be combined to calculate the lineage of a query. The authors also introduced a syntax-dependent definition and did prove that this definition equivalent to the declarative one.

Definition 13 (Lineage of a Relational Algebra Operator)**.** Let Op be an relational algebra operator with inputs R_1, ..., R_n. The *lineage* of a tuple $t \in Op(R_1, \ldots, R_n)$ is a list $\langle\, {R_1}^*, \ldots, {R_n}^* \,\rangle$ where ${R_i}^* \subseteq R_i$ such that:

1. $Op({R_1}^*, \ldots, {R_n}^*) = \{t\}$
2. $\forall i \in \{1, \ldots, n\} : \forall t^* \in {R_i}^* : Op({R_1}^*, \ldots, \{t^*\}, \ldots, {R_n}^*) \neq \emptyset$
3. $({R_1}^*, \ldots, {R_n}^*)$ is the maximal among all lists fulfilling conditions (1) and (2).

Note that condition (1) is a stricter version of sufficiency: instead of requiring that t is in the result of evaluating query Q over the provenance, this condition requires that only t is returned. While this definition works for the operators considered in Cui and Widom (2000a), it would fail for any operator that returns more than one result tuple for a single input tuple. For example, consider extending relational algebra with a "multiple projection" operator that takes multiple projection lists as an input parameter and projects each input tuple onto all of these lists. For such an operator, it may be impossible to find an input that produces exactly one result tuple, since if evaluated over a single input tuple this operator would return multiple result tuples. Similar issues can arise from condition (2). This condition works for the operators considered in Cui and Widom (2000a), but it is possible to construct new operators for which it fails. For instance, consider an operator that takes a single relation as input, sorts this relation on one of its attributes, and then groups each two adjacent with respect to the sort order and returns for each group the first tuple in sort order. For any group with less than two tuples, e.g., if the input does only contain one tuple, no output is produced. The lineage for this operator is the empty set for any input, because any input tuple would fail condition (2). Finally, requiring maximality can be problematic when dealing with operators like set difference that check for the non-existence of input tuples (we will discuss this further in Section 2.4).

Definition 13 only defines the lineage for queries consisting of a single operator. The lineage for queries consisting of multiple operators is defined through transitivity. Intuitively, if an input tuple t_{in} belongs to the lineage of a tuple t_{op} in the input of an operator and t_{op} belongs to the lineage of a result tuple t of the operator according to Definition 13, then t_{in} belongs to the lineage of t according to the query. In the definition below we use $\Cup$ to denote the element-wise union of two lists of sets, e.g., $(S_1, S_2) \Cup (S_3, S_4) = (S_1 \cup S_3, S_2 \cup S_4)$. Also recall that $\circ$ denotes concatenation of tuples and lists.

Definition 14 (Lineage of a Query). Let Q be a query $Op(Q_1, \ldots, Q_m)$ where Op is an operator with inputs with inputs R_1, ..., R_m. The *lineage* $\textsc{Lin}(Q, D, t)$ of a tuple $t \in Q(D)$.

$$\begin{aligned}
Q_{cheapOrders} &:= \Pi^{\textcircled{0}}_{Customer}(\mathsf{Orders}^{\textcircled{2}} \bowtie^{\textcircled{1}} \sigma_{Price<3.00}{}^{\textcircled{3}}(\mathsf{Items}^{\textcircled{4}}))\\
Q_1 &:= \mathsf{Orders} \bowtie \sigma_{Price<3.00}(\mathsf{Items})\\
Q_2 &:= \mathsf{Orders}\\
Q_3 &:= \sigma_{Price<3.00}(\mathsf{Items})\\
Q_4 &:= \mathsf{Items}
\end{aligned}$$

Figure 2.4: Operators identifiers (red labels) and subqueries of query $Q_{cheapOrders}$

- If $Q := \mathsf{R}$, then $\textsc{Lin}(Q, D, t) = \{t\}$
- Otherwise, $Q := Op(Q_1, \ldots, Q_m)$. Let $({S_1}^*, \ldots, {S_k}^*)$ be the lineage of t wrt. Op. Consider tuples $(t_1, \ldots, t_k)$ such that $t_i \in {S_i}^*$ for $i \in \{1, \ldots, k\}$ and let T denote the set of all such tuples. Then the lineage of t is defined as

$$\bigcup_{(t_1,\ldots,t_k)\in T} \textsc{Lin}(Q_1, D, t_1) \circ \ldots \circ \textsc{Lin}(Q_k, D, t_n)$$

Note that by defining lineage through transitivity, the definition makes the implicit assumption that provenance itself is transitive. No problems arise from this assumption for positive relational algebra queries, but this assumption is violated by set difference, one of the non-monotone operators supported by Cui and Widom (2000a) and Cui *et al.* (2000). We will discuss this further in Section 2.4.

Example 20 (Lineage)**.** Reconsider query $Q_{cheapOrders}$. For convenience we assign each operator an identifier and show the subquery rooted at each operator of this query (Figure 2.4). The intermediate results produced by each of these subqueries is shown in Figure 2.5. We trace the lineage of result tuple c_1 (customer Peter). There are two tuples in the join result (t_{11} and t_{12}), whose Customer value is "Peter". Evaluating the join over the set contain only these two tuples returns only c_1 (Definition 13, condition 1). Each of these tuples produces a non-empty result (Definition 13, condition 2) and no other tuple can be added to the set without violating either condition 1 or condition 2. Tuple t_{11}

Result of Q_1

Customer	Item	NumItems	Date	Price	Weight (lb)	id
Peter	Lettuce	3	2020-01-03	0.99	2.5	t_{11}
Peter	Lettuce	3	2020-01-04	0.99	2.5	t_{12}

Result of Q_2

Customer	Item	NumItems	Date	id
Peter	Lettuce	3	2020-01-03	t_{21}
Peter	Oranges	1	2020-01-03	t_{22}
Peter	Lettuce	3	2020-01-04	t_{23}
Bob	Oranges	2	2020-01-04	t_{24}
Alice	Peanuts	3	2020-01-04	t_{25}

Result of Q_3

Item	Price	Weight (lb)	id
Lettuce	0.99	2.5	t_{31}

Result of Q_4

Item	Price	Weight (lb)	id
Lettuce	0.99	2.5	t_{41}
Oranges	2.49	3	t_{42}
Apples	3.99	6	t_{43}
Bok choy	1.99	1.5	t_{44}
Peanuts	3.99	2	t_{45}

Figure 2.5: Results of subqueries of query $Q_{cheapOrders}$

(t_{12}) was produced by joining tuples t_{21} (t_{23}) with tuple t_{31}.

$$
\begin{aligned}
\textsc{Lin}(Op_0, D, c_1) &= \langle\, \{t_{11}, t_{12}\} \,\rangle \\
\textsc{Lin}(Op_1, D, t_{11}) &= \langle\, \{t_{21}\}, \{t_{31}\} \,\rangle \\
\textsc{Lin}(Op_1, D, t_{12}) &= \langle\, \{t_{23}\}, \{t_{31}\} \,\rangle \\
\textsc{Lin}(Op_2, D, t_{21}) &= \langle\, \{o_1\} \,\rangle \\
\textsc{Lin}(Op_2, D, t_{23}) &= \langle\, \{o_3\} \,\rangle \\
\textsc{Lin}(Op_3, D, t_{31}) &= \langle\, \{t_{41}\} \,\rangle \\
\textsc{Lin}(Op_4, D, t_{41}) &= \langle\, i_1 \,\rangle
\end{aligned}
$$

Using this per operator lineage, we compute the lineage of the query based on Definition 14 through transitivity and get:

$$\textsc{Lin}(Q_{cheapOrders}, D, c_1) = \langle\, \{o_1, o_3\}, \{i_1\} \,\rangle$$

Note that lineage, even though it is declaratively defined in Definition 13 and Definition 14, is not syntax independent. Intuitively, this is true because it is defined over the syntactic structure of a relational algebra expression. Cui and Widom (2000a) also introduced an operational definition of lineage which consists of rules computing the per-operator lineage and this definition was proven to fulfill the conditions of Definition 13 and Definition 14. We show this definition (with slightly adapted notation) below. Recall that $[\![Q]\!]_D$ denotes the result of evaluating query Q over database D.

Definition 15 (Operational Definition of Operator Lineage)**.** Let D be a database, R_1 and R_2 be relations, and t a result tuple. The operational semantics of the lineage of the operators of $\mathcal{RA}^{agg}$ is defined below.

$$\begin{aligned}
\textsc{Lin}(\sigma_\theta(\mathsf{R}), D, t) &= \langle\, \{t\}\,\rangle \\
\textsc{Lin}(\Pi_A(\mathsf{R}), D, t) &= \langle\, [\![\sigma_{A=t}(\mathsf{R})]\!]_D\,\rangle \\
\textsc{Lin}(R_1 \bowtie R_2, D, t) &= \langle\, t[\textsc{Sch}(R_1)], t[\textsc{Sch}(R_2)]\,\rangle \\
\textsc{Lin}(\gamma_{G;aggr(A)}, D, t) &= \langle\, [\![\sigma_{G=t.G}(\mathsf{R})]\!]_D\,\rangle \\
\textsc{Lin}(R_1 \cup R_2, D, t) &= \langle\, [\![\sigma_{\textsc{Sch}(R_1)=t}(R_1)]\!]_D, [\![\sigma_{\textsc{Sch}(R_2)=t}(R_2)]\!]_D\,\rangle \\
\textsc{Lin}(R_1 - R_2, D, t) &= \langle\, \{t\}, [\![R_2]\!]_D\,\rangle
\end{aligned}$$

We invite the reader to verify that the lineage computed using the rules from Definition 15 is same as lineage computed using Definition 13 and Definition 14. In contrast to why-provenance, lineage is not syntax independent. For instance, Cheney *et al.* (2009) shows examples of equivalent queries that have different lineage.

Example 21 (Syntax Dependence of Lineage, adapted from Cheney *et al.* (2009), proof of Proposition 2.4)**.** Consider a relation $R(A, B)$ with instance $\{(1,2),(1,3)\}$ and the two queries shown below which are equivalent under set semantics.

$$Q_1 \coloneqq \mathsf{R} \qquad\qquad Q_2 \coloneqq \Pi_{A,B}(\mathsf{R} \bowtie \rho B \to C(\mathsf{R}))$$

The result tuple $(1,2)$ is derived from $(1,2)$ by query Q_1 while for query Q_2, this result can be produced in two ways: either by joining $(1,2)$ with itself or by joining $(1,2)$ with $(1,3)$. Thus,

$$\begin{aligned}
\textsc{Lin}(Q_1, D, (1,2)) &= \langle\, \{(1,2)\}\,\rangle \\
\textsc{Lin}(Q_2, D, (1,2)) &= \langle\, \{(1,2)\}, \{(1,2),(1,3)\}\,\rangle
\end{aligned}$$

Cheney *et al.* (2009) observed that if tuples are associated with identifiers that allow us to determine which relation they belong too, then lineage can be defined to be a subset of D instead of defining it as a sequences of sets of tuples. While this simplifies definitions, this transformation is lossy: the sequence of subsets in the lineage encodes some information about the structure of the query for which it was derived for. To be precise, we can "read" reconstruct from the lineage the sequence of leaf nodes of the relational algebra tree for the query. However, it is questionable whether this information can be used in any meaningful way, because the lineage does not encode any other information about the query, e.g., how these the relations accessed by the query were combined.

Cui *et al.* (2000) did extent lineage for bag semantics. Two alternative semantics were provided. One is basically the same as the one for set semantics which minor changes to the operational semantics to deal with bags. The second one is closer to why-provenance in nature in that it is a set of alternative derivations (the *derivation set* of a result). A derivation set for a set of n result tuples can be of size exponential in n. The reason for this exponential blow-up is that if a tuple t appears in the query result more than once, then the derivation set contains all of the possible ways to derive each duplicate. For instance, consider a relation $R(A, B)$ with instance $\{\{\,(1, 2), (1, 2)\,\}\}$. Evaluating a query $Q \coloneqq \Pi_A(R)$ over this instance under bag semantics yields $\{\{\,(1), (1)\,\}\}$. Each of the two duplicates of tuple (1) can be derived each of the duplicates of $(1, 2)$ leading to 4 possible combinations (which duplicate in the result is generated by which duplicate in the input). We will refrain from discussing the intricacies of this model further, because the semiring-based model for the provenance of queries under bag semantics that we will discuss in Section 2.3.4 is clearly superior. This model is based on sound principles and only requires polynomial space to encode the provenance of a query under bag semantics. Intuitively, the blow-up in the representation is based on considering the duplicates in a bag to have an inherent identity, but then not considering this identity when tracking provenance.

2.3.3 Where-Provenance

In addition to why-provenance, Buneman *et al.* (2001) introduced a second type of provenance called **Where-provenance**. Where-provenance, like why-provenance, also encodes data dependencies. However, in contrast to why-provenance, this model tracks copying of values instead of sufficiency of inputs. The model presented in Buneman *et al.* (2001) and ported to relational algebra in Cheney *et al.* (2009) tracking copying of values at the granularity of attribute values. We adopt the notation from Cheney *et al.* (2009) for identifying attribute values (with a minor modification). If t is a tuple from a relation R with schema $\mathbf{R} = (A_1, \ldots, A_n)$, then we use (R, t, A_i) to denote the value of attribute A_i of tuple t. We refer to such triples as cells. The where-provenance of a cell in the result of a query is then defined to be a subset of all cells from the input database.

Example 22 (Where-provenance)**.** Query $Q_{VisaOrderedItems}$ returns names of customers that have a Visa credit card and have at least ordered once. The value of cell $(Q, v_1, Name)$ (Peter) obviously was copied from $(Customers, c_1, Name)$. However, note that the join condition enforces that the value of attribute Name for any result tuple has to be equal to the value of attribute Customer of all Orders tuples it is derived from. That is, the where-provenance for $(Q, v_1, Name)$ should also include $(Orders, o_1, Customer)$, $(Orders, o_2, Customer)$, and $(Orders, o_3, Customer)$.

$$Q_{VisaOrders} \coloneqq \Pi_{Name}(\textsf{Orders} \bowtie_{Customer=Name} \sigma_{Card=Visa}(Customers))$$

Name	id
Peter	v_1
Alice	v_2

As illustrated in the example above, where-provenance should not just consider *direct copying* of values as in, e.g., projection, but also how *indirect copying* through equality constraints enforced by the query, e.g., in a join or selection, cause the values of a attribute in the query result to be equal to the values of an input attribute. The definition of where-provenance shown below is adapted from Cheney *et al.* (2009) using slightly different notation.

Definition 16 (Where-provenance). Let Q be a query with result schema $(A_1, \dots, A_n)$, D a database, t a tuple in $Q(D)$, and $A \in \{A_1, \dots, A_n\}$. Furthermore, let R_1 be a relation with schema $(B_1, \dots, B_m)$ and R_2 a relation with schema $(C_1, \dots, C_k)$. The where-provenance of value $t.A$ according to Q and D, denoted as $\textsc{Where}(Q, D, t, A)$, is defined as shown below.

$$\begin{aligned}
\textsc{Where}(\mathsf{R}, D, t, A) &= \{(R, t, A)\} \\
\textsc{Where}(\sigma_\theta(\mathsf{R}), D, t, A) &= \textsc{Where}((R), D, t, A) \\
\textsc{Where}(\Pi_U(\mathsf{R}), D, t, A) &= \textsc{Where}(\mathsf{R}, D, t, A) \\
\textsc{Where}(\rho_{E \to F}(\mathsf{R}), D, t, A) &= \textsc{Where}(\mathsf{R}, D, t, G) \textbf{ where } \; G[E \to F] = A \\
\textsc{Where}(R_1 \cup R_2, D, t, A) &= \textsc{Where}(R_1, D, t, A_i) \\
&\quad \cup \textsc{Where}(R_2, D, t, B_i) \textbf{ for } A = A_i \\
\textsc{Where}(R_1 \bowtie R_2, D, t, A) &= \\
&\begin{cases}
\textsc{Where}(R_1, D, t, A) & \text{if } A \in \textsc{Sch}(R_1) - \textsc{Sch}(R_2) \\
\textsc{Where}(R_2, D, t, A) & \text{if } A \in \textsc{Sch}(R_2) - \textsc{Sch}(R_1) \\
\textsc{Where}(R_1, D, t, A) & \text{if } A \in \textsc{Sch}(R_1) \cap \textsc{Sch}(R_2) \\
\quad \cup \textsc{Where}(R_2, D, t, A) &
\end{cases}
\end{aligned}$$

This definition of where-provenance is syntax dependent.

Example 23 (Syntax Dependence of Where-provenance). Consider the two queries shown below that are equivalent under set semantics.

$$\begin{aligned}
Q_1 &\coloneqq \Pi_{Customer,Item}(Orders) \\
Q_2 &\coloneqq \Pi_{Customer,Item}(Orders) \bowtie \Pi_{Customer}(Orders)
\end{aligned}$$

Let us compute the where-provenance of the Customer attribute of result tuple $(Peter, Lettuce)$. For query Q_1 this cell is copied from $(Orders, o_1, Customer)$ and $(Orders, o_3, Customer)$. The where-provenance for this cell according to query Q_2 additionally contains $(Order, o_2, Customer)$ because of the redundant self-join that matches o_1 and o_3 with o_2.

Bhagwat *et al.* (2004) extended this definition to be syntax independent, by merging the where-provenance of all queries that are equivalent to the input query.

Definition 17 (Syntax Independent Where-Provenance)**.** Consider a query Q, database D, and $t \in Q(D)$. Let A be an attribute in the schema of Q, then

$$\text{IW}\textsc{here}(Q, D, t, A) = \bigcup_{Q \equiv Q'} \text{W}\textsc{here}(Q, D, t, A)$$

The definition above cannot be used directly to compute IWHERE, because there are infinitely many queries that are equivalent to a query Q. For example, $Q \bowtie Q$, $Q \bowtie Q \bowtie Q$, … are all equivalent to Q under set semantics. As observed in Bhagwat *et al.* (2005) and Bhagwat *et al.* (2004), for any input query Q there has to exists a finite set of queries $Q_1, \ldots, Q_n$ such that the union of the where-provenance of these queries is equal to the syntax independent where-provenance. To see why this has to be the case note that the size of the set $\text{IW}\textsc{here}(Q, D, t, A)$ is limited by the size of D and, thus, has to be finite. Thus, there has to exist a number n such that for any set of $n + 1$ queries equivalent to Q at least two queries have the same where-provenance. Bhagwat *et al.* (2004) did present an algorithm that, given an input query Q, generates a set of queries such that the union of their where-provenance is equal to the syntax independent where-provenance of Q. The main idea behind the construction applied by this algorithm it to extend the input query with additional joins through which additional cells can be incorporated into the provenance. Intuitively, under syntax independent where-provenance, the provenance of a cell in the result of a query contains all input cells that have the same value as the output cell. For instance, for the result tuple n_1 from query $Q_{custnames}$ from Figure 2.2, $\text{IW}\textsc{here}(Q_{custnames}, D, n_1, Name)$ contains not just $(Customer, c_1, Name)$, the cell from which the value Peter is copied from by this query, but also all other occurrences of this value. In the example database from Figure 2.1 these are:

$$\{(Orders, o_1, Customer),$$
$$(Orders, o_2, Customer),$$
$$(Orders, o_3, Customer)\}$$

Several other forms of where-provenance have been considered in the literature. Glavic *et al.* (2013b) and Glavic (2010) presented a version of where-provenance called **Copy-Contribution Semantics** (C-CS) that tracks where-provenance at the granularity of tuples: the provenance of a result tuples contains all input tuples for which at least one (all) attribute values have been copied to this result tuple. Buneman *et al.* (2008) presented a type of where-provenance for nested relational calculus and nested relational algebra and for the nested update language (an update language for nested relations). Provenance annotations are represented as colors that are propagated through operations in this model. Buneman *et al.* (2008) did formalize semantic properties that should hold for where-provenance over nested data. A provenance semantics is *copying* if for any value-color pair in the output of an operation this value-color pair exists somewhere in the input. That is, provenance is attached to values as colors and transformations (queries and updates) may not break the association between a value and its color. Cheney *et al.* (2014) introduce an expressive provenance model that generates **traces** for the execution of a query. Where-provenance can be extracted from such traces. We will revisit this model in Section 2.6.

2.3.4 Provenance Polynomials and the Semiring Annotation Model

Why-provenance, lineage, and where-provenance all have in common that they track data dependencies and are based on sufficiency (and other properties). Next we introduce the semiring annotation framework (Green and Tannen, 2017; Karvounarakis and Green, 2012; Green *et al.*, 2007a) which is a general framework for expressing set semantics, bag semantics, various extensions of the relational model such as incomplete databases, and provenance through a general type of annotated relations. Provenance polynomials, the most general provenance model that can be expressed in this framework is the first model we introduce that is based on computability and, thus, also encodes how inputs were combined to produce a result. Furthermore, many other provenance models including why-provenance and lineage can be expressed in this model. This also enables these models to be compared in terms of their expressive power. An interesting outcome of the comparison is that while not specifically designed for this purpose, we will see that

$$k_1 \oplus_{\mathcal{K}} k_2 = k_2 \oplus_{\mathcal{K}} k_1 \qquad (\textbf{commutativity of addition})$$

$$k_1 \otimes_{\mathcal{K}} k_2 = k_2 \otimes_{\mathcal{K}} k_1 \qquad (\textbf{commutativity of multiplication})$$

$$(k_1 \oplus_{\mathcal{K}} k_2) \oplus_{\mathcal{K}} k_3 = k_1 \oplus_{\mathcal{K}} (k_2 \oplus_{\mathcal{K}} k_3) \qquad (\textbf{associativity of addition})$$

$$(k_1 \otimes_{\mathcal{K}} k_2) \otimes_{\mathcal{K}} k_3 = k_1 \otimes_{\mathcal{K}} (k_2 \otimes_{\mathcal{K}} k_3) \qquad (\textbf{associativity of multiplication})$$

$$k_1 \oplus_{\mathcal{K}} \mathbb{0}_{\mathcal{K}} = k_1 \qquad (\textbf{neutral element of addition})$$

$$k_1 \otimes_{\mathcal{K}} \mathbb{1}_{\mathcal{K}} = k_1 \qquad (\textbf{neutral element of multiplication})$$

$$k_1 \otimes_{\mathcal{K}} \mathbb{0}_{\mathcal{K}} = \mathbb{0}_{\mathcal{K}} \qquad (\textbf{annihilation by zero})$$

$$k_1 \otimes_{\mathcal{K}} (k_2 \oplus_{\mathcal{K}} k_3) = (k_1 \otimes_{\mathcal{K}} k_2) \oplus_{\mathcal{K}} (k_1 \otimes_{\mathcal{K}} k_3) \qquad (\textbf{multiplication distributes over addition})$$

Figure 2.6: Equational Laws of Commutative Semirings

why-provenance also encodes computability. In the following, we will first introduce $\mathcal{K}$-relations, the datamodel of the semiring annotation framework, and queries over this model. Afterwards, we will discuss provenance polynomials, the most general type of provenance expressible in this framework, and will show how some of the provenance types we have discussed so far can be expressed as $\mathcal{K}$-relations.

K-relations

The foundation of the semiring framework are $\mathcal{K}$**-relations**, which are relations where tuples are annotated with elements from an annotation domain K. What is specific to $\mathcal{K}$-relations is that the domain of annotations K has to be equipped with two binary operations $\oplus_{\mathcal{K}}$ and $\otimes_{\mathcal{K}}$ (addition and multiplication) and two distinguished elements $\mathbb{0}_{\mathcal{K}}$ and $\mathbb{1}_{\mathcal{K}}$ that are the neutral elements of $\oplus_{\mathcal{K}}$ and $\otimes_{\mathcal{K}}$, respectively. It is required that the mathematical structure $\mathcal{K} = (K, \oplus_{\mathcal{K}}, \otimes_{\mathcal{K}}, \mathbb{0}_{\mathcal{K}}, \mathbb{1}_{\mathcal{K}})$ is a **commutative semiring**. That is, these operations have to obey the equational laws shown in Figure 2.6. These operations will be used to define a query semantics for $\mathcal{K}$-relations that is independent of the choice of annotation domain. The rationale for requiring that the operations comply with the equational laws of commutative semirings will become clear in the next subsection.

Formally, $\mathcal{K}$-relations are functions that take a tuple as input and return the tuple's annotation from the annotation domain K. The distinguished zero element ($\mathbb{0}_{\mathcal{K}}$) of the semiring is used to annotate tuples that are not part of the relation. That is, all tuples with the same schema as the relation are mapped to an annotation. Thus, if the domain of an attribute is infinite (e.g., floating point numbers), then any $\mathcal{K}$-relation over such a domain would be infinite in size. To avoid having to deal with infinite relations, it is required that only finitely many tuples are annotated with non-zero elements from K. Technically, these relations are still infinite in size, but can be finitely represented by only explicitly listing tuples whose annotation is not $\mathbb{0}_{\mathcal{K}}$. To simplify the definition of $\mathcal{K}$-relations, we will follow Green *et al.* (2007a) and assume that there is a domain $\mathcal{U}$ that is used as the domain for all attributes.

Definition 18 (K-relations)**.** Let $\mathcal{K} = (K, \oplus_{\mathcal{K}}, \otimes_{\mathcal{K}}, \mathbb{0}_{\mathcal{K}}, \mathbb{1}_{\mathcal{K}})$ be a commutative semiring and $\mathcal{U}$ be a universal domain of values. An n-ary $\mathcal{K}$-relation R is a function $\mathcal{U}^n \rightarrow K$ such that the set $\{t \mid t \in \mathcal{U}^n \wedge R(t) \neq 0\}$ is finite.

Because $\mathcal{K}$-relations are functions it is customary to use $R(t)$ to denote the annotation associated with tuple t, i.e., the result of applying function R to t. Before discussing the semantics of queries over this model, let us first show how set and bag semantics as well as some extensions of the relational data model can be expressed as $\mathcal{K}$-relations through an appropriate choice of semirings.

- **Natural Numbers** ($\mathbb{N} = (\mathbb{N}, +, \cdot, 0, 1)$): the semiring of natural numbers with standard addition and multiplication can be used to model bag semantics by annotating each tuple with its multiplicity, i.e., the number of duplicates of this tuple that exist in the relation. Tuples that do not exist are annotated with 0 (zero duplicates of such a tuple exist).

- **Boolean Semiring** ($\mathbb{B} = (\{\top, \bot\}, \vee, \wedge, \bot, \top)$): the semring whose elements are the boolean constants true ($\top$) and false ($\bot$) with disjunction ($\vee$) as addition and conjunction ($\wedge$) as multiplication can be used to model set semantics. Tuples that exist are annotated with $\top$ and tuples that do not with $\bot$.

- **Possible Worlds Semiring** ($\mathbb{W} = (2^W, \cup, \cap, \emptyset, W)$): An incomplete database models uncertainty in data by encoding it as a set of deterministic databases called **possible worlds**. Intuitively, each world records one possible state of the real world and it is unknown which possible world corresponds to the actual state of the real world. Let us associate each possible world with an identifier and let W denote the set of all these identifiers. An alternative way to encode an incomplete database is then to annotate each tuple with the set of identifiers of the worlds it appear in. That is, the annotation of a tuple is an element from the powerset of W (2^W).

We limit the discussion to these three semiring here, because our main interest in semirings is their use in modeling data provenance. In Section 2.9 we will provide some literature references to work using $\mathcal{K}$-relations for different purposes.

Example 24 (Encoding Relational Data Models as $\mathcal{K}$-relations)**.** Figure 2.7 shows examples of set, bag, and incomplete relations (shown on the left) and their $\mathcal{K}$-relational encoding (shown on the right). All of these relations have a single attribute Name. The relation shown on the top is a set semantics relation with two tuples. In the $\mathcal{K}$-relational framework, set semantics can be encoded using the Boolean semiring. The two tuples that are part of this relation are annotated with $\top$. All other possible tuples of this relation are annotated with $\bot$. As mentioned before, we use the finite representation of $\mathcal{K}$-relations where all tuples not explicitly listed in the table are assumed to be annotated with $\mathbb{0}_{\mathcal{K}}$ (are not in the relation). Below this relation we show an example of a bag semantics relation which contains two duplicates of the tuple ($Peter$) and three duplicates of tuple ($Alice$). Using semiring $\mathbb{N}$, each tuple is annotated with its multiplicity, e.g., ($Peter$) is annotated with 2. Finally, on the bottom left we show an incomplete relation with two possible worlds w_1 and w_2. Alice exists in both worlds (we are

Set semantics with semiring $\mathbb{B}$

Name
Peter
Alice

Name	$\mathbb{B}$
Peter	$\top$
Alice	$\top$

Bag semantics with semiring $\mathbb{N}$

Name
Peter
Peter
Alice
Alice
Alice

Name	$\mathbb{N}$
Peter	2
Alice	3

Incomplete Databases with semiring $\mathbb{W}$

w_1
Name
Alice
Bob

w_2
Name
Peter
Alice

Name	$\mathbb{W}$
Bob	$\{w_1\}$
Alice	$\{w_1, w_2\}$
Peter	$\{w_2\}$

Figure 2.7: Examples how to encode set semantics, bag semantics, and incomplete databases as $\mathcal{K}$-relations – many different variants of the relational datamodel can be expressed using an appropriate choice of semiring to annotate tuples.

certain that Alice exists). Bob and Peter, however, only appear in one of the two worlds. Using semiring $\mathbb{W}$ for the set of worlds $\{w_1, w_2\}$ this incomplete relation is encoded by annotating each tuple with the set of world is appears in, e.g., $(Peter)$ only appears in world w_2 and, thus, is annotated with $\{w_2\}$.

Abusing notation, we will will sometimes use $\mathcal{K}$ to refer to a semiring as well as its domain.

Positive Relational Algebra Over K-relations

Defining a query semantics over a data model like $\mathcal{K}$-relations which allows many different types of relational semantics to be encoded as

$$
\begin{array}{rcl}
R_1 \cup R_2 = R_2 \cup R_1 & \quad & k_1 \oplus_{\mathcal{K}} k_2 = k_2 \oplus_{\mathcal{K}} k_1 \\
(R_1 \cup R_2) \cup R_3 = R_1 \cup (R_2 \cup R_3) & & (k_1 \oplus_{\mathcal{K}} k_2) \oplus_{\mathcal{K}} k_3 = k_1 \oplus_{\mathcal{K}} (k_2 \oplus_{\mathcal{K}} k_3) \\
R_1 \cup \emptyset = R_1 & & k_1 \oplus_{\mathcal{K}} \mathbb{0}_{\mathcal{K}} = k_1 \\
R_1 \times R_2 = R_2 \times R_1 & & k_1 \otimes_{\mathcal{K}} k_2 = k_2 \otimes_{\mathcal{K}} k_1 \\
(R_1 \times R_2) \times R_3 = R_1 \times (R_2 \times R_3) & & (k_1 \otimes_{\mathcal{K}} k_2) \otimes_{\mathcal{K}} k_3 = k_1 \otimes_{\mathcal{K}} (k_2 \otimes_{\mathcal{K}} k_3) \\
R_1 \times \{()\} = R_1 & & k_1 \otimes_{\mathcal{K}} \mathbb{1}_{\mathcal{K}} = k_1 \\
R_1 \times \emptyset = \emptyset & & k_1 \otimes_{\mathcal{K}} \mathbb{0}_{\mathcal{K}} = \mathbb{0}_{\mathcal{K}} \\
 & & \\
R_1 \times (R_2 \cup R_3) & & k_1 \otimes_{\mathcal{K}} (k_2 \oplus_{\mathcal{K}} k_3) \\
=(R_1 \times R_2) \cup (R_1 \times R_3) & & =(k_1 \otimes_{\mathcal{K}} k_2) \oplus_{\mathcal{K}} (k_1 \otimes_{\mathcal{K}} k_3)
\end{array}
$$

Figure 2.8: Relational algebra equivalences and corresponding semiring laws

annotation is challenging. One major advantage of the semiring model is that it defines a version of relational algebra that is defined for any semiring and that coincides with standard set and bag semantics for a appropriate choices of semirings. This semantics combines the annotations of the input tuples of an operator using the operations of the semiring to produce the annotation of an output tuple. Semiring addition is used for disjunctive use of inputs as in union and projection and multiplication is used for conjunctive use of inputs as is the case for join. The reader may wonder what informed the choice of commutative semirings as the annotation structure in $\mathcal{K}$-relations? Green *et al.* (2007a) observed that there exists a common core of equivalence laws that hold over relational algebra operators under many variants of the relational model including set semantics, bag semantics, incomplete databases, and several others. These laws are shown in Figure 2.8 on the left. Specifically, cross product and union are commutative and associative, the empty set is the neutral element of union, the cross product of a relation with the empty set returns the emptyset, the singleton relation containing the empty tuple is the neutral element of cross product[2], and cross product distributes over union. In Figure 2.8 we pair each

[2] Note that for incomplete databases, the neutral element is the incomplete relation that is the singleton set containing the empty tuple in every possible world.

relational algebra equivalence with a particular semiring law. Note that there is a one-to-one correspondence between each of these laws if we replace relations with semiring elements and replace union (cross product) with addition (multiplication). This correspondence informs the choice of semirings as the structures that govern how annotations are propagated from input tuples to query results. Since $\mathcal{K}$-relations are functions, a convenient way of defining the semantics of an algebra operator is to define the $\mathcal{K}$-relation that is the result of an operator in a point-wise manner by stating how the annotation of a result is computed from the annotations of tuples in the operators input(s). In fact, Green *et al.* (2007a) defined the positive relational relational algebra ($\mathcal{RA}^+$) over $\mathcal{K}$-relations in this fashion.

Definition 19 ($\mathcal{RA}^+$ over $\mathcal{K}$-relations)**.** For a semiring $\mathcal{K}$ and tuple t and condition θ, we use $\theta(t)$ to denote a function that returns $\mathbb{1}_{\mathcal{K}}$ if t fulfills the condition θ and $\mathbb{0}_{\mathcal{K}}$ otherwise. Let R, R_1, and R_2 be $\mathcal{K}$-relations and t with the same schema as the result of an algebra operator.

- **Rename:** $\rho_{A \to B}(R)(t) = R(t[B \to A])$
- **Projection:** $\Pi_U(R)(t) = \sum_{t=t'[U]} R(t')$
- **Selection:** $\sigma_\theta(R)](t) = R(t) \otimes_{\mathcal{K}} \theta(t)$
- **Natural Join:** $(R_1 \bowtie R_2)(t) = R_1(t[\mathbf{R}_1]) \otimes_{\mathcal{K}} R_2(t[\mathbf{R}_2])$
- **Union:** $(R_1 \cup R_2)(t) = R_1(t) \oplus_{\mathcal{K}} R_2(t)$

Example 25 (Queries over $\mathcal{K}$-relations)**.** Figure 2.9 shows the relations Customers and Orders from our running example as $\mathbb{B}$- (encoding set semantics), $\mathbb{N}$- (encoding bag semantics), and $\mathbb{W}$-relations (encoding an incomplete database with worlds $\{w_1, w_2, w_3, w_4\}$). Annotations are shown on right of tuples. We assume that each input tuple appears once under bag semantics (semiring $\mathbb{N}$) and have made some rather arbitrary choices of which possible worlds a tuple belongs to for the incomplete versions of these relations (semiring $\mathbb{W}$). All tuples shown here are assumed to exist under set semantics (are annotated with $\top$). Consider the evaluation of query $Q_{custWithLargeOrders}$ (Figure 2.9c)

Name	**Age**	**Card**	$\underline{\mathbb{B}}$	$\underline{\mathbb{N}}$	$\underline{\mathbb{W}}$
Peter	39	Visa	$\top$	1	$\{w_1, w_3\}$
Alice	25	AE	$\top$	1	$\{w_1, w_2, w_3\}$
Bob	25	Visa	$\top$	1	$\{w_1, w_3\}$
Astrid	26	Master	$\top$	1	$\{w_1, w_3\}$

(a) Customers

Customer	**Item**	**NumItems**	**Date**	$\underline{\mathbb{B}}$	$\underline{\mathbb{N}}$	$\underline{\mathbb{W}}$
Peter	Lettuce	3	2020-01-03	$\top$	1	$\{w_1\}$
Peter	Oranges	1	2020-01-03	$\top$	1	$\{w_2, w_3\}$
Peter	Lettuce	3	2020-01-04	$\top$	1	$\{w_3, w_4\}$
Bob	Oranges	2	2020-01-04	$\top$	1	$\{w_1, w_3\}$
Alice	Peanuts	3	2020-01-04	$\top$	1	$\{w_3\}$

(b) Orders

$$Q_{custWithLargeOrders} := \Pi_{Name}(\mathsf{Customers} \bowtie_{Name=Customer} Q_{largeOrders})$$
$$Q_{largeOrders} := \Pi_{Customer}(\sigma_{NumItems>3}(\mathsf{Orders}))$$

(c) Query $Q_{custWithLargeOrders}$

Customer	$\underline{\mathbb{B}}$	$\underline{\mathbb{N}}$	$\underline{\mathbb{W}}$
Peter	$\top$	2	$\{w_1, w_3, w_4\}$
Alice	$\top$	1	$\{w_3\}$

(d) Result of Subquery $Q_{largeOrders}$

Name	$\underline{\mathbb{B}}$	$\underline{\mathbb{N}}$	$\underline{\mathbb{W}}$
Peter	$\top$	2	$\{w_1, w_3\}$
Alice	$\top$	1	$\{w_3\}$

(e) Query Result

Figure 2.9: Example for $\mathcal{K}$-relational query semantics

which returns names of customers which have issued at least one order with three or more items. The result of this query is shown in Figure 2.9e. Figure 2.9d shows the result of the subquery $Q_{largeOrder}$ which returns the customers of orders with three or more items. This subquery first applies a selection to check that orders have three or more items. The annotations of Orders tuples for which the condition evaluates to true are multiplied with $\mathbb{1}_{\mathcal{K}}$ (retained unmodified) while the annotations of all other input tuples are multiplied with $\mathbb{0}_{\mathcal{K}}$. Thus, the later (e.g., the second order) will be annotated with $\mathbb{0}_{\mathcal{K}}$ in the result of the selection which means that these tuples are not in the result of the selection. The remaining tuples are then projected onto their Customer attribute resulting in the intermediate result shown in Figure 2.9d. The annotation of a tuple in the result of a projection is computed by summing up the annotations of input tuples projected onto this output. In our example, the result tuple ($Peter$) is derived from the first and third order. Thus, its annotation is the sum of the annotations of these two tuples. For semiring $\mathbb{B}$ (set semantics), we get $\top \vee \top = \top$ (the tuple exists). For bag semantics ($\mathbb{N}$), we get $1 + 1 = 2$, i.e., the tuple ($Peter$) appears twice in the result of $Q_{largeOrders}$. For semiring $\mathbb{W}$ (incomplete data), the tuple exists in worlds $\{w_1\} \cup \{w_2, w_3\} = \{w_1, w_2, w_3\}$.

The result of $Q_{largeOrders}$ is then joined with relation Customers. For instance, the first tuple of Customers joins with the first result tuple of $Q_{largerOrders}$. The annotations of these two tuples are multiplied. We get:

$$\top \wedge \top = \top \qquad 1 \cdot 2 = 2 \qquad \{w_!, w_3\} \cap \{w_1, w_3, w_4\} = \{w_1, w_3\}$$

Finally, this tuple is projected onto its Name attribute. There is only one tuple with Name equal to *Peter*. This output tuple, thus, is mapped to the same annotation as the input tuple from which it was produced. We invite the reader to verify that the annotations correctly encode the query result under set semantics (the two tuples exist), under bag semantics (the first result tuple appears twice and the second one appears once), and under incomplete query semantics (the first tuple is in the result in possible worlds w_1 and w_3 and the second tuple is in the result in possible world w_3).

Homomorphisms

An advantage of generalizing the relational data model and query semantics to semiring annotations is that it allows us to apply results from universal algebra to understand the relationship between extensions of the relational model. Of specific importance are **homomorphisms**. A homomorphism from a semiring $\mathcal{K}_1$ to a semiring $\mathcal{K}_2$ is a mapping from the domain of $\mathcal{K}_1$ to the domain of $\mathcal{K}_2$ which is compatible with the semiring structure.

Definition 20 (Semiring Homomorphisms)**.** Let $\mathcal{K}_1$ and $\mathcal{K}_2$ be semirings, a mapping $h : K_1 \rightarrow K_2$ is a homomorphism if for all $k_1, k_2 \in K_1$, we have:

$$\begin{aligned} h(k_1 \oplus_{\mathcal{K}_1} k_2) &= h(k_1) \oplus_{\mathcal{K}_2} h(k_2) \\ h(k_1 \otimes_{\mathcal{K}_1} k_2) &= h(k_1) \otimes_{\mathcal{K}_2} h(k_2) \\ h(\mathbb{0}_{\mathcal{K}_1}) &= \mathbb{0}_{\mathcal{K}_2} \\ h(\mathbb{1}_{\mathcal{K}_1}) &= \mathbb{1}_{\mathcal{K}_2} \end{aligned}$$

Green *et al.* (2007a) observed that semiring homomorphisms can be lifted to homomorphisms between $\mathcal{K}$-relations by applying the homomorphism to the annotation of each tuple of a relation:

$$h(R)(t) = h(R(t))$$

This idea can be extended to homomorphisms over databases in the obvious way: applying the homomorphism to each relation in the database. Since queries over $\mathcal{K}$-relations use the operations of $\mathcal{K}$ to compute the annotation of a query result, it follows that homomorphisms commute with queries (Green *et al.* (2007a), Proposition 3.5):

$$h(Q(D)) = Q(h(D))$$

As we will see in the following, homomorphisms are an essential tool for studying the relative informativeness of provenance models that can be expressed in the semiring annotation framework. Note that if for any query and database there exists a homomorphism $h : \mathcal{K}_1 \rightarrow \mathcal{K}_2$ then it is possible to compute the result of any query in $\mathcal{K}_2$ by evaluating the query in $\mathcal{K}_1$ and then applying h to derive the query result in $\mathcal{K}_2$ (using the equivalence shown above). If this is the case, then $\mathcal{K}_1$ is more informative than $\mathcal{K}_2$, because the $\mathcal{K}_1$ annotation of a query result contains sufficient information for computing the result in $\mathcal{K}_2$. This will become more clear in the following when we discuss concrete examples. In Section 3.1 we will revisit homomorphisms and show that homomorphism over relations annotated with a semiring encoding provenance can be used to implement view maintenance.

Provenance Polynomials

Now we are finally ready to introduce **provenance polynomials**, the most informative provenance model expressible in the semiring annotation model. We will show that provenance tracked using this semiring enjoys the computability property (Definition 9) for any other semiring (and, thus, set and bag semantics). The basic idea behind provenance polynomials is to represent provenance as symbolic expressions that record how the annotation of a query result tuple was computed by combining the annotations of input tuples using addition and multiplication. The annotations of input tuples are represented as variables. The elements of the provenance polynomial semiring are polynomials with natural number coefficients and exponents over these variables. We will $\mathbb{N}[X]$ to denote the set of all such polynomials for a set of variables X.

- **Provenance Polynomials** $\mathbb{N}[X] = (\mathbb{N}[X], +\cdot, 0, 1)$

Example 26 (Provenance Polynomials)**.** Figure 2.10 shows Customers and Orders as $\mathbb{N}[X]$-relations. Each tuple is annotated with a variable identifying the tuple, e.g., customer Bob is assigned variable x_3. Consider the annotations of the two result tuples of $Q_{custWithLargeOrders}$ shown in Figure 3.1d. The first tuple is annotated with $x_1 \cdot (y_1 + y_3)$. This annotation records that this tuple was produced by joining (multiplication), the tuple corresponding to customer Peter (x_1) with the first (y_1) and third tuple (y_3) of the Orders relation. Similarly, the second result tuple was produced by joining the tuple for Alice x_4 with the fifth tuple (y_5) of the Orders relation.

The semiring $\mathbb{N}[X]$ enjoys the important property that for any other semiring $\mathcal{K}$ and assignment $\mu : X \to K$ (mapping each variable to an element from $\mathcal{K}$), there exists a unique homomorphism $\textsc{eval}_\mu : \mathbb{N}[X] \to \mathcal{K}$ by applying μ to replace variables in polynomials with elements from $\mathcal{K}$ and then evaluating the resulting symbolic expressions in $\mathcal{K}$.

Definition 21 ($\textsc{eval}_\mu$)**.** Let $\mathcal{K}$ be a semiring and $\mu : X \to K$ a valuation from X to K. We define $\textsc{eval}_\mu : \mathbb{N}[X] \to \mathcal{K}$ as shown below.

Name	**Age**	**Card**	$\mathbb{N}[X]$
Peter	39	Visa	x_1
Alice	25	AE	x_2
Bob	25	Visa	x_3
Astrid	26	Master	x_4

(a) Customers

Customer	**Item**	**NumItems**	**Date**	$\mathbb{N}[X]$
Peter	Lettuce	3	2020-01-03	y_1
Peter	Oranges	1	2020-01-03	y_2
Peter	Lettuce	3	2020-01-04	y_3
Bob	Oranges	2	2020-01-04	y_4
Alice	Peanuts	3	2020-01-04	y_5

(b) Orders

$$\begin{aligned} Q_{custWithLargeOrders} &:= \Pi_{Name}(\mathsf{Customers} \bowtie_{Name=Customer} Q_{largeOrders}) \\ Q_{largeOrders} &:= \Pi_{Customer}(\sigma_{NumItems>3}(\mathsf{Orders})) \end{aligned}$$

(c) Query $Q_{custWithLargeOrders}$

Name	$\mathbb{N}[X]$
Peter	$x_1 \cdot (y_1 + y_3)$
Alice	$x_4 \cdot y_5$

(d) Query Result

Figure 2.10: Example $\mathbb{N}[X]$-relations

$$\begin{aligned} \text{EVAL}_\mu(k_1 + k_2) &= \text{EVAL}_\mu(k_1) \oplus_{\mathcal{K}} \text{EVAL}_\mu(k_2) \\ \text{EVAL}_\mu(k_1 \cdot k_2) &= \text{EVAL}_\mu(k_1) \otimes_{\mathcal{K}} \text{EVAL}_\mu(k_2) \\ \text{EVAL}_\mu(x) &= \mu(x) \end{aligned}$$

Intuitively, the fact that EVAL_μ is a semiring homomorphism means that $\mathbb{N}[X]$ generalizes the computation in any other semiring $\mathcal{K}$ by recording the structure of the computation without making any assumption of its semantics apart from requiring that it obeys the laws of commutative semirings.[3] Note that this property means that provenance polynomials fulfill our computability requirement, because we

[3] This result follows from the fact that $\mathbb{N}[X]$ is the *free object* in the *variety* of semirings. In universal algebra, a variety is the set of all algebraic structures

can evaluate a query in semiring $\mathbb{N}[X]$ and then derive the query result in any other semiring using $\textsc{eval}_\mu$.

Example 27 (From provenance polynomials to other semirings)**.** Let us apply $\textsc{eval}_\mu$ to derive the original set, bag, and incomplete database results of $Q_{custWithLargeOrders}$ (Figure 2.9e) from the $\mathbb{N}[X]$ result shown in Figure 3.1d. Consider the annotation of the first result tuple:

$$x_1 \cdot (y_1 + y_3)$$

To determine the result under set semantics we substitute the variables with the $\mathbb{B}$-annotations from Figure 2.9a and Figure 2.9b:

$$\mu(x_1) = \top \qquad \mu(y_1) = \top \qquad \mu(y_3) = \top$$

To compute the query result under set semantics we apply $\textsc{eval}_\mu$ to the polynomial to replace variables based on μ and evaluate the resulting expression using the addition ($\vee$) and multiplication ($\wedge$) operations of the semiring $\mathbb{B}$:

$$\begin{aligned}
&\textsc{eval}_\mu(x_1 \cdot (y_1 + y_2)) \\
=&\textsc{eval}_\mu(x_1) \wedge (\textsc{eval}_\mu(y_1 + y_2)) \\
=&\textsc{eval}_\mu(x_1) \wedge (\textsc{eval}_\mu(y_1) \vee \textsc{eval}_\mu(y_2)) \\
=&\top \wedge (\top \vee \top) = \top
\end{aligned}$$

For bag semantics, we replace variables with multiplicity 1 (each tuple in the input appears once in our example) and get:

$$\textsc{eval}_\mu(x_1 \cdot (y_1 + y_2)) = 1 \cdot (1 + 1) = 2$$

of a given signature σ (the number and arity of operations of the structure) that obey a set of equational laws. For example, semirings have signature $\sigma = (2, 2, 0, 0)$, because they have two binary operations (addition and multiplication) and two 0-ary operations (the constants $\mathbb{0}_\mathcal{K}$ and $\mathbb{1}_\mathcal{K}$) and obey the laws shown in Figure 2.6. The free object in a variety is a structure whose elements are congruence classes of symbolic expressions that combine a set of variables using the variety's operations with respect to the equational laws of the variety. For instance, for semirings and a set of variables $X = \{x_1, x_2, x_3\}$, $x_1 \cdot (x_2 + x_3)$ and $(x_1 \cdot x_2) + (x_1 \cdot x_3)$ are two symbolic expressions from the free semiring ($\mathbb{N}[X]$) that belong to the same congruence class, because of the distributivity law of semirings. For any free object over variables X, any valuation $\mu : X \to K$ into the domain of another algebraic structure can be extended to a homomorphism between algebraic structures by applying μ to the variables in the symbolic expression and then evaluating the resulting expression in the target structure.

That is, this tuple appears twice in the query result under bag semantics. Finally, for the incomplete database version of these relations, we replace variables as shown below.

$$\mu(x_1) = \{w_1, w_3\} \qquad \mu(y_1) = \{w_1\} \qquad \mu(y_3) = \{w_3, w_4\}$$

Applying $\textsc{eval}_\mu$ we get:

$$\textsc{eval}_\mu(x_1 \cdot (y_1 + y_2)) = \{w_1, w_3\} \cap (\{w_1\} \cup \{w_3, w_4\}) = \{w_1, w_3\}$$

Other Provenance Semirings

We now will discuss several other provenance semirings which, like provenance polynomials, encode provenance as symbolic expressions. The difference between $\mathbb{N}[X]$ and these other provenance semirings is that additional equivalences are assumed to hold. This means that polynomials (provenance expressions) that are not equivalent under $\mathbb{N}[X]$ may be equivalent under some of these semirings. Intuitively, this means that these semirings are less informative and can only correctly model the provenance for semirings $\mathcal{K}$ that fulfill these additional laws enforced by the provenance semiring. Interestingly, some of these semirings correspond to provenance models we have introduced earlier, thus, shedding light on their expressiveness and on which provenance model is the right choice for set and bag semantics.

Green (2011) and Green (2009) studied query equivalence and containment for $\mathcal{K}$-relations. Two queries are equivalent for semiring $\mathcal{K}$, if they return the same result on every $\mathcal{K}$-database.[4] In addition, to proving new complexity results, this work also investigated how the choice of semirings affects query equivalence. A result that is of specific interest for our study of provenance is that queries are equivalent under bag semantics ($\mathbb{N}$) if and only if they are equivalent under $\mathbb{N}[X]$. This means that $\mathbb{N}[X]$ is a syntax independent provenance model bag semantics.

[4] Containment is somewhat harder to define over annotated relations, because this requires a notion "smaller than" to compare the annotations two relation assign to a tuple. Green (2009) utilized the *natural order* of a semiring for this purpose which orders elements based on addition. An element $k_1 \in K$ is smaller than or equal to an element $k_2 \in K$ with respect to the natural order if there exists an element k_3 such that $k_1 \oplus_\mathcal{K} k_3 = k_2$.Kostylev *et al.* (2013) studied containment for $\mathcal{K}$-relations independent of the natural order.

$$R \cup R = R \qquad k \oplus_{\mathcal{K}} k = k \qquad \textbf{(idempotence of addition)}$$

$$R \bowtie R = R \qquad k \otimes_{\mathcal{K}} k = k \qquad \textbf{(idempotence of multiplication)}$$

$$R \cup \Pi_{\mathbf{R}}(R \bowtie S) = R \qquad k_1 \oplus_{\mathcal{K}} (k_1 \otimes_{\mathcal{K}} k_2) = k_1 \qquad \textbf{(absorption)}$$

Figure 2.11: Relational algebra equivalences that hold under set semantics, but not under bag semantics. For each equivalence, we show the corresponding semiring equivalence.

Note that in Cheney *et al.* (2009) it was claimed that provenance polynomials are syntax dependent. While queries that are equivalent under set semantics may not be equivalent under $\mathbb{N}[X]$-relational semantics, this is to be expected, because additional equivalences hold under set semantics that do not hold under bag semantics. We show these equivalences in Figure 2.11. Namely, join and union are idempotent, and the union of a relation R with the result of joining this relation with another relation is equal to R. As in Figure 2.8 we can identify equivalence laws over semiring operations that correspond to these equivalences. For example, idempotence of join and union correspond to idempotence of the multiplication and addition operations of the semiring. As we will see in the following, the semiring of positive boolean algebra expressions obeys precisely these additional equivalences and, thus, is the right choice of provenance model for set semantics. Provenance according to this semiring is exactly the same as minimal why-provenance. That is, this is another justification for the choice of minimal why-provenance as a provenance model for set semantics queries and another way to prove that minimal why-provenance is syntax independent for set semantics.

We now introduce provenance semirings $\mathsf{PosBool}[\mathsf{X}]$, $\mathsf{Why}[X]$, and $\mathsf{Which}[X]$ and explain why these semirings correspond to the minimal why-provenance, why-provenance, and lineage provenance models.[5] For each of these semirings it will be insightful to consider two equivalent

[5] The lineage provenance model only corresponds to $\mathsf{Which}[X]$ if we ignore the query syntax information encoded by this model as discussed in Section 2.3.2.

definitions: one as congruence classes of polynomials based on the semiring laws and additional equivalences and one that is a more direct way of defining this semiring. Figure 2.12, inspired by Green and Tannen (2017), shows how these provenance semirings (and additional provenance semirings that we do not discuss here) are derived from $\mathbb{N}[X]$ by enforcing additional equivalences. We also show which semirings have the same equivalences as bag and set semantics and which of the provenance models we have discussed earlier correspond to which semirings (shown as dashed boxes besides the semirings that encode these models).

- **Positive Boolean Algebra Semiring**
  $\mathsf{PosBool[X]} = (\mathsf{PosBool[X]}, \vee, \wedge, \bot, \top)$:
  The elements of the $\mathsf{PosBool[X]}$ are positive boolean formulas over a set of variables X. Addition is logical *or* (with neutral element $\bot$) and multiplication is logical *and* (with neutral element $\top$). Note that boolean algebra fulfills the three algebraic laws for set semantics shown in Figure 2.11, e.g., $k_1 \vee k_1 = k_1$. For the alternative definition of $\mathsf{PosBool[X]}$ as equivalence classes of polynomials, we apply all equivalence from Figure 2.11, e.g., $x_1 + (x_1 \cdot x_2)$ and x_1 are considered to be the same in this semiring.

- **Why-provenance Semiring**
  $\mathsf{Why}[X] = (2^{2^X}, \cup, \Cup, \emptyset, \{\emptyset\})$:
  The elements of this semiring are set of sets of tuples with union as addition and pair-wise union ($\Cup$) as multiplication. Addition and multiplication in this semiring are idempotent, but the absorption law does not hold, e.g., $\{w_1\} \cup (\{w_1\} \Cup \{w_2\}) = \{w_1, (w_1 \cup w_2)\} \neq \{w_1\}$. For the alternative definition, polynomials are equated based on idempotence of addition and multiplication, e.g., $x_1 + x_1 = x_1$. This semiring encodes why-provenance.

- **Which-provenance Semiring**
  $\mathsf{Which}[X] = (2^X \cup \{\bot\}, \cup_+, \cup_\times, \bot, \emptyset)$:
  The elements of this semiring are sets of variables (tuple identifiers) and a special element $\bot$ which denotes that a tuple does not exist. Operations $\cup_+$ and $\cup_\times$ are both regular set union, but differ in how

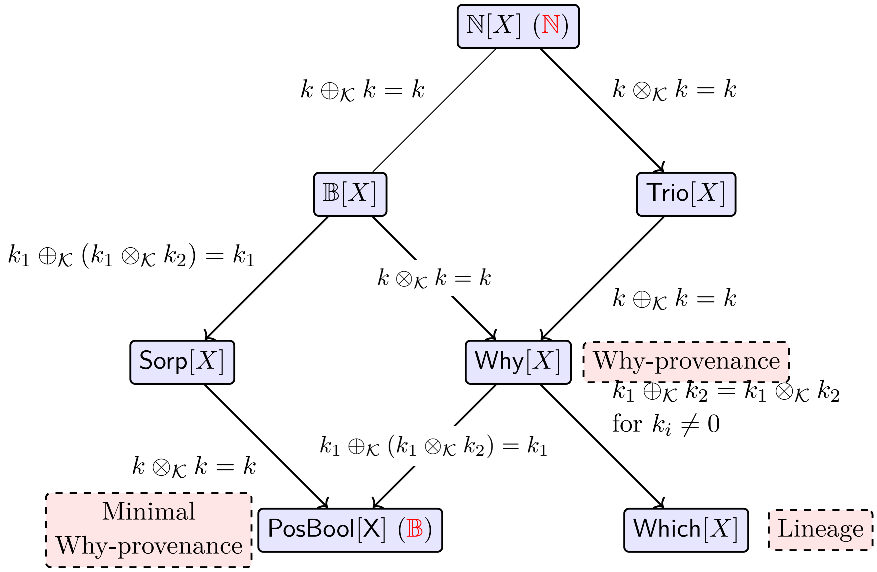

$\mathbb{N}[X]$ ($\mathbb{N}$)
$k \oplus_{\mathcal{K}} k = k$
$k \otimes_{\mathcal{K}} k = k$
$\mathbb{B}[X]$
Trio$[X]$
$k_1 \oplus_{\mathcal{K}} (k_1 \otimes_{\mathcal{K}} k_2) = k_1$
$k \otimes_{\mathcal{K}} k = k$
$k \oplus_{\mathcal{K}} k = k$
Sorp$[X]$
Why$[X]$
Why-provenance
$k_1 \oplus_{\mathcal{K}} k_2 = k_1 \otimes_{\mathcal{K}} k_2$ for $k_i \neq 0$
$k_1 \oplus_{\mathcal{K}} (k_1 \otimes_{\mathcal{K}} k_2) = k_1$
$k \otimes_{\mathcal{K}} k = k$
Minimal Why-provenance
PosBool[X] ($\mathbb{B}$)
Which$[X]$
Lineage

Figure 2.12: Provenance semirings organized by their informativeness from most informative (top) to least informative (bottom). Solid edges indicate that the starting point of the edge fulfills less equivalences as the end point of the edge. Each edge is labeled with the additional equivalence that is introduced. The semirings shown in red indicate the least informative provenance semiring that is sufficient to model the provenance of the semiring shown in red, i.e., query equivalence in these two semirings is the same. For instance, queries are equivalent in PosBool[X] iff they are equivalent under set semantics ($\mathbb{B}$). Dashed nodes indicate which provenance model a semiring corresponds to, e.g., Why$[X]$ encodes Why-provenance.

they deal with the special element $\perp$: $k_1 \cup_+ \perp = \perp \cup_+ k_1 = k_1$ while $k_1 \cup_\times \perp = \perp \cup_\times k_1 = \perp$. With the exception of how the zero element of the semiring is treated, addition is equivalent to multiplication in this semiring. Furthermore, both operations are idempotent and fulfill the absorption law. This semiring corresponds to the lineage provenance model if we ignore the order of leaves in the algebra tree of the query encoded by the lineage model.

Figure 2.12 shows several additional models that each correspond to a subset of the equivalences from Figure 2.11. Semiring $\mathbb{B}[X]$ and Trio$[X]$ are of no further interest to us here, but Sorp$[X]$ where addition is idempotent and absorptive, will be relevant for modeling the provenance of recursive queries in Section 2.5.

Where-provenance

Foster *et al.* (2008) extended the semiring framework to track the provenance for path queries over XML data.[6] The addition operation of the semiring is used to combine the annotations of multiple paths that match a query and multiplication is used to combine the annotations of the individual elements on a path. Using an encoding of relations as XML data and relational algebra queries as path queries, the approach can be used to compute a type of where-provenance for relational algebra expressions. Interestingly, this encoding of a relation as an annotated XML-document allows annotations to be placed on attribute-values, relations, and the database itself in addition to the annotation of tuples as supported by $\mathcal{K}$-relations.

In the next subsection we will review the PI-CS model from Glavic and Alonso (2009a) that was shown be equivalent to provenance polynomials for $\mathcal{RA}^+$ queries. Another provenance model of interest that is powerful enough to encode provenance polynomials is the provenance games model from Köhler *et al.* (2013) that defines provenance for Datalog queries through an interpretation of the evaluation of such queries as two player games. This model has the advantage of encoding the input query's structure and, thus, is advantageous for debugging queries. Since this model is intimately related to negation, we will delay its discussion until Section 2.4.

2.3.5 Perm Influence Contribution Semantics

Glavic (2010) and Glavic and Alonso (2009a) introduced a provenance model called **Perm influence contribution semantics** or **PI-CS** for short. The model represents provenance as bags of lists of tuples and was partially inspired by the lineage model. Note the difference to lineage which is a list of sets of tuples. This model has in common with lineage that it has a syntax-driven semantics which is shown to fulfill a set of declarative conditions. The provenance of a query result according to this model is a bag of **witness lists**. A witness list, is a list containing one element for each relation accessed by the query.

[6] This work did use unordered XML.

$$\mathcal{PI}(\mathsf{R}, D, t) = \{\{ < u >^n | \; u^n \in R \wedge u = t \}\}$$

$$\mathcal{PI}(\sigma_C(Q_1), D, t) = \{\{ < u >^n | \; u^n \in Q_1(D) \wedge u = t \}\}$$

$$\mathcal{PI}(\Pi_A(Q_1), D, t) = \{\{ < u >^n | \; u^n \in Q_1(D) \wedge u.A = t \}\}$$

$$\begin{aligned}\mathcal{PI}(Q_1 \bowtie_C Q_2, D, t) = \{\{ < u, v >^{n \times m} | \; & u^n \in Q_1(D) \wedge u = t.\mathbf{Q_1} \\ & \wedge v^m \in Q_2(D) \wedge v = t.\mathbf{Q_2} \}\}\end{aligned}$$

$$\begin{aligned}\mathcal{PI}(Q_1 \cup Q_2, D, t) = & \{\{ < u, \perp >^n | \; u^n \in Q_1(D) \wedge u = t \}\} \\ & \cup \{\{ < \perp, u >^n | \; u^n \in Q_2(D) \wedge u = t \}\}\end{aligned}$$

Figure 2.13: Compositional Semantics for PI-CS

Each element of a witness list is either a tuple from the corresponding relation or the special element $\perp$. The declarative conditions that define the PI-CS provenance for a query result include the ones defined for lineage with some adjustments to deal with bags, to account for the fact that the provenance is a set of lists of tuples instead of list of sets of tuples, and to deal with nested subqueries, set difference, and outer joins. Notably, this model was the first to support nested subqueries (Glavic and Alonso, 2009b).

We show the compositional semantics of PI-CS for $\mathcal{RA}^+$ in Figure 2.13.[7] The provenance semantics of each operator is defined using bag comprehensions. We use t^n to denote n duplicates of a tuple t, e.g., $t^n \in Q_1(D)$ means that tuple t appears in $Q_1(D)$ with multiplicity n.

Glavic *et al.* (2013b) showed that for positive relational algebra ($\mathcal{RA}^+$), the provenance polynomial for a query result can be extracted from its PI-CS provenance using the function h shown below. WLOG assume that Q accesses relations $R_1, \ldots R_m$. We use $w[i]$ to denote the i^{th} element of a witness list w.

[7] To be consistent with other provenance definitions discussed so far, we use a notation that is slightly different from Glavic (2010), Glavic *et al.* (2013b), and Glavic and Alonso (2009a).

$$h(\mathcal{PI}(Q,D,t)) = \sum_{w^n \in \mathcal{PI}(Q,D,t)} n \cdot \prod_{i=1}^{m} mon(w[i]))$$

$$mon(e) = \begin{cases} w[i] & \text{if } w[i] \neq \perp \\ 1 & \text{otherwise} \end{cases}$$

This definition has to be interpreted as follows: each witness list is translated into a monomial (a product of variables). $\perp$ elements are replaced with 1 (do not contribute to the result). The polynomial corresponding to a bag of witness lists is then the sum of the translation of each monomial multiplied by the witness lists multiplicity in the bag.

Example 28 (PI-CS Provenance)**.** As an example for PI-CS provenance, we compute the provenance of query $Q_{custWithLargeOrders}$ using this model. This query accesses two relations: Customers and Orders. Every witness list for the query consists of a tuple from the customer relation and a tuple of orders relations representing two tuples that got joined. We show $\mathcal{PI}(Q_{custWithLargeOrders}, D, t)$ for the two result tuples of this query below. The first result tuple was produced by joining x_1 with y_1 and x_1 with y_3. Thus, its PI-CS provenance contains two witness lists $< x_1, y_1 >$ and $< x_1, y_3 >$. The second tuple was produced by joining x_4 with y_5. Its provenance consists of a single witness list $< x_4, y_5 >$.

Name	$\mathcal{PI}$
Peter	$\{\{< x_1, y_1 >, < x_1, y_3 >\}\}$
Alice	$\{\{< x_4, y_5 >\}\}$

Using function h, we get back the provenance polynomials for these tuples:

$$h(\{\{< x_1, y_1 >, < x_1, y_3 >\}\}) = (x_1 \cdot y_1) + (x_1 \cdot y_3)$$
$$h(\{\{< x_4, y_5 >\}\}) = x_4 \cdot y_5$$

2.4 Non-monotone Queries, Negation, and Why-not Provenance

So far we have discussed how the existence of a tuple in the result of a query can be explained by the existence of tuples in the input database (and how such tuples were combined to produce the result). This is sensible for monotone queries. However, when dealing with non-monotone queries, e.g., queries with negation, the non-existence of inputs may be necessary for a producing a result. The notions of sufficiency and necessity (Definition 2 and Definition 3) fail to provide a full picture of why a result exists for non-monotone queries.

Example 29 (Influence of Missing Tuples). Set difference is an example of a non-monotone operator which checks for non-existence of tuples in its right input. Consider a two relations $R(A)$ and $S(B)$ and assume we want to compute $Q \coloneqq R - S$ under set semantics. We also show how to express this query as a Datalog rule below to point out the negation causing the non-monotonicity of this operation. Evaluating the query over the database instance shown below, a single result tuple (1) is returned. Obviously, (1) has to exist in R for (1) to be in the result of the query. Additionally, (1) has to not exist in S. That is, the existence of (1) in the query result depends on the fact that (1) does not exist in S.

$$Q \coloneqq R - S \qquad (\textbf{relational algebra})$$

$$Q(X) :- R(X), \neg S(X) \qquad (\textbf{Datalog})$$

R

A
1
2

S

B
2
3

Query Result

A
1

This example motivates the need to reason about both existing and missing tuples when dealing with non-monotone queries. Our notions of sufficiency and necessity / minimality are based on existing tuples alone and, thus, can not track such dependencies of outputs on missing inputs. Extending provenance models with support for general types of negation is challenging, because the number of tuples that could exist, but do not, can be very large or even infinite (if the domain of an attribute is infinite). For example, consider a relation $R(A, B, C, D, E)$

and assume that the data type of all attributes is 64-bit integers. Then there are $(2^{64})^5 \approx 2 \cdot 10^{100}$ tuples that could belong to R. Naturally, the vast majority of these tuples would not be part of this relation, even if it is contains, say, 100 billion (10^{11}) tuples. In the following, we will first discuss provenance models for non-monotone queries covering set difference, aggregation, Datalog with negation, and first-order logic. Afterwards, we will discuss how provenance models which support negation unify provenance and why-not provenance (explaining missing answers).

2.4.1 Set Difference

The lineage and PI-CS provenance models we have discussed before support set difference. However, both models are subject to limitations that are ultimately rooted in the fact that they are based on per-operator sufficiency and do not consider missing inputs as potential causes for a query result. Under PI-CS only tuples from the left input of a set difference are considered to belong the provenance. For Lineage, all tuples from the right input are in the provenance for every output of the set difference. This gross overestimation was motivated based on the recognition that non-existence of inputs can be required for the set difference to the produce a result. However, as we will see in the following, this approach can still produce incorrect results. The limitations of these provenance models are most obvious in queries with double negation (computing the difference of a relation and the result of another set difference operation), e.g., $R - (S - T)$. For such a query, tuples from relation T can contribute to the result by removing tuples from relation S which would have otherwise removed tuples from R.

Example 30 (Set Difference with Lineage and PI-CS). Below we show three tables R, S, and T. The query $Q := R - (S - T)$ returns a single result (1). Intuitively, this tuple is in the result, because (i) (1) exists in R and (ii) tuple (1) exists in T causing (1) to not be in the result of subquery $S - T$. Using the lineage and PI-CS models, the provenance of this query result is:

- **Lineage**: $\langle\, \{r_1\}, \emptyset, \emptyset\, \rangle$
- **PI-CS**: $\{\{ < r_1, \perp, \perp > \}\}$

Note that both models fail to detect the influence of t_1 on the query result.

R

A	id
1	r_1

S

B	id
1	s_1

T

C	id
1	t_1

Query Result

A
1

Monus Semirings

Geerts and Poggi (2010) extended the semiring model with support for set difference by introducing a monus operation. The resulting mathematical structures are called **monus semirings** or *m-semirings* for short. The monus operation $\ominus_{\mathcal{K}}$ is based on the **natural order** $\preceq_{\mathcal{K}}$ which orders the elements of a semiring based on the addition operation of the semiring. Specifically, $k_1 \preceq_{\mathcal{K}} k_2$ if there exists k_3 such that $k_1 \oplus_{\mathcal{K}} k_3 = k_2$. For some semirings, the natural order is a partial order. These semirings are called *naturally ordered.* A counter example is the semiring $\mathbb{Z}$ because $k_1 \preceq_{\mathbb{Z}} k_2$ for any k_1 and k_2 in $\mathbb{Z}$. For example, $1 \preceq_{\mathbb{Z}} -3$ and $-3 \preceq_{\mathbb{Z}} 1$ because $1 + -4 = -3$ and $-3 + 4 = 1$. The monus $k_1 \ominus_{\mathcal{K}} k_2$ is defined to be the smallest k_3 such that $k_2 \oplus_{\mathcal{K}} k_3 \succeq_{\mathcal{K}} k_1$. The monus operation is only well-defined if (i) the semiring is naturally ordered and (ii) for all $k_1, k_2 \in K$, the set $\{k_3 \mid k_1 \preceq_{\mathcal{K}} k_2 + k_3\}$ has a unique smallest element.

Definition 22 (Set Difference Over K-relations)**.** Let $\mathcal{K}$ be a m-semiring and R and S be two K-relations with the same arity. The set difference over $\mathcal{K}$-relations is defined as:

$$\begin{aligned}(\mathsf{R} - \mathsf{S})(t) &\coloneqq \mathsf{R}(t) \ominus_{\mathcal{K}} \mathsf{S}(t)\\ k_1 \ominus_{\mathcal{K}} k_2 &\coloneqq \min_{\preceq_{\mathcal{K}}}\{k_3 \mid k_1 \preceq_{\mathcal{K}} k_2 \oplus_{\mathcal{K}} k_3\}\end{aligned}$$

Semirings $\mathbb{N}$, $\mathbb{B}$, and the provenance polynomial semiring are all m-semirings. For semiring $\mathbb{N}$, the monus operation is the truncating minus ($k_1 \ominus_{\mathbb{N}} k_2 = k_1 \dot{-} k_2 = max(0, k_1 - k_2)$), for $\mathbb{B}$ the monus operation is $k_1 \ominus_{\mathbb{B}} k_2 = k_1 \wedge \neg k_2$, and for $\mathbb{N}[X]$ the coefficients of each monomial are subtracted using the truncating minus, for example

$$
\begin{aligned}
&(3 \cdot x_1^2 \cdot x_2 + 1 \cdot x_3 \cdot x_4^3 + 1 \cdot x_5) \ominus_{\mathbb{N}[X]} (1 \cdot x_1^2 \cdot x_2 + 5 \cdot x_3^5 + 3 \cdot x_5)) \\
=&(3 \dot{-} 1) \cdot x_1^2 \cdot x_2 + (1 \dot{-} 0) \cdot x_3 \cdot x_4^3 + (0 \dot{-} 5) \cdot x_3^5 + (1 \dot{-} 3) x_5) \\
=&2 \cdot x_1^2 \cdot x_2 + 1 \cdot x_3 \cdot x_4^3
\end{aligned}
$$

However, the provenance polynomial semiring looses its generality property when moving from semirings to m-semirings. Geerts and Poggi (2010) showed that a more general structure is needed whose elements are symbolic expressions involving the monus operation. We will use $\mathbb{N}_{\ominus}[X]$ to denote this m-semiring.

Example 31 (Set Difference with Monus Semirings)**.** Reconsider the database instance and query $Q \coloneqq R - (S - T)$ from Example 30. Below we show a bag semantics (m-semiring $\mathbb{N}$) and provenance (m-semiring $\mathbb{N}_{\ominus}[X]$) version of this database. Tuple (1) appears thrice in the result of the query under bag semantics. The provenance annotation of this tuple is $r_1 - (s_1 - t_1)$. As an example for the universality property of $\mathbb{N}_{\ominus}[X]$, we apply the homomorphism EVAL_{μ} corresponding to valuation μ $\mu(r_1) = 3$, $\mu(s_1) = 1$, and $\mu(t_1) = 2$ which models our example bag semantics instance to derive the bag query result from the $\mathbb{N}_{\ominus}[X]$ result:

$$
\begin{aligned}
\text{EVAL}_{\mu}(r_1 - (s_1 - t_1)) &= \text{EVAL}_{\mu}(r_1) \dot{-} (\text{EVAL}_{\mu}(s_1) \dot{-} \text{EVAL}_{\mu}(t_1)) \\
&= 3 \dot{-} (2 \dot{-} 1) = 3
\end{aligned}
$$

R

A	$\mathbb{N}$	$\mathbb{N}_{\ominus}[X]$
1	3	r_1

S

B	$\mathbb{N}$	$\mathbb{N}_{\ominus}[X]$
1	1	s_1

T

C	$\mathbb{N}$	$\mathbb{N}_{\ominus}[X]$
1	2	t_1

Query result

C	$\mathbb{N}$	$\mathbb{N}_{\ominus}[X]$
1	$3 \dot{-} (1 \dot{-} 2) = 3$	$r_1 - (s_1 - t_1)$

As pointed out in Amsterdamer *et al.* (2011c), there exists an equivalence that holds under both set and bag semantics relational algebra, but does not hold for all monus semirings. That is, m-semirings may be considered to be too general for modeling set difference. Other attempts to extend semirings with support for difference also do not fulfill all of these equivalences. Green *et al.* (2009) annotates tuples with integer numbers (the ring $\mathbb{Z}$) and defines difference using the subtraction operation of this ring. This semantics does differs from both set and bag semantics. Amsterdamer *et al.* (2011d) extended the semiring model with support for aggregation (more on this later) and applied this extension to define a semantics for set difference. This model does not correctly model difference for bag semantics.

The divergence of semantics for set difference over K-relations and the discussion in Amsterdamer *et al.* (2011c) point out a critical design decision that has to be made for any provenance model that is constructed based on a given set of algebraic equivalences: what is the set of query semantics that the provenance model should support and what is the set of equivalences that hold for all of these query semantics. For example, m-semirings do not enforce an equivalence for set difference holds under both set and bag semantics. The trade-off is between generality (less equivalences means that a larger class of query semantics is supported by the provenance model) and complexity of the provenance model and size of the representation (additional equivalences can be used to compress provenance).

2.4.2 Aggregation

Amsterdamer *et al.* (2011d) did present an extension of the semiring provenance model to support aggregation. A major challenge of dealing with aggregation over K-relations is that the values of result tuples (aggregation function results) may depend on the annotation of input tuples. For example, when calculating the sum over a column under bag semantics, the multiplicity ($\mathbb{N}$ annotation) of a tuple affects the result of the sum.

Example 32 (Aggregation over Semirings)**.** Query $\gamma_{sum(A);(}\mathsf{R})$ computes the sum over column A. Under bag semantics, the number of duplicates of the two input tuples affects the sum. Over the example database the sum is calculated as $3 \cdot 2 + 5 \cdot 3 = 21$. The result tuple (21) appears once (is annotated with 1). Now consider evaluating the query over a $\mathbb{N}[X]$-relation. We cannot assign a concrete value to the aggregation result, because the sum depends on what annotations are represented by variables x_1 and x_2.

R

A	$\mathbb{N}$	$\mathbb{N}[X]$
3	2	x_1
5	3	x_2

$\mathbb{N}$ **Query Result**

A	$\mathbb{N}$
21	1

$\mathbb{N}[X]$ **Query Result**

A	$\mathbb{N}[X]$
?	1

The example above motivates the need to encode how the result of an aggregation function is computed based on the annotations of tuples. For some semirings like $\mathbb{N}$, the result is a concrete value. To preserve the generality of provenance semirings like $\mathbb{N}[X]$ we cannot settle on concrete values, because the aggregation result will be different under different valuations. For instance, applying the valuation $\mu(x_1) = 1$ and $\mu(x_2) = 2$ over the database shown above, the sum evaluates to $3 \cdot 1 + 5 \cdot 2 = 13$. Amsterdamer *et al.* (2011d) addressed this issue by defining symbolic expressions that record how the result of an aggregation function is combined by aggregating the result of pairing values (from an attribute) with the annotation of the tuple they belong to. For instance, in the example we have to combine the annotation of the first tuple with its value in attribute A (3) and add the result to the combination of the annotation of the second with this tuple's A attribute value (5). If we can define an appropriate class of symbolic expressions that encode such computations, then we can specialize them (using semiring homomorphisms extended to these type of expressions) to compute a concrete aggregation result. Note that such expressions contain two types of operations:

- **Aggregation**: we need a binary operation to be able to aggregate values, e.g., addition corresponds to the sum aggregation function.
- **Pairing aggregation function domain values with semiring values**: we need a binary operation that takes an attribute value (a value from the domain of the aggregation function's input) and a semiring annotation and returns a value from the aggregation function domain.

In the example above, we assumed implicitly that the "pairing" function for integers (input to the sum aggregation function) and $\mathbb{N}$

annotations multiplies the two inputs. This is sensible for bag semantics, because if an input tuple has multiplicity x and its value in the attribute over which we are aggregating over is y, then the tuple contributes $x \cdot y$ to the sum. In Amsterdamer *et al.* (2011d) aggregation functions are formalized as **monoids**. A monoid $M = (M, +_M, \mathbb{0}_M)$ is a set M equipped with a binary operation $+_M$ that is associative, commutative, and has neutral element $\mathbb{0}_M$. Many common aggregation functions including **min**, **max**, and **sum** (the monoid for sum can also express count) are monoids. We show the definition for some aggregation functions below.

$$\mathsf{SUM} := (\mathbb{R}, +, 0) \quad \mathsf{MIN} := (\mathbb{R}, min, \infty) \quad \mathsf{MAX} := (\mathbb{R}, max, -\infty)$$

We would expect the "pairing" operation $*_{\mathcal{K},M} : \mathcal{K} \times M \rightarrow M$ for an aggregation monoid M and semiring $\mathcal{K}$ to fulfill several natural equivalences:

- If we union two sets of tuples (semiring addition), then the aggregation function result computed for the union of these sets should be the same as aggregating the aggregation results for the two sets. Similarly, if we have a tuple with an attribute value $m_1 +_M m_2$ annotated with k, then the result of aggregating over this tuple should be the same as aggregating over two tuples with annotation k and values m_1 and m_2, respectively. That is, the pairing operation should distribute over semiring and monoid addition.

$$(k_1 \oplus_{\mathcal{K}} k_2) *_{\mathcal{K},M} m = k_1 *_{\mathcal{K},M} m +_M k_2 *_{\mathcal{K},M} m$$

$$k *_{\mathcal{K},M} (m_1 +_M m_2) = k *_{\mathcal{K},M} m_1 +_M k *_{M,\mathcal{K}} m_2$$

- Tuples that do not exist (annotated with $\mathbb{0}_{\mathcal{K}}$) should not affect the result of aggregation.

$$\mathbb{0}_{\mathcal{K}} *_{\mathcal{K},M} m = \mathbb{0}_M$$

- Tuples whose value in the attribute we are aggregating over is the neutral element of the aggregation function ($\mathbb{0}_M$) should not contribute to the result of aggregation independent of the tuple's annotation.

$$k *_{\mathcal{K},M} \mathbb{0}_M = \mathbb{0}_M$$

- The pairing operation behaves as multiplication in the semiring, e.g., in $\mathbb{N}$ multiplication of natural numbers is both the multiplication operation of the semiring as well as the "pairing" operation.

$$(k_1 \otimes_{\mathcal{K}} k_2) *_{\mathcal{K},M} m = k_1 *_{\mathcal{K},M} (k_2 *_{\mathcal{K},M} m)$$

- Tuples annotated with the one element of the semiring, e.g., tuples with multiplicity 1 in $\mathbb{N}$, contribute exactly their attribute value to the aggregation result.

$$\mathbb{1}_{\mathcal{K}} *_{\mathcal{K},M} m = m$$

A mathematical structure that fulfills the laws shown above is called a $\mathcal{K}$-semimodule. For instance, the semimodule for sum aggregation (SUM) and bag semantics (semiring $\mathbb{N}$) is simply multiplication, the semimodule for min aggregation (MIN) and bag semantics ($\mathbb{N}$) is: return ∞ for annotation 0 and the input domain value m otherwise. The semimodule for min aggregation and set semantics ($\mathbb{B}$) returns m for annotation $\top$ and ∞ otherwise.

As mentioned above, for provenance semirings[8], it is not possible to define a sensible semimodule operation, because the aggregation result value cannot be determined unless a homomorphism is applied to map the input relation to a semiring like $\mathbb{N}$ where this operation is well-defined. Amsterdamer *et al.* (2011d) addressed this problem by defining an extended aggregation domain whose values are bags of elements from $K \times M$. Such elements are written as $k \otimes m$ instead of (k, m). Note that $K \times M$ with bag union and the empty bag is a monoid. That is, we can aggregate over such bags. Note that elements from the monoid M we started with are embedded in the new aggregation domain through the mapping $\iota : M \to K \otimes M$ defined as $\iota(m) = \mathbb{1}_{\mathcal{K}} \otimes m$. A semimodule over such symbolic expressions can then be defined as

[8] But also for some other non-provenance semirings

multiplying the semiring part of each pair in a bag by the semiring element we are pairing with:

$$k *_{\mathcal{K}\otimes M} \biguplus k_i \otimes m_i = \biguplus (k \otimes_{\mathcal{K}} k_i) \otimes m_i$$

Now we take the congruence of the elements from $\mathcal{K} \times M$ with respect to the monoid addition and semiring operations satisfying the semimodule laws. The resulting structure is denoted as $\mathcal{K} \otimes M$. As a notational convention, elements from $\mathcal{K} \otimes M$ are written as sums instead of bags, e.g., the bag $\{\{\, k_1 \otimes m_1, k_1 \otimes m_1, k_2 \otimes m_2 \,\}\}$ would be written as $k_1 \otimes m_1 +_{\mathcal{K}\otimes M} k_1 \otimes m_! +_{\mathcal{K}\otimes M} k_2 \otimes m_2$. For some semirings-monoid combinations, the monoid $\mathcal{K} \otimes M$ is isomorphic to M. That is, computing aggregations in $\mathcal{K} \otimes M$ is the same as aggregation in M. For example, this is the case for $\mathbb{N}$ and all of the aggregation monoids presented above. For such $\mathcal{K}$-M pairs, the operation $*_{\mathcal{K}\otimes M}$ behaves exactly like semimodule $*_{\mathcal{K},M}$. That is, an aggregation in $\mathcal{K} \otimes M$ corresponds directly to a concrete aggregation result value.

Any semiring homomorphism $h : \mathcal{K}_1 \rightarrow \mathcal{K}_2$ can be lifted to a monoid homomorphism $h^M : \mathcal{K}_1 \otimes M \rightarrow \mathcal{K}_2 \otimes M$ by applying h to the semiring part of the pairs in an $\mathcal{K}_1 \otimes M$-element:

$$h^M(\sum k_i \otimes m_i) = \sum h(k_i) \otimes m_i$$

To model the output of aggregation over $\mathcal{K}$-relations, Amsterdamer *et al.* (2011d) map the input relation to a $\mathcal{K}$-relation whose attribute values are elements from $\mathcal{K} \otimes M$ using ι and sum up input values using the addition operation of the monoid $\mathcal{K} \otimes M$. For aggregation without group-by, the result is an $\mathcal{K}$-relation with a single tuple annotated with $\mathbb{1}_{\mathcal{K}}$ (aggregation always return a single result tuple) whose attribute value is the symbolic aggregation result. This construction may seem opaque, but will become clear in the example shown below.

Example 33 (Aggregation over $\mathcal{K} \otimes M$-relations). Reconsider the $\mathbb{N}[X]$-relation and query $\gamma_{sum(A)}(R)$ from Example 32. To evaluate this aggregation we first use ι to lift the input from a $\mathcal{K}$-relation over numbers to a $\mathcal{K}$-relation over $\mathbb{N}[X] \otimes \mathsf{SUM}$. The result is shown below on the left. To calculate the sum over this lifted relation, we then use $*_{\mathbb{N}[X]\otimes\mathsf{SUM}}$ to pair a tuple's A value with its $\mathbb{N}[X]$ annotation and then sum up the resulting $\mathbb{N}[X] \otimes \mathsf{SUM}$ values using the addition operation of $\mathbb{N}[X] \otimes \mathsf{SUM}$ as shown below. Recall that $*_{\mathbb{N}[X]\otimes\mathsf{SUM}}$ is defined as multiplication of the semiring part of an $\mathcal{K} \otimes M$ element with the input annotation.

$$
\begin{aligned}
& x_1 *_{\mathbb{N}[X]\otimes\mathsf{SUM}} (1 \otimes 3) +_{\mathbb{N}[X]\otimes\mathsf{SUM}} x_1 *_{\mathbb{N}[X]\otimes\mathsf{SUM}} (1 \otimes 5) \\
& = x_1 \otimes 3 +_{\mathbb{N}[X]\otimes\mathsf{SUM}} x_2 \otimes 5
\end{aligned}
$$

Note that the result value in $\mathbb{N}[X] \otimes \mathsf{SUM}$ encodes the computational steps required to compute the sum in any other semiring. To illustrate this, let us recompute the query result in $\mathbb{N}$ using a valuation

$$
\mu(x_1) = 2 \qquad\qquad \mu(x_2) = 3
$$

which maps the variables of the $\mathbb{N}[X]$-relation to the $\mathbb{N}$ annotations from Example 32. Lifting homomorphism $\textsc{eval}_\mu$ to $\mathbb{N}[X] \otimes \mathsf{SUM}$ and applying the lifted homomorphism to the query result shown below, we get the result of the sum under bag semantics:

$$
\begin{aligned}
& \textsc{eval}_\mu^{\mathsf{SUM}}(x_1 \otimes 3 +_{\mathbb{N}[X]\otimes\mathsf{SUM}} x_2 \otimes 5) \\
= & \textsc{eval}_\mu(x_1) \otimes 3 +_{\mathbb{N}\otimes\mathsf{SUM}} \textsc{eval}_\mu(x_2) \otimes 5 \\
= & 2 \otimes 3 +_{\mathbb{N}\otimes\mathsf{SUM}} 3 \otimes 5
\end{aligned}
$$

Recall that for semiring $\mathbb{N}$ and monoid SUM, $\otimes$ behaves exactly like the semimodule $*_{\mathbb{N},\mathsf{SUM}}$ and $+_{\mathbb{N}\otimes\mathsf{SUM}}$ behaves exactly like $+_{\mathsf{SUM}}$. Thus, the symbolic expression corresponds to:

$$
\begin{aligned}
= & 2 *_{\mathbb{N},\mathsf{SUM}} 3 +_{\mathsf{SUM}} 3 *_{\mathbb{N},\mathsf{SUM}} 5 \\
= & 6 + 15 = 21
\end{aligned}
$$

R over domain $\mathbb{N}[X] \otimes \mathsf{SUM}$

A	$\mathbb{N}[X]$
$1 \otimes 3$	x_1
$1 \otimes 5$	x_2

Query Result

sum(A)	$\mathbb{N}[X]$
$x_1 \otimes 3 +_{\mathbb{N}[X]\otimes\mathsf{SUM}} x_2 \otimes 5$	1

Amsterdamer *et al.* (2011d) also considered aggregation with group-by. Aggregation with group-by is challenging if the group-by values are symbolic (produced by a previous aggregation step). In this case, the group membership of tuples is no longer fixed, but can vary under different valuations. A similar problem arises when applying a selection to filter the result of an aggregation operator, because the unknown aggregation function result determines whether a tuple would be filtered out or not. To be able to represent the result of such operations, the symbolic expressions from $\mathcal{K} \otimes M$ are extended with equality comparisons. We do not further expand on the formal definition of this extended symbolic expressions here, but instead provide an example. See Amsterdamer *et al.* (2011d) for the detailed construction.

Example 34 (Filtering Aggregation Results)**.** Continuing with Example 33, consider an extended query $\sigma_{sum(A)=15}(\gamma_{sum(A)}(R))$. Whether the single result tuple of the aggregation fulfills the condition $sum(A) = 15$ depends on the value of $sum(A)$. In semiring $\mathbb{N}[X]$, $sum(A)$ is a symbolic expression that does not correspond to a concrete number. For some valuations, the selection condition may hold while for others it may fail. As mentioned above, Amsterdamer *et al.* (2011d) resolves this issue by extending the domain of tuple annotations with comparisons between $\mathcal{K} \otimes M$ values. Semiring homomorphisms are lifted to these extended structure by applying them to the semiring values of an element. For semrings and monoid pairs where $\mathcal{K} \otimes M$ is isomorphic to M, the equalities are evaluated and are replaced with $\mathbb{1}_{\mathcal{K}}$ if the condition evaluates to true and with $\mathbb{0}_{\mathcal{K}}$ otherwise. The result of the selection is shown below. The tuple exists if the value of the tuple is equal to 15. This is reflected in the tuple's annotation.

Query Result

sum(A)	$\mathbb{N}[X]$
$x_1 \otimes 3 +_{\mathbb{N}[X]\otimes \mathsf{SUM}} x_2 \otimes 5$	$[x_1 \otimes 3 +_{\mathbb{N}[X]\otimes \mathsf{SUM}} x_2 \otimes 5 = 1 \otimes 15]$

Now let us apply the lifted homomorphism for the valuation shown below (the first does not exist and the second tuple has multiplicity 3).

$$\mu(x_1) = 0 \qquad\qquad \mu(x_2) = 3$$

Applying the lifted homomorphism the value of attribute $sum(A)$ evaluates to 15. Since this part works exactly as in Example 33 we omit the details.

$$
\begin{aligned}
&\textsc{eval}_{\mu}^{\mathsf{SUM}}([x_1 \otimes 3 +_{\mathbb{N}[X]\otimes\mathsf{SUM}} x_2 \otimes 5 = 1 \otimes 15]) \\
=&\begin{cases} 1 & \text{if } \textsc{eval}_{\mu}^{\mathsf{SUM}}(x_1 \otimes 3 +_{\mathbb{N}[X]\otimes\mathsf{SUM}} x_2 \otimes 5) = \textsc{eval}_{\mu}^{\mathsf{SUM}}(1 \otimes 15) \\ 0 & \text{otherwise} \end{cases} \\
=&\begin{cases} 1 & \text{if } \mu(x_1) \otimes 3 +_{\mathbb{N}\otimes\mathsf{SUM}} \mu(x_2) \otimes 5 = \mu(1) \otimes 15 \\ 0 & \text{otherwise} \end{cases} \\
=&\begin{cases} 1 & \text{if } 0 \otimes 3 +_{\mathbb{N}\otimes\mathsf{SUM}} 3 \otimes 5 = 1 \otimes 15 \\ 0 & \text{otherwise} \end{cases} \\
=&1
\end{aligned}
$$

The Lineage and PI-CS provenance models also support aggregation. However, these models only compute a sufficient subset of the input as the provenance for an aggregation result. That is, in contrast to the extended semiring model, these approaches do not enjoy the *computability* property for aggregation, because it is not possible to recompute the aggregation result based on the provenance alone without knowing the query.

2.4.3 Datalog with Negation (First-Order Logic Queries)

We now discuss provenance models for first-order logic queries, that is queries that are expressed as formulas in first order logic. Such queries may contain both existential and universal quantification as well as negation. An alternative syntax for such queries is $\textsc{Datalog}^{\neg}$ (Datalog with negation but no recursion).[9]

Semiring Provenance for FOL

Grädel and Tannen (2017) and Tannen (2017) did present an extension of the semiring framework for FOL model checking. In this framework,

[9] Datalog rules are existential in nature. A universal quantification can be expressed by rewriting it based on DeMorgan's rules: $\forall x : \phi(x) \Leftrightarrow \neg\exists x : \neg\phi(x)$.

$$\pi[\![R(\mathbf{x})]\!]_{\nu} = \pi(R(\nu(\mathbf{x}))) \qquad \pi[\![\neg\, R(\mathbf{x})]\!]_{\nu} = \pi(\neg\, R(\nu(\mathbf{x})))$$

$$\pi[\![x \,\mathbf{op}\, y]\!]_{\nu} = \text{ if } \nu(x) \,\mathbf{op}\, \nu(y) \text{ then } \mathbb{1}_{\mathcal{K}} \text{ else } \mathbb{0}_{\mathcal{K}} \quad \pi[\![\neg\varphi]\!]_{\nu} = \pi[\![\mathbf{nnf}(\varphi)]\!]_{\nu}$$

$$\pi[\![\varphi_1 \vee \varphi_2]\!]_{\nu} = \pi[\![\varphi_1]\!]_{\nu} + \pi[\![\varphi_2]\!]_{\nu} \quad \pi[\![\varphi_1 \wedge \varphi_2]\!]_{\nu} = \pi[\![\varphi_1]\!]_{\nu} \cdot \pi[\![\varphi_2]\!]_{\nu}$$

$$\pi[\![\exists x\, \varphi]\!]_{\nu} = \sum_{a \in A} \pi[\![\varphi]\!]_{\nu[x \mapsto a]} \qquad \pi[\![\forall x\, \varphi]\!]_{\nu} = \prod_{a \in A} \pi[\![\varphi]\!]_{\nu[x \mapsto a]}$$

Figure 2.14: Computing the annotation of a first-order logic formula φ based on a $\mathcal{K}$-interpretation π and valuation μ.

facts (grounded atoms) are annotated with elements from a semiring $\mathcal{K}$. Based on such a *$\mathcal{K}$-interpretation* π and a valuation μ of the free variables of a formula φ to values from a domain of values A, the formula evaluates to a value $\pi[\![\varphi]\!]_{\nu}$ from $\mathcal{K}$. The zero element of a semiring denotes false and non-zero elements denote various shades of truth. For instance, for semiring $\mathbb{B}$ the annotation of a formula is its truth value under classical first-order logic and under semiring $\mathbb{N}$ the annotation is the number of proof trees for the formula given the valuation. Of course, here we are mainly interested in provenance semirings and the fact that queries with negation can be expressed as FOL formula.

Tannen (2017) used an interesting trick to deal with negation. Instead of extending the semiring structure with a new operation that models negation, the input formula is transformed into *negation normal form* to restrict the use of negations to the atoms of the formula. A formula is in negation normal form, if negation only occurs in the form of negative facts. For a formula φ, $\mathbf{nnf}(\varphi)$ denotes the equivalent formula in negation normal form. Any FOL formula can be brought into negation normal form by pushing negation through operations using DeMorgan's rules, e.g., $\neg\exists x : \varphi \Leftrightarrow \forall d : \neg\varphi$ and $\neg(\varphi_1 \wedge \varphi_2) \Leftrightarrow \neg\varphi_1 \vee \neg\varphi_2$. For example, the negation normal form of the formula $\neg\exists x : \mathsf{marriedto}(Peter, x)$ which checks that nobody is married to Peter is $\mathbf{nnf}(\varphi_{notMarried}) = \forall x : \neg\, \mathsf{marriedto}(Peter, x)$. To deal with negative facts (missing tuples) in formulas, Tannen (2017) defined $\mathcal{K}$-interpretations to assign separate annotations to the positive and negative version of a fact. Not all possible such assignments are sensible. It can neither be the case that for a $\mathcal{K}$-interpretation π and fact $R(\mathbf{a})$ we have $\pi(R(\mathbf{a})) = \pi(\neg R(\mathbf{a})) = \mathbb{0}_{\mathcal{K}}$ (it is not possible for both $R(\mathbf{a})$ and its negation to be false) or $\pi(R(\mathbf{a})) \neq \mathbb{0}_{\mathcal{K}}$ and $\pi(\neg R(\mathbf{a})) \neq \mathbb{0}_{\mathcal{K}}$ (both $R(\mathbf{a})$ and its negation are true).[10]

Figure 2.14 shows the rules for computing the $\mathcal{K}$-annotation of a formula φ based on a valuation μ and $\mathcal{K}$-interpretation π. Conjunction and universal quantification are translated into semiring multiplication, disjunction and existential quantification are translated as semiring addition. This is sensible, because in semiring $\mathbb{B}$ multiplication is conjunction and addition is disjunction and quantification can be rewritten into disjunction (conjunction) by grounding the formula. Note that **op** denotes a comparison operator: either $=$ or $\neq$.

Provenance tracking for first-order formula has to take into account the dual nature of facts. Towards this goal, Grädel and Tannen (2017) and Tannen (2017) introduced a semiring structure whose elements are polynomials over two sets of variables: variables from X which are used exclusively to annotate positive facts and variables from $\bar{X}$ are exclusively used to annotate negative literals. Note that this is a rediscovery of an earlier approach (Damásio *et al.*, 2013) which applied this idea to track provenance for recursive Datalog queries with negation. We will discuss this approach in more detail in Section 2.5.2. For any variable $x \in X$, there exists a corresponding variable $\bar{x} \in \bar{X}$ and vice versa. For any interpretation using this semiring, we have to require that if x annotates a fact $R(\mathbf{a})$, then $\bar{x}$ can only annotate $\neg\, R(\mathbf{a})$ (and vice versa). The elements of the new provenance semiring are then defined to be equivalence class of polynomials from $\mathbb{N}[X \cup \bar{X}]$ based on the congruence $x \cdot \bar{x} = 0$. The resulting structure is denoted by $\mathbb{N}[X, \bar{X}]$. Intuitively, this congruence encodes the logic equivalence $R(\mathbf{a}) \wedge \neg\, R(\mathbf{a}) \equiv false$.

[10] We are oversimplifying here for educational purposes. Grädel and Tannen (2017) uses a more general type of consistency and completeness check and related it to properties of the semiring that is used.

A $\mathbb{N}[X, \bar{X}]$-interpretation π fulfills two purposes: assigning a truth value to each fact (and its negation) and decide for which facts we want to track provenance. Interestingly, it is also possible to leave the truth value of some facts undecided. Below, we show all feasible combinations for annotating $R(\mathbf{a})$ and $\neg R(\mathbf{a})$ in π and their meaning. If we annotate $R(\mathbf{a})$ with 1 (or 0), this corresponds to asserting the fact $R(\mathbf{a})$ (negated fact $\neg R(\mathbf{a})$), but not tracking provenance for it. Annotating $R(\mathbf{a})$ with a variable x and $\neg R(\mathbf{a})$ with 0 we assert that $R(\mathbf{a})$ is true and that we want to track provenance for this positive fact. Analog, annotating $\neg R(\mathbf{a})$ with a variable $\bar{x}$ and $R(\mathbf{a})$ with 0 we assert that $R(\mathbf{a})$ is false and that we want to track provenance for this negative fact. By setting $R(\mathbf{a}) = x$ and $\neg R(\mathbf{a}) = \bar{x}$, we leave the truth of $R(\mathbf{a})$ undecided. Note that $R(\mathbf{a}) = 0$ and $\neg R(\mathbf{a}) = 0$ lead to incompleteness, because it cannot be the case the both the fact and its negation are false. $R(\mathbf{a}) = 1$ and $\neg R(\mathbf{a}) = 1$ leads to inconsistency, because it cannot be the case that both a fact and its negation are true.

$\pi(R(\mathbf{a})) = 1$	$\pi(\neg R(\mathbf{a})) = 0$	(**true, no provenance**)
$\pi(R(\mathbf{a})) = 0$	$\pi(\neg R(\mathbf{a})) = 1$	(**false, no provenance**)
$\pi(R(\mathbf{a})) = x$	$\pi(\neg R(\mathbf{a})) = 0$	(**true, track provenance**)
$\pi(R(\mathbf{a})) = 0$	$\pi(\neg R(\mathbf{a})) = \bar{x}$	(**false, track provenance**)
$\pi(R(\mathbf{a})) = x$	$\pi(\neg R(\mathbf{a})) = \bar{x}$	(**undetermined**)

A $\mathbb{N}[X, \bar{X}]$-interpretation π with undetermined facts represents a set of possible worlds, one for each assignment of the undetermined facts to truth values. The annotation of a formula computed for such an interpretation encodes the provenance of the formula in each of these worlds. By applying a homomorphism to such an annotation that replaces the annotation of undetermined facts with one of four other options shown above, we get the provenance of the formula in this possible world. Thus, undetermined facts can be used for reverse reasoning: given a search space of models (determined by the choice of undetermined facts) find models that fulfill certain desirable properties. We will discuss such hypothetical reasoning in more detail in Section 3.2.

Example 35 (Dual Polynomial Provenance)**.** Consider the query $R - S$ over relations $\mathsf{R}(\mathrm{A})$ and $\mathsf{S}(\mathrm{B})$ and let us assume that the value domain is $A = \{1, 2, 3\}$. We can express this query as the following FOL formula:

$$\varphi_{R-S} \coloneqq \mathsf{R}(x) \wedge \neg \mathsf{S}(x)$$

Note that x is a free variable in this formula. All valuations for x for which φ_{R-S} evaluates to true (an annotation other than $\mathbb{0}_{\mathcal{K}}$) are the answers of the query. The $\mathbb{N}[X, \bar{X}]$-interpretation shown below encodes the instances of relations R and S where (i) $R(1)$ and $R(2)$ exists and we want to track provenance for these facts, (ii) $R(3)$ does not exist and we do not want to track provenance for this negative fact, and (iii) $S(1)$, $S(2)$, and $S(3)$ do not exist and we want to track provenance for these negative facts.

$$\begin{array}{llll} \pi(R(1)) = x_1 & \pi(\neg\, R(1)) = 0 & \pi(S(1)) = 0 & \pi(\neg\, S(1)) = \bar{y_1} \\ \pi(R(2)) = x_2 & \pi(\neg\, R(2)) = 0 & \pi(S(2)) = 0 & \pi(\neg\, S(2)) = \bar{y_2} \\ \pi(R(3)) = 0 & \pi(\neg\, R(3)) = 1 & \pi(S(3)) = 0 & \pi(\neg\, S(3)) = \bar{y_3} \end{array}$$

Consider the three possible valuations for x over A:

$$\mu_1(x) = 1 \qquad \mu_2(x) = 2 \qquad \mu_3(x) = 3$$

The annotations of the formula under these three valuations (query answers) are computed by multiplying (conjunction) the annotations assigned to the relevant R and S facts by the $\mathbb{N}[X, \bar{X}]$-interpretation π. For example, for query result (1) (the valuation μ_1), the annotation is $x_1 \cdot \bar{y_1}$, the result exists if $R(1)$ is true and $S(1)$ is false.

$$\pi[\![\varphi_{R-S}]\!]_{\mu_1} = x_1 \cdot \bar{y_1} \quad \pi[\![\varphi_{R-S}]\!]_{\mu_2} = x_2 \cdot \bar{y_2} \quad \pi[\![\varphi_{R-S}]\!]_{\mu_3} = 0 \cdot \bar{y_3} = 0$$

Provenance Games

Köhler *et al.* (2013) introduced **provenance games**, a graph-based provenance model for non-recursive Datalog queries with negation. This

$$\mathsf{Q}(X) :\!-\ \mathsf{R}(X,Y), \neg\, \mathsf{R}(Y,X) \qquad (r)$$

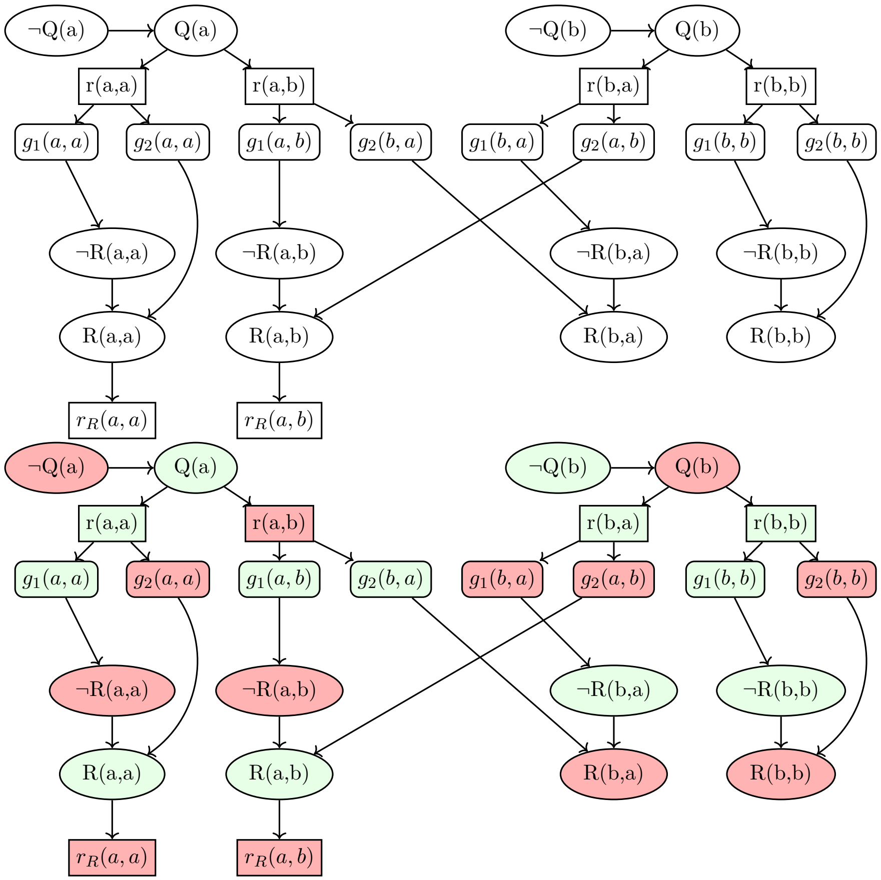


Figure 2.15: Example instantiated game (top) and solved game (bottom).

approach constructs a graph modeling the evaluation of the query. This type of graphs are called *instantiated games*. An instantiated game $G_{Q,D}$ for a query Q and database D can be interpreted as the game graph of a two player game (in the game-theoretic sense). Köhler *et al.* (2013) demonstrated that the existence of winning strategies for a player starting at one of the positions (nodes) of the game is intimately related to the success / failure of rules and goals as well as to the existence of tuples. By labeling each node in the graphs as winning (green) if there exists a winning strategy for a player starting at this node and loosing (red) otherwise, we get what we will refer to as a *solved game* $G^{\gamma}_{Q,D}$.[11] Some of the edges in a solved game correspond to moves in the game that are disadvantageous for the player taking such a move. It turns out that "good" moves that are "provenance-relevant" and "bad" moves should be removed from the provenance graph. The result of removing all edges corresponding to bad moves is called a provenance game and is denoted by $\Gamma_{Q,D}$. The provenance game for an idb tuple t is then the subgraph of $\Gamma_{Q,D}$ containing all nodes reachable from the tuple node for t.

Instantiated Games The nodes of such a graph represent the grounded atoms, goals, and rules of a Datalog program wrt. to the active domain of a database and the also encode which tuples exist in the database. For each idb and edb relation, the graph contains for each tuple that can be formed using values of the active domain of the database a positive and negated tuple node. The positive version represents the claim that the tuple exists and the negated version represents the claim that the tuple does not exist. Furthermore, the graph contains a node for each grounded rule instance and grounded versions of goals in a grounded rule's body. For a grounded rule there exists a node labelled $r(\mathbf{c})$ where $\mathbf{c}$ is the tuple of constants bound to the variables of the rule (sorted based on the order of occurrence of variables in the rule). A goal node labelled $g_i(\mathbf{c})$ denotes the i^{th} grounded goal of a grounded rule with arguments $\mathbf{c}$. Rules nodes are connected to goal nodes. Positive goals

[11] The type of two player games considered here permit draws. However, instantiated games FOL queries (non-recursive Datalog without recursion) do not have drawn positions.

are connected to negated tuples, negated goals are connected to positive tuples. idb tuples are connected to the rules that derive them. Negated tuples are connected to positive tuples. Furthermore, there are special nodes (fact nodes) for tuples of edb relations. Positive edb tuple nodes are connected to such fact nodes.

Example 36 (Instantiated Provenance Game)**.** Figure 2.15 shows the instantiated game the following query and database:

$$\mathsf{Q}(X) :- \mathsf{R}(X,Y), \neg R(Y,X) \qquad (r)$$
$$D := \{\mathsf{R}(a,a), \mathsf{R}(a,b)\}$$

Note that the active domain of the database and query is $\{a, b\}$. Thus, there are two tuple nodes for the idb predicate Q ($\mathsf{Q}(a)$ and $\mathsf{Q}(b)$). There are four possible ground instances of rule r ($r(a,a)$, $r(a,b)$, $r(b,a)$, and $r(b,b)$). The instantiated game contains two fact nodes representing the two tuples in the database. Grounded rule nodes are connected to the nodes for tuples they derive and to grounded goal nodes. For example, consider the following ground instance of rule r:

$$\mathsf{Q}(a) :- \underbrace{\mathsf{R}(a,a)}_{g_1(a,a)}, \underbrace{\neg R(a,a)}_{g_2(a,a)}$$

The tuple node $\mathsf{Q}(a)$ is connected to this grounded rule ($r(a,a)$). This node, in turn, is connected to nodes representing the two grounded goals in the body of this grounded rule instance. The first goal asserts the existence of edb tuple $\mathsf{R}(a,a)$ and is connected to negated tuple node $\neg R(a,a)$ while the second goal asserts that $\mathsf{R}(a,a)$ does not exist and is connected to tuple node $R(a,a)$.

The reason why positive goals are connected to negated tuples (and vice versa) will become clear when we discuss the interpretation of such a graph as a two-player game.

Solved Games An instantiated game can be interpreted as the game graph of a two player game. In a game graph the nodes represent *positions* of the game and the edges represent allowable *moves*. A *play* of such a game starts in one of the positions. The two players playing the game take turns picking moves (chose one of the outgoing edges of the current node). If the current position has no outgoing edges (valid moves), then the player whose turn it is looses the play. A position of a game is winning, if the player starting in this position has a winning *strategy*, i.e., if no matter what moves the opponent takes, the starting player can force a win. A position is loosing if there exist a winning strategy for the opponent (the player that does not start in this position). We can label the positions of a game as won (green) and lost (red) based on the existence of winning strategies. The reason for this excursion into game theory is that, as was demonstrated in Köhler *et al.* (2013), the following holds:

- **Tuples**: A tuple exists iff the position corresponding to the tuple is won.
- **Rules**: A grounded rule is successful iff the position corresponding to the rule is lost.
- **Goals**: A goal is successful if the position corresponding to the goal is won.

That is, in a precise sense a solved game encodes query evaluation.

Example 37 (Solved Game). Figure 2.15 (bottom) show the solved version of the instantiated game discussed in Example 36. Tuple node $\mathsf{Q}(a)$ is a winning position, that is this tuple belongs to the result of the query. Only one of two grounded rules deriving this result succeeds. This grounded rule $r(a, b)$ is lost in the solved game. The grounded rule is successful, because all of the gaols in its body succeeded (the corresponding goal nodes are won). The first goal succeeds because tuple $R(a, b)$ exists (is won). The second goal succeeds, because tuple $R(b, a)$ does not exist (is lost). The other grounded rule deriving this result $(r(a, a))$ failed, because it's second goal fails. This goal $(\neg R(a, a))$ failed because tuple $R(a, a)$ exists.

Note that the structure of the instantiated game, specifically how goals are connected to tuple nodes and the existence of fact nodes,

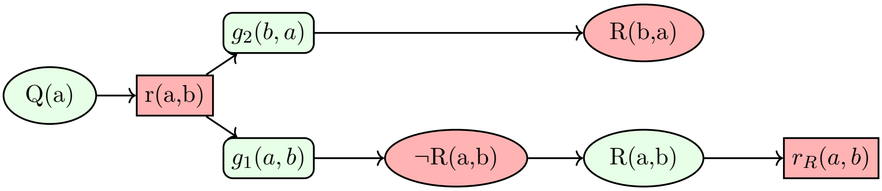


Figure 2.16: Subgraph of the solved provenance game encoding the provenance of $\mathsf{Q}(a)$. This tuple's existence is justified by the successful grounded rule instance $r(a,b)$. This grounded rule succeeds, because tuple $\mathsf{R}(a,b)$ exists and tuple $\mathsf{R}(b,a)$ does not exist.

is chosen such that correspondence between won / lost positions and existing / missing tuples holds.

Provenance Game Given a winning position, not all outgoing moves may guarantee a win. In particular, moving to a losing position is a "winning" move, while moving to a wining position is a "loosing" move, because the opponent can force a win from this position. Intuitively, the existence of winning moves is what causes a position to be winning and the lack of winning moves (all moves lead to winning positions) is what causes a position to be lost.

Example 38 (Provenance Game)**.** Continuing with Example 37, Figure 2.16 shows the provenance game for $\mathsf{Q}(a)$, i.e., the subgraph of the solved game from Figure 2.15 containing only nodes reachable from $\mathsf{Q}(a)$ through winning edges. For example, there are two edges starting in $\mathsf{Q}(a)$ corresponding to the two grounded rule instances $r(a,a)$ and $r(a,b)$. Only the later is successful (lost in the game). Moving from $\mathsf{Q}(a)$ is a loosing move, because node $r(a,a)$ is won and the subtree rooted at this node is not included in the provenance game for $\mathsf{Q}(a)$. Intuitively, this is what we would expect, because only successful rule derivations contribute to the existence of a query result. All nodes reachable from $r(a,b)$ are reachable through winning moves and, thus, are included in the provenance game for $Q(a)$.

Note that a provenance game explains both successful derivations of existing tuples as well as why the derivation of missing tuples failed. In Section 2.4.3 we will discuss how provenance games are used to unify provenance with why-not provenance (explaining why a tuple is not in the result of a query).

Condensing Provenance Games Note that some nodes in provenance games only exist to ensure the correspondence between winning strategies and the existence of tuples. Such nodes can be removed without loosing information if we forsake the interpretation of provenance graphs as games. Lee *et al.* (2018) and Lee *et al.* (2017) introduced a provenance graph model that encodes precisely the same information as provenance games, but avoids such redundancies. These graphs only have a single node for each tuple t instead of a negated and positive tuple node. Furthermore, the rule nodes are labelled as “won” (green) if the grounded rule is successful.

Lee *et al.* (2018) proved that $\mathbb{N}[X, \bar{X}]$-provenance can be extracted efficiently from provenance games and, thus, also from the provenance graph model of Lee *et al.* (2017). The main difference between such provenance graphs / games and FOL semiring annotation is (i) that these models are more verbose and (ii) that they encode which rules were used to derive a result. Thus, for use cases like debugging Datalog queries where the provenance should accurately reflect the derivation using the rules of the query, provenance games are preferable, while for use cases where the query syntax is irrelevant the FOL semiring is a superior choice.[12]

Why-not provenance and Queries with Negation

Provenance for queries explains how results of a query are derived from the query’s inputs. In contrast, why-not provenance explains why a query result is missing. Why-not provenance has sometimes also been

[12] Note that it is possible to track rule derivation in semirings by annotating each rule with an annotation and multiply monomials corresponding to a grounding of a rule r with the annotation of r. This further reduces the distinction between the two model. One remaining difference though is that the provenance game model “foctorizes” provenance based on the structure of the query. This is exploited in Lee *et al.* (2018) to choose a factorization that is worst-case optimal in size using the factorization techniques from Olteanu and Závodný, 2015.

referred to as explanations for missing answers. We will use the term why-not provenance.

Why-not provenance techniques can be classified (Herschel *et al.*, 2017) based on whether they explain a missing answer based on missing and existing input tuples (*instance-based* explanations) or identify which parts of the input query caused the failure to derive the missing answer (*query-based*).

Example 39 (Instance-based versus Query-based Explanations). Relations train$(from, to)$ and city$(name, pop)$ shown below store train connections and cities with their population. Query Q returns cities that have a population of at least 6 million people and are starting points of train connections. Given such a query, a user may be surprised to see that Chicago is not in the result. That is, the user is interested in an explanation for why $(Chicago) \notin Q(D)$. Instance-based approaches explain such missing answers based on the input database. For example, they may enumerate tuples that could be inserted into D such that evaluating Q over the resulting database returns the missing answer. In our example, inserting, e.g., $(Chicago, 10,000,000)$ into relation city would be sufficient for producing the missing answer. Query-based explanations assume that the input database is correct. The failure to derive the missing answer is explained by identifying parts of the query that can be held responsible for not producing the missing answer. In our example, one possible such explanation is that the selection's condition is too restrictive. For instance, if we change the condition to $pop > 2,000,000$, then Chicago would be returned by the query.

$$\Pi_{from}(\mathsf{train} \bowtie_{from=name} \sigma_{pop>6,000,000}(\mathsf{city}))$$

train

from	**to**
Chicago	New York
Chicago	Portland
New York	Boston

city

from	**pop**
Chicago	2,695,598
New York	8,336,817
Boston	692,600
Portland	654,741

Q

from
New York

Traditionally, provenance and why-not provenance have been studied independently of each other. However, as observed in Köhler *et al.* (2013), for queries with general negation there is no clear distinction between the two problems. Explaining why a t is not in the result of a query Q is equivalent to asking why t is in the result of the complement $Q^{\mathcal{C}}$ of the query. The complement $Q^{\mathcal{C}}$ of a query Q returns all tuples of the schema of Q that are not returned by Q. Let us use $\textsc{Tup}_{Q,\mathcal{U}}$ to denote all tuples over domain $\mathcal{U}$ with the same arity as Q, i.e., all tuples that could be produced by Q. For the complement $Q^{\mathcal{C}}$ or Q and every database D we require that:

$$\forall t \in \textsc{Tup}_{Q,\mathcal{U}} : t \in Q^{\mathcal{C}}(D) \Leftrightarrow t \notin Q(D)$$

Note that the complement of a query is only has finite results over finite domains. An alternative to defining the complement over some universal domain $\mathcal{U}$ is to require that only values that exist in the database or query are considered as viable attribute values for tuples. Note that this is the so-called active domain of the database $\textsc{adom}(D)$. Similar to the notation above we use $\textsc{Tup}_{Q,\textsc{adom}(D)}$ to denote all tuples with the same schema as the result of query Q with values from the active domain of the database D. The alternative definition of the complement of a query is:

$$\forall t \in \textsc{Tup}_{Q,\textsc{adom}(D)} : t \in Q^{\mathcal{C}}(D) \Leftrightarrow t \notin Q(D)$$

Note that the complement of a query Q from a class of queries $\mathcal{C}$ may not be expressible in this class. For example, the complement of positive relational algebra queries ($\mathcal{RA}^{+}$) is not expressible in $\mathcal{RA}^{+}$, because it requires the use of negation.

Example 40 (Complement of a Query). Consider the Datalog query shown below that returns cities that are starting points of train connections over a database with a single edb relation $\mathsf{train}(\mathit{from}, \mathit{to})$. The complement of this query can be computed by negating Q. However, this would result in an unsafe rule. To ensure safety we have to "guard" the negated atom $\mathsf{Q}(X)$ by ensuring that X appears positively in the body. Towards this goal we use an unary predicate adom that contains all values from the activate domain of the database. This is computed by projecting the edb relation on each of its attributes.

$$\begin{aligned}
\mathsf{Q}(X) &:- \mathsf{train}(X, Y)\\
\mathsf{Q}^C(X) &:- \mathsf{adom}(X), \neg\, \mathsf{Q}(X)\\
\mathsf{adom}(X) &:- \mathsf{train}(X, Y)\\
\mathsf{adom}(X) &:- \mathsf{train}(Y, X)
\end{aligned}$$

Evaluating these queries over the edb instance shown below, Q returns Chicago and New York (these cities are starting points of train connections) and $Q^{\mathcal{C}}$ returns Portland and Boston (these cities are not starting points of train connections).

train

from	**to**
Chicago	New York
Chicago	Portland
New York	Boston

Q

X
Chicago
New York

$Q^{\mathcal{C}}$

X
Portland
Boston

Based on the realization that why-not provenance and why-provenance are two sides of the same coin for provenance models that support classes of queries that are closed under taking complements (which requires support for negation), we can use provenance models for such classes of queries to explain missing answers. However, instance-based approaches for missing answer can track dependencies on missing input tuples. That is, we have to use provenance models, such as the provenance games and dual polynomials model, that can track missing inputs as dependencies.

Example 41 (Explaining Missing Answers with Provenance)**.** Using provenance games (or the equivalent graph model from Lee *et al.* (2018) and Lee *et al.* (2017)) or the FOL semiring approach, we can track the provenance of missing answers. In this example, we will show both the FOL semiring approach and show provenance graphs according to Lee *et al.* (2018). The boolean query shown below as a FOL formula and as

train

from	**to**
Portland	Chicago
Boston	Portland

train (Dual Polynomials)

$$\begin{aligned}
\pi(\mathsf{train}(Portland, Portland)) &= 0 & \pi(\neg\mathsf{train}(Portland, Portland)) &= 1\\
\pi(\mathsf{train}(Portland, Chicago)) &= 1 & \pi(\neg\mathsf{train}(Portland, Chicago)) &= 0\\
\pi(\mathsf{train}(Portland, Boston)) &= 0 & \pi(\neg\mathsf{train}(Portland, Boston)) &= 1\\
\pi(\mathsf{train}(Chicago, Chicago)) &= 0 & \pi(\neg\mathsf{train}(Chicago, Chicago)) &= \bar{x}_2\\
\pi(\mathsf{train}(Chicago, Boston)) &= 0 & \pi(\neg\mathsf{train}(Chicago, Boston)) &= \bar{x}_3\\
\pi(\mathsf{train}(Boston, Chicago)) &= 0 & \pi(\neg\mathsf{train}(Boston, Chicago)) &= 1\\
\pi(\mathsf{train}(Boston, Boston)) &= 0 & \pi(\neg\mathsf{train}(Boston, Boston)) &= 1
\end{aligned}$$

Figure 2.17: Example train connections and $\mathbb{N}[X, \bar{X}]$-interpretation for computing the why-not provenance explaining why Chicago has no outgoing train connections

Datalog query returns true if there are no train connections starting in Chicago. For simplicity, we assume that the active domain of the database can be accessed through a predicate adom. Recall that for the FOL semiring approach we need to first translate the formula φ into negation normal form (**nnf**), because this approach only supports negation at the fact level.

$$\begin{aligned}
\varphi &:= \neg\exists x : \mathsf{train}(Chicago, x)\\
\mathbf{nnf}(\varphi) &:= \forall x : \neg\mathsf{train}(Chicago, x)
\end{aligned}$$

$$Q() :- \mathsf{adom}(X), \mathsf{train}(Chicago, X) \qquad (r_1)$$

Figure 2.17 shows an example instance of the train relation. Assuming that Portland, Chicago, and Boston are the only cities, we can construct a $\mathbb{N}[X, \bar{X}]$-interpretation of all positive and negated facts. Here we assign all facts a truth value, but only track negated facts for train connections starting in Chicago by assigning them negated provenance tokens ($\bar{x}_1$,

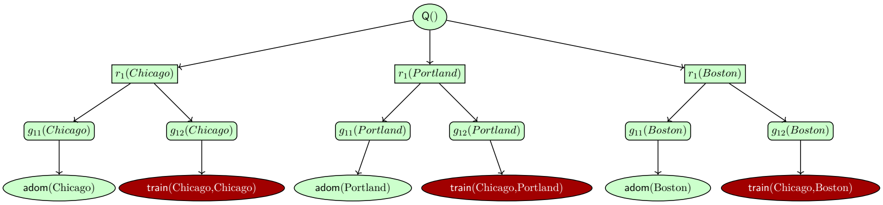


Figure 2.18: Provenance graphs explaining why there are no outgoing train connections from Chicago.

$\bar{x_2}$, and $\bar{x_3}$). Using this interpretation the annotation for the formula φ is derived as follows:

$$\begin{aligned}
&\pi[\![\mathbf{nnf}(\varphi)]\!]_\nu \\
=&\prod_{c \in \{Chicago, Portland, Boston\}} \pi(\neg\mathsf{train}(Chicago, c)) \\
=&\pi(\neg\mathsf{train}(Chicago, Chicago)) \\
&\cdot \pi(\neg\mathsf{train}(Chicago, Portland)) \\
&\cdot \pi(\neg\mathsf{train}(Chicago, Boston)) \\
=&\bar{x_1} \cdot \bar{x_2} \cdot \bar{x_3}
\end{aligned}$$

The result can be interpreted as follows: there are no outgoing train connections from Chicago as long as all three possible such train connections do not existing the database. Figure 2.18 shows the provenance graph according to Lee *et al.* (2018) for the Datalog version of this query. Recall the green (light) nodes are successful goal and rules (existing tuples) and red (dark) nodes are failed rules and goals (missing tuples). The query returns true, because there are three grounded versions of rule r_1 (with $X = Chicago$, $X = Portland$, and $X = Boston$) that successfully derive this result. Each of these grounded rule instances is successful, because both of its goals are successful. The first goal of each of these grounded rules is successful, because the binding for X exists in the active domain (adom). The second goals are successful, because the corresponding outgoing train connection does not exist. Observe the differences between these two provenance models. The FOL semiring expression is much more concise, but the provenance graph provides additional information about how the rule of the query has derived the result.

While such provenance models are expressive enough to explain missing answers, directly applying them to explain missing answers is not practically feasible. The reason is that, as mentioned above, the number of possible tuples is typically very large. Approaches for why-not provenance address this issue by using compact representations of provenance such as pattern-based summaries (Lee *et al.*, 2020) or constraint-based symbolic models (Herschel and Hernandez, 2010; Herschel *et al.*, 2009), or by just returning part of the provenance (Wu *et al.*, 2014; Wu *et al.*, 2013). We will discuss such representations and how they are computed in detail in Section 4.1 and Section 4.3. Reconsider Example 39 and let us assume that any natural number below 100 million is viable as a value for attribute pop (a city's population). Then inserting any tuple $(Chicago, c)$ for $6,000,000 < c < 100,000,000$ into relation city would cause $(Chicago)$ to appear in the query result. This set of tuples can be compactly encoded as a *C-table* (Imieliński and Lipski Jr, 1984), i.e., a relation where attribute values can be constants and variables from a set Σ and each tuple is associated with a so-called *local condition* which is a logical formula that constraints under which conditions the tuple exists.

$$(Chicago, x) \textbf{ for } x > 6,000,000 \wedge x < 100,000,000$$

$$(\textbf{Encoding why-not provenance compactly using constraints})$$

This type of encoding of the why-not provenance has been used by, e.g., Herschel and Hernandez (2010) and Herschel *et al.* (2009).

The causality framework from Meliou *et al.* (2010) can also be used to compute which missing inputs are causes for missing query answers by slightly adapting the definition of causes as shown below. This definition assumes that a set of possible missing input tuples D^n with $D^n \cap D = \emptyset$ is provided by the user.[13]

[13] D^n is called the set of endogenous tuples. For the positive case (called Why-so Meliou *et al.* (2010)), the user can decide which tuples from the input database should be consider as possible causes (are *endogenous*) and which ones should not be considered as causes (are *exogenous*).

Definition 23 (Actual Causes (Missing Answers))**.** A tuple $t_{cause} \notin D$ is an actual cause for a tuple $t \notin Q(D)$, if there exists $\Gamma \subseteq D^n - \{t_{cause}\}$, called a contingency, such that the following conditions hold:

$$t \notin (D \cup \Gamma)$$
$$t \in (D \cup \Gamma \cup \{t_{cause}\})$$

Reconsider scenario from example 40. We are interesting in knowing why there are no outgoing train connections from Chicago. We consider the missing train connections $\mathsf{train}(Chicago, Portland)$ and $\mathsf{train}(Chicago, Boston)$ as endogenous and all other missing tuples as exogenous. Both of the endogenous tuples are counterfactual (and, thus, also actual) causes for the empty result tuple (the query is boolean). That means, that the query returns true (the empty tuple) as long as one of these two tuples is inserted into the input database.

2.5 Recursion and Iteration

In this section, we discuss provenance models that support recursion or iterative queries. The challenge dealing with recursion is that under certain provenance semantics, recursive queries can lead to infinite provenance, because there may exist infinitely many derivations of a result. We will use Datalog to express recursive queries and will discuss semiring provenance for recursive Datalog queries.

As a prerequisite, we review the semantics of non-recursive Datalog over $\mathcal{K}$-relations. Recall the semantics for Datalog queries discussed in Section 1.3.3. For now, we will only consider unions of conjunctive queries (the query class $\mathcal{UCQ}$), i.e., Datalog programs without negation and recursion where all rules have the same idb predicate in their head, say Q, and no idb predicates appear in the body of rules. In terms of expressive power, this class of queries is equivalent to positive relational algebra without generalized projection and where selection (and join) conditions are limited to equality comparisons. Consider such a Datalog program with n rules:

$$
\begin{aligned}
&r_1 : Q(\overline{X_1}) :\!- \mathsf{R}_{11}(\overline{X_{11}}), \ldots, \mathsf{R}_{1k_1}(\overline{X_{1k_1}})\\
&\ldots\\
&r_n : Q(\overline{X_n}) :\!- \mathsf{R}_{n1}(\overline{X_{n1}}), \ldots, \mathsf{R}_{nk_n}(\overline{X_{nk_n}})
\end{aligned}
$$

We now define the semantics of $\mathcal{UCQ}$ queries over $\mathcal{K}$-relations following Green *et al.* (2007a). Note that our notation differs slightly from the one used in Green *et al.* (2007a). A valuation φ for a rule r is a binding of the variables of the rule to constants from the database. The set of all constants that occur in the database is referred to as the *active domain* $\textsc{adom}(D)$ of the database D). For a Datalog rule r we use $r[\varphi]$ to denote the result of applying the valuation φ to the rule r. We refer to $r[\varphi]$ as a *grounded rule*. For an input database D, the body of a grounded rule evaluates to either true of false. We say a grounded rule $r[\varphi]$ derives a tuple t if the head of $r[\varphi]$ is $Q(t)$. We use $\mathsf{Ground}(r, t)$ to denote the set of all grounded instances of rule r which derive t.

The annotation of a result tuple t of a $\mathcal{UCQ}$ query Q evaluated over a $\mathcal{K}$-database D is the sum over all the annotation of all grounded rules of Q wrt. D. The annotation corresponding to a grounded rule is the product of the annotations associated to all atoms appearing the body of the rule.

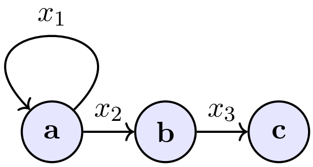
$$
Q(t) \coloneqq \sum_{r \in Q} \quad \sum_{r[\varphi] \in \mathsf{Ground}(r,t)} \quad \prod_{R_i(t') \in body(r[\varphi])} R_i(t') \tag{\textbf{Semantics of $\mathcal{UCQ}$ queries over K-relations}}
$$

Note the correspondence to $\mathcal{RA}^{+}$over $\mathcal{K}$-relations. In relational algebra, a Datalog rule can be expressed as join over all relations in the body of the rule and a projection corresponding to the head of the rule. Thus, each grounded rule's body corresponds to a single join result (computed as the multiplication of input tuple annotations). Multiple grounded instances of a rule may derive the same output tuple (multiple join results may be projected onto the same output tuple). Thus, we need to sum up (projection) the annotations for all of these join results. Furthermore, a result may be derived by more than one rule (belongs to multiple inputs that are "unioned" together). Thus, the annotations for each of these rules should be summed up.

Example 42 ($\mathcal{UCQ}$ over K-relations)**.** Figure 2.19 shows a $\mathcal{UCQ}$ query which returns the end points of paths of length up to 2 that exists in an input graph. The edb relation edge stores the edge relation of the input graph. For example, there are two grounded rules that produce result tuple $t_{a,a} = (a, a)$ and succeed over the example database:

$$\mathsf{up2hop}(a,a) :- \mathsf{edge}(a,a)$$
$$\mathsf{up2hop}(a,a) :- \mathsf{edge}(a,a), \mathsf{edge}(a,a)$$

The annotation corresponding to the first grounded rule is x_1 and the one for the second one is $x_1 \cdot x_1 = {x_1}^2$. Thus, this tuple is annotated with $x_1 + {x_1}^2$.



$$r_1 : \mathsf{up2hop}(X,Y) :- \mathsf{edge}(X,Y)$$
$$r_2 : \mathsf{up2hop}(X,Z) :- \mathsf{edge}(X,Y), \mathsf{edge}(Y,Z)$$

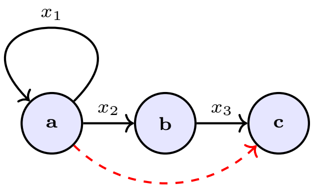
edge

start	**end**	$\mathbb{N}[X]$
a	a	x_1
a	b	x_2
b	c	x_3

Query Result (up2hop)

X	**Y**	$\mathbb{N}[X]$
a	a	$x_1 + {x_1}^2$
a	b	$x_2 + x_1 x_2$
b	c	x_3
a	c	$x_2 x_3$

Figure 2.19: Datalog over $\mathcal{K}$-relations



$$\mathsf{TC}(X,Y) :\!- \mathsf{edge}(X,Y)$$
$$\mathsf{TC}(X,Z) :\!- \mathsf{TC}(X,Y), \mathsf{edge}(Y,Z)$$

edge

start	**end**	$\mathbb{N}[X]$
a	a	x_1
a	b	x_2
b	c	x_3

Query Result (TC)

X	**Y**	$\mathbb{N}^{\infty}[[X]]$
a	a	$x_1 + {x_1}^2 + {x_1}^3 \dots$
a	b	$x_2 + x_1x_2 + {x_1}^2x_2 + {x_1}^3x_2 + \dots$
b	c	x_3
a	c	$x_2x_3 + x_1x_2x_3 + {x_1}^2x_2x_3 + \dots$

Figure 2.20: Example recursive query: transitive closure of the edge relation of a directed graph. Additional edges that are part of the transitive close, but not the input edge relation are shown as dashed, red arrows. The provenance of edges in the result of the query that start in node a have an infinite number of derivations that differ in the number of repetitions of the self-loop.

When moving from $\mathcal{UCQ}$ to Datalog with recursion, the formula for computing the annotation of a query result is no longer sufficient. The reason is that the body of a grounded rule may reference annotations of idb tuples that depend on the annotation of the tuple at the rule's head.

Example 43 (Recursive Datalog over $\mathcal{K}$-relations)**.** Figure 2.20 shows a Datalog program that computes the transitive closure (TC) of the edge relation of a directed graph. Each derivation of a query result through the rules of the query corresponds to a path in the graph. In contrast to our previous example, these paths may be of arbitrary length. For instance, there are infinitely many ways of deriving the result (a, a) by taking the edge annotated with x_1 a given number of times. Figure 2.21 shows the proof trees for some of these derivations. Thus, the provenance polynomial is an infinite sum:

$$\sum_{i=1}^{\infty} {x_1}^i$$

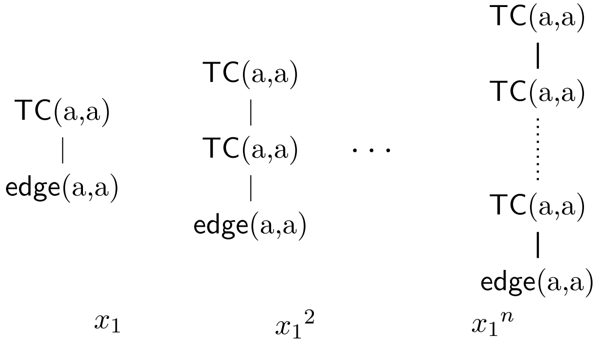


Figure 2.21: Proof trees for result $\mathsf{edge}(a, a)$

To be able to express infinite sums we have to extend $\mathbb{N}[X]$ with support for infinite sums. The solution is to use the semiring $\mathbb{N}^\infty[[X]]$ whose elements are *formal power series*. A formal power series is a (possibly infinite) sum of monomials with coefficients (that may be infinite) such as the sum shown above.

Green *et al.* (2007a), the paper that introduced the semiring provenance model, did present an algorithm for computing such formal power series. Specifically, the authors did demonstrate that the provenance of a recursive (positive) Datalog query of query result is the solution of an algebraic system of equation whose variables are the annotations of edb and idb tuples. The least fixpoint of such a system of equations is then the result of the recursive query evaluated over an input $\mathcal{K}$-database. For example, the part of the equation for the result tuple $\mathsf{T}(a, a)$ from Figure 2.20 is:

$$x_{\mathsf{TC}(a,a)} = x_1 + x_{\mathsf{TC}(a,a)}x_1 \qquad (2.2)$$

Luttenberger and Schlund (2014) showed that for certain queries and semirings, it is possible to represent the provenance as regular expressions. For example, the infinite sum from the example above can be written as the regular expression ${x_1}^*$. Algorithms for solving algebraic systems of equations over semirings have received attention in other contexts. For instance, Esparza *et al.* (2007), Luttenberger and

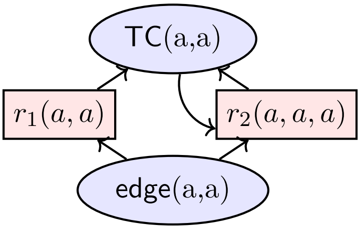


Figure 2.22: A provenance graph according to Köhler *et al.* (2012) for the transitive closure query result $\mathsf{TC}(a,a)$

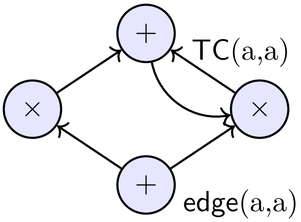


Figure 2.23: Provenance Circuit for $\mathsf{TC}(a,a)$

Schlund (2013), and Esparza *et al.* (2014) present methods for solving such equation systems.

2.5.1 Circuits and Provenance Graphs for Datalog Derivations

An alternative to encoding all possible derivations using formal power series is to encode the algebraic equation system that determines the annotations of tuples as a graph. Köhler *et al.* (2012) did introduce a graph-based provenance model for Datalog with recursion. Such graphs contain nodes representing tuples and nodes representing grounded rules (the labels of such nodes encode the valuation). Figure 2.22 shows an example of such a graph for tuple $\mathsf{TC}(a,a)$ in the result of the transitive closure query from Figure 2.20. Note how the graph encodes Equation (2.2): rule nodes encode multiplication while multiple incoming edges for a tuple node represent addition. One advantage of such provenance graphs is that they are better suited for debugging complex queries, because they are based on the syntactic structure of the queries making it easier to identify which parts of a program caused a bug. Similar data structures have also been used as compressed representations of such algebraic equation systems, e.g., see Esparza *et al.* (2014). Another related representation are circuits. Deutch *et al.* (2014) studied the applicability of **circuits** for encoding semiring annotations that are the result of recursive Datalog queries and identified for which classes of semirings this is possible (semirings which are *absorptive*). Circuits also have been used to compactly represent provenance for non-recursive queries with negation (monus-semirings) in Amarilli *et* *al.* (2015) and Senellart (2017). Figure 2.23 shows the provenance circuit for $\mathsf{TC}(a, a)$ using the notation from Senellart (2017). Note the correspondence to the provenance graph from Figure 2.22. Deutch *et al.* (2014) presented several optimization for such circuits that are based on absorption. For instance, self-dependencies can be removed, because they are redundant for absorptive semirings.

2.5.2 Recursive Datalog with Negation Under Well-founded Semantics and Answer Set Programming

Provenance Circuits and formal power series are defined for Datalog without negation. Several semantics have been proposed for Datalog queries that contain both recursion and negation, e.g., stratified semantics which is only applicable to programs without cycles (recursion) that contain negation. Damásio *et al.* (2013) introduced a provenance model for the so-called well-founded semantics (Van Gelder *et al.*, 1991; Flum *et al.*, 1997; Kemp *et al.*, 1995) which is defined for any recursive Datalog programs with negation, even programs where recursion goes through negation. A canonical example is the program shown below which, interpreted under well-founded semantics, computes that winning positions of a two-player game (the type of game we discussed in Section 2.4.3).

$$\mathsf{win}(X) :- \mathsf{move}(X, Y), \neg \mathsf{win}(X)$$

The provenance model of Damásio *et al.* (2013) uses the m-semiring $\mathcal{K}_{WhyNot}$ of equivalence classes of boolean formulas over a set of variables. The set of variables used is specific to a Datalog program P. It contains one variable for each grounded rule of the program (this model also tracks transformations) and two variables $x_{r(\mathbf{a})}$ and $x_{\neg r(\mathbf{a})}$ for each grounded atom in the Herbrand Base $\mathbb{H}$ of P. As mentioned before this work was the first to use separate provenance tokens for positive and negative facts. This trick is also used by the FOL semiring approach described in Section 2.4.3. Recall that the Herbrand Base of a logic program consists of all grounded atoms based on the active domain of the program. Note that in contrast to the definition of Datalog programs we have used where the program and edb database instance are separate, here the program includes the data as logical facts. Similar to the $\mathcal{K}$-interpretations from Section 2.4.3, an interpretation of a logic program is then a mapping that assigns to each atom in the $\mathbb{H}$ a Boolean formula, i.e., an element of semiring $\mathcal{K}_{WhyNot}$. Damásio *et al.* (2013) presented fixpoint algorithms that compute the interpretations of positive programs based on a generalization of the immediate consequence operator and of programs with negation using a fixpoint computation that closely mirrors the alternating fixpoint computation of well-founded semantics (Van Gelder *et al.*, 1991). Furthermore, Damásio *et al.* (2013) also applied this model to track the provenance of answer set programming. Answer set programming extends Datalog with non-determinism. One way to express this is through rules with disjunctions in their head. Intuitively, a rule encodes a set of possible worlds (any of the atoms in the head could be true). An answer set program may have multiple solutions. Extending their work on semiring annotations for first-order logic, Grädel and Tannen (2020) studied semiring annotations for fixed-point logic and games.

2.6 Transformation Provenance

When discussing provenance models in this chapter, we have mostly focused on the data dependencies encoded by these models. However, we have already seen that some models track additional information related to how a data item was produced by a transformation. For example, the provenance polynomials model distinguishes between conjunctive and disjunctive use of inputs. Another example are the provenance traces of Cheney *et al.* (2014) which provide a detailed account of a query's computation and provenance games which record how the rules of a Datalog query are used to derive a result. Provenance models for updates such as the MV-semiring provenance model Arab *et al.* (2018b) and Arab *et al.* (2016) and the model from Bourhis *et al.* (2020) also track which transformations (updates) affected a result. We will discuss these models in more depth in Section 2.7.

In this section, we will discuss models that track transformation dependencies and take a closer look at the transformation dependencies tracked by the models we have introduced earlier. Note that, as mentioned earlier, for why-not provenance this type of provenance has been called *query-based* explanations.

2.6.1 Provenance for data integration and exchange

We start by reviewing provenance models that track transformation dependencies in data integration and exchange. For that we briefly review the foundations of data exchange to the extend necessary to understand these provenance models. We refer the interested reader to Fagin *et al.*, 2005b, the seminal paper that formalized data exchange. In data exchange, we are given a schema mapping $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ which consists of a source schema $\mathbf{S}$, a target schema $\mathbf{T}$, and a set of logical constraints Σ relating the elements of the two schemata, and an instance $\mathbf{I}$ of the source schema. The task is to translate instance $\mathbf{I}$ into an instance $\mathbf{J}$ of $\mathbf{T}$ such that $(\mathbf{I}, \mathbf{J})$ fulfills Σ. The constraints Σ are expressed in some logical formalism, a common one being the language of *source-to-target tuple-generating dependencies* (*st-tgds*), An st-tgd is FOL formula of the form

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

where ϕ is a conjunction of atoms from the source schema and ψ is a conjunction of atoms of over the target schema. Intuitively, a st-tgd specifies the requirement that if a certain set of tuples exist in the source instance than a corresponding set of tuples has to existing in the target instance. A *solution* for schema mapping is any target instance $\mathbf{J}$ (an instance of $\mathbf{T}$) such that $(\mathbf{I}, \mathbf{J}) \models \Sigma$. Note that the existentials in ϕ are used to deal with elements (attributes) from the source schema that have no correspondence in the target, i.e., we have to "invent" values for such attributes. Because of the existence of existentials and because of the open world assumption that is made in data exchange (the target instance is allowed to contain data that does not stem from applying the schema mapping to the source instance), data exchange scenarios typically have infinitely many solutions. Fagin *et al.* (2005b) demonstrated how to represent a subspace of these solutions relevant for answering a certain class of queries as an incomplete database called

Source Schema

LivesAt($\underline{\text{Name}}$, City)

Source Instance

LivesAt

Name	City
Alice	Chicago
Bob	New York

Target Schema

Person($\underline{\text{Name}}$,Address)
Address($\underline{\text{Id}}$,City)

Universal Solution

Person

Name	Address
Alice	n_1
Bob	n_2

Address

Id	City
n_1	Chicago
n_2	New York

st-tgds

$$\sigma_1 : \forall x, y : LivesAt(x, y) \rightarrow \exists z : Person(x, z), Address(z, y)$$

Figure 2.24: Example data exchange scenario

the *universal solution*. While the set of possible worlds represented by a universal solutions are typically infinite, using an encoding of incomplete relations as tables with labelled null values (V-tables, see Imieliński and Lipski Jr (1984)). Queries over the target instance are then answered using certain answer semantics, i.e., only results that exist in every possible world are returned. Importantly, union of conjunctive queries can be answered efficiently over the universal solution for a data exchange scenario. However, we will not dive deeper into these intricacies of data exchange.

Example 44 (Data Exchange Scenario). Consider the data exchange scenario shown in Figure 2.24. The source schema consists of a relation LivesAt that stores the names of persons and the city they live. This schema is mapped to a target schema with two relations Person and Address storing the name of persons and the the address they live at. The attribute address of relation Person is a foreign key to relation Address that stores the identifier and city of addresses. The tgd σ_1 vertically partitions the LivesAt relation into the Person and Address relations. In the source, addresses do not have an identifier. The existentially quantified variable z is used to ensure that when splitting a LivesAt tuple into a Person and Address tuple, the association between the person and their address is preserved. The example source instances stores the addresses of two persons. A universal solution for this data exchange scenario is shown on the bottom right of Figure 2.24. Labelled nulls n_1 and n_2 are used to represent the unknown identifiers of the Chicago and New York address, respectively.

It has been shown that it is possible to use query languages, e.g., SQL, to compute universal solutions. These query languages do not support incompleteness[14], but it is possible to simulate the semantics of labelled nulls to the degree that is sufficient for query answering by using Skolem functions.[15]

Routes for Schema Mappings

Chiticariu and Tan (2006) introduce a provenance model for schema mappings that uses "routes" to explain why a tuple is in the target instance of a solution for a schema mapping. A route is a sequence of "satisfaction steps" each of which of the form $K_1 \overset{\sigma,h}{\rightarrow} K_2$. Here $\sigma : \forall \phi(\mathbf{x}) \rightarrow \exists \mathbf{y} : \psi(\mathbf{x}, \mathbf{y})$ is a tgd, K_1 is an instance of $(\mathbf{I}, \mathbf{J})$ that fulfills σ and h is a homomorphism such that $h(\phi(\mathbf{x})) \subseteq K_1$ and $K_2 = K_1 \cup h(\psi(\mathbf{x}, \mathbf{y}))$. Intuitively, a route step is the "application" of a tgd whose premise is fulfilled and records the consequence of the tgd for a particular premise.

Example 45 (Routes)**.** One possible route for Example 44 is shown below. Using the homomorphism $\{x \mapsto Alice, y \mapsto Chicago, z \mapsto n_1\}$,

[14] While SQL supports incompleteness through `NULL` values, it does not support the open world assumption, queries do not return certain answers, and there is no support for labelled nulls.

[15] The use of function symbols to encode which values an unknown value depends on has a long-standing tradition in data integration (Hull and Yoshikawa, 1990; Arocena *et al.*, 2013) and has been the basis of more expressive schema mapping languages such as SO-tgds (Fagin *et al.*, 2005a).

mapping σ derives the target instance tuples $Person(Alice, n_1)$ and $Address(n_1, Chicago)$.

$$\{LivesAt(Alice, Chicago)\} \overset{\sigma_1,h}{\rightarrow} \{Person(Alice, n_1), Addres(n_1, Chicago)\}$$
$$h = \{x \mapsto Alice, y \mapsto Chicago, z \mapsto n_1\}$$

Mapping and Transformation Provenance

Glavic *et al.* (2010) and Glavic *et al.* (2011) introduced a provenance model that tracks the transformation dependence of a tuple on the schema mappings that were used to derive it as well as the part of the (relational algebra) transformation that implements these mappings which is responsible for producing the result. This two types of provenance are referred to as **mapping provenance** and **transformation provenance** in this work. The approach annotates parts of relational algebra transformations with the mappings that they implement. The transformation provenance for a result tuple $t \in Q(D)$ for a query (relational algebra expression) Q and database D is a set of annotated versions of Q where each operator is associated with a boolean value $\{0, 1\}$. Each of the annotated trees in the transformation provenance of a query result encodes which operators did contribute to the result wrt. to a *witness list* in the PI-CS provenance of the query result. Recall that the PI-CS provenance (Example 28) consists of witness lists which, for SPJU queries, are alternative derivations of a result tuple. In the tree for a witness list w, operators annotated with 1 do contribute to t while the remaining operators (annotated with 0) do not. An operator of the query does contribute to a query result t wrt. to a witness list w if its evaluating the subquery rooted at this operator over the tuples from the witness list returns a non-empty result. The intuition is that if this combination of input tuples did produce an empty result for the operator, then this operator did not contribute to the result given this subset of the input.

Definition 24 (Transformation Provenance)**.** An annotated algebra tree for a query Q is a pair $(Tree_Q, \theta)$ where $Tree_Q = (V, E)$ is a tree that contains a node for each algebra operator used in Q and $\theta : V \in$

$Tree_q \rightarrow \{0,1\}$ is a function that associates each operator in the tree with an annotation from $\{0,1\}$. We define a preorder on the nodes to give each node an identifier. Let $I(op)$ denote the identifier for a node representing operator *op* and let Q_{op} denote the subtree rooted at operator *op*. Furthermore, let D be a database and $t \in Q(D)$. Recall that $\mathcal{PI}(Q,D,t)$ denotes the PI-CS provenance of t. The transformation provenance of an output tuple t is the set $\mathcal{T}(Q,t)$ of annotated-trees defined as follows:

$$\mathcal{T}(Q,t) = \{(Tree_Q, \theta_w) \mid w \in \mathcal{PI}(Q,D,t)\}$$

$$\theta_w(op) = \begin{cases} 0 \text{ if } Q_{op}(w) = \emptyset \\ 1 \text{ otherwise} \end{cases}$$

Note that transformation provenance can be used to track transformation dependencies for relational algebra statements (and SQL based on a canonical translation between relational algebra and parts of SQL query blocks) independent of whether they represent a mapping or not. For tracking provenance of schema mappings that are implemented as transformations in relational algebra, Glavic *et al.*, 2010 assumes that for each tgd σ in Σ implemented by a query Q, there exists a function $\mu_{\sigma,Q}$ that maps operators of Q to 1 if they implement σ and 0 otherwise. Note that an operator may be part of the implementation of more than one mapping. The **mapping provenance** for a result tuple wrt. to a witness list w of the tuple's provenance is then the set of tgds which are implemented precisely by the operators in the transformation provenance (the function $\mu_{\sigma,Q}$ for the tgd is equal to θ_w for all operators).

Example 46 (Transformation and Mapping Provenance). Figure 2.25 shows a mapping scenario mapping persons, some with associated addresses, to a person and address relation in the target. In the target there is a single relation Person recording which person lives at which address. Tgd σ_1 maps persons and their address to the target while σ_2 maps only persons to ensure that persons without associated address are also copied to the target. The transformation implementing these two tgds (query Q_{σ_1,σ_2}) uses an outer join to match persons with their

Source Schema

PersonName($\underline{\text{Name}}$)
LivesAt($\underline{\text{Name}}$,$\underline{\text{Address}}$)

Source Instance

LivesAt

Name	**City**
Alice	Chicago
Bob	New York

PersonName

Name
Alice
Bob
Peter

Target Schema

Person(Name, City)

Universal Solution produced by Q_{σ_1,σ_2}

Person

Name	**City**
Alice	Chicago
Bob	New York
Peter	$f_{City}(Peter)$

st-tgds

$$\sigma_1 : \forall x, y : PersonName(x) \wedge LivesAt(x, y) \rightarrow Person(x, y)$$

$$\sigma_2 : \forall x : PersonName(x) \rightarrow \exists y : Person(x, y)$$

transformations implementing tgds σ_1 and σ_2

$$Q_{\sigma_1,\sigma_2} := \Pi^{\textcircled{0}}_{Name,\textbf{if}\ \textsc{isnull}(City)\ \textbf{then}\ f_{City}(Name)\ \textbf{else}\ City}(\mathsf{PersonName}^{\textcircled{2}} \mathbin{⟕}^{\textcircled{1}}_{Name=Name} \mathsf{LivesAt}^{\textcircled{3}})$$

Figure 2.25: Data exchange scenario with two tgds mapping persons with addresses and persons without addresses.

addresses (if they exist). Here we use $\textbf{if}\,\theta\,\textbf{then}\,e_1\,\textbf{else}\,e_2$ to denote an expression that evaluates to e_1 if condition θ holds and e_2 otherwise. Furthermore, $\textsc{isnull}(e)$ returns true if e evaluates to null and false otherwise. f_{City} is a Skolem function used to generate an address value (modeling a labelled null) that is unique for a given name. We show the identifier associated with an operator in a red cycle besides the operator. The projection, outer join, and access of relation PersonName implement both mappings. The access of relation LivesAt, however, is only relevant for σ_1. Consider result tuple $t_a = (Alice, Chicago)$. The single witness list in the provenance of this tuple is $< (Alice), (Alice, Chicago) >$. The subqueries rooted at each of the query's operators returns a non-empty result for the tuples $\{(Alice), (Alice, Chicago)\}$. Thus, the transformation provenance $\mathcal{T}(Q_{\sigma_1,\sigma_2}, t_a)$ contains of a single annotated algebra tree where all operators are annotated with 1 (are in the provenance). The mapping provenance is $\{\sigma_1\}$, because σ_1 is the only tgd which is implemented by all operators of the query.

2.6.2 Schema Mappings Provenance through $\mathcal{M}$-semirings

Semiring provenance models the computation of a transformation by recording how inputs have been combined. The advantage of these models is that they have been formally proven to capture the essence of the computation, because the result of the computation can be computed based on the provenance (recall that we referred to this property as *computability*). To be more precise, the annotation of a result tuple in any semiring $\mathcal{K}$ can be reconstructed based on its $\mathbb{N}[X]$ (provenance polynomial) annotation and the annotation of input tuples (encoded as a valuation of the variables in the polynomial to elements of $\mathcal{K}$). However, such semiring annotations to not capture which parts of the transformation affected the result. The model has to be extended to keep track of this information.

Karvounarakis (2009) did extend semirings for the tracking the provenance of schema mappings in data / update exchange (Karvounarakis *et al.*, 2013; Fagin *et al.*, 2005b). In update exchange (Karvounarakis *et al.*, 2013; Green *et al.*, 2010; Ives *et al.*, 2008; Green *et al.*, 2007b; Ives *et al.*, 2005), a set of peers exchange information based a mapping between their schemata. In Karvounarakis *et al.* (2013), updates to the data of one peer are translated into updates over the database of other peers based on mappings between the schemata of these peers. In update exchange, provenance can be used to track how data is derived and to reason about trust (some peers may mistrust updates coming from some other peers). However, this also requires knowing based on which mapping a result was produced. Karvounarakis (2009) and Karvounarakis *et al.* (2013) did extend semirings with with unary functions $\mathcal{M} = \{m_1, \ldots, m_n\}$ with $m : K \rightarrow K$ for all $m \in \mathcal{M}$. These functions represent the application of mappings. The resulting structure is referred to as $\mathcal{M}$-semirings. Any such function m is required to commute with addition (mapping application commute with union) and return $\mathbb{0}_{\mathcal{K}}$ if the input is $\mathbb{0}_{\mathcal{K}}$ (a mapping cannot create outputs based on non-existing inputs).

$$m(\mathbb{0}_{\mathcal{K}}) = \mathbb{0}_{\mathcal{K}} \qquad\qquad m(k_1 \oplus_{\mathcal{K}} k_2) = m(k_1) \oplus_{\mathcal{K}} m(k_2)$$

The elements of the $\mathcal{M}$-semiring for tracking the provenance of schema mappings are symbolic expressions build from semiring operations and applications of the mapping functions. For example, if a result tuple is annotated with $m_3(x_1 \cdot x_2) + m_2(x_3)$ then this tuple was derived by "applying" schema mapping m_3 to the input tuples annotated with variables x_1 and x_2 or applying schema mapping m_2 to the tuple annotated with x_3.

2.6.3 Provenance Traces

Cheney (2007) argued that provenance is intimately related to the notion of program slicing. A program slice for a result (or value of a variable in a program) consists of the parts of the program that are sufficient to compute the result. Mapping this idea to databases, a slice of a query would be the part of the query responsible for producing a result. In that sense, a program slice is a type of transformation provenance. Perera *et al.* (2012) extended the idea of program slices to provenance trace slices that explain the evaluation of functional programs. Intuitively, a provenance trace slice tracks both data dependencies and transformation dependencies and models the part of an execution trace for the program that produced a result of interest. Cheney *et al.* (2014), based on this earlier work, developed a tracing mechanism for queries expressed in nested relational calculus ($\mathcal{NRC}$) over nested multisets (bag semantics). The **provenance trace** of a query over a database is essentially an unrolling of the query, i.e., it shows how subexpressions of the query are applied to values to compute the result. This model has been proven to be *computable*, i.e., Cheney *et al.*, 2014 presented an evaluation semantics called *trace replay* and proved that the result produced by replaying a trace is the same as the result of the original query for which the trace was produced. Multisets are modelled by the approach as mappings from tuple identifiers to tuples where $[id].t$ denotes a tuple t with identifier id.

Traces can be *sliced* to find the part of the computation that is relevant for producing some part of the query's output. The user provides as input a pattern that describes which part of the query result they want to track. Patterns are nested values where some elements may be *"holes"*. A hole $\Box$ indicates that we do not want to track provenance for this part of the nested value. Given a pattern and a trace for a query and database, the *trace slice* can be derived by removing from the trace parts that are not relevant for producing the part of the result described by the pattern.

Example 47 (Provenance Trace)**.** Figure 2.26 shows a query that sums up the values of attribute A for all tuples from relation R where the value of attribute B is greater than 2. We show both a SQL and an $\mathcal{NRC}$ version of this query. For readers unfamiliar with $\mathcal{NRC}$, the result relation is constructed using the relation ($\{\}$) and tuple constructors ($\langle\rangle$) to create a relation with a single tuple with an attribute Total whose value is the result of summing up (**sum**) the result of a set comprehension (the result of such a comprehension $\bigcup\{e' \mid x \in e\}$ is the union of the relations produced by substituting variable x in e' with all values from the result of expression e). This comprehension uses a conditional expression ($\mathbf{if}(\cdot,\cdot,\cdot)$) to generate for each tuple x from R with either the empty set (if $B \leq 2$) or with a projection of the tuple on attribute A (implemented using the tuple constructor). That

Query expressed in SQL and $\mathcal{NRC}$

```
SELECT sum(A) AS Total FROM R WHERE B > 2;
```

$$Q = \{\langle \mathit{Total} : \mathbf{sum}\Big(\bigcup\{\mathbf{if}(x.B > 2, \{\langle A : x.A\rangle\}, \{\}) \mid x \in R\}\Big)\rangle\}$$

R

id	A	B
$[r_1]$	2	3
$[r_2]$	3	1
$[r_3]$	5	4

Query Result

id	Total
ϵ	7

Provenance trace for the query

$$\begin{aligned} T = \{\langle \mathit{Total} :& \mathbf{sum}(\bigcup\{\mathbf{if}(x.B > 2, \{\langle A : x.A\rangle\}, \{\}) \mid x \in R\} \triangleright \{\\ & [r_1].\mathbf{if}(x.B > 2, \{\langle A : x.A\rangle\}, \{\}) \triangleright_{\mathit{true}} \{\langle A : 2\rangle\},\\ & [r_2].\mathbf{if}(x.B > 2, \{\langle A : x.A\rangle\}, \{\}) \triangleright_{\mathit{false}} \{\},\\ & [r_3].\mathbf{if}(x.B > 2, \{\langle A : x.A\rangle\}, \{\}) \triangleright_{\mathit{true}} \{\langle A : 5\rangle\}\\ & \})\rangle\} \end{aligned}$$

Trace slice for result tuple $\langle \mathit{ttl} : 7\rangle$

$$\begin{aligned} T = \{\langle \mathit{Total} :& \mathbf{sum}(\bigcup\{\mathbf{if}(x.B > 2, \{\langle A : x.A\rangle\}, \{\}) \mid x \in R\} \triangleright \{\\ & [r_1].\mathbf{if}(x.B > 2, \{\langle A : x.A\rangle\}, \{\}) \triangleright_{\mathit{true}} \{\langle A : 2\rangle\},\\ & [r_2].\square,\\ & [r_3].\mathbf{if}(x.B > 2, \{\langle A : x.A\rangle\}, \{\}) \triangleright_{\mathit{true}} \{\langle A : 5\rangle\}\\ & \})\rangle\} \end{aligned}$$

Figure 2.26: Example Provenance Trace

is, this comprehension returns a relation with a single attribute A that contains the A values of all tuples where $B < 2$. Figure 2.26 also shows an example instance and the result of this query over this instance as well as the trace for this query and the trace slice for the aggregation result. Note how the trace mirrors the structure of the query, but the set comprehension has been unrolled, showing the computation for each of the three input tuples. Furthermore, the trace records for each conditional expressions both its result and whether the condition of the expression evaluated to true or false for an input. For instance, for the first tuple the condition evaluated to true and, thus the projection of tuple $\langle A : 2, B : 3 \rangle$ on A is returned $(\langle A : 2 \rangle)$. For the slice of the query, the expression that evaluated to the empty set (for the second tuple) has been replaced with a hole $\Box$, because this tuple does not affect the result.

Comparing provenance traces with transformation provenance, routes, and $\mathcal{M}$-semirings, both provenance traces and transformation provenance determine which syntactic constructors of a query contribute to a result. However, transformation provenance, while being defined based on data dependencies, does not convey data dependencies directly while they are encoded in the traces. Furthermore, trace slices have a stronger theoretical foundation. $\mathcal{M}$-semirings only record application of transformations (mappings), but do not record the syntactic structure of Σ in the derivations. Comparing provenance traces with plain semiring provenance (without mappings), provenance traces are more verbose and may sometimes produce an overestimation of what inputs are relevant for a slice. However, the syntactic structure encoded in the traces, similar to transformation provenance, makes them better suited for, e.g., debugging use cases.

2.6.4 Query-based Explanations for Why-not Provenance

Transformation dependencies have also been studied in the context of why-not provenance where these are called *query-based* explanations. A query-based explanation for a missing answer records which parts (e.g., operators) of the query are responsible for the failure to derive the result. Chapman and Jagadish (2009) first introduced query-based explanations for why-not provenance that are determined based on data dependencies using the lineage provenance model. The approach determines based on the missing answers of interest which tuples from the input database may contribute to these answers. These tuples are referred to as *unpicked data items*. Lineage is then used to determine manipulations (operators) for which an unpicked data item or a *successor* of an unpicked data item (a tuple whose lineage contains the unpicked data item) is in the input of the operator, but no successor of this unpicked data item is in the output. Such operators are called *picky manipulations*. The intuition is that such an operator filters out data items that could potentially have been used to derive the missing answer. The explanation for a why-not question is then the set of *frontier picky manipulations*.

Example 48 (Query-based Explanations). Consider a query returning all students with GPA higher than 3.0:

$$\Pi_{name}(\sigma_{GPA>3.0}(students))$$

Assume that we are interesting in knowing why Peter is not in the result. For example, if a single student named Peter exists in the input with a GPA of 2.5, then the selection operator would be identified as frontier picky, because it filters out the only input tuple $(Peter, 2.5)$ that could been used to produce the missing answer $(Peter)$.

The why-not definition of Chapman and Jagadish (2009) has several limitations. Notably, it may produce false negatives (operators that can be held responsible for the missing answer may not be part of an explanation) and false positives (operators whose modification have no effect on the missing answer may be part of an explanation). Follow-up work such as Bidoit *et al.*, 2015; Bidoit *et al.*, 2014 has focused on resolving some of these limitations.

Example 49 (False Positives and Negatives). As an example for a false negative consider a query over relations person(name,home,work) and address(aid,city) that joins persons with their work addresses they live at:

$$\Pi_{name,city}(person \bowtie_{work=aid} address)$$

An example instance is shown below.

name	**home**	**work**
Peter	1	2

aid	**city**
1	New York
2	Boston

For instance, we may be interested in knowing why tuple `(Peter,New York)` is not in the result. This result could be produced by changing the join condition to join on $home = aid$. However, the join is not identified as frontier picky, because it contains a success of tuple `(Peter,1,2)` that is compatible with the missing answer.

Query Repairs

A query-based explanation informs the user which operators in a query are responsible for the failure to return the missing answer of interest. However, this may not be sufficient for the user to determine how to *repair* the query such that the missing answer is returned. Bidoit *et al.*, 2016 referred to this problem as *query refinement*. We will use the term *query repair*.

Example 50 (Query Repair)**.** Continuing with Example 49, we can repair this query by changing the join condition to:

$$home = aid$$

2.7 Updates and Transactions

We now discuss provenance models for updates and transactions. There are two main difference between queries and updates: (i) updates change the state of the database and (ii) typically a database state is the results of a history of updates, so transformation provenance is important to track which updates affected which version of a tuple. Buneman *et al.* (2008) did present one of the first provenance models for updates. This provenance model was defined for nested relational calculus ($\mathcal{NRC}$) queries and a nested update language ($\mathcal{NUL}$) defined in this work. In this work, provenance is defined as annotations (colors) on data. The authors present both an explicit provenance semantics (the input query determines how colors are propagated) as $\mathcal{NRC}$ and $\mathcal{NUL}$ over colored data (in the nested relational model used in this work, annotated objects can be modeled as value-annotation pairs) and an implicit provenance semantics (colors are not accessible to the query and are propagated automatically). Since the semantics is defined based on "copying" of objects from a nested input instance to a nested output instance, this is essentially a type of where-provenance. In Vansummeren and Cheney (2007) this model was applied to track the provenance of SQL DML operations. While this work represented an important step forward, it is subject to a few limitations: (i) transformation provenance is not tracked, i.e., there is no record of which update in a sequence of update operations created a particular object in a nested relational instance; (ii) dependencies that are not copying of values are not tracked. For instance, when two sets e_1 and e_2 are combined with union, then the resulting set object is considered to be created by the query. While the provenance annotations of elements from the two input sets are propagated, we loose the information that the output was derived from the sets e_1 and e_2; (iii) the approach does not support transactions.

Databases allow updates to be grouped into transactions that are executed atomically and in isolation (at least conceptually). Database systems apply concurrency control protocols to ensure the so-called ACID properties (atomicity, consistency, isolation, and durability). While concurrency control is a fascinating topic, a thorough overview of this field is beyond the scope of this work. What is relevant in terms of provenance tracking is that concurrency control protocols enable multiple transaction to be executed in parallel while maintaining consistency. The notion of correctness that is typically applied is serializability which requires that the concurrent execution of a transactional history (a set of transactions with an execution order for all or their operations) is equivalent in a precise sense to a serial execution of the involved transaction, i.e., a history where there is no interleaving of the operations of transactions. While serializability is desirable, it can come at a high price. Thus, relaxed versions of concurrency control protocols such as snapshot isolation (Berenson *et al.*, 1995) have been developed that do not guarantee serializability. For the purpose of developing a provenance model for transactional updates, this entails that we need to reason about how tuple versions created by one transaction depend on tuple versions created by other (perhaps concurrently running) transactions.

2.7.1 MV-semirings

Arab *et al.* (2018b) and Arab *et al.* (2016) did extend the semiring provenance model to support updates and transactions in addition to queries. The approach is a strict generalization of the semiring model with support for updates where the annotation of a result tuple of a query explains both how this tuple was generated from input tuples from the current version of the database as well as how these input tuples were created by a transaction history from past tuple versions (tuples that no longer exist in the current version of the database). This work did introduce multi-version semirings (MV-sermirings), a specific class of semirings that are used to encode how tuples are derived by a transactional history. For a semiring $\mathcal{K}$ a corresponding MV-semiring $\mathcal{K}^\nu$ can be constructed whose elements are symbolic expressions consisting of function symbols that represent applications of updates, addition, multiplication, and elements of the embedded semiring $\mathcal{K}$. The elements of an MV-semiring $\mathcal{K}^\nu$ are equivalence classes of such symbolic expression based on evaluation of addition and multiplication in semiring $\mathcal{K}$ for operands that are in $\mathcal{K}$. That is, except for symbolic expressions involving unary function symbols (representing application of updates) $\mathcal{K}^\nu$ behaves exactly as $\mathcal{K}$ which means that $\mathcal{K}^\nu$ is backward-compatible to $\mathcal{K}$ in terms of query evaluation. Intuitively, a $\mathcal{K}^\nu$-relation encodes the history of a $\mathcal{K}$-relation. Arab *et al.* (2018b) showed how to create the current version of a $\mathcal{K}$-database created by a transactional history by evaluating the symbolic expression that is the $\mathcal{K}^\nu$-annotation of a tuple by interpreting the function symbols of this expression as functions $\mathcal{K} \rightarrow \mathcal{K}$. In the following, we will describe this provenance model in more detail, but will only show the application of this model by example instead of formally defining the semantics of updates and transactions over this model. We refer the reader to Arab *et al.* (2018b) and Arab *et al.* (2016) for update and transaction semantics.

Example 51 (Transaction Provenance (Arab *et al.*, 2018b))**.** Figure 2.28a shows an example database storing information about bank accounts and overdrafts. MV-semiring annotations are shown on the left of each tuple. Suppose a user Bob executed the Transactions T_5 shown in Figure 2.27 under the snapshot isolation concurrency control protocol

(Berenson *et al.*, 1995). This transaction implements a new policy of giving a \$100 bonus to all savings accounts and an additional \$300 bonus to all savings accounts with a balance higher than \$5000. The database instance after the execution of Transaction T_5 is shown in Figure 2.28b. Attribute values affected by an update are highlighted in red. Meanwhile, Alice did withdraw money (\$1500) from her checking account which triggered Transaction T_6. This transaction inserts an overdraft record into the relation `Overdraft(cust,bal)` since the total balance of Alice's accounts is negative after the withdrawal. The states of the `Account` and `Overdraft` relations after the execution of both transactions are shown in Figure 2.28c. Alice, surprised to receive an overdraft notice, checks her account. She observes that the total balance of her accounts is positive and, thus, she should not have received the \$100 overdraft. This unexpected result is caused by a concurrency anomaly called *write-skew* (Berenson *et al.*, 1995) which can occur under snapshot isolation. Under snapshot isolation, each Transaction T executes over a private snapshot which contains changes made by transactions that committed before T's start. Hence, Transactions T_6 sees the previous balance of \$1000 for Alice's savings account and after the withdrawal of \$1500 from her checking account, it computes a total balance of $1000 + (-1100) = -100 < 0$. In the MV-semiring model the annotation of a tuple version encodes the complete derivation history of the tuple, i.e., which updates have created the tuple version using which previous tuple versions. For instance, the annotation of the overdraft tuple in the database version created by transaction T_6 shows that this tuple became valid after the commit of this transaction at version 16 (database versions are identified by a totally ordered domain of version identifiers) and was produced by an insert statement executed by this transaction at version 15. The inserted tuple in turn was generated by joining (multiplication) two tuples that were both committed at version 4 by transaction T_1. Each of these tuples were created by an insert of transaction T_1 executed at versions 2 and 3, respectively. From this tuple's annotation and the annotations of the tuples recording Alice's account balances we can determine that the overdraft was based on transaction T_6 seeing an outdated account balance (not including Bob's changes) while the current balance of Alice's accounts includes these

T	SQL	Time

T_5	`UPDATE Account SET bal = bal + 100`
	`WHERE typ = 'Savings';`
		10

T_6	`UPDATE Account SET bal = bal - 1500`
	`WHERE cust = 'Alice' AND typ = 'Checking';`
		11

T_5	`UPDATE Account SET bal = bal + 300`
	`WHERE typ = 'Savings' AND bal > 5000 ;`
		12

T_5	`COMMIT;`	13

T_6	`INSERT INTO Overdraft`
	`(SELECT cust, a1.bal + a2.bal`
	`FROM Account a1, Account a2`
	`WHERE a1.cust = 'Alice' AND a1.cust = a2.cust`
	`AND a1.typ≠a2.typ AND a1.bal + a2.bal < 0);`
		14

T_6	`COMMIT;`	15

Figure 2.27: Example audit log for a transactional history

changes (and, thus, does not warrant an overdraft penalty).

Version Annotations

Before defining MV-semirings, we first define the t unary function symbols called version annotations that are used in the symbolic expressions of an MV-semirings. A version annotation $X^{id}_{T,\nu}(k)$ denotes that an operation of type X (update U, insert I, delete D, or commit C) that was executed at time $\nu - 1$ by transaction T affected a previous version of a tuple with identifier id and previous provenance k. Assuming domains of tuple identifiers $\mathbb{I}$, version identifiers $\mathbb{V}$ (this model assumes a totally ordered domain of versions and each update of a history is executed at a particular version), and transaction identifiers $\mathbb{T}$ (each transaction is assigned a unique identifier from $\mathbb{T}$, $\mathbb{A}$ denotes the set of all version annotations:

$$I^{id}_{T,\nu}, U^{id}_{T,\nu}, D^{id}_{T,\nu}, C^{id}_{T,\nu} \;\; \textbf{for} \;\; id \in \mathbb{I}, \nu \in \mathbb{V}, T \in \mathbb{T} \tag{2.3}$$

Account

	cust	typ	bal
$C^1_{T_1,4}(I^1_{T_1,2}(x_1))$	Alice	Checking	400
$C^2_{T_1,4}(I^2_{T_1,3}(x_2))$	Alice	Savings	1000
$C^3_{T_2,3}(I^3_{T_2,1}(x_3))$	Peter	Savings	4990

Overdraft

cust	bal

(a) Database before execution of T_5 and T_6

Account

	cust	typ	bal
$C^1_{T_1,4}(I^1_{T_1,2}(x_1))$	Alice	Checking	400
$C^2_{T_5,14}(U^2_{T_5,11}(C^2_{T_1,4}(I^2_{T_1,3}(x_2))))$	Alice	Savings	1100
$C^3_{T_5,14}(U^3_{T_5,13}(U^3_{T_5,11}(C^3_{T_2,3}(I^3_{T_2,1}(x_3)))))$	Peter	Savings	5390

(b) Database after execution of T_5

Account

	cust	typ	bal
$C^1_{T_6,16}(U^1_{T_6,12}(C^1_{T_1,4}(I^1_{T_1,2}(x_1))))$	Alice	Checking	-1100
$C^2_{T_5,14}(U^2_{T_5,11}(C^2_{T_1,4}(I^2_{T_1,3}(x_2))))$	Alice	Savings	1100
$C^3_{T_5,14}(U^3_{T_5,13}(U^3_{T_5,11}(C^3_{T_2,3}(I^3_{T_2,1}(x_3)))))$	Peter	Savings	5390

Overdraft

	cust	bal
$C^4_{T_6,16}(I^4_{T_6,15}(U^1_{T_6,12}(C^1_{T_1,4}(I^1_{T_1,2}(x_1)) \cdot C^2_{T_1,4}(I^2_{T_1,3}(x_2)))))$	Alice	-100

(c) Database after execution of T_6

Figure 2.28: Running example database states

The nesting of version annotations in an $\mathcal{K}^\nu$-annotation of a tuple t encodes the sequence of operations that lead to the creation of tuple t and from which previous tuple versions it was derived from.

Example 52. Consider the $\mathbb{N}[X]^\nu$-relation `Account` in Figure 2.28b. The second tuple is annotated with $C^2_{T_5,14}(U^2_{T_5,11}(C^2_{T_1,4}(I^2_{T_1,3}(x_2))))$, i.e., it was created by an update of Transaction T_5, which updated a tuple that was inserted by T_1. Based on the outermost commit annotation, this tuple version was committed at version 13.

MV-semiring Annotation Domain

Fixing a semiring $\mathcal{K}$, the domain of semiring $\mathcal{K}^\nu$ is defined based on the set of all finite symbolic expressions P following the syntax defined by the grammar shown below where $k \in K$ and $\mathcal{A} \in \mathbb{A}$.

$$P := k \mid P + P \mid P \cdot P \mid \mathcal{A}(P) \tag{2.4}$$

The semantics of these expressions is defined in Definition 25 and Figure 2.29. Note that $+$ and $\cdot$ in these expressions are used to encode that a tuple depends on multiple input tuples, e.g., a query such as the one

Evaluation of expressions with operands from K

$$k + k' = k \oplus_{\mathcal{K}} k' \qquad k \cdot k' = k \otimes_{\mathcal{K}} k' \qquad (\text{if } k \in K \wedge k' \in K)$$

Equivalences involving version annotations

$$\mathcal{A}(\mathbb{0}_{\mathcal{K}}) = \mathbb{0}_{\mathcal{K}} \qquad \mathcal{A}(k + k') = \mathcal{A}(k) + \mathcal{A}(k')$$

Figure 2.29: Equivalences that hold for $\mathcal{K}^{\nu}$

used by the insert of example Transaction T_6 or an update that modifies two tuples that are distinct in the input to be the same in the output (e.g., `UPDATE Account SET typ = ’Savings’`). For example, consider a query $\Pi_{typ}(Account)$ evaluated over the instance from Figure 2.28a. The result tuple `(Savings)` is derived from the second and third tuple in the `Account` table (the two tuples with this value in attribute `typ`) and, thus, would be annotated with $C^2_{T_1,4}(I^2_{T_1,3}(x_2)) + C^3_{T_2,3}(I^3_{T_2,1}(x_3))$. Recall that addition represents alternative use of tuples. The elements on an MV-semiring are equivalence classes (denoted as $[]_{\sim}$) of these symbolic expression wrt. the standard semiring laws as shown in Figure 2.6 and additional equivalences shown in Figure 2.29. These equivalences equate expressions in the “embedded” semiring $\mathcal{K}$ to be evaluated in semiring $\mathcal{K}$ ($k_1 + k_2 = k_1 +_{\mathcal{K}} k_2$). The remaining two equivalences intuitively mean that updating a non-existing tuple does not lead to an existing tuple ($\mathcal{A}(0_{\mathcal{K}}) = 0_{\mathcal{K}}$) and that addition distributes over updates (applying an update to the union of two relations is the same as applying the update to the inputs of the union and returning the union of the result).

Definition 25. Let $\mathcal{K} = (K, \oplus_{\mathcal{K}}, \otimes_{\mathcal{K}}, \mathbb{0}_{\mathcal{K}}, \mathbb{1}_{\mathcal{K}})$ be a commutative semiring. The MV-semiring $\mathcal{K}^{\nu}$ corresponding to $\mathcal{K}$ is the structure

$$\mathcal{K}^{\nu} = (K^{\nu}, +_{\mathcal{K}^{\nu}}, \cdot_{\mathcal{K}^{\nu}}, [0_{\mathcal{K}}]_{\sim}, [1_{\mathcal{K}}]_{\sim})$$

where $\cdot_{\mathcal{K}^{\nu}}$ and $+_{\mathcal{K}^{\nu}}$ are defined as

$$[k]_{\sim} \cdot_{\mathcal{K}^{\nu}} [k']_{\sim} = [k \cdot k']_{\sim} \qquad [k]_{\sim} +_{\mathcal{K}^{\nu}} [k']_{\sim} = [k + k']_{\sim}$$

The addition and multiplication operations of an MV-semiring output a symbolic expression by connecting the inputs with $+$ or $\cdot$ and then return the equivalence class for this expression. Consider semiring $\mathbb{N}$ and recall that this semiring is used to encode bag semantics relations by annotating each tuple with a natural number representing its multiplicity. For example, assume a tuple t is annotated with the $\mathbb{N}^{\nu}$-expression $U^{1}_{T,\nu}(3 \cdot 6)$. Here 3 and 6, elements from the embedded semiring $\mathbb{N}$, represent multiplicities. Applying equivalence $k \cdot k' = k \cdot_{\mathcal{K}} k'$, we can evaluate $3 \cdot 6 = 3 \otimes_{\mathbb{N}} 6 = 18$. Thus, t appears with multiplicity 18 and was updated by an update (U) of transaction T. The update was run at time $\nu - 1$ and, thus, the tuple became valid at time ν.

An important result proven in Arab *et al.*, 2018b is that semiring homomorphisms any $h : \mathcal{K}_1 \rightarrow \mathcal{K}_2$ can be lifted to a MV-semiring homomorphisms $h^{\mu} : \mathcal{K}^{\nu}{}_1 \rightarrow \mathcal{K}^{\nu}{}_2$ that preserves the structure of symbolic expressions by applying the homomorphism h to each element from $\mathcal{K}_1$ that occurs in such a symbolic expression. This implies that $\mathbb{N}[X]^{\nu}$, the MV-semiring version of the provenance polynomial semiring is the most general MV-semiring and, thus, the right choice for capturing provenance. Furthermore, Arab *et al.*, 2018b also introduced an "unversioning" homomorphism that maps $\mathcal{K}^{\nu}$-relation to a $\mathcal{K}$-relation by interpreting the version annotations as functions from $\mathcal{K}$ to $\mathcal{K}$. Intuitively, this homomorphism evaluates the history encoded in annotations of tuples to compute the current version of the relation.

2.7.2 Provenance for Hyperplane Update Queries

Bourhis *et al.* (2020) did present an algebraic provenance model for hyperplane update queries which is a restricted update language that is nonetheless of interest, because a sound and complete axiomatization of set equivalence for this class of update queries exists (Karabeg and Vianu, 1991). The authors identify the "correct" algebraic structure to capture these axioms which also allows a normal form to be defined that removes redundant sub-expressions in a provenance annotation to reduce the storage size of provenance. For instance, under set semantics (which is what is studied in this work), inserting the same tuple twice as a part of a single transaction[16] is redundant. Importantly, homomorphisms between algebraic structures used for provenance annotations

[16] Note that in this work concurrent executions of transactions is not considered.

propagate through updates. In this work update operations (transactions) are themselves annotated and these annotations propagate to the results of an update (transaction). Thus, this approach also encodes transformation provenance. Given that homomorphisms commute with operations, this has the advantage that the effect of aborting a transaction on the database produced by a history of transactions can be computed from the result of the history.

2.8 Summary and Conclusions

In this chapter, we have discussed a wide range of provenance models and studied their properties. We saw how progressively more complex query classes, require more sophisticated provenance models. Generalizing computations based on the algebraic properties, the foundation of the semiring annotation frameworks and its extensions, has proven to lead to expressive and general provenance models. Such models do enjoy the *computability* property, and, thus, are often more informative than provenance models based on *sufficiency* and *necessity*. In addition, the study of such general provenance models has also lead to a better understanding how provenance models compare with respect to their expressive power.

2.9 Additional References

It is not possible to do full justice to the broad body work on provenance models in this article. Instead we provide the reader with a list of additional references that may be of interest.

2.9.1 Surveys

Past surveys on database provenance have also covered and compared provenance models. Herschel *et al.* (2017) did cover why and why-not provenance models. Cheney *et al.* (2009) did present a detailed introduction and comparison of provenance models. However, the field has evolved significantly in the decade after the publication of this work. In this chapter we tried to cover as much as possible of this evolution.

Karvounarakis and Green (2012) and Green and Tannen (2017) did provide an overview of the $\mathcal{K}$-relational model.

2.9.2 Causality, Interventions, Responsibility, Resilience, Sensitivity

Causality in databases has been studied extensively in recent years. For instance, Freire *et al.* (2015) studied the complexity of resilience and responsibility for self-join free conjunctive queries and Freire *et al.* (2020) studied queries with self-joins. Bertossi and Salimi (2013) did establish interesting connections between causality, database repairs, and consistency-based diagnosis. Salimi *et al.* (2016) did provide a critique of the notion of responsibility from Meliou *et al.* (2010) and proposed an alternative definition based an interpretation rooted in probabilistic / incomplete databases.

2.9.3 Semirings and Annotations

Geerts *et al.* (2012) did study annotation structures for RDF queries. To deal with the `OPTIONAL` construct of the SparQL query language (essentially an outer join), the authors introduce spm-semirings, and extension of semirings with an operator that realizes the specific version of difference needed to implement `OPTIONAL`. Kostylev *et al.* (2013) studied the containment of queries over $\mathcal{K}$-relations. While this problem was previously studied in Green (2009), Kostylev *et al.* (2013) drops the assumption that containment is based on the natural order of semirings. Kostylev and Buneman, 2012 investigated the semantics of queries over relations annotated with multiple semirings. Buneman *et al.*, 2013 study an annotation model where the boundaries between data and annotations are blurred. Queries are free to interpret part of the input as data or as annotations.

Annotation can serve purposes different from provenance, e.g., they can be used to store documentation. If the purpose of annotations is not provenance, then propagating annotations based on data dependencies (provenance) may not be what the user wants. Consequently, general annotation management systems like DBNotes (Chiticariu *et al.*, 2005), Mondrian (Geerts *et al.*, 2006), and BDBM (Eltabakh *et al.*, 2008) allow more control over how annotations propagate. InsightNotes (Ibrahim *et al.*, 2015) enables the user to summarize annotations using data mining techniques. Annotation management is itself a broad field that we can not discuss in detail in this article.

3

Applications

In this chapter, we will discuss several motivating applications for data provenance and will review the requirements these applications impose on the provenance models we have discussed in Chapter 2. We will reflect on these requirements when discussing provenance systems and algorithms for capturing, storing, and analyzing provenance in Chapter 4.

3.1 View Maintenance, What-if Analysis, and Provisioning

The potential for provenance to support **maintenance** of materialized views (Gupta, Mumick, *et al.*, 1995) has been recognized early-on (Cui and Widom, 2001; Buneman *et al.*, 2002; Cong *et al.*, 2012). When the results and provenance for a view are materialized, then the effect of deleting tuples from the input database on the view can be determined based on the provenance of the view alone. As discussed in Section 2.9.3, homomorphisms commute with queries over $\mathcal{K}$-relations and, thus, factor through queries. Note that given the provenance in semiring $\mathbb{N}[X]$, deletion of a set of tuples is a homomorphism (replacing the variables that correspond to these tuples with 0). This application of homomorphisms is called **deletion propagation**. Deletion propagation

Name	Age	Card	$\mathbb{N}[X]$
Peter	39	Visa	x_1
Alice	25	AE	x_2
Bob	25	Visa	x_3
Astrid	26	Master	x_4

(a) Customers

Customer	Item	NumItems	Date	$\mathbb{N}[X]$
Peter	Lettuce	3	2020-01-03	y_1
Peter	Oranges	1	2020-01-03	y_2
Peter	Lettuce	3	2020-01-04	y_3
Bob	Oranges	2	2020-01-04	y_4
Alice	Peanuts	3	2020-01-04	y_5

(b) Orders

$$\begin{aligned} Q_{custWithLargeOrders} &:= \Pi_{Name}(\mathsf{Customers} \bowtie_{Name=Customer} Q_{largeOrders}) \\ Q_{largeOrders} &:= \Pi_{Customer}(\sigma_{NumItems>3}(\mathsf{Orders})) \end{aligned}$$

(c) Query $Q_{custWithLargeOrders}$

Name	$\mathbb{N}[X]$
Peter	$x_1 \cdot (y_1 + y_3)$
Alice	$x_4 \cdot y_5$

(d) Query Result

Figure 3.1: Example of using an $\mathbb{N}[X]$-relation for deletion propagation

is a special case of view maintenance where we want to update the materialized result $Q(D)$ of a query Q to reflect the deletion of a set of tuples ΔD from D, i.e., we want to compute $Q(D - \Delta D)$ from $Q(D)$ and ΔD.

Example 53 (Deletion Propagation with Provenance). In Figure 3.1 we revisit a previous example for provenance polynomials. Recall that the query shown in this figure returns customers with large orders. Assume we want to evaluate how the deletion of tuple y_1 (highlighted in red) from the input affects the result of the query. Such a deletion can be implemented as a semiring homomorphism $\mathbb{N}[X] \rightarrow \mathbb{N}[X]$ defined through a variable assignment μ that is the identity for all tuples we want to keep and maps all variables to 0 (recall that tuples are annotated with 0 are not part of the relation) for tuples we would like to delete.

We then apply this homomorphism to the query result to evaluate the effect of the deletion on the result of the query. The homomorphism $\textsc{eval}_\mu$ based on the variable assignment μ shown below implements the deletion of y_1.[1]

$$\mu(x) = \begin{cases} 0 & \textbf{if } x = y_1 \\ x & \textbf{otherwise} \end{cases}$$

For the first tuple in the query result we get

$$\textsc{eval}_\mu(x_1 \cdot (y_1 + y_3)) = x_1 \cdot (0 + y_3) = x_1 \cdot y_3$$

That is the tuple still exists when y_1 is deleted.

In general, provenance models for positive queries are not sufficient to deal with insertions, because, as discussed in Section 2.4.3, they do not record the provenance of missing answers. A more powerful model that supports queries with negation and missing answers (these two are intimately related as has been shown in Köhler *et al.* (2013)) is needed to deal with both inserts and deletions of input data (Grädel and Tannen, 2017; Xu *et al.*, 2018; Lee *et al.*, 2017; Lee *et al.*, 2018). Ikeda and Widom (2010) and Ikeda *et al.* (2011b) use provenance information to selectively refresh outputs of a workflow based on changes to input data. Psallidas and Wu (2018b) use provenance to refresh visualizations in the context of an interactive visualization system. In what-if analysis, the effect of hypothetical changes to data on a query result is evaluated. Thus, what-if analysis is a specific type of view maintenance. If the possible hypothetical scenarios are known up-front then one can provision for these scenarios, that is compute auxiliary data structures upfront to speed-up the subsequent what-if analysis. Provisioning is typically based on compacting and simplifying provenance expressions (Deutch *et al.*, 2013a; Deutch *et al.*, 2013b; Assadi *et al.*, 2016). Assadi *et al.* (2016) did study algorithms that approximate answers to what-if queries by creating sketches for a given set of hypothetical scenarios expressed as subsets of the database that can be combined to form scenarios.

[1] Recall that any assignment of variables to a elements of a semiring $\mathcal{K}$ extends to a homomorphism $\textsc{eval}_\mu : \mathbb{N}[X] \to \mathcal{K}$ that replaces each variable x in a polynomial with $\mu(x)$ and then evaluates the resulting expression in $\mathcal{K}$.

Example 54 (What-if). Let us provision for two hypothetical scenarios for the database and query from Figure 2.10. Say we want to evaluate the effect of removing either all customers with Visa cards or all customers with Master cards. We can provision for these scenarios by replacing all variables corresponding to these subsets of the input database with a single variable representing the scenario, e.g., replacing x_1 and x_3 (customers with Visa cards) with x_{visa}, and all other variables with a multiplicity (1 for all tuples in the example). Evaluating the query over this "conditioned" database we get the relation shown below.

Name	$\mathbb{N}[X]$
Peter	$2 \cdot x_{visa}$
Alice	x_{master}

By assigning x_{visa} and x_{master} to either zero or one, we can evaluate four different hypothetical scenarios:

- $x_{visa} = 1 \land x_{master} = 1$: this is the original database.
- $x_{visa} = 1 \land x_{master} = 0$: deletion of all customers with Master cards. Under this scenario only the first query result tuple exists.
- $x_{visa} = 0 \land x_{master} = 1$: deletion of all customers with Visa cards. Under this scenarios only the second query result exists.
- $x_{visa} = 0 \land x_{master} = 0$: deletion of all Visa and Master card customers resulting in an empty query result.

3.2 View Update and How-to Analysis

While view maintenance is concerned with computing how modifications to the inputs of a query impact the query's result, **view update** takes as input a change ΔQ to the result of a query Q and the task is to compute an update ΔD to the input database such that $Q(D \cup \Delta D) = Q(D) \cup \Delta Q$. Deletion propagation is a restricted version of the view update problem where only deletions to the view are allowed and these have to be translated into deletions of the input database. Note that there are instances of the view update problem where no solution exists, because either there exists no input D' such that $\Delta Q \subseteq Q(D')$ or any such D' causes side-effects on the view, i.e., additional undesired tuples to be inserted or some of tuples $Q(D)$ to be deleted. Naturally, one would like to optimize for solutions that minimize side-effects on the view (or produce no side-effects if this is possible). Given an instance of the view update problem, ideally one would like to also minimize the side-effects on the input database. That is, we only want changes that are necessary for producing the requested update to the view.

As discussed above, provenance polynomials are sufficient for evaluating the effect of input deletions on the result of a positive relational algebra query (or equivalently a Datalog query without recursion and negation). This implies that the provenance of a query is also enough to find solutions for the deletion propagation problem for both set and bag semantics (and in fact for any semiring). To see why this is the case consider the following brute force approach which takes as input a query Q, an input $\mathcal{K}$-database D, and an update ΔQ. The approach first evaluates Q in $\mathbb{N}[X]$ over an abstractly-tagged version of D, i.e., a $\mathbb{N}[X]$-database $D_{\mathbb{N}[X]}$ where each tuple t for which $D(t) \neq \mathbb{0}_{\mathcal{K}}$ holds is annotated with a unique variable x_t. We then enumerate all possible assignments μ of variables from the provenance polynomials of the query result $Q(D_{\mathbb{N}[X]})$ to elements from $\mathcal{K}$ such that $\mu(x_t) \preceq_{\mathcal{K}} D(t)$ (we can only reduce annotations which corresponds to a deletion). For each such μ we apply the homomorphism $\textsc{eval}_\mu$ to test whether the desired deletion is achieved and what the side-effects on the view are (other tuples that got deleted). The source side-effects are simply the number of variables mapped to $\mathbb{0}_{\mathcal{K}}$ by μ. Of course this brute force algorithm is highly inefficient because the number of possible assignments is 2^n where n is the number of tuples in the input. Buneman *et al.* (2002) studied the complexity of the view deletion problem for union of conjunctive queries when either the side-effects on the input or on the view should be minimized and noted its connection to annotation placement (essentially view update for annotated relations). Cong *et al.* (2012) did built on this work and investigated the complexity of these problems for subclasses of the cases considered by Buneman *et al.* (2002). How-to queries (Meliou and Suciu, 2012) are a variation of the view update problem, where the user can specify constraints on what changes to the input database are allowable.

Example 55 (View Deletion with Provenance)**.** Reconsider the query result from Figure 3.1d and let us explore the following view deletion problem for bag relations: we want to delete tuple (*Peter*) while minimizing side-effects on the query result (view). For the sake of this example let us assume that all input tuples appear with multiplicity 1. That means we have to find an assignment of variables to either 0 or 1 that is a solution to the following optimization problem:

$$\begin{aligned} &\textbf{maximize} \; \textsc{eval}_\mu(x_4 \cdot y_5) && \text{(keep } (\mathit{Alice}) \text{ if possible)} \\ &\textbf{subject to} \; \textsc{eval}_\mu(x_1 \cdot (y_1 + y_3)) = 0 && \text{(delete } (\mathit{Peter})\text{)} \end{aligned}$$

Two solutions without view side-effects are:

- Delete Peter from the customer relation ($x_1 = 0$)
- Delete orders y_1 and y_3 ($y_1 = 0$ and $y_3 = 0$)

For example, under the first solution the updated query result is:

Name	$\mathbb{N}[X]$
Peter	0
Alice	1

To use set semantics instead of bag semantics we simply use semiring $\mathbb{B}$ instead of $\mathbb{N}$.

Readers familiar with linear programming or constraint optimization in general may have observed already that the optimization problem we solved in the example above can be expressed as an integer linear program. In fact, view update problems have been cast as constraint optimization problems in the past (Shu, 2000). Another example are How-to queries that have also been implemented in Meliou and Suciu (2012) using MILP (mixed integer linear programming).

3.3 Error Diagnosis and Debugging

One of the most wide-spread use cases for data provenance is debugging of operations and data. For instance, data and transformation dependencies are used to track an erroneous or suspicious query result back to the data and/or operations that caused this result. Examples of this usage of provenance and related techniques are error diagnosis with view-conditioned causality (Meliou *et al.*, 2011), retroactive debugging of transactions with provenance (Niu *et al.*, 2017a), exposing intermediate results for debugging SQL queries (Grust *et al.*, 2011; Dietrich and Grust, 2015), debugging for dataflow programs (Gulzar *et al.*, 2017a; Gulzar *et al.*, 2017b; Interlandi *et al.*, 2018) including watch-points for declarative programs and delta debugging, and automated fault localization (Alvaro *et al.*, 2015).

3.4 Metadata Management, Versioning, and Reproducibility

Provenance provides a record of how data was derived from other data and by which transformations. Such information is essential for finding relevant data in large data repositories, to understand the version history of datasets and to provide evidence of how data was derived for reproducibility of computational experiments.

3.4.1 Metadata Management and Versioning

The emergence of data lakes has resulted in a need to track dataset and code versions as well as their inter-dependencies. Systems like Apache Atlas (Apache, 2017) and Ground (Hellerstein *et al.*, 2017) manage coarse-grained workflow provenance to enable users to make sense of dataset derivation in such an uncontrolled environment. Similarly, the need for provenance to keep track of dataset versions was also recognized in the context of the DataHub project (Bhardwaj *et al.*, 2015; Bhattacherjee *et al.*, 2015; Chavan *et al.*, 2015). In the context of workflow systems, it has been recognized that in addition to tracking the provenance of a workflow execution it is also necessary to track the provenance of the evolution of the workflow specification (Koop, 2016; Callahan *et al.*, 2006). For example, this idea has been implemented in VisTrails (Scheidegger *et al.*, 2008) and Vizier (Brachmann *et al.*, 2019).

3.4.2 Tracking Sources in Data Integration and Curation

Data integration and curation techniques enable heterogeneous data from multiple sources to be cleaned and transformed to prepare it for analysis. For example, virtual data integration enables data from multiple sources to be queried using a unified schema. Provenance information can help to track data from multiple sources and explains how data was transformed by the integration process. Ives *et al.* (2008) and Green *et al.* (2010) use provenance to track trust for update exchange in a collaborative data sharing system. The TRAMP system (Glavic *et al.*, 2010) tracks the provenance of schema mappings in data exchange and their implementation as transformations in a declarative query language. SPIDER (Alexe *et al.*, 2006) uses transformation and data dependencies to explain how tuples are derived through schema mappings. Dong and Srivastava (2013) uses provenance to explain data fusion decisions. DeepDive (Niu *et al.*, 2012) utilizes provenance for knowledge-base construction.

3.4.3 Reproducibility

Reproducibility of computational experiments is a challenging task (Janin *et al.*, 2014; Pimentel *et al.*, 2019; Freire *et al.*, 2012; Davison *et al.*, 2014), because the results of an experiment and whether it will be possible to repeat the experiment may depend on details of the execution environment. Furthermore, just setting up the right environment is insufficient for reproducibility since the right operations have to be executed in the right order to repeat a computation. For example, consider a complex analysis workflow that requires the right set of shell scripts to be executed in the right order and requires the right input data to be placed in a particular location in the user's file system. Provenance can be used to complement approaches such as containerization and packaging of environments (Chirigati *et al.*, 2013; Pham *et al.*, 2015; That *et al.*, 2017; Pham *et al.*, 2013). Both workflow and OS provenance information have been used successfully to aide reproducibility (Freire *et al.*, 2011; Freire and Silva, 2012; Chirigati *et al.*, 2013; Pimentel *et al.*, 2019; Freire and Chirigati, 2018; Pham *et al.*, 2013; Pham *et al.*, 2015).

3.5 Incomplete and Probabilistic Databases

Provenance has proven to be essential in probabilistic databases (Van den Broeck and Suciu, 2017; Suciu, 2020; Suciu *et al.*, 2011; Benjelloun *et al.*, 2006). In this field, provenance information is often referred to as lineage. It was shown that provenance in semiring PosBool[X] (see Section 2.3.4) can be used to compute the probability of the results of a query evaluated over a tuple independent database, i.e., a database where tuples are independent events associated with a marginal probability (the probability that the tuple exists). Importantly, tuple independent databases are insufficient as a closed representation system for probabilistic data. That is, it is not possible to evaluate queries over the result of another query. Closedness can be achieved if provenance is maintained (Benjelloun *et al.*, 2006; Van den Broeck and Suciu, 2017; Fink *et al.*, 2012). We omit the technical details here and instead refer the interested reader to one of the excellent overview articles on the topic (Van den Broeck and Suciu, 2017; Suciu, 2020; Suciu *et al.*, 2011).

Example 56 (Probabilistic Query Processing with Provenance)**.** Reconsider the example from Figure 3.1d. Let us assume that the input is a tuple-independent database with marginal probabilities:

$$\begin{array}{llll} p(x_1) = 1.0 & p(x_2) = 0.8 & p(x_3) = 0.2 & p(x_4) = 0.3 \\ p(y_1) = 0.4 & p(y_2) = 0.3 & p(y_3) = 0.2 & p(y_4) = 0.1 \\ p(y_5) = 0.3 & & & \end{array}$$

The semantics of queries over a probabilistic database are defined using the so-called possible world semantics. Given a probabilistic database

$$\mathcal{D} = \{D_1, \ldots, D_n\}$$

which is a set of possible worlds, deterministic databases that each encode one possible state of the real world, and a probability distribution P over these worlds, the result $Q(\mathcal{D})$ of a query Q over $\mathcal{D}$ is the set of worlds created by evaluating Q over each possible world in $\mathcal{D}$:

$$Q(\mathcal{D}) = \{Q(D) \mid D \in \mathcal{D}\}$$

The probability of a possible query result R is the sum of the probabilities of all possible worlds that yield R:

$$P[Q(\mathcal{D}) = R] = \sum_{D \in \mathcal{D} : Q(D) = R} P(D)$$

Note that, while clean and simple, this definition is not practical, because for typical probabilistic data models, the number of possible worlds can be exponential in the size of the representation. For instance, the number of possible worlds encoded by a tuple-independent probabilistic database D with n tuples is 2^n (all subsets of D). Thus, it is preferable to evaluate queries over a more compact encoding. Without presenting all of the technical details, recall from Section 3.2 that for sufficiently expressive provenance models (enjoying the computability property discussed in Section 2.1.7), we can evaluate the effect of all possible deletions of input tuples on the result of query using only the provenance of the query result. This can be exploited for probabilistic query processing: each assignment of the variables in the provenance to 0 or 1 corresponds to one possible world of the input probabilistic database. Thus, we can (i) evaluate the query with provenance tracking (e.g., semiring $\mathsf{PosBool}[\mathsf{X}]$) and (ii) treat each variable x as a random variable such that $P[X = 0] = 1 - p(x)$ and $P[X = 1] = p(x)$. The problem of computing the probability of a query result then boils down to computing the probability that the provenance formula evaluates to true.[2]

Consider the first result tuple $t_{Peter} = (Peter)$ of our example query. Under semiring $\mathsf{PosBool}[\mathsf{X}]$ its annotation is

$$x_1 \wedge (y_1 \vee y_2)$$

Recall that for independent events $P(A \wedge B) = P(A) \cdot P(B)$ and $P(A \vee B) = 1 - (1 - P(A)) \cdot (1 - P(A))$. Since the input tuples are assumed to be independent in an tuple-independent probabilistic database, the

[2] Note that this is equivalent to the well-known problem of weighted model counting which is known to be intractable (#P-hard). See, e.g., Van den Broeck and Suciu (2017).

probability of this formula being true is:

$$\begin{aligned} p(t_{Peter}) &= p(x_1) \cdot (1 - (1 - p(y_1)) \cdot (1 - p(y_3)) \\ &= 1 \cdot (1 - (1 - 0.4) \cdot (1 - 0.2)) \\ &= (1 - 0.6 \cdot 0.8) = 0.52 \end{aligned}$$

Note that in general even if the inputs are independent, queries may introduce correlations and, thus, the simple and efficient formulas used above are often not applicable.

In addition to providing a technical tool for computing probabilities and for achieving closedness for queries over probabilistic data, provenance can also aide in explaining how uncertainty in input data or uncertainty introduced by heuristic curation operations causes the output of an analysis to be uncertain. Note that most cleaning and curation algorithms are heuristic in nature in that they choose one possible repair among the space of all possible repairs. While methods for evaluating computations over probabilistic and uncertain data can quantify the uncertainty of analysis results, provenance is needed to identify its causes (Yang *et al.*, 2015; Brachmann *et al.*, 2019).

3.6 Security, Privacy, and Auditing

Provenance information can be utilized for a wide range of use cases related to security and privacy.

3.6.1 Access Control

Fine-grained access control (Wang *et al.*, 2007) enables access control policies to be specified at the level of tuples or individual cells (attribute-values of a tuple). When evaluating queries over databases with fine-grained access control policies, it is necessary to ensure that no information about tuples and attribute-values a user has no access to are exposed by query results. According to Wang *et al.* (2007), a query evaluation algorithm should be *sound* (it does only return answers that would be in the result of the query without access control, perhaps with attribute values replaced with NULL values to "mask" them) and *secure* which is defined as that the same answer is returned for all databases that are indistinguishable under the access control policies (e.g., two databases that only differ in tuples that are access restricted by the policy are equivalent because these tuples cannot be exposed).

The purpose of **provenance-based access control** (Nguyen *et al.*, 2013; Park *et al.*, 2012) is to enable access control policies that are based on provenance information. For example, a user should be given access to any data that is derived from data they have produced.

3.6.2 Auditing

Many organizations keep records of their data operations to detect security breaches, for forensic investigations, or to comply with regulations. **Audit logs** (Becker and Chambers, 1988) keep a record of operations executed on a database. Audit logs can be used to prove that a business complies with data management regulations such as who in the organization should be allowed what type of access to which data. When combined with time travel, audit logging provides a full record of past operations and states of a database (Arab *et al.*, 2018b). Provenance information can complement such audit logs and time travel by providing data and control dependencies. For instance, to understand which data was affected by a security breach, we need to understand which data was modified directly or was indirectly affected by the actions of a compromised accounts. Provenance for transactional workloads (Arab *et al.*, 2018b) provides precisely this information.

Another type of auditing that has been proposed in the literature is to determine whether a query exposed any sensitive information. This line or work typically defines disclosure based on intervention: if deleting a tuple from the input does not affect the result of the query, then the information of this tuple is assumed to not have been exposed (Agrawal *et al.*, 2004). Note that this is essentially the notion of counterfactual causality we have discussed in Section 2.1.4. Kaushik *et al.* (2013) and Kaushik and Ramamurthy (2011) study how to determine this type of causes efficiently. Methods for capturing counterfactual causes are also applicable to this problem. We discussed in Section 2.1.4 that counterfactual causes are insufficient for queries that are disjunctive in nature. Thus, the notion of disclosure used for auditing queries is too weak to prevent exposure of sensitive information for such queries.

3.7 Explanations for Outcomes

The need to explain an outcome arises in many application domains. While there is no commonly accepted definition of what constitutes an explanation, a common theme is that explanations are high-level descriptions of root causes for an observation meant for consumption by an end user. Covering the large and diverse body of work on explanations in detail is far beyond the scope of this article. We refer the reader to Glavic *et al.* (2021) for a recent overview article. Here we limit the discussion to clarifying how provenance information can aide explanation tasks.

3.7.1 Explaining Query Results with Provenance Summaries

Provenance in its raw form, while providing sufficient information for explaining outcomes, may be to detailed to be suitable for human consumption. For instance, the size of fine-grained provenance for an aggregation query, e.g., using lineage, may be linear in the input size. Ideally, higher-level explanations should be generated based on provenance to explain an output. Techniques that have been used to produce such explanations include

- **Declarative summaries.** Declarative descriptions of what data belongs to the provenance, e.g., through selection patterns (El Gebaly *et al.*, 2014; Roy and Suciu, 2014; Wu and Madden, 2013; Lee *et al.*, 2020) can be used to summarize provenance. For example, the provenance contains all cities where `state = IL`.

- **Taxonomies**. Taxonomies can be used to replace sets of tuples with a higher-level concept that subsumes them (Cate *et al.*, 2015; Glavic *et al.*, 2015), e.g., replacing a set of persons that are working in politics with the concept `politician`.

- **Collapsing provenance graphs**. These techniques hierarchically group nodes of a complex provenance graph into subgraphs. The user can explore the provenance by collapsing and expanding such subgraphs. For instance, the nodes of a workflow provenance graph may be grouped into modules (Biton *et al.*, 2007; Biton *et al.*, 2008).

- **Natural language explanations**. Some approaches use *natural language* to describe provenance (Deutch *et al.*, 2017; Deutch *et al.*, 2020; Deutch *et al.*, 2018a).

We will discuss techniques for provenance summarization in more depth in Section 4.2.

3.7.2 Explanations for Machine Learning Models and Predictions

Explainability of machine learning predictions is one of the fundamental research challenges of today. To understand a prediction one needs to understand why a machine learning model did produce a prediction for a given input, but also how the model was generated and how the training data and training method indirectly influenced the prediction, e.g., to detect bias and ensure fairness criteria (Salimi *et al.*, 2018; Farnadi *et al.*, 2018; Jagadish *et al.*, 2019). In addition to aiding explainability and fairness analysis, provenance has been used to diagnose problems in ML pipelines (Zhang *et al.*, 2017). Furthermore, causality (which is intimately related to provenance) has been used to guide users on what non-sensitive information they can safely make public without risking that some sensitive information can be inferred from this data using a set of machine learning models (Fernandez *et al.*, 2019). Recently, provenance and methods from the ML community (the influence functions from Koh and Liang (2017)) have been combined to explain errors in the result of queries that have access to the predictions of an ML model. By combining these two techniques, the errors can be traced back to the training data based on which the model was created. Deutch and Frost (2019) generates a minimal perturbation of the input to be classified fulfilling a set of user provided constraints that causes an requested change to the classification outcome. Wu *et al.* (2020) uses a provenance model for linear algebra operations (Yan *et al.*, 2016) to incrementally update regression models.

3.8 Additional Applications

3.8.1 Query Equivalence and Containment Checking

Query equivalence and containment are fundamental problems that have been studied extensively in database research. Unfortunately, these problems are computationally hard or undecidable, even for relatively simple query classes. There is a lack of practical, best-effort techniques for proving or disproving equivalence and containment of queries. Provenance models that use symbolic expressions to encode the evaluation of a query over multiple database instances, can be utilized to reason about the behavior of a query over all possible database instances as required for query equivalence and containment. Recent work on checking equivalence of relational queries (Chu *et al.*, 2018) explored this idea. This work did extended $\mathcal{K}$-relations by defining a new class of mathematical structure called U-semirings which have additional operations representing, e.g., negation, and are used to prove query equivalence.

3.8.2 Performance Monitoring and Observability

Understanding and debugging the performance behavior of DISC systems and other distributed systems can be challenging. Even at the most basic level, performance monitoring requires the orchestration of monitoring tools across the machines of a cluster, gathering and aggregation of the collected information, continuous monitoring for *"interesting"* events, and analysis of root causes for events of interest. At the most basic level, provenance can aid in the drill down involved when trying to understand what causes an event. Even low-level performance monitoring is already challenging in a distributed setting. When debugging the behavior of a DISC system which execute programs written in a high-level language, then the additional challenge arises of how to connect the high-level operations expressed by the user to the low-level performance events captured by monitoring tools. Provenance information (Gulzar *et al.*, 2017a) for such higher-level dataflow languages can help to track performance information at the right level of detail (Olston and Reed, 2011; Interlandi *et al.*, 2016).

3.8.3 Recommendations and Query-by-Example

The need to recommend data or workflow steps arises in many applications. For instance, to guide a user to construct a data wrangling pipeline (Kandel *et al.*, 2011; Guo *et al.*, 2011) by recommending the most likely next step based on a repository of past interactions. A detailed record of how data has been derived from other data and through which operations can provide important features for learning recommendations. In Scheidegger *et al.* (2008) a query-by-example approach based on provenance was proposed to create workflows by analogy. Similarly, Query-by-explanation (Deutch and Gilad, 2016; Deutch and Gilad, 2019) constructs a query based on data examples with provenance. Ives *et al.* (2012) proposed to employ methods from link analysis (e.g., PageRank) to identify important nodes in provenance graphs and use this information for the purpose of recommendation.

4

Provenance Capture, Storage, and Querying

Having discussed formal provenance models and applications of provenance, in this chapter we will discuss algorithms and systems for capturing, storing, compressing, summarizing, and querying data provenance. We will start by reviewing different mechanisms for representing provenance information. A major focus will be methods for compact representation of provenance for storage as well as how to summarize it for human consumption. Afterwards, we will cover algorithms for capturing provenance information and discuss how these algorithms can be implemented on-top of database systems or as extensions of DBMS kernels. Finally, we will discuss approaches for querying and analyzing provenance information and give an overview of provenance management systems.

4.1 Storage, Compression, and Summarization

When storing provenance we have to decide where to store it and how to encode it. Either we can store provenance inside the system for which we are recording provenance, e.g., store the provenance of relational queries inside the database, or outside the system. Storing provenance inside the system has the advantage that we do not have to ship information between systems. However, this comes at the potential cost that we inherit the limitations of the data model of the system. For example, a native graph model may be more efficient for storing a graph-based provenance model than a relational database. Storing provenance outside of the system that executes the transformations for which we want to capture provenance is not subject to these limitations, but has the disadvantage that we do not benefit from the functionality build into the system for managing the recorded provenance information. For example, if we store the provenance of a relational query inside a relational database, then we can query provenance using the database's query language. Furthermore, when capturing provenance we have to transfer this data from the system under observation to the system we are use for provenance storage. None of these two approaches is fundamentally superior to the other. Rather they represent different trade-offs. In addition to choosing where to store provenance information we have to decide how to encode it. As we will see in the following, the choice of encoding can not only affect size, but may also affect the runtime of provenance capture and querying.

4.1.1 Storing Provenance As Trees and Graphs

We start with a trivial observation, that nonetheless will be relevant to our discussion of provenance compression in the following: many of the provenance models we have discussed in Chapter 2 have a natural representation as trees or graphs. One example are provenance games which are defined as graphs. Another example are provenance polynomials which can be expressed as expression trees which are binary trees whose internal nodes are labeled with "+" and "×" and whose leaf nodes are labeled with variables. For example, the expression tree for the polynomial $x_1 \cdot (x_2 + x_3)$ is shown in Figure 4.1 (left). Even for provenance models like Lineage that encode binary relationships between the inputs and outputs of a transformation, graph-based representations may be beneficial if we want to track provenance across multiple transformations (or for intermediate results of a single transformation).

Example 57 (Using graphs to store provenance for multiple transformations)**.** Consider a user running a query $Q_1 := \gamma_{hobby;avg(age) \rightarrow avgage}(\mathsf{person})$

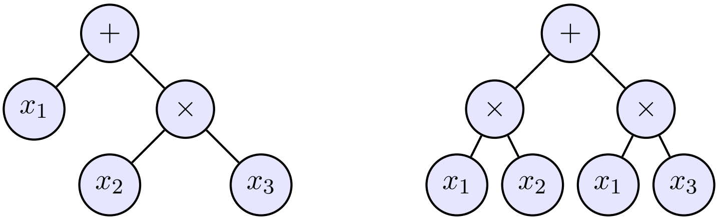
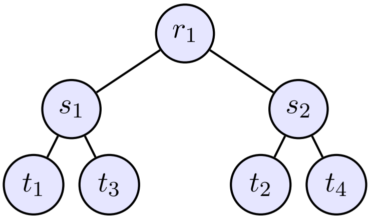


Figure 4.1: Encoding provenance polynomial $x_1 \cdot (x_2 + x_3)$ as an expressions tree. This polynomial can also be written as $x_1 \cdot x_2 + x_1 \cdot x_3$ resulting in an equivalent, but larger tree.



person

name	**hobby**	**age**	id
Alice	Hiking	45	t_1
Peter	Movies	32	t_2
Bob	Hiking	35	t_3
Gert	Movies	52	t_4

Q_1

hobby	**avgage**	id
Hiking	40	s_1
Movies	40	s_2

Q_2

cnt	id
2	r_1

Figure 4.2: Storing Lineage for multiple transformations using graphs.

which returns the average age of persons pursuing a particular hobby. Afterwards, the user runs a query $Q_2 \coloneqq \gamma_{count(*) \rightarrow cnt}(\sigma_{avgage \geq 40}(Q_1))$ to determine the number of hobbies where the average age of persons that pursue this hobby is greater than 39. An example database is shown in Figure 4.2. The lineage of the final result r_1 is $\{s_1, s_2\}$, the two tuples in the result of Q_1. Each of these tuples depends on the two input tuples belonging to the tuple's group. We can encode this information as a graph where the nodes are tuples and the edges represent lineage relationships between inputs and outputs (Figure 4.2, top). For instance, s_1 is connected to t_1 and t_3, the tuples that are in tuple s_1's lineage.

4.1.2 Compressing Provenance

Provenance models that are based on data dependencies (and potentially encode additional information such how inputs have been combined) encode a relationship between the outputs and the inputs of a transformation. Thus, in the worst-case the size of the provenance may be quadratic in the size of the input and output. When provenance is recorded for multiple transformations as in Example 57 or for intermediate results produced by a transformation, then the provenance's size can increase by a factor that depends on the number of transformations (or number of intermediate results if we choose to break down a transformation into multiple steps). In the following we will discuss techniques for compressing provenance information. Some of these techniques are applicable to any provenance model that can be encoded as a dependency graph while others exploit the semantics of specific provenance models.

4.1.3 Sharing Common Subexpressions

The reason we emphasized the representation of provenance as trees and graphs, is that this representation allows provenance to be compressed by sharing common subgraphs instead of repeating them. For example, consider the provenance polynomial shown below:

$$x_1 \cdot (x_2 + x_3) + x_4 \cdot (x_2 + x_3)$$

The subexpression $(x_2 + x_3)$ appears twice. Figure 4.3 (top) shows the expression tree for this polynomial. We can compress the polynomial by sharing the tree $(x_2 + x_3)$ resulting in the directed acyclic graph (DAG) shown in Figure 4.3 (bottom). The observation that provenance can be compressed was already made in early work on provenance compression (Chapman *et al.*, 2008; Anand *et al.*, 2009). Note that these approaches are mostly independent of the provenance model as long as provenance is represented as a graph.

In addition to sharing common subexpressions, other compression techniques that have been explored in Chapman *et al.* (2008) and Anand *et al.* (2009) are encoding sets of dependencies based on subset or subsequence relationships, e.g., instead of listing all dependencies of a data

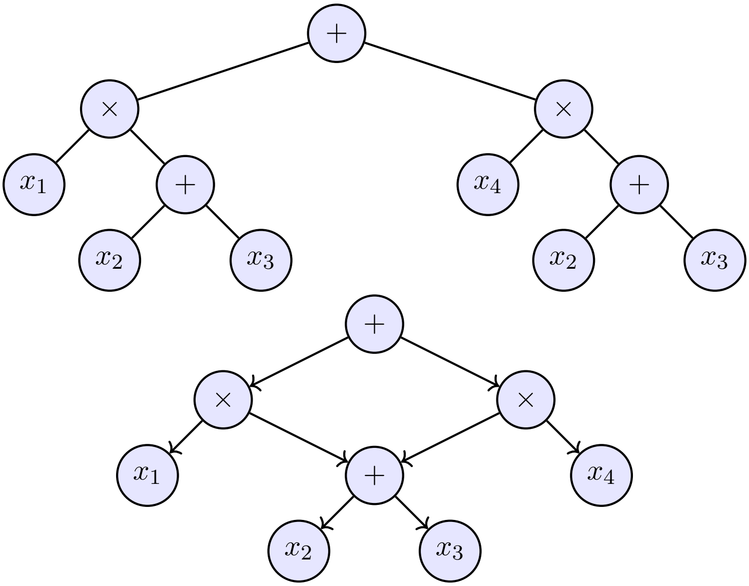


Figure 4.3: Compressing the tree representation of a provenance polynomial by sharing a common subexpression $(x_2 + x_3)$ resulting in a more compact representation as a DAG.

item, one may state how its set of dependencies differs from the set of dependencies of another data item. For example, consider two data items d_1 and d_2 with dependencies $S_1 = \{a, b, c, d\}$ and $S_2 = \{b, c, d, e\}$, respectively. The set of dependencies for d_2 can be expressed as $S_1 \cup \{e\} - \{a\}$. Furthermore, basic assumptions about the structure of provenance, e.g., dependencies are assumed to be transitive, can be used to further compress provenance. For example, if data item d_1 depends directly on data item d_2 and d_2 depends on d_3 then under the assumption that data dependencies are transitive, d_1 depends on d_3. If data dependencies are transitive, then it suffices to only store direct dependencies, because all remaining dependencies can be generated by computing the transitive closure of the set of direct dependencies. Generalizing the example of transitivity, provenance can be compressed by stating declarative constraints that have to hold for the provenance and then removing from the provenance all dependencies that can be inferred based on these constraints.

4.1.4 Factorizing Provenance

For provenance models that are based on algebraic structures such as provenance polynomials and any other provenance model that can be expressed in the semiring framework, compression can be achieved by rewriting expressions using the algebraic laws of these structures (e.g., distributivity). For example, consider the polynomial $x_1 \cdot x_2 + x_1 \cdot x_3$. Using distributivity, this polynomial can be written as $x_1 \cdot (x_2 + x_3)$. As another example, consider the $\mathsf{PosBool}[\mathsf{X}]$ expression $x_1 \vee (x_1 \wedge x_2)$. In semiring $\mathsf{PosBool}[\mathsf{X}]$, absorption can be used to compress formulas. For instance, $x_1 \vee (x_1 \wedge x_2)$ is equivalent to x_1. In addition to sharing of common subexpressions using distributivity, formulas can sometimes be further compressed by using graphs to share common subexpressions in different parts of an expression. We already discussed an example for such sharing when discussing general compression strategies above. Reconsider the example polynomial shown in Figure 4.3. An alternative, equivalent, way to encode such sharing is to allow subexpressions to be assigned to variables. For example, the polynomial shown in this figure is $x_1 \cdot (x_2 + x_3) + x_4 \cdot (x_2 + x_3)$. We can give an identity to the subexpression $(x_2 + x_3)$, e.g., $x_{sub} = (x_2 + x_3)$. Then the polynomial can be written as

$$\begin{aligned} x_{sub} &= x_2 + x_3 \\ x_1 \cdot x_{sub} + x_4 \cdot x_{sub} & \end{aligned}$$

Note that in this example we could have also used distributivity to rewrite the polynomials as

$$(x_1 + x_4) \cdot (x_2 + x_3)$$

.

In general, however, graph-based representations may be more compact than any equivalent tree-based representation produced by distributivity. Note that these types of graphs are the arithmetic version of the Boolean circuits that are used for encoding the provenance of recursive queries as discussed in Section 2.5.1 except that for non-recursive queries these circuits do not contain cycles.

Given these mechanisms for compressing a provenance polynomial, an interesting question is how to efficiently find the smallest possible encoding of the polynomials. We can study either the offline problem of compressing an existing provenance polynomial or the online problem of computing a compressed representation during provenance capture. The latter has the advantage that it not just reduces the size of the provenance, but also may improve the runtime of provenance capture. A variant of the latter that has been studied extensively is query evaluation in *factorized databases* (Olteanu and Schleich (2016), Olteanu and Závodný (2015), and Olteanu and Závodny (2011)). Specific attention has been paid to the problem of selecting a factorization structure for a query that results in the worst-case optimal size of the factorized provenance polynomials for this query. Here by factorization structure, we mean that provenance polynomials are structured according to a "plan" for the query. Compression of provenance is only one application of this work. Its main focus is factorization of the data itself.

As an example of the factorization structures mentioned above, consider the query $\Pi_A(R(A,B) \bowtie_{A=C} S(C,D))$. The flat provenance of a query result is a sum $\sum r_i \cdot s_j$ for some set of indexes i, j and where r_i is a variable annotating a tuple from R and s_j is a variable annotating a tuple from S. One possible factorization structure for this query is $\sum_i r_i \cdot (\sum_j s_{ij})$ which avoids repetition of each annotation r_i of a tuple from R. This is achieved by multiplying r_1 with the sum of all annotations of tuples from S that join with a the tuple of R annotated with r_i. The details of how the worst-case optimal factorization structure is chosen is beyond the scope of this paper. We refer the reader to Olteanu and Závodný (2015) and Olteanu and Schleich (2016). There are several systems that implement this approach, e.g., FDB (Bakibayev *et al.*, 2012) is a database that natively uses factorized representations of data and the Pug system (Lee *et al.*, 2018) applies the technique from Olteanu and Závodný (2015) to rewrite an input Datalog program to compute provenance graphs with worst-case optimal size. Compressed representations of algebraic and Boolean expressions have been studied in many other contexts, e.g., for tractable query evaluation over probabilistic data and for solving specific classes of intractable problems (e.g., constraint solving) in polynomial time.

4.1.5 Encoding Provenance in the Host Data Model

So far we have discussed how to compactly encode provenance information. As mentioned in the beginning of this section, another choice we have to make is where to store provenance information. If we choose to store provenance natively inside the system for which we are recording provenance, then one option to realize this idea is to encode provenance in the native data model of this system (we refer to this as the *host* data model). We will review three such storage themes here.

Relational Encoding of Provenance Polynomials in Perm

This approach was pioneered in the Perm system Glavic (2010), Glavic and Alonso (2009b), Glavic and Alonso (2009c), Glavic *et al.* (2013b), and Glavic and Alonso (2009a) for encoding provenance according to the PI-CS model as annotations on data. Recall that for positive relational algebra this model encodes provenance polynomials. Here we will focus on the encoding of provenance polynomials only and refer the interested reader to Glavic and Alonso (2009c) and Glavic *et al.* (2013b) for a detailed description. Consider a positive relational algebra query Q and let $(R_1, \ldots, R_n)$ be the relations accessed by Q in the order they appear as leaf nodes (left to right) of the relational algebra tree for Q. For instance, for query $R \cup (S \bowtie T)$, we get (R, S, T). To encode the provenance polynomial $p = Q(D)(t)$ for a tuple t in the result of Q over a database D, we normalize the polynomial as follows:

1. **Sum of product form**: Using the equational laws of semirings we transform p into a sum of products: $\sum_{i=1}^{m} \prod_{j=1}^{k} x_{ij}$.

2. **Ordering variables**: Then variables are sorted based on which leaf node they stem from.[1] Furthermore, we pad monomials with ones such that each monomial has n elements where n is the number of relations accessed by the query. That is, the polynomial is now of the form $\sum_{i=1}^{m} \prod_{j=1}^{n} e_{ij}$ where each e_{ij} is either 1 or a

[1] In general, it is not possible to determine with certainty which leaf node a variable stems from using the polynomial alone. Here we will ignore this aspect. In the context of the Perm system this problem does not arise, because the PI-CS provenance model used there explicitly encodes this order.

variable x stemming from the annotation of a tuple in relation R_j.

3. **Encoding monomials**: Each monomial $e_1 \cdot \ldots \cdot e_n$ is then encoded as a single tuple by concatenating the result tuple t and tuples $t_1, \ldots, t_n$. If $e_i = x$ for some variable x, then t_i is the tuple from relation R_i annotated with x. Otherwise, t_i is a tuple of null values with the same arity as R_i.[2]

4. **Encoding the polynomial**: The polynomial p is encoded as the set of tuples generated for its monomials.

Note that the approach from Glavic and Alonso (2009a) represents provenance using input tuples in lieu of variables. The rationale behind this decision is that it enables queries over provenance to access attribute values of input tuples in the provenance which is not directly possible when abstract variable symbols are used. However, for applications where only the identity (variables) of input tuples is needed this would be overkill. For such applications, one can use a different encoding of variables, e.g., using unique tuple identifiers.

Example 58 (Relational encoding of provenance polynomials)**.** Figure 4.1 shows an example query that returns the names of customers and of employees that work for at least one department. Consider the result tuple $(Peter)$. There are three tuples in the result of the two joins that are projected onto this result tuple corresponding to the three departments Peter is working for. The provenance polynomial for this result is $e_1 \cdot (x_1 \cdot d_1 + x_3 \cdot d_2 + x_3 \cdot d_3) + c_1$. The sum of products representation of this polynomial is:

$$e_1 \cdot x_1 \cdot d_1 + e_1 \cdot x_2 \cdot d_2 + e_1 \cdot x_3 \cdot d_3 + c_1$$

Ordering variables based on the order of relation accessed in the algebra tree for the query and padding the monomials with ones we get:

[2] We assume that the input database is abstractly tagged, i.e., each input tuple is annotated with a unique variable. Furthermore, we assume that there no tuples in the input that only consist of null values. For databases that violate this assumption, we can use an additional Boolean column per input relation that stores whether $e_i = 1$.

$$(e_1 \cdot x_1 \cdot d_1 \cdot 1) + (e_1 \cdot x_2 \cdot d_2 \cdot 1) + (e_1 \cdot x_3 \cdot d_3 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot c_1)$$

The relational encoding of the polynomials for all three answers of this query is shown on the bottom of Figure 4.4. Note how the 4 monomials of the provenance polynomial for $(Peter)$ are encoded as 4 tuples and how variables are represented by the tuples that are annotated with these variables in the input database.

A few remarks are in order. The advantage of this encoding of polynomials is that the provenance is stored alongside the data as a single relation that is suited well for queries that retrieve the provenance for some result tuples only (by applying a selection to filter result tuples) or retrieve only parts of the provenance (by applying selections or projections to filter out parts of provenance that are of interest). We will discuss this point in more detail in Section 4.4. A disadvantage of this encoding is that it is quite verbose and does not support some of the storage optimizations we discussed above. This encoding of provenance was used for a larger class of algebraic operators including aggregation and nested subqueries with correlations (Glavic *et al.*, 2013b; Glavic and Alonso, 2009b).

Relational Encoding of Provenance Polynomials in Orchestra

A similar encoding is used in the Orchestra system (Green *et al.*, 2010; Karvounarakis *et al.*, 2010) to store the provenance of schema mappings in update exchange. In Orchestra, a set of peers in a network specify mappings between their schemas. Based on these schema mappings and a provenance-based trust policy, Orchestra translates updates to one peer into updates to the databases of other nodes in the network. The provenance model used in this work is the extension of the semiring model with functions that represent applications of schema mappings. Recall that we already did discuss this model in Section 2.6.2. Like the encoding used by Perm, monomials are encoded as the tuples (or some unique identifier such as a key) whose variables appear in the monomial. However, this approach materializes intermediate results at the granularity of schema mappings. While this comes at the cost

emp

eid	name	$\mathbb{N}[X]$
1	Peter	e_1
2	Bob	e_2
3	Alice	e_3

empdept

eid	did	$\mathbb{N}[X]$
1	1	x_1
1	2	x_2
1	3	x_3
2	1	x_4
3	2	x_5

dept

did	dname	$\mathbb{N}[X]$
1	Sales	d_1
2	IT	d_2
3	HR	d_3

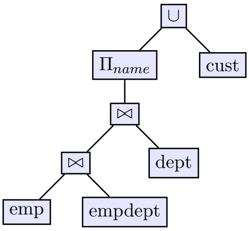
cust

cid	name	$\mathbb{N}[X]$
1	Peter	c_1
2	Malice	c_2

Query

```
SELECT name
FROM emp
     NATURAL JOIN empdept
     NATURAL JOIN dept
UNION ALL
SELECT name
FROM cust
```

Algebra tree



Query result annotated with provenance

SCH(Q)	emp		empdep		dept		cust		Padded
name	**eid**	**name**	**eid**	**did**	**did**	**dname**	**cid**	**name**	Monomial
Peter	1	Peter	1	1	1	Sales	NULL	NULL	$e_1 \cdot x_1 \cdot d_1 \cdot 1$
Peter	1	Peter	1	2	2	IT	NULL	NULL	$e_1 \cdot x_2 \cdot d_2 \cdot 1$
Peter	1	Peter	1	3	3	HR	NULL	NULL	$e_1 \cdot x_3 \cdot d_3 \cdot 1$
Peter	NULL	NULL	NULL	NULL	NULL	NULL	1	Peter	$1 \cdot 1 \cdot 1 \cdot c_1$
Bob	2	Bob	2	1	1	Sales	NULL	NULL	$e_2 \cdot x_4 \cdot d_1 \cdot 1$
Alice	3	Alice	3	2	2	IT	NULL	NULL	$e_3 \cdot x_5 \cdot d_2 \cdot 1$
Malice	NULL	NULL	NULL	NULL	NULL	NULL	2	Malice	$1 \cdot 1 \cdot 1 \cdot c_2$

Figure 4.4: Relational encoding of provenance polynomials. The provenance polynomial for each result tuple has been normalized by rewriting it into a sum of products and ordering variables of monomials based on the order of relation accesses in the relational algebra tree for the query. Result tuples whose provenance consists of more than one monomial are represented as multiple tuples in the encoding. For instance, the provenance of $(Peter)$ is $e_1x_1d_1 + e_1x_2d_2 + e_1x_3d_3 + c_1$.

of having to materializing these intermediate results, the benefit is that it may result in an encoding that is more compact compared to the Perm encoding, because of sharing of common subexpressions and factorization.

Example 59 (Factorization and Sharing)**.** By splitting the provenance polynomials into parts corresponding to individual schema mappings, Orchestra generates a factorization that reflects this structure. For example, consider the two mappings and $\mathbb{N}[X]$-instance for the relations used in the schema mappings shown in Figure 4.5. Orchestra would create two relations to store the provenance of m_1 and m_2. These relations store all successful bindings of the variables of a tgd to values. The provenance table for m_1 records that the provenance of tuple $t_d = \mathsf{Department}(HR)$ is $x_1 + x_2$. The provenance table for m_2 records that tuple $\mathsf{ManagedDepartment}(HR)$ is derived by joining t_d with $\mathsf{Manager}(Alice, HR)$ and $\mathsf{Manager}(Astrid, HR)$, i.e., its provenance is the annotation of $\mathsf{Department}(HR)$ multiplied with y_1 and y_2. This corresponds to the factorized representation of the provenance polynomial as shown as a graph in Figure 4.5 or as the expression with assignment shown below.

$$x_d = x_1 + x_2$$

$$y_1 \cdot x_d + y_2 \cdot x_d$$

Provenance Graphs in Pug

The Pug system's provenance graph model (Lee *et al.*, 2018) for Datalog queries uses factorization based on the query structure (recall that we discussed this model briefly in Section 2.4.3). By default Pug does not optimize this factorization to ensure that the structure of the provenance reflects the structure of user's query which is important for debugging queries. However, if the user is only interested in provenance polynomials, then, as mentioned above, Pug can rewrite the query to generate a worst-cast optimal factorization of the polynomial.

$$m_1 : \forall x, y : Employee(x, y) \rightarrow Department(y)$$

$$m_2 : \forall x : Department(x), Manager(y, x) \rightarrow ManagedDepartment(x)$$

Employee

name	**dept**	$\underline{\mathbb{N}[X]}$
Peter	HR	x_1
Bob	HR	x_2

Department

dept	$\underline{\mathbb{N}[X]}$
HR	$x_1 + x_2$

Manager

name	**dept**	$\underline{\mathbb{N}[X]}$
Alice	HR	y_1
Astrid	HR	y_2

Managed Department

dept	$\underline{\mathbb{N}[X]}$
HR	$(x_1 + x_2) \cdot (y_1 + y_2)$

m_1's provenance

name	**dept**
Peter	HR
Bob	HR

m_2's provenance

name	**dept**
Alice	HR
Astrid	HR

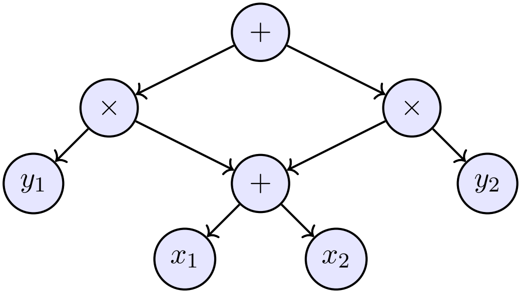


Figure 4.5: Example schema mapping and Orchestra provenance encoding.

Where-Provenance in DBNotes

As yet another example of a relational encoding we discuss how where-provenance is stored in the DBNotes system. Recall that where-provenance (Section 2.3.3) uses attribute-level granularity. The where provenance of the value of an attribute A of a tuple t is a set of input attribute values. Each such values is identified through a unique tuple identifier and the attribute's name. The schema of each relation R is extended with one additional attribute A_a for each attribute A of the relation. These attributes are used to store annotations. If the where-provenance of one or more attribute values of a tuple contains more than one input attribute value, then the tuple is replicated n times where n is the maximum number of elements in the where-provenance of any attribute of the tuple.

Example 60 (DBNotes where-provenance storage scheme)**.** Consider the where provenance of evaluating the query student ∪ teacher over the database shown below. We show the set of input attribute values that contribute to a result attribute value as a superscript for this value. The relational encoding of where-provenance encodes result tuple ($Peter$) using two tuples to store the two attribute values in the where-provenance of the name attribute of this tuple.

student

name	id
Peter	s_1
Bob	s_2

teacher

name	id
Peter	t_1
Alice	t_2

Query result with where-provenance

name
Peter$^{\{t_1.name, t_1.name\}}$
Bob$^{\{s_2.name\}}$
Alice$^{\{t_2.name\}}$

Relational encoding of the query's where-provenance

name	**name**$_a$
Peter	$s_1.name$
Peter	$t_1.name$
Bob	$s_2.name$
Alice	$t_2.name$

4.1.6 Encoding Provenance using User-Defined Types

An alternative to using the native data model of the system for which we are capturing provenance is to use the extensibility mechanisms of such a system to define new data types for representing provenance information. Of course this is only possible if the system has an extensible type system. For example, most database systems support user-defined datatypes (UDTs) and user-defined functions (UDFs). For instance, the ProvSQL system (Senellart *et al.*, 2018) uses UDTs to store provenance circuits for monus-semiring provenance, the extension of provenance polynomials with support for set difference we discussed in Section 2.4.1. As we will see in Section 4.3, encoding provenance using a UDT provides us with more flexibility for how to store provenance, but can lead to large tuple sizes when the provenance of a tuple is encoded as a single attribute value. However, most DBMS are optimized for large quantities of moderately sized tuples. In Senellart *et al.* (2018) this problem is circumvented by storing nodes of a circuit as separate values instead of encoding the whole circuit as a single value. We will continue this discussion in Section 4.3.

4.1.7 Native Provenance Storage

Yet another option is to extend the storage and execution engine of a database system to support provenance. This option is the most flexible, but is also the most invasive. This option gives us the flexibility to piggyback provenance on existing data structures that are created during query evaluation. For example, the Smoke system (Psallidas and Wu, 2018a) utilizes data structures such as hash tables created by implementations of join and aggregation operators to store provenance with less overhead.

4.1.8 Standardized Representations of Provenance

The storage formats for provenance we have discussed so far are mostly specific to a provenance model and the main focus is on how to compactly represent provenance to enable efficient capture and querying. In this article, we have been using the term provenance model in a narrow sense to denote a syntax that is used to encode provenance information as well as a semantics that defines what the provenance of a transformation, e.g., a query, is. We have discussed such provenance models in depth in Chapter 2. A separate body of work has focused on creating standardized representations of provenance information that are generic enough to apply to a wide range of provenance use cases. To make clear the distinction between the models discussed in Chapter 2 and such models, we will refer to the later as *standardized representations*. Because of their generic nature, such standardized representations can serve as a storage format for exchanging provenance information among provenance-enabled systems. Two such standardized representations are the PROV model (Moreau and Groth, 2013) and the open provenance model (OPM) discussed in Moreau *et al.* (2011). We briefly discuss the PROV model here.

PROV is a graph-based model. A PROV graph consists of a number of different node types:

- *entities* are the objects whose provenance we want to track (represented as rectangles in PROV graphs)

- *activities* are actions that consume and produce entities (represented as rectangles with rounded edges in PROV graphs)

- *agents* control, trigger, and execute activities (represented as "house shapes" in PROV graphs)

Figure 4.6 shows the provenance of a query Q (an activity) executed by a person named Bob (an actor) at the granularity of relations (the entities). Note that we use ovals to denote agents. Edges in PROV graphs represent provenance relationships between the elements (of the graph). Important types of edges are:

- *wasDerivedFrom* edges connect entities to other entities they were derived from, i.e., such edges encode data dependencies

- *wasGenereatedBy* edges connect entities to the activities involved in their creation (in our terminology this a type transformation dependency)

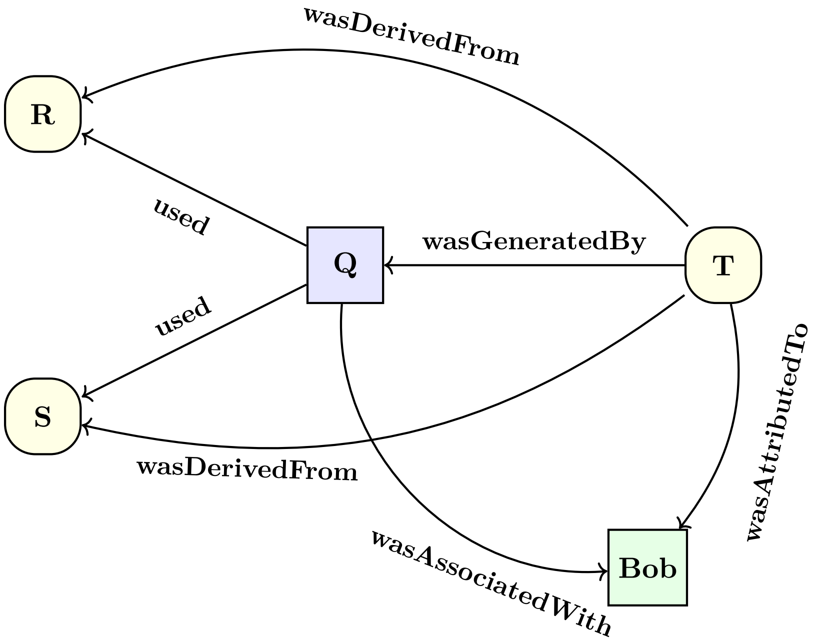


Figure 4.6: Example PROV graph recording the derivation of a query result relation (an entity) from multiple input relations (entities). The query (an activity) was executed by user Bob (an agent).

- *used* edges connect activities to the entities consumed by the activities
- *wasAttributeTo* edges connect entities to the agents that can be credited with their creation
- *wasAssociatedWith* edges connect activities to the agents that controlled, executed, or triggered them

In our example shown in Figure 4.6, the query activity Q *was attribute to* Bob who submitted the query. The query's result *was also attributed to* Bob. The query result *was generated by* by the query which *used* the two input relations R and S.

4.2 Provenance Summarization

So far we have focused on how effective storage strategies are in reducing the size of provenance information and what their cost is. However, size and performance of compression are not the only factors that should be considered. Other important factors are how a storage strategy affects the performance of querying, or more generally, analyzing provenance information and, if provenance will be consumed by a human, how the storage strategy affects understandability. In this section, we will discuss techniques for producing summaries of provenance for human consumption. These differ from the compression techniques discussed earlier in that (i) understandability is a major concern and that (ii) aggressive compression is required to produce summaries that are small enough to be consumed by a human. To achieve these goals, most techniques permit summaries to over- and/or under-approximate the provenance. That is, not all provenance may be represented by the summary (under-approximation) or some of the data represented by the summary may not be in the provenance (over-approximation).

4.2.1 Pattern- and Constraint-based Summaries

One common method for summarizing provenance is to use a declarative description of the data that belongs to a query's provenance. A common type of declarative summaries are selection patterns which are tuples over constants and variables paired with constraints on the values these variables can take. For instance, the pattern $(Peter, x_{age}); x_{age} > 30 \land x_{age} \leq 40$ "covers" all tuples for which the first attribute value is the constant "Peter" and the second attribute is an integer between 30 and 40. Many approaches for summarizing provenance have used a restricted form of such patterns for which variables are unconstrained (represent any value) and do not repeat. Such patterns can be compactly represented by using a "don't care" value $*$ that denotes variables (since variables are unique within a pattern and are not constrained, their identity is irrelevant). For instance, $(Peter, *)$ is a pattern that represents all persons named "Peter" no matter what their age is.

Such patterns have been used to identify subsets of an aggregation query's provenance that have a large effect on the result, e.g., find branches that contributed the most to a company's sales. These approaches use intervention to measure the impact the subset of the input data described by a pattern has on a query result: what is the change to the query result when deleting this subset from the database. Recall that we already discussed a provenance model based on intervention in Section 2.1.4. Wu and Madden (2013) presented Scorpion, an intervention-based system for explaining aggregate query results. Roy and Suciu (2014) presented a similar approach that also takes inclusion dependencies (foreign keys) into account to only consider interventions that respect the constraints of the input database's schema. For example, an intervention that removes a teacher has to also remove the students that are advised by the teacher (assuming that there is foreign key from students to teacher). Selection patterns have also been used in Lee *et al.* (2020) to summarize why-not provenance. This work uses sampling to scale why-not provenance to non-trivial database sizes. Declarative summaries of data have been used extensively for applications other than data provenance, e.g., to explain classifications outcomes (Vollmer *et al.*, 2019; El Gebaly *et al.*, 2018; El Gebaly *et al.*, 2014) or for interactive data exploration (Joglekar *et al.*, 2019). One challenge in all of these approaches is that the number of candidate patterns, while polynomial in the size of the input data, is exponential in the number of attributes. Approaches that explore the search space exhaustively are, thus, limited to relatively small schemas (in terms of the number of attributes in the schema). Some approaches like, e.g., Lee *et al.* (2020), Wu and Madden (2013), and El Gebaly *et al.* (2014), use heuristics to prune the search space by considering only a subset of all possible pattern candidates as well as using sampling to speed-up the evaluation of a pattern's impact. In addition to the impact that deleting a subset of the data corresponding to a pattern has on the query result, other measures may be used to measure the "quality" of a pattern-based explanation such as (i) the size of the explanation (if a set of pattern is returned); (ii) the amount of non-provenance data covered by the pattern (does the pattern mostly describe provenance information); (iii) and the informativeness of the pattern, i.e., how much new information is conveyed by the pattern, e.g., a pattern consisting

tax

name	city	state	tax-2019	tax-2020
Eton Fragrance	Los Angles	CA	13,232,343	14,532,343
Bill Smith	San Antonio	CA	9,123,123	12,123,123
William Entryway	Lake Washington	WA	8,122,233	6,122,233
Bob Pretzel	Chicago	IL	1,432,232	1,032,232
Pete Everyman	Chicago	IL	4,534	15,534
Alice Smith	Chicago	IL	14,543	22,543

```
SELECT sum(tax-2020 - tax-2019) AS taxincr
FROM tax
```

taxincr
1,919,000

Figure 4.7: Using pattern-based summaries to explain the outcome of an aggregate query based on intervention (finding influential subsets of the provenance).

only of placeholders does not convey any new information. Approaches differ in what metrics they optimize for and which hard constraints they enforce for explanations. For example, given an aggregate query, direction (higher or lower), and a database, the approach described in Roy and Suciu (2014) selects a pattern that has the largest impact on the result of the aggregation query.

Example 61 (Summarizing provenance with patterns). Consider the tax dataset shown in Figure 4.7. We are using the query shown in this figure to calculate the total increase in tax revenue from 2019 to 2020. A pattern based summarization technique can be used to explain which are the major contributors to the observed increase. For instance, we may employ the technique from Roy and Suciu, 2014 that returns a pattern such that removing the data covered by the pattern leads to the greatest decrease of the aggregation result.[3] In this example, some persons' tax has increased (Eton, Bill, Pete, and Alice) while the tax of the remaining persons decreased. It is easy to see that in this case the pattern that leads to the greatest decrease of the result is $\mathit{state} = CA$ (or $(*, *, CA, *, *)$ if we use the placeholder notation introduced above),

[3] This method also reasons about foreign key constraints, but since our example uses a single relation this is not relevent here.

since both Eton and Bill saw a large tax increase in 2020 while most other persons did see a decrease. While Pete and Alice also saw an increase in tax, their total contribution is not enough to offset other decreases that would be included for patterns that also cover these two persons.

4.2.2 Summarization with Ontologies

An alternative to using queries to describe provenance data declaratively, is to use a hierarchy of concepts encoded as an ontology to summarize data. For instance, we can replace many instances of a more specific concept with a general concept that subsumes them. For example, if all cities in California are in the provenance then we may represent this set of cities using the more general concept, e.g., "CaliforniaCity" or "USCity". To use an ontology to summarize (provenance) data, we need to relate the constants / tuples occurring in the database with concepts in the ontology. For instance, one way to achieve this is to require that all constants from the database (the database's active domain $\textsc{adom}(D)$) exist as base concepts (concepts that are not subsuming any other concepts) in the ontology:

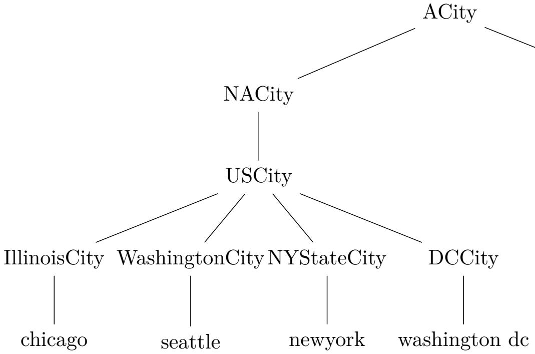
Definition 26 (Ontology)**.** Given a database D with active domain $\textsc{adom}(D)$, an ontology is a pair $(\mathcal{C}, \sqsubseteq)$ where

- $\mathcal{C} \subseteq \textsc{adom}(D)$ is a set of concepts
- $\sqsubseteq$, called the subsumption relation, is a reflexive and transitive binary relation over $\mathcal{C}$

We require that $\forall c \in \textsc{adom}(D) : \neg\exists c' \in \mathcal{C} : c' \sqsubset c$.

Summarization with ontologies and summarization with queries are more closely related to each other than one may expect. Cate *et al.* (2015) showed how to automatically derive an ontology from queries based on query containment. For instance, the query

$$Q_{\text{largeCAcities}} \coloneqq \Pi_{cityname}(\sigma_{state=CA \wedge population > 100{,}000}(\textsf{cities}))$$



Train

fromCity	**toCity**
new york	washington dc
new york	chicago
chicago	seattle
washington dc	seattle
berlin	paris
berlin	munich
paris	lyon
paris	dijon

2hop

X	**Y**
new york	seattle
chicago	washington dc
berlin	lyon
berlin	dijon
...	...

Figure 4.8: Example train connection database and city ontology from Glavic *et al.* (2015).

which computes cities in California with more than 100,000 inhabitants is contained in

$$Q_{\text{CAcitites}} := \Pi_{cityname}(\sigma_{state=CA}(\mathsf{cities}))$$

which returns all cities in California.

Thus, the concept corresponding to $Q_{\text{largeCAcitites}}$ is a subconcept of the concept corresponding to Q_{CAcities}. Cate *et al.* (2015) demonstrated how to use an ontology to generalize a missing answer, i.e., the user provides as input a non-answer to a query and the approach generalizes this missing answer to a *most general explanation* which is a tuple of concepts from an ontology that covers only missing answers, includes the missing answer provided by the user, and none of the concepts can be generalized further without covering also existing answers. Glavic *et al.* (2015) utilizes the concept of most general explanations to summarize why- and why-not provenance for unions of conjunctive queries using taxonomies and demonstrated how to use Datalog to compute such summaries. Data X-ray Wang *et al.* (2015) implements an approximate algorithm that find explanations based on a taxonomy in linear time.

Example 62 (Summarization with ontologies). Consider the ontology and database shown in Figure 4.8. The Datalog query shown below returns train connections with one intermediate stop.

$$\begin{aligned}\mathsf{2hop}(X,Y) &:- \mathsf{bidir}(X,Z), \mathsf{bidir}(Z,Y)\\ \mathsf{bidir}(X,Y) &:- \mathsf{Train}(X,Y)\\ \mathsf{bidir}(Y,X) &:- \mathsf{Train}(X,Y)\end{aligned}$$

A user may wonder why there are no train connections from Chicago to Berlin with one intermediate stop. The why-not provenance of this missing answer consists of all failed derivations of the first rule with X = chicago, Y = berlin, and Z bound to any city. However, there is not just no train connection from Chicago to Berlin, there are no train connections from any American city to any European city. Thus, instead of listing all of these derivations, we can use the ontology from Figure 4.8 to compactly explain this more general observation:

$$\begin{aligned}\mathsf{2hop}(\mathsf{NACity}, \mathsf{EuropeanCity}) :- \ &\mathsf{bidir}(\mathsf{NACity}, \mathsf{ACity}),\\ &\mathsf{bidir}(\mathsf{ACity}, \mathsf{EuropeanCity})\end{aligned}$$

4.3 Provenance Capture

To be able to benefit from provenance information, we have to devise techniques for **capturing** provenance. Methods for provenance capture differ in the types of transformations they support, in how they capture provenance, and in when provenance is captured (during the execution of a transformation or after the fact). We will first discuss what aspects of these methods are of interest to us and then discuss individual classes of methods in more detail.

- **When to capture provenance?** Methods that capture provenance proactively during the execution of a transformation have traditionally been called **eager** while methods that capture provenance retroactively are referred to as **lazy**. Eager methods have the disadvantage that we pay the overhead of provenance capture for all transformations upfront. The additional computational resources used to capture provenance may be wasted if the captured provenance information is never used. Lazy methods have the disadvantage that it may not always be possible to capture provenance retroactively, because we may lack relevant information that is needed for capture. Note that eager and lazy are only two extremes of a spectrum: hybrid methods capture some information during transaction execution and then use this information later on for retroactive provenance tracking. These methods have the advantage that they enable lazy capture for larger classes of transformations without having to pay the high upfront cost of provenance capture of many eager methods.

- **What to information is required for capture?** For methods that are lazy (or hybrid), we will discuss what information needs to be captured during transformation execution and what overhead this entails. For instance, as we will see in the following, provenance for queries and transactions can be captured retroactively using the time travel and audit logging facilities build into many DBMS.

- **Backward vs. forward tracing?** To compute provenance we can either start from the result of a transformation and then trace these results backwards to the inputs they depend on or we can start from the input data and track its impact on (intermediate) results of the transformation. We refer to the earlier as **backwards tracing** and the later as **forward tracing**. A prevalent type of forward tracing is **annotation propagation**. In annotation propagation we annotate the input data with provenance tokens that are propagated and combined during transformation execution to produce results that are annotated with provenance information.

- **External vs. internal capture?** To capture provenance we can either extend the system on which we are executing transformations with native provenance capture capabilities (we call this approach **internal** capture) or we limit our interactions with the system to the API provided by the system and compute provenance information outside of the system (we refer to this approach as **external** capture). Of course, it is possible for a method to do some work internally and some work externally. A specific type of external capture is *language-level instrumentation* where the provenance computation is expressed in the same language as the transformation for which we want to capture provenance. One example of such a technique is to use SQL to capture the provenance of SQL queries, e.g., see Glavic *et al.* (2013b), Cui *et al.* (2000), and Bhagwat *et al.* (2004).

4.3.1 Backward Tracing (Inversion)

Inversion computes the provenance for a data item in the result of a query Q over a database D by inverting the query. However, queries are typically not injective, i.e., the query may produce the same result for two input databases. Such queries are not invertible in the mathematical sense. This implies that it in general it is not possible to compute provenance using just the query result and query. Inversion methods, thus, either require access to the input data or other types of additional information to be able to capture provenance.

An early example of a provenance capture method for the Lineage model (introduced in Section 2.3.2) that employs inversion was described in Cui *et al.* (2000). This procedure takes as input an $\mathcal{RA}^{agg}$ query and divides this query into *"blocks"* where some operators (e.g., aggregation and difference) are block boundaries. Starting from a result tuple t, the approach then recursively traces the tuple back to their provenance in the database one block at a time. Note that this requires the result of the root operator of each block to be materialized (or to be recomputed on the fly when needed). Blocks are classified based on the types of operators that occur in the block. Cui *et al.* (2000) defines a separate tracing procedure for each block type which consists of running one or more queries that trace the input or intermediate provenance produced for a parent block back to the inputs of a block.

Tracing SPJ Queries

Let us consider first tracing an SPJ query, i.e., a query consisting only of selection, projection, and joins. Recall that the lineage of a query result is a list of subsets $\langle\, {R_1}^*, \ldots, {R_n}^* \,\rangle$ of the input relations R_i accessed by the query that fulfills the conditions of Definition 13 and Definition 14. Definition 13 defines the lineage of a single operator by requiring that when replacing the operator's inputs with the lineage of a result tuple t, we get the result set $\{t\}$. Furthermore, each tuple $t^* \in {R_i}^*$ together with ${R_j}^*$ for all $j \neq i$ is sufficient for producing the result. Finally, $\langle\, {R_1}^*, \ldots, {R_n}^* \,\rangle$ is maximal among all subsets fulfilling the other two conditions.

The tracing procedure for SPJ queries consists of a single query plus an operator $Split_{A_1,\ldots,A_m}(R)$ which returns a list of projections of R, one for each A_i:

$$Split_{A_1,\ldots,A_m}(R) = \langle\, \Pi_{A_1}(R), \ldots, \Pi_{A_m}(R) \,\rangle$$

Consider an SPJ query of the form shown below. Note that any SPJ query can be brought into this form by applying standard relational algebra equivalences.

$$\Pi_A(\sigma_\theta(R_1 \bowtie \ldots \bowtie R_m))$$

Given such an SPJ query, the lineage of one result tuple t consists of all tuples from an input relation R_i that contribute to a result of the join $(R_1 \bowtie \ldots \bowtie R_m)$ that fulfills the selection condition θ and is projected onto t by the projection of Q. This set of tuples can be computed by evaluating the join and selection and then filtering out tuples that are not projected on t using a condition $A = t$. Then the split operator is applied to generate the individual ${R_i}^*$ that constitute the lineage of the result.

sales

item	**shop**	**quantity**	<u>id</u>
Coffee	Chicago	2	s_1
Coffee	Schaumburg	3	s_2
Tea	Springfield	10	s_3
Tea	Chicago	1	s_4

items

item	**price**	<u>id</u>
Coffee	13	i_1
Tea	7	i_2

Query result

item
Coffee
Tea

Figure 4.9: Computing lineage using inversion.

Definition 27 (Lineage SPJ Tracing)**.** Let D be a database and t be a tuple in $Q(D)$ for a query Q:

$$\Pi_A(\sigma_\theta(R_1 \bowtie \ldots \bowtie R_m))$$

The tracing query $TQ_{t,Q}$ shown below computes the lineage of t wrt. Q and D.

$$TQ_{t,Q} := Split_{\textsc{Sch}(R_1),\ldots,\textsc{Sch}(R_m)}(\sigma_{\theta \wedge A=t}(R_1 \bowtie \ldots \bowtie R_m))$$

Before presenting an example of applying the tracing procedure, let us briefly reason about how it compares against the input query in terms of performance The input query and tracing query differ in three points: (i) the tracing query applies additional selection conditions; (ii) there is no projection in the tracing query; and (iii) the tracing query applies the split operator. The additional selection conditions may result in the tracing query being significantly smaller. However, if this condition is not very selective, then the tracing query may actually be slower than the input query, because of the larger result size (no projection on A) and the cost of the split operator. Nonetheless, it is not unreasonable to assume that in most cases the lineage of a single result tuple of an SPJ query will be small compared to the size of the input.

Example 63 (Computing Lineage According to Cui *et al.* (2000))**.** Figure 4.9 shows an example sales database. We demonstrate the inversion approach using the query shown below which returns items with sales (quantity times price) of more than \$20.

$$Q_{HighSaleItems} \coloneqq \Pi_{item}(\sigma_{quanitity \cdot price > 20}(sales \bowtie items))$$

The tracing query for the result tuple $t_{coffee} = (Coffee)$ according to Definition 27 is

$$\begin{aligned} &TQ_{t_{coffee}, Q_{HighSaleItems}} \coloneqq \\ &\qquad Split_{\textsc{Sch}(sales), \textsc{Sch}(items)}(\sigma_{quanitity \cdot price > 20}(sales \bowtie items)) \end{aligned}$$

Evaluating this query over the example database we get:

$$\langle \{(Coffee, Chicago, 2), (Coffee, Schaumburg, 3)\}, \{(Coffee, 13)\} \rangle$$

That is, coffee is in the result, because of the first two tuples from relation sales and the first tuple of relation items.

ASPJ Queries

The SPJ tracing procedure can be extended to support aggregation. Recall from the operational definition of lineage (Definition 15) that the lineage of an aggregation operator (with group-by) consists of all input tuples that have the same group-by value as the result tuple of interest. For now consider an aggregation query as shown below.

$$\gamma_{G;fA}(\Pi_A(\sigma_\theta(R_1 \bowtie \ldots \bowtie R_m)))$$

Thus, the input tuples of the aggregation that contribute to a result tuple t can be retrieved through a selection $\sigma_{G=t.G}$, i.e., all input tuples that have the same values in the group-by attributes G as the output t. The extended tracing query for a query with a single aggregation as the last operation is:

$$TQ_{t,Q} \coloneqq Split_{\textsc{Sch}(R_1), \ldots, \textsc{Sch}(R_m)}(\sigma_{\theta \wedge G=t.G}(R_1 \bowtie \ldots \bowtie R_n))$$

Multi-block Queries

As mentioned above, more complex queries are split into blocks such that each block contains at most one aggregation operator. The reason for introducing block-wise tracing is that a single tracing query is only sufficient to invert a single such block, but tracing across blocks requires access to intermediate results. For instance, if there is a query with two levels of aggregation, e.g., counting the number of departments (outer aggregation) that have more than 10 employees (inner aggregation with group-by), then the first tracing query can only determine which output tuples of the inner aggregation contribute to the final query result. A second tracing query has to be used to then capture the lineage of these tuples back to the input database. We refer the interested reader to Cui *et al.* (2000) for a detailed description of the algorithm for splitting a query into multiple blocks and for tracing procedures for set operations.

Discussion

The backward tracing procedure according to Cui *et al.* (2000) we have introduced in this section does only require access to the input database and query to compute provenance. However, it is more efficient if intermediate results of subqueries corresponding to tracing blocks are stored as materialized views. Cui and Widom (2000b) discussed heuristics for choosing which views to create for this purpose. As we will discuss in the following, some forward tracing procedures can capture the provenance of a query using a single query instead of tracing it one block at a time. Furthermore, they support more informative provenance models such as provenance semirings. Another optimization proposed in Cui and Widom (2000b) is that if the query result contains keys of all or some of the input relations, then the values of these tuples for a query result of interest can be used to directly fetch tuples that belong to the lineage from the input relations using a selection on the key.

4.3.2 Annotation Propagation (Forward Tracing)

An alternative to inversion is to annotate tuples from the input database with their provenance and then propagate and combine these annotations during query processing to produce results annotated with provenance. Annotation propagation is particularly well-suited for provenance models that are defined using annotation propagation such as the semiring model and its extensions. However, any type of provenance can be computed using this approach as long as the annotated result produced for a subquery contains sufficient information to calculate provenance annotations for down-stream operators (ancestors). Fortunately, this is the case for the provenance models we have discussed in Chapter 2.

Computing Semiring Annotations

As one example for an annotation propagation approach, we discuss an approach for calculating semiring provenance using the examples of provenance polynomials and lineage. In Section 4.3.3 we will discuss possible ways of implementing annotation propagation.

Example 64 (Annotation Propagation)**.** Recall from Section 2.3.4 and Section 2.3.4 the semiring of provenance polynomials and the one used for lineage:[4]

- $\mathsf{Which}[X] = (2^X \cup \{\perp\}, \cup_+, \cup_\times, \perp, \emptyset)$ **(lineage)**
- $\mathbb{N}[X] = (\mathbb{N}[X], +\cdot, 0, 1)$ **(provenance polynomials)**

We now revisit computing provenance for query $Q_{HighSaleItems}$ from Example 63 bottom-up with forward tracing (annotation propagation) instead of inversion. Recall that the example database used for this is shown in Figure 4.9. We show subqueries of this query below. Figure 4.10 shows the intermediate and final results of the query using annotation propagation (which for the case of semirings is the same as $\mathcal{K}$-relational query semantics as discussed in Section 2.3.4). In the input relations every tuple is annotated with unique variable (e.g., using tuple identifiers). Subquery Q_2 is a natural join of the two input relations. In $\mathcal{K}$-relational semantics, each result tuple of a join is annotated with the product of the annotations of the two input tuples it is derived from. In semiring $\mathsf{Which}[X]$ multiplication is defined as set union. Query Q_1 then

[4] As discussed in Chapter 2 the semiring version of lineage differs from lineage by not tracking the order in which relations are accessed by a query.

applies a selection to the result of Q_2 which filters out the last tuple (annotated with $s_4 \cdot i_2$). Finally, query $Q_{HighSaleItems}$ projects the result of Q_1 on attribute `item`. In $\mathcal{K}$-relational algebra, an output tuple of a projection operator is annotated with the sum of the annotations of all input tuples projected onto this output. For example, tuple $(Coffee)$ is derived from the first two tuples in the result of query Q_2 and, thus, is annotated with $s_1 \cdot i_1 + s_2 \cdot i_1$. Addition in semiring $\mathsf{Which}[X]$ is set union. Thus, modulo the split of lineage into subsets of the individual input relations of the query, forward tracing with semirings produces the same result as inversion.

$$\begin{aligned} Q_{HighSaleItems} &\coloneqq \Pi_{item}(\sigma_{quanitity \cdot price > 20}(sales \bowtie items)) \\ Q_1 &\coloneqq \sigma_{quanitity \cdot price > 20}(sales \bowtie items) \\ Q_2 &\coloneqq sales \bowtie items \end{aligned}$$

Note that the slight difference in semantics between forward tracing (annotation propagation) and backward tracing (inversion) is not an inherent property of these two fundamental approaches, but due to our choice to semiring used in forward tracing.

4.3.3 Extending Systems for Provenance Capture

We now discuss how database systems can be extended to support provenance capture. While our focus is on relational databases, the methods we will discuss are of broader interest, e.g., for implementing provenance capture on big data platforms such as Spark (Zaharia *et al.*, 2010) or MapReduce (Dean and Ghemawat, 2004).

Instrumenting Queries for Capture

One common method for implementing provenance capture by annotation propagation is to use an encoding of annotated relations in the host data model (as discussed in Section 4.1.5) or using user-defined types (as discussed in Section 4.1.6) and implement annotation propagation over this encoding by **instrumenting** the input query for which we want to capture provenance. Here by instrumenting we mean rewriting the query such that it returns this encoding of query results annotated

sales

item	**shop**	**quantity**	$\mathsf{Which}[X]$	$\mathbb{N}[X]$
Coffee	Chicago	2	$\{s_1\}$	s_1
Coffee	Schaumburg	3	$\{s_2\}$	s_2
Tea	Springfield	10	$\{s_3\}$	s_3
Tea	Chicago	1	$\{s_4\}$	s_4

items

item	**price**	$\mathsf{Which}[X]$	$\mathbb{N}[X]$
Coffee	13	$\{i_1\}$	i_1
Tea	7	$\{i_2\}$	i_2

Annotated result of Q_1

item	**shop**	**quantity**	**price**	$\mathsf{Which}[X]$	$\mathbb{N}[X]$
Coffee	Chicago	2	13	$\{s_1, i_1\}$	$s_1 \cdot i_1$
Coffee	Schaumburg	3	13	$\{s_2, i_1\}$	$s_2 \cdot i_1$
Tea	Springfield	10	7	$\{s_3, i_2\}$	$s_3 \cdot i_2$
Tea	Chicago	1	7	$\{s_4, i_2\}$	$s_4 \cdot i_2$

Annotated result of Q_2

item	**shop**	**quantity**	**price**	$\mathsf{Which}[X]$	$\mathbb{N}[X]$
Coffee	Chicago	2	13	$\{s_1, i_1\}$	$s_1 \cdot i_1$
Coffee	Schaumburg	3	13	$\{s_2, i_1\}$	$s_2 \cdot i_1$
Tea	Springfield	10	7	$\{s_3, i_2\}$	$s_3 \cdot i_2$

Annotated result of $Q_{HighSaleItems}$

item	$\mathsf{Which}[X]$	$\mathbb{N}[X]$
Coffee	$\{s_1, s_2, i_1\}$	$s_1 \cdot i_1 + s_2 \cdot i_1$
Tea	$\{s_3, i_2\}$	$s_3 \cdot i_2$

Figure 4.10: Computing lineage and provenance polynomials bottom-up with annotation propagation.

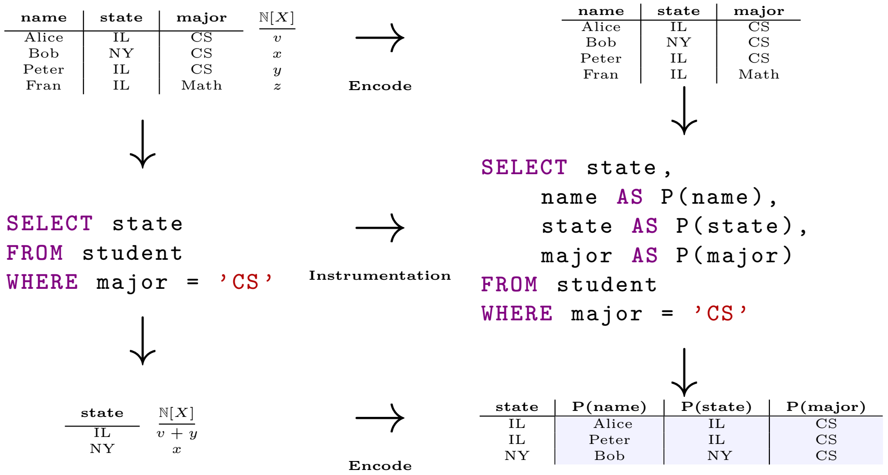
with provenance. Many of the provenance management systems we will discuss in Section 4.5 such as Perm, GProM, DBNotes, Orchestra, ProvSQL and many others implement this approach.

As one example of instrumentation we now discuss the instrumentation approach implemented in Perm (Glavic *et al.*, 2013b; Glavic and Alonso, 2009a)[5] to compute the relational encoding of provenance polynomials discussed in Section 4.1.5. Figure 4.11 shows an overview of how this approach works. On the left-hand side of this figure we show the evaluation of a query that returns the home states of students that study Computer Science. For convenience, we write the query in SQL, but note that the instrumentation rules developed in Glavic and Alonso (2009a) are defined for relational algebra queries. For the example query, `IL` in the result depends on the the input tuples annotated with v and y and, thus, is annotated with $v + y$. Similarly, tuple `NY` depends on x. To calculate the relational encoding of provenance as introduced before, the input query is instrumented to produce this encoding. This is achieved in two steps: first the attributes of input relation `student` are duplicated to produce the relational encoding of the input $\mathbb{N}[X]$-relation. A renaming function $P()$ is applied to create unique names for the duplicated attributes storing provenance information. An additional purpose of the naming scheme applied by function P is to identify which attributes store the provenance of which input table. For sake of the exposition we have simplified this function to only take an attribute name as an input. The definition from Glavic *et al.* (2013b) additionally takes as input the query and name of the relation the attribute belongs to. This additional information is used to ensure that provenance attribute names are unique and to identify the table access operator of the query whose provenance they are recording. For instance, for a query $R \bowtie R$ over a relation $R(A, B)$, the provenance attribute names are:

```
prov_R_a, prov_R_b, prov_R_1_a, prov_R_1_b
```

The initial values of the annotation attributes are then propagated by instrumenting each operator of the query. As shown in Figure 4.11 the provenance polynomial $v + y$ for result tuple `IL` is encoded as the

[5] The GProM (Arab *et al.*, 2018a) system also implements this method.



name	state	major	$\mathbb{N}[X]$
Alice	IL	CS	v
Bob	NY	CS	x
Peter	IL	CS	y
Fran	IL	Math	z

name	state	major
Alice	IL	CS
Bob	NY	CS
Peter	IL	CS
Fran	IL	Math

```
SELECT state
FROM student
WHERE major = 'CS'
```

```
SELECT state,
    name AS P(name),
    state AS P(state),
    major AS P(major)
FROM student
WHERE major = 'CS'
```

state	$\mathbb{N}[X]$
IL	$v + y$
NY	x

state	P(name)	P(state)	P(major)
IL	Alice	IL	CS
IL	Peter	IL	CS
NY	Bob	NY	CS

Figure 4.11: Capturing a relational encoding of provenance polynomials with instrumentation (Arab *et al.*, 2018a).

first two tuples in the result of the instrumented query. The first encodes variable v as the input tuple annotated with v while the second encodes variable y as the input tuple annotated with y. Glavic and Alonso (2009a) demonstrated that this type of instrumentation is correct, i.e., the instrumented query produces the relational encoding of provenance polynomials for the result of the query evaluating under $\mathbb{N}[X]$-relational (provenance polynomial) semantics.

The instrumentation framework of Glavic and Alonso (2009a) implements instrumentation as a rewrite operator $\cdot^{+}$. This operator takes as input the query Q for which provenance should be captured and returns a query Q^{+} that captures the provenance of Q. The operator $\cdot^{+}$ is defined as a recursive set of rewrite rules — one for each type of algebra operator supported by the approach. During the rewriting we maintain a list of provenance attributes $\mathcal{P}Q$ for the instrumented result of a subquery Q. Figure 4.12 shows the rewrite rules implementing $\cdot^{+}$ (top) and the rules that determine $\mathcal{P}Q$ (bottom). Recall that the relational encoding of provenance polynomials used by this approach encodes a tuple t annotated with a polynomial p as a set of tuples each of which encodes one monomial of the polynomial in sum of products form. The provenance attributes of a query consist of the attributes of

Structural Rewrite

$$
\begin{aligned}
&Q = R: && Q^{+} = \Pi_{\mathbf{R},\mathbf{R}\rightarrow P(\mathbf{R}))}(R) && \textbf{(R1)}\\
&Q = \sigma_{\theta}(Q_1): && Q^{+} = \sigma_{\theta}({Q_1}^{+}) && \textbf{(R2)}\\
&Q = \Pi_{A}(Q_1): && Q^{+} = \Pi_{A,\mathcal{P}(Q^{+})}({Q_1}^{+}) && \textbf{(R3)}\\
&Q = \gamma_{G;f(a)\rightarrow x}(Q_1): && Q^{+} = \Pi_{G,x,\mathcal{P}(Q^{+})}(\gamma_{G;f(a)\rightarrow x}(Q_1) && \textbf{(R4)}\\
& && \qquad ⟕_{G=_{\epsilon}X}\ \Pi_{G\rightarrow X,\mathcal{P}({Q_1}^{+})}({Q_1}^{+})) && \\
&Q = \delta(Q_1): && Q^{+} = Q_1^{+} && \textbf{(R5)}
\end{aligned}
$$

$$
\begin{aligned}
&Q = Q_1 \bowtie_{\theta} Q_2: && Q^{+} = \Pi_{\mathbf{Q_1},\mathbf{Q_2},\mathcal{P}(Q^{+})}({Q_1}^{+} \bowtie_{\theta} {Q_2}^{+}) && \textbf{(R6)}\\
&Q = Q_1 ⟕_{\theta} Q_2: && Q^{+} = \Pi_{\mathbf{Q_1},\mathbf{Q_2},\mathcal{P}(Q^{+})}({Q_1}^{+} ⟕_{\theta} {Q_2}^{+}) && \textbf{(R7)}
\end{aligned}
$$

$$
\begin{aligned}
&Q = Q_1 \cup Q_2: && Q^{+} = ({Q_1}^{+} \times \mathit{null}(\mathcal{P}({Q_2}^{+}))) && \textbf{(R8)}\\
& && \qquad \cup (\Pi_{\mathbf{Q_1},\mathcal{P}(Q^{+})}({Q_2}^{+} \times \mathit{null}(\mathcal{P}({Q_1}^{+})))) && \\
&Q = Q_1 \cap Q_2: && Q^{+} = \Pi_{\mathbf{Q_1},\mathcal{P}(Q^{+})}(\delta(Q_1 \cap Q_2) && \textbf{(R9)}\\
& && \qquad \ltimes_{\mathbf{Q_1}=_{\epsilon}X}\ \Pi_{\mathbf{Q_1}\rightarrow X,\mathcal{P}({Q_1}^{+})}({Q_1}^{+}) && \\
& && \qquad \ltimes_{\mathbf{Q_1}=_{\epsilon}Y}\ \Pi_{\mathbf{Q_2}\rightarrow Y,\mathcal{P}({Q_2}^{+})}({Q_2}^{+})) && \\
&Q = Q_1 - Q_2: && Q^{+} = \Pi_{\mathbf{Q_1},\mathcal{P}(Q^{+})}(\delta(Q_1 - Q_2) && \textbf{(R10)}\\
& && \qquad \ltimes_{\mathbf{Q_1}=_{\epsilon}X}\ \Pi_{\mathbf{Q_1}\rightarrow X,\mathcal{P}({Q_1}^{+})}({Q_1}^{+}) && \\
& && \qquad \times\ \mathit{null}(\mathcal{P}({Q_2}^{+}))) &&
\end{aligned}
$$

Provenance Attribute List

$$
\mathcal{P}(Q) = \begin{cases} \mathcal{P}(Q_1) & \textbf{if } Q = \sigma_{\theta}(Q_1) \mid \Pi_{A}(Q_1) \mid \gamma_{G;f(a)\rightarrow x}(Q_1) \mid \delta(Q_1)\\ P(R) & \textbf{if } Q = R\\ \mathcal{P}(Q_1) :: \mathcal{P}(Q_2) & \textbf{otherwise} \end{cases}
$$

Figure 4.12: Algebraic rewrite rules for instrumenting a query to capture *PI-CS* provenance which corresponds to provenance polynomials for positive queries.

all leaf nodes (table access operators) of the algebra tree of the query. Thus, $\mathcal{P}(Q)$ for a subquery Q consists of the list of attributes from the leaf nodes occurring in Q. For unary operators these are all provenance attributes for the input of the operator (first case in Figure 4.12). For binary operators (e.g., join or union) this is the concatenation (denoted as ::) of the list of provenance attributes for the left input (Q_1) and the right input (Q_2) of the query (last case in Figure 4.12). For table access operators, $P(R)$ denotes the provenance attributes for this relation.

We now discuss some of the rules presented in Figure 4.12. For each table access operator, a projection is used in the instrumented query (**R1**) to duplicate the attributes of the relation R (here $\mathbf{R}$ denotes the schema of relation R) and rename them as provenance attributes. For a selection, the provenance polynomial of each result tuple is the same as the corresponding input tuple (**R2**). The provenance polynomial for a result tuple t of projection (**R3**) consists of the sum of annotations for all input tuples u of the projection that are projected on t. Since addition is expressed as union in the encoding, the rewrite rule just projects the result of the instrumented input on the projection expressions A and the provenance attributes of the input. In the result, the tuple t and its annotation are encoded as the set of tuples containing the provenance encoding of all input tuples u projected onto t. Each result tuple of a join (**R6**) is annotated with the product of the annotations of the two tuple that were joined. In the relational encoding, this is achieved by concatenating the provenance attributes of the left and the right input. An output tuple in the result of a union operator is annotated with the sum of the annotations of this tuple in the left and right input (**R8**). Thus, in the instrumented version we can just union the two inputs. However, the provenance attribute lists from the two inputs will not be union compatible, because the subtrees rooted at the two inputs of the union operator contain disjoint sets of leaf nodes. To make the inputs union compatible, the provenance attributes of both inputs are padded with null values such that both input have the provenance attributes of both inputs (recall that in the relational encoding if all provenance attributes for an input relation are null this is interpreted as a 1, the neutral element of multiplication of the semiring). Let us consider aggregation, as an example of an operator for which the PI-CS provenance produced by this instrumentation differs from the extension of semirings for this operator. The PI-CS provenance of a result tuple t of a group-by aggregation operator consists of the provenance of all input tuples that belong to the group based on which t was generated. The instrumentation rule for aggregation (**R4**) joins the result of aggregation in the original query with the instrumented input of the aggregation on the group-by attributes.

Example 65 (Computing provenance polynomials using instrumentation according to Glavic and Alonso, 2009a)**.** Let us apply the instrumentation techniques to the query $Q_{HighSaleItems}$ and its subqueries from Example 63. The results of evaluating these queries under $\mathbb{N}[X]$ semantics is shown in Figure 4.10:

$$\begin{aligned}
Q_{HighSaleItems} &\coloneqq \Pi_{item}(\sigma_{quanitity \cdot price > 20}(sales \bowtie items)) \\
Q_1 &\coloneqq \sigma_{quanitity \cdot price > 20}(sales \bowtie items) \\
Q_2 &\coloneqq sales \bowtie items
\end{aligned}$$

For ease of presentation we discuss the instrumentation of this query bottom-up. The final result of instrumentation is shown below (we abbreviate $Q_{HighSaleItems}$ as Q_{HSI} and abbreviate some of the attribute names). First **R1** is applied to duplicate the attributes of relations `sales` and `items`. Afterwards, the natural join (Q_2) is instrumented by joining $sales^+$ with $items^+$ and adding a projection to reorder the attributes (non-provenance attributes before provenance attributes). The selection following the join is instrumented by applying the selection to its rewritten input. Finally, the projection of $Q_{HighSaleItems}$ is instrumented by appending the provenance attributes of Q_1 to the projection output. Figure 4.13 shows the result produced by the instrumented query. For convenience, we show which attributes are query result attributes and which attributes are provenance attributes (and which input relation they encode). Furthermore, we show the monomial corresponding to each tuple on the right.

Query result	sales			items		
item	**P(item)**	**P(shop)**	**P(quantity)**	**P(item)**	**P(price)**	$\mathbb{N}[X]$
Coffee	Coffee	Chicago	2	Coffee	13	$s_1 \cdot i_1$
Coffee	Coffee	Schaumburg	3	Coffee	13	$s_2 \cdot i_1$
Tea	Tea	Springfield	10	Tea	7	$s_4 \cdot i_2$

Figure 4.13: Query result for produced by query $Q_{HighSaleItems}$ instrumented to capture a relational encoding of provenance polynomials.

$$
\begin{aligned}
sales^{+} &= \Pi_{item,shop,quantity,item \rightarrow P(i),shop \rightarrow P(s),quantity \rightarrow P(q)}(sales)\\
items^{+} &= \Pi_{item,price,item \rightarrow P(i),price \rightarrow P(p)}(items)\\
Q_2{}^{+} &= \Pi_{item,shop,quantity,price,P(i),P(s),P(q),P(i),P(p)}(sales^{+} \bowtie items^{+})\\
Q_1{}^{+} &= \sigma_{quanitity \cdot price > 20}(Q_2{}^{+})\\
Q_{HSI}{}^{+} &= \Pi_{item,P(i),P(s),P(q),P(i),P(p)}(\sigma_{quanitity \cdot price > 20}(sales \bowtie items))
\end{aligned}
$$

Piggy-backing Capture on Query Processing

An alternative to instrumentation is to extend the execution engine of a database system to propagate provenance during query execution. While this approach is more invasive than instrumentation, there are several potential benefits: (i) we have more flexibility in designing the data structures used to store provenance, e.g., provenance data structures can be shared across multiple tuples or across operators in a query plan; and (ii) data structures that are generated during query processing, e.g., the hash table used by a hash join operator, can be reused for provenance capture. For instance, the Ariadne system implements this approach to capture provenance in a data streaming system (Glavic *et al.*, 2013a). Another example of a provenance system extending a database execution engine is TripleProv (Wylot *et al.*, 2014). Psallidas and Wu (2018a) presented Smoke which implements sharing of query execution data structures for provenance capture and query execution in an in-memory, compilation-based query engine. For example, the hash tables used for hash join and hash-based group-by aggregation are extended to allow the mappings between input and output tuple identifiers encoding the provenance of an operator to be materialized efficiently.

Control Dependencies Logging and Replay

Yet another approach is to only capture "control flow" information at query runtime and then capture data dependencies during a second execution of the query which operates on provenance encoded as sets of identifiers instead of over the input data of the query. This approach was first proposed in Müller and Grust (2015) where it was realized by translating SQL queries into Python programs and then using data- and control-flow analysis techniques from the programming languages community for realizing the two-step capture process. Müller *et al.* (2018) ported this approach for using instrumentation in SQL to capture provenance. To capture an input query's provenance, the query is instrumented to write control-dependencies (e.g., which tuples pass filter conditions and ordering of tuples in windows) to a log table (in the implementation this log table is stored as a relation). Then in a second step the query is instrumented again, but this time to compute provenance. This version of the query just operates on sets of tuple identifiers encoding provenance instead of data. This phase replays control-flow decision based on the log of control flow decision produced in the first phase since such decisions are typically based on data values. For example, which input tuples pass a selection is determined based on whether the selection condition evaluates to true or not. Importantly, this approach supports a very large subclass of SQL including recursion and window functions. It was shown to significantly outperform Perm (Glavic *et al.*, 2013b) for several TPC-H queries with nested subqueries while being slower for some queries without these features. However, this may in part be due to the fact that a less informative provenance model was used and that tuple identifiers instead of full tuples are captured.

Capturing Provenance for Updates and Queries Retroactively

To capture the provenance of a query Q, instrumentation evaluates an instrumented query over the same input database to produce data annotated with provenance. However, this requires that both Q and the instrumented query are executed over the same database state. Thus, this approach can not be applied if Q was executed at some point in past and the database has been updated since then. However, for database systems that support time travel (Jensen and Snodgrass, 1999), i.e., enable access to past version of the database, the provenance of Q can be captured by going back in time to the database version that was valid when Q was executed and run the instrumented query over this database version. For example, this idea was described in Zhang and Jagadish (2010). Of course, for this to be feasible, we need to know precisely at what time the query was executed. One way to achieve this is to rely on audit logging (Becker and Chambers, 1988) which keeps a record of executed SQL statements. Most major DBMS support audit logging.

Similar challenges are faced when capturing the provenance of update operations. Since updates modify the database, after running an update, time travel is required to capture the provenance of the update retroactively by accessing the state of the database before the update. Capturing provenance information for updates (and transactions) entails two additional challenges: (i) to execute an instrumented update we would have to create a copy of the temporal snapshot valid before the update to evaluate this update to capture provenance and (ii) capturing provenance retroactively for updates belong to transactions that were executed under weaker isolation levels (Berenson *et al.*, 1995) requires us to understand the interaction of transactions under such lower isolation levels to know which versions of a data item was seen or modified by which statement of a transaction. Gawlick and Radhakrishnan, 2011 were the first to realize that audit logging paired with temporal databases are sufficient for capturing provenance information. Arab *et al.* (2018b) and Arab *et al.* (2016) introduced the first approach for capturing provenance for transactions and updates using the provenance model discussed in Section 2.7. This work introduced **reenactment**,

	Account			Provenance for the First Update			Provenance for the Second Update			u_1	u_2
	cust	**typ**	**bal**	**P(cust,u_1)**	**P(typ,u_1)**	**P(bal,u_1)**	**P(cust,u_2)**	**P(typ,u_2)**	**P(bal,u_2)**	$\mathcal{U}_1$	$\mathcal{U}_2$
$C^2_{T_5,14}(U^2_{T_5,11}(C^2_{T_1,4}(c_2)))$	Alice	Savings	1100	Alice	Savings	1000	Alice	Savings	1100	T	F
$C^3_{T_5,14}(U^3_{T_5,13}(U^3_{T_5,11}(C^3_{T_2,3}(c_3))))$	Peter	Savings	5390	Peter	Savings	4990	Peter	Savings	5090	T	T

Figure 4.14: Relational encoding of the provenance and intermediate results for relation `Account` with respect to Transaction T_5.

a method that uses queries, time-travel, and audit logging to capture provenance of updates using queries. Importantly, this method does not entail any changes to the database and does not require past states of the database to be replicated for capture. Reenactment is based on the observation that it is possible to simulate the operations of updates and even transactional histories under lower isolation levels using queries. Such queries are called reenactment queries. With reenactment, provenance of past queries and transactions is captured retroactively incurring only a small overhead on transaction execution and at the same time reducing provenance capture costs compared to an approach that directly instruments updates. Reenactment uses a relational encoding of provenance similar to the one for provenance polynomials discussed in Section 4.1.5.

Example 66 (Relational Encoding of Transaction Provenance)**.** Recall Example 51 which illustrated how a transaction anomaly (a so-called write-skew which can occur under snapshot isolation can be detected using the MV-semiring model from Arab *et al.* (2016). Figure 4.14 shows the relational encoding of the provenance (annotation) of the state of the `Account` relation after the execution of the example transaction T_5. The MV-semiring annotation encoded by a tuple are shown on the left. In addition to the attributes of the `Account` relation, additional provenance attributes are used to store the previous tuple versions before the second and before the first update of the transaction the tuple was derived from. Furthermore, additional attributes record transformation dependencies, i.e., which of the two updates of the transaction did affect the current tuple version (after transaction T_6's commit).

This relational encoding is produced through reenactment. The details of constructing a reenactment query are beyond the scope of this paper. However, we will show an example for the critical step of translating updates (and transactions) into queries.

Example 67. Assume a user is interested in evaluating the effect of an update over the current version of a bag semantics relation `Emp(name,salary)` which increases the salary of all employees by \$500 if their current salary is less than \$1000. For simplicity, assume that the user is not interested in provenance (we use semiring $\mathbb{N}$ instead of $\mathbb{N}[X]^{\nu}$). This request is expressed using an `REENACT` statement in GProM (Arab *et al.*, 2018a), the system implementing reenactment:

```
REENACT(
  UPDATE Emp SET salary = salary + 500
  WHERE salary < 1000;
);
```

To reenact this update over the current version of relation `Emp`, we can construct a reenactment query which returns the new state of `Emp` produced by the update. This state is computed as a union between the set of tuples that would not be updated (do not fulfill the update's condition) and the updated versions of tuples that fulfill the update's condition (we have to increase their salary by `500`):

```
SELECT * FROM Emp
WHERE NOT(salary < 1000)
UNION ALL
SELECT name, salary + 500 AS salary, b FROM Emp
WHERE salary < 1000;
```

To reenact a sequence of updates, a common table expression (CTE) is generated for the result of each update and used as input to the next update in the sequence. For example,

```
REENACT(
  UPDATE Emp SET salary = salary + 500
  WHERE salary < 1000;
  UPDATE Emp SET salary = 0
  WHERE salary < 500;
);
```

can be reenacted using the query shown below.

```
WITH up1 AS (
   SELECT * FROM Emp WHERE NOT(salary < 1000)
   UNION ALL
   SELECT name, salary + 500 AS salary, b
   FROM Emp WHERE salary < 1000)
SELECT * FROM up1 WHERE NOT(salary < 500)
UNION ALL
SELECT name, 0 AS salary, b FROM up1
WHERE salary < 500
```

4.3.4 Optimizing Provenance Capture

Given the potential size of provenance information, capturing provenance can result in quite significant overhead for query processing. There are several ways of how the runtime and storage requirements of provenance capture can be improved: (i) provenance capture can utilize existing data structures that are generated during query evaluation (we already discussed this approach in Section 4.3.3; (ii) the capture process can be optimized to directly produce compressed provenance reducing the size and potentially runtime of capture; (iii) specialized query optimization techniques can be used that are fine-tuned for the characteristics of provenance capture queries.

Generation of compressed provenance during capture has been studied intensively. Bao *et al.* (2012) investigated how to compress data dependencies by breaking a query into multiple parts and materializing data dependencies for each part. This allows common subexpressions to be shared, and, thus, reduces provenance size. For instance, two result tuples t_1 and t_2 may both depend on an intermediate result s_1 that in turn depends on input tuples $i_1, \ldots, i_{100}$. We can compress this provenance by splitting it into the dependencies of t_1 and t_2 on s_1 and the dependencies of s_1 on i_j for $j \in \{1, \ldots, 100\}$. This is closely related to how factorization and arithmetic circuits are used to compress provenance polynomials, e.g., in Olteanu and Závodný (2015). Amsterdamer *et al.* (2011b) studied the problem of rewriting a query Q into a query Q' that is equivalent to Q under set semantics and produces provenance (using the provenance polynomial model) of minimal size (defined as having less and or "smaller" monomials). The SubZero system (Wu *et al.*, 2012) optimizes queries over provenance by deciding when to re-execute parts of a computation and when to use materialized provenance information. Furthermore, the system compresses provenance by pairing up sets of input and output data items for which all elements are data dependent on each other, e.g., a set of data dependencies $\{(o_1, i_1), (o_1, i_2), (o_1, i_3), (o_2, i_1), (o_2, i_2), (o_2, i_3), (o_3, i_1), (o_3, i_2), (o_3, i_3)\}$ can be compressed as $\{o_1, o_2, o_3\} \times \{i_1, i_2, i_3\}$. Note that we can interpret this compression scheme as factorizing a set of dependencies as a cross product. The Smoke system (Psallidas and Wu, 2018a) mentioned above optimizes lineage capture for known workloads, e.g., by partitioning a backwards provenance index (mapping output tuple identifiers to input tuple identifiers) based on group-by attributes to enable queries using the same group-by attributes (or a sub- or superset of these group-by attributes) to be refreshed to reflect a filter criterion (the filtered result of one query is traced back to the relevant subset of its provenance and then the other queries that should be refreshed are reevaluated over the provenance instead of the full database).

As an example of (iii), consider Niu *et al.* (2018) and Niu *et al.* (2017b) which did present heuristic and cost-based optimization techniques that are targeted at diverse provenance capture and analysis tasks. The authors observed that database optimizers are not well equipped to deal with provenance capture queries created by instrumentation, because of the unusual structure exhibited by these queries. Since an exhaustive exploration of the search space is not feasible for query optimization in general, database optimizers focus on optimizations that benefit typical use cases. Niu *et al.* (2017b) demonstrated that algebraic rewrites that target common issues for provenance capture operations such as prevalence of operators that block join reordering can significantly improve the performance of capture queries (by several orders of magnitude in some cases).

4.4 Querying, Exploring, and Visualizing Provenance

In addition to provenance capture and storage capabilities, provenance management systems should support the user in analyzing the captured provenance information by providing query, analysis, and visualization capabilities for provenance information.

4.4.1 Querying Provenance

Two fundamental approaches for querying provenance information have been proposed. Either the query language for which provenance should be captured is extended with new language features for querying (and potentially capturing) provenance or a new query language specialized for provenance is proposed. In the following we will refer to the language for which we are capturing provenance as the *target language* and the data model used by this query language the *target data model*. Note that the first approach is not applicable for, e.g., workflow provenance, where the transformations for which provenance is captured are not declarative queries.

Extending Query Languages for Provenance Support

When provenance information is encoded in the target data model, then the target language can be used to query provenance information. For instance, the Perm (Glavic *et al.*, 2013b) and GProM (Arab *et al.*, 2018a) systems implement this approach. These systems extend a target language, e.g., SQL, by adding support for provenance capture as a query feature. This enables flexible combination of provenance capture with queries over provenance information and data.

Example 68 (Querying Provenance with SQL)**.** The syntax of provenance capture in GProM is `PROVENANCE OF (Q)` where `Q` is the query for which provenance should be captured. For example, the query shown below captures the provenance of a query counting the number of students per country:

```
PROVENANCE OF (
  SELECT country, count(*) AS ttlstudents
  FROM students
  GROUP BY country
)
```

The `PROVENANCE OF` construct is a query construct that can be used in almost any place where a `SELECT` block would be allowed. This can be used to specify provenance capture and queries over provenance in a single query. For instance the query shown below captures the provenance of a query that returns the number of students with a GPA higher than 3.0 and then computes the number of students per country that are in the provenance of this query.

```
SELECT count(DISTINCT PROV_student_name) AS num_stds
FROM (PROVENANCE OF (
         SELECT country, count(*) AS ttlstudents
         FROM students
         WHERE gpa > 3.0
         GROUP BY country))
```

As another example of the benefits of having the full expressive power of SQL available for querying provenance, consider computing the difference between the provenance of two queries Q1 and Q2 (which we assume access the same input, i.e., their provenance is union compatible).

```
WITH prov1 AS (
  PROVENANCE OF (
    Q1
),
prov2 AS (
  PROVENANCE OF (
    Q2
)
SELECT PROV_COUNTRY FROM prov1 WHERE
EXCEPT ALL
SELECT PROV_COUNTRY FROM prov2 WHERE
UNION ALL
SELECT PROV_COUNTRY FROM prov2 WHERE
EXCEPT ALL
SELECT PROV_COUNTRY FROM prov1 WHERE
```

Provenance-specific Query Languages

Several query languages specific for provenance have been proposed. When designing new query languages for provenance we have the freedom to tailor the language for provenance. However, often it is necessary to query provenance alongside the data it is describing which would require many of the features of the target language to be integrated into the provenance query language. (Karvounarakis *et al.*, 2010) presented the ProQL query language for querying provenance graphs for schema mappings encoding provenance polynomials. The language is inspired by query languages for semi-structured data. A query in ProQL returns subgraphs of a provenance graph, a distinguished set of nodes, and optionally evaluates the provenance polynomial corresponding to the graph in a specific application semiring under a given assignment of variables to semiring values.

Example 69 (ProQL Queries)**.** The query shown below retrieves all derivations of tuples from a relation cities where attribute country is equal to US. The query returns the subgraph of the provenance containing these derivations (specified in using `INCLUDE PATH` clause) and returns these city tuples (specified using the `RETURN` clause). The `FOR` clause specifies variable bindings. Variables can be bound to tuples and mappings. In the for clause, the user can also specify how tuples bound to variables should be connected in the graph.

```
FOR [cities $x]
WHERE $x.country = US
INCLUDE PATH [$x] <-+ []
RETURN $x
```

As mentioned above the language also allows the result returned by a query to be evaluated in a specific semiring. For that the user has to specify the semiring of choice and can optionally specified what annotations are assigned to input tuples. For example, the query shown below counts the number of derivations (semiring $\mathbb{N}$ selected by specifying `NUMBER OF DERIVATIONS`) while assigning each input tuple the default multiplicity of 1.

```
EVALUATE NUMBER OF DERIVATIONS OF {
    FOR [cities $x]
    WHERE $x.country = US
    INCLUDE PATH [$x] <-+ []
    RETURN $x
} ASSIGNING EACH leaf_node $y {
  CASE $y in cities and $y.country = US: SET 1
  DEFAULT : SET 0
}
```

Anand *et al.* (2010) introduces QLP, a query language for provenance graphs (the language was developed for querying workflow provenance, but could be applied to other provenance graph too). This is a closed language, i.e., query results are valid provenance graphs that can be queried further.

Example 70 (QLP Queries)**.** In the QLP language, queries are specified as paths of interest. Keyword `derived` specifies that nodes should be connected, possibly indirectly. For example, the three example queries from Anand *et al.* (2010) shown below intuitively mean:

- Find the subgraph of the provenance containing all number deriving node 19
- Find the subgraph containing all nodes derived from 6
- Find the subquery containing all paths leading through an activity `Slicer:1`.

```
* derived 19
6 derived *
* through Slicer:1 derived *
```

Deutch *et al.* (2015b), Deutch *et al.* (2018b), and Deutch *et al.* (2015a) introduced an approach for provenance capture that captures provenance for a Datalog query encoded as derivation trees. The user can specify tree patterns to state which parts of the provenance they are interested in. Given a user query, the approach instruments the query to capture provenance that matches the users query.

Optimizing Queries over Provenance

The performance of provenance queries can be affected significantly by the choice of storage layout. Heinis and Alonso (2008) did investigate how to speed up forward (which outputs depend on an input) and backward (on which inputs does an output depend on) queries over provenance graphs. This work uses a generalization of a common relational encoding of interval data that allows for efficient reachability queries without having to resort to recursive queries whose evaluation is quite inefficient in most relational systems. The optimization techniques of Niu *et al.* (2017b) and Niu *et al.* (2018) are also applicable for optimizing queries over provenance data in addition to optimizing provenance capture.

4.4.2 Provenance Visualization and Exploration

The size of fine-grained provenance information can be overwhelming for users. In addition to supporting queries over provenance, this problem can be addressed by creating suitable, possibly interactive, visualizations that empower the user to explore provenance information. Some examples of visualization approaches for provenance are Hoekstra and Groth (2014), Suriarachchi *et al.* (2015), Borkin *et al.* (2013), Kunde *et al.* (2008), and Rio and Silva (2007).

4.5 Provenance Management Systems

In the preceding sections we have discussed methods and algorithms for managing provenance. We now give a brief (and incomplete) overview of systems for managing provenance information.

4.5.1 Relational Systems

The **Perm** system extends PostgreSQL with support for provenance capture implemented using instrumentation. The system was the first to support complex SQL features like nested subqueries. Provenance capture using the relational encoding of provenance discussed in Section 4.1.5 is integrated into SQL as a query feature that can be combined with other SQL features to enable complex queries over provenance. **GProM** (Arab *et al.*, 2018a) implements this approach in a middleware that supports multiple database backends. This system implement provenance-aware query optimization techniques (Niu *et al.*, 2017b), supports provenance tracking for transactions using reenactment (Arab *et al.*, 2018b), and has been used as a platform to study other extensions of the relational model that can be implemented using implementation such as uncertain (Feng *et al.*, 2019) and temporal data management (Dignös *et al.*, 2019). The **PUG** system (Lee *et al.*, 2018) extends GProM with support for capturing and summarizing provenance for queries with negation and, thus, also why-not provenance. **Smoke** (Psallidas and Wu, 2018a) implements provenance capture in a main-memory database system utilizing existing data structures created during query processing for capture. Müller *et al.* (2018) implemented provenance capture for a large subset of SQL including recursive queries, window functions, and nested subqueries using the two-stage approach described in Section 4.3.3. The **Orchestra** update exchange system tracks semiring provenance for schema mappings in a distributed setting. **Trio** (Agrawal *et al.*, 2006) is system for probabilistic data management that eagerly captures provenance of SQL queries. The lineage provenance capture approach described in Section 4.3.1 was implemented in the **WHIPS** data warehousing system (Cui *et al.*, 2000).

4.5.2 Big Data Platforms

Titian (Interlandi *et al.*, 2016) tracks provenance information for Apache Spark. The system has been used as a platform to implement debugging frameworks for big data analytics (Gulzar *et al.*, 2017a). **Pebble** (Diestelkämper and Herschel, 2020) tracks provenance of nested in Apache Spark. To improve the performance of provenance capture, Pebble instruments queries to capture row level provenance during query execution and then combines this with instance-independent information about the structural transformations (e.g., unnesting) applied by the query to restore fine-grained provenance at the level of individual nested elements withing a tuple. The **Lipstick** system (Amsterdamer *et al.*, 2011a) captures semiring provenance for a subset of the PIG language that is implemented on top of MapReduce. Other systems in this space are **Newt** (Logothetis *et al.*, 2013), **RAMP** (Ikeda *et al.*, 2011a), **HadoopProv** (Akoush *et al.*, 2013), and **PROVision** (Zheng *et al.*, 2019).

4.5.3 Why-not Provenance

The first instance-based approach was presented in Huang *et al.* (2008). The first query-based approach in Chapman and Jagadish (2009). **Artemis** (Herschel *et al.*, 2009) computes instance-based why-not provenance using techniques from incomplete databases (C-tables as introduced in Imieliński and Lipski Jr (1984)) to compactly represent the large space of missing answers and uses constraint solving to find such explanations. **Conseil** (Herschel, 2013) computes hybrids of instance- and query-based explanations for missing answers. **Ted++** (Bidoit *et al.*, 2015) computes syntax-independent query-based explanations. The **Pug** system (Lee *et al.*, 2018) unifies why and why-not provenance through provenance tracking for queries with negation and efficiently creates approximate summaries of why-not provenance. **ConQueR** (Tran and Chan, 2010) explains missing answer by refining the input query such that it returns the missing answer.

5

Connection to Other Research Fields

While the term data provenance mainly emerged in the database and HPC communities, many of the concepts and methods applied in database provenance are closely related to work in other research fields such as programming languages and compilers, formal languages, logic programming, and security. Often, the development of these methods predates their use by the database community. In this chapter, we discuss these lines of work and explain how the relate to database provenance techniques and concepts discussed in the previous chapters.

5.1 Dataflow Analysis, Controlflow Analysis, and Program Slicing

In programming languages and compiler design it is crucial to understand the flow of control and information in a program, either statically for all possible inputs or dynamically for a particular given input. For instance, many compiler optimizations rely on knowing which variables the values of a variable at a position in a program depends on. Extracting this type of information is referred to as **dataflow** and **controlflow** analysis (Allen, 1970). Data and controlflow can either be analyzed statically, i.e., for all possible inputs, or dynamically for a given input. Static analysis is typically done using worst-case analysis, i.e., it creates

Input Program

```
1 int f(int x) {
2   y = 0;
3   z =  0;
4   if (x < 20)
5     y = 15;
6   else
7     z = x;
8   print("%u,%u", y, z);
9 }
```

Slice for y at line 9

```
1 int f(int x) {
2   y = 0;
3
4   if (x < 20)
5     y = 15;
6
7
8   print("%u,%u", y, z);
9 }
```

Figure 5.1: Example Program

an over-approximation of all possible dependencies for any possible input. Mapping these concepts to the terminology for provenance we have used in this article, dataflow analysis produces data dependencies.

Example 71 (Data- and Controlflow Analysis)**.** Consider the C program shown below. Let us consider the value of variable z at line 8. The values of variable `z` at this line and which other variables it depends on is determined by control flow decisions. If variable x's value is greater than or equal to 20 then line 7 gets executed and z is assigned the current value of x. That is for some inputs the value of variable z depends on the value of x at line 2.

Another related program analysis technique is program slicing. **Program slicing** (Weiser, 1981) identifies which part of the program (called a slice) is relevant for computing the value of a variable at a particular statement. Like data and control flow analysis, program slicing can be done statically (returning a slice that is sufficient for producing the variable's value for any possible input) or dynamically (return a slice specific to a particular input).[1] Program slices are computed using dataflow analysis. Oversimplifying a bit, we need to include all statements that are control or data dependencies for the variables at the statement of interest. Cheney (2007) noted the relationship between program

[1] Note that computing a minimal slice is undecidable. This was proven in Weiser (1981) through a reduction from the halting problem.

slicing and data provenance. Note that using our terminology, a program slice consists of the parts of the program that are transformation dependencies of the output of interest.

Example 72 (Program Slicing). To compute a static program slice for the output variable y at line 9, we determine the dependencies of variable y at 9. The value of y may depend on the value of x because the execution of line 5 that changes the value of y depends on the value of x (if x is less than 20). Furthermore, if x is larger than or equal to 20 then the final value of y will be the value assigned at line 2. Thus, these statements and the if statement have to be included in the slice.

5.2 Taint Analysis

In security applications, there is often a need to analyze how inputs affect the execution of a program. For example, this information may be used to determine misuse of program inputs that enable exploits. **Taint analysis** (Schwartz *et al.*, 2010) is a method for tracking dependencies by tainting data values in the program's input and then propagating these taints through the execution of the program. Annotation propagation techniques for provenance capture are closely related to taint tracking. The main difference is that taints are typically mere sets of identifiers while provenance annotations may store more detailed information, e.g., provenance polynomials also record how inputs have been combined to compute a result.

5.3 Symbolic Execution

Symbolic execution is a program analysis technique which reasons about all possible executions of a program. Instead of a concrete input, the input of a program under symbolic execution is a set of symbolic values that can represent any possible input. In the symbolic execution of a program, logical formula are constructed for possible execution paths in the program that encode constraints on the values of variables that have to hold to follow this execution path. To find concrete inputs that lead to particular execution path, satisfiability solvers (Moura and Bjørner, 2011) are used to find concrete inputs that result in a particular

```
 1  static int f(char x, char y) {
 2      if (x == 'a')
 3          return -1;
 4      if (x == 'b') {
 5          if (y == 'a') {
 6              return 0;
 7          }
 8          return 1;
 9      }
10      return 2;
11  }
```

Figure 5.2: Concrete and Symbolic Execution

execution path. Symbolic execution is applied in test case generation, because it allows the automatic generation of test cases that cover all execution paths of a program.

Example 73 (Symbolic Execution)**.** Consider the program shown in Figure 5.2. Function `f` takes two parameter. In symbolic execution we assign two symbolic values, say v_x and v_y to these two input parameters. Then symbolic expressions (logical formula) are constructed for execution paths. For example, consider the execution path that does not execute the body of the first if statement and which executed line 6 because the conditions of the if statements on lines 4 and 5 evaluated to true. This execution path is taken if $v_x \neq a$ (the condition of the first if statement fails), $v_x = b$ (the condition of the second if statement evaluates to true), and $v_y = a$ (the condition of the second if statement evaluates to true). Thus, the symbolic expression associated with this execution path is:

$$v_x \neq a \wedge v_x = b \wedge v_y = a$$

To find a concrete input for function f that exercises this execution path, we can use a constraint solver (Moura and Bjørner, 2011) to find a satisfying assignment for the symbolic expression. The assignment $v_x = b$ and $v_y = a$ is the only satisfying assignment for this formula.

As discussed in Chapter 2 the concept of symbolic expressions is also used by several provenance models, e.g., provenance polynomials.

Similar to symbolic execution this enables reasoning about evaluation of queries over all possible inputs. For instance, as outlined in Section 3.2 and Section 3.5 we can use this property for solving the view deletion and probabilistic query processing problems.

5.4 Applications of Semirings

In addition to their use in provenance models, the use of semirings in many areas of computer science predates their use for provenance. For instance, semirings can be used to evaluate properties of context-free grammars such that ambiguity of a grammar (Chomsky and Schützenberger, 1959). Another application of semirings are variants of shortest path problems over graphs (Mohri, 2002; Ramusat *et al.*, 2018). To support such applications, solvers for equation systems over semirings have been developed (e.g., Esparza *et al.* (2014)). Gondran and Minoux (2008) provides a detailed discussion of semirings and their applications.

5.5 Justifications and Debugging for Logic Programming

Debugging of logic programs has been studied intensively in the past. Many techniques developed by this field are related to provenance. For example, we already discussed the provenance model from Damásio *et al.* (2013). Caballero *et al.* (2015), Brain *et al.* (2007), Brain and Vos (2005), Comini *et al.* (1995), Drabent *et al.* (1988), and Pereira (1986) are examples of approaches for debugging logic programs.

6

Summary and Conclusions

In this article we have provided a comprehensive overview of research on database provenance. Provenance is an active research area in databases as well as in other fields of computer science. Great progress has been made since provenance started to receive more widespread attention in the early 2000s. While concepts similar to provenance have been studied much earlier in other fields (see Chapter 5), the study of database provenance has lead to new developments that go beyond what has been proposed in other fields. One noteworthy contribution are formal provenance models which often provide stronger guarantees than, e.g., data dependency tracking and taint tracking in programming languages. Apart from formal provenance semantics, methods for capturing, querying, and storing provenance information have emerged and have lead to the integration of provenance functionality in data management systems. While there are several good surveys that cover database provenance (e.g., Cheney *et al.*, 2009; Herschel *et al.*, 2017), they either do not provide a detailed introduction of provenance concepts or have been published to long ago to cover more recent developments in the field. The goal of this article is to close this gap. Because this article is meant as a technical introduction to fundamental concepts in provenance research in databases, we had to exclude many interesting lines of work on data provenance in fields other than databases and even from the database community. For instance, there is a large body of work on workflow provenance that we merely touched upon.

Acknowledgements

This work is supported in part by NSF Awards OAC-1640864 and IIS-1956123. The conclusions and opinions in this work are solely those of the authors and do not represent the views of the National Science Foundation.

References

Abiteboul, S., R. Hull, and V. Vianu. (1995). *Foundations of Databases*. Addison-Wesley.

Agrawal, P., O. Benjelloun, A. D. Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom. (2006). “An Introduction to ULDBs and the Trio System”. *IEEE Data Engineering Bulletin*. 29(1): 5–16.

Agrawal, R., R. Bayardo, C. Faloutsos, J. Kiernan, R. Rantzau, and R. Srikant. (2004). “Auditing compliance with a hippocratic database”. In: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment. 516–527.

Akoush, S., R. Sohan, and A. Hopper. (2013). “HadoopProv: Towards Provenance As A First Class Citizen In MapReduce”. *TaPP*.

Alexe, B., L. Chiticariu, and W. Tan. (2006). “SPIDER: a schema mapPIng DEbuggeR”. In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment. 1179–1182.

Allen, F. (1970). “Control flow analysis”. *Proceedings of a symposium on Compiler optimization*. 5(7): 1–19.

Alvaro, P., J. Rosen, and J. M. Hellerstein. (2015). “Lineage-driven Fault Injection”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 331–346.

Amarilli, A., P. Bourhis, and P. Senellart. (2015). "Provenance Circuits for Trees and Treelike Instances". In: *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*. 56–68. DOI: 10.1007/978-3-662-47666-6_5. URL: https://doi.org/10.1007/978-3-662-47666-6%5C_5.

Amsterdamer, Y., S. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. (2011a). "Putting Lipstick on Pig: Enabling Database-style Workflow Provenance". *Proceedings of the VLDB Endowment*. 5(4): 346–357.

Amsterdamer, Y., D. Deutch, T. Milo, and V. Tannen. (2011b). "On Provenance Minimization". In: *Proceedings of the 30th Symposium on Principles of Database Systems (PODS)*. 141–152.

Amsterdamer, Y., D. Deutch, and V. Tannen. (2011c). "On the Limitations of Provenance for Queries with Difference". In: *TaPP '11: 3rd USENIX Workshop on the Theory and Practice of Provenance*.

Amsterdamer, Y., D. Deutch, and V. Tannen. (2011d). "Provenance for Aggregate Queries". In: *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 153–164.

Anand, M. K., S. Bowers, and B. Ludäscher. (2010). "Techniques for efficiently querying scientific workflow provenance graphs." In: *EDBT*. Vol. 10. 287–298.

Anand, M. K., S. Bowers, T. McPhillips, and B. Ludäscher. (2009). "Efficient Provenance Storage over Nested Data Collections". In: *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*. 958–969.

Apache. *http://atlas.apache.org/*. (Accessed on 2017).

Arab, B., S. Feng, B. Glavic, S. Lee, X. Niu, and Q. Zeng. (2018a). "GProM - A Swiss Army Knife for Your Provenance Needs". *IEEE Data Engineering Bulletin*. 41(1): 51–62.

Arab, B., D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. (2016). "Reenactment for Read-Committed Snapshot Isolation". In: *Proceedings of the 25th ACM International Conference on Information and Knowledge Management*. 841–850.

Arab, B., D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. (2018b). "Using Reenactment to Retroactively Capture Provenance for Transactions". *IEEE Transactions on Knowledge and Data Engineering*. 30(3): 599–612. DOI: 10.1109/TKDE.2017.2769056.

Arocena, P. C., B. Glavic, and R. J. Miller. (2013). "Value Invention for Data Exchange". In: *Proceedings of the 39th International Conference on Management of Data*. 157–168.

Assadi, S., S. Khanna, Y. Li, and V. Tannen. (2016). "Algorithms for Provisioning Queries and Analytics". In: *19th International Conference on Database Theory, ICDT 2016, Bordeaux, France, March 15-18, 2016*. 18:1–18:18. DOI: 10.4230/LIPIcs.ICDT.2016.18. URL: https://doi.org/10.4230/LIPIcs.ICDT.2016.18.

Bakibayev, N., D. Olteanu, and J. Závodný. (2012). "FDB: A query engine for factorised relational databases". *Proceedings of the VLDB Endowment*. 5(11): 1232–1243.

Bao, Z., H. Köhler, L. Wang, X. Zhou, and S. W. Sadiq. (2012). "Efficient provenance storage for relational queries". In: *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*. Ed. by X. Chen, G. Lebanon, H. Wang, and M. J. Zaki. ACM. 1352–1361. ISBN: 978-1-4503-1156-4. DOI: 10.1145/2396761.2398439. URL: https://doi.org/10.1145/2396761.2398439.

Bates, A. and W. U. Hassan. (2019). "Can Data Provenance Put an End To the Data Breach?" *IEEE Security & Privacy*. 17(4): 88–93. DOI: 10.1109/MSEC.2019.2913693. URL: https://doi.org/10.1109/MSEC.2019.2913693.

Bates, A., D. Tian, K. R. B. Butler, and T. Moyer. (2015). "Trustworthy Whole-System Provenance for the Linux Kernel". In: *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. Ed. by J. Jung and T. Holz. USENIX Association. 319–334. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/bates.

Becker, R. A. and J. M. Chambers. (1988). "Auditing of data analyses". *Journal on Scientific and Statistical Computation*: 747–760.

Benjelloun, O., A. D. Sarma, A. Y. Halevy, and J. Widom. (2006). "ULDBs: Databases with Uncertainty and Lineage". In: *Proceedings of the 32th International Conference on Very Large Data Bases (VLDB).* 953–964.

Berenson, H., P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. (1995). "A critique of ANSI SQL isolation levels". *ACM SIGMOD Record.* 24(2): 1–10.

Bertossi, L. and B. Salimi. (2013). "Causality in Databases, Database Repairs, and Consistency-Based Diagnosis".

Bhagwat, D., L. Chiticariu, W. C. Tan, and G. Vijayvargiya. (2005). "An Annotation Management System for Relational Databases". *VLDB J.* 14(4): 373–396. DOI: 10.1007/s00778-005-0156-6. URL: https://doi.org/10.1007/s00778-005-0156-6.

Bhagwat, D., L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. (2004). "An Annotation Management System for Relational Databases". In: *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases.* 900–911.

Bhardwaj, A., A. Deshpande, A. J. Elmore, D. Karger, S. Madden, A. Parameswaran, H. Subramanyam, E. Wu, and R. Zhang. (2015). "Collaborative data analytics with DataHub". *Proceedings of the VLDB Endowment.* 8(12): 1916–1919.

Bhattacherjee, S., A. Chavan, S. Huang, A. Deshpande, and A. Parameswaran. (2015). "Principles of dataset versioning: Exploring the recreation/storage tradeoff". *Proceedings of the VLDB Endowment.* 8(12): 1346–1357.

Bidoit, N., M. Herschel, and A. Tzompanaki. (2015). "Efficient Computation of Polynomial Explanations of Why-Not Questions". In: *Proceedings of the 24th ACM International Conference on Information and Knowledge Management, CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015.* Ed. by J. Bailey, A. Moffat, C. C. Aggarwal, M. de Rijke, R. Kumar, V. Murdock, T. K. Sellis, and J. X. Yu. ACM. 713–722. ISBN: 978-1-4503-3794-6. DOI: 10.1145/2806416.2806426. URL: https://doi.org/10.1145/2806416.2806426.

Bidoit, N., M. Herschel, K. Tzompanaki, *et al.* (2014). "Query-Based Why-Not Provenance with NedExplain". In: *Extending Database Technology (EDBT).*

Bidoit, N., M. Herschel, and K. Tzompanaki. (2016). “Refining SQL Queries based on Why-Not Polynomials”. In: *8th USENIX Workshop on the Theory and Practice of Provenance (TaPP 16)*. Washington, D.C.: USENIX Association.

Biton, O., S. Cohen-Boulakia, S. Davidson, and C. Hara. (2008). “Querying and Managing Provenance through User Views in Scientific Workflows”. *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*: 1072–1081.

Biton, O., S. Cohen-Boulakia, and S. B. Davidson. (2007). “Zoom* userviews: Querying relevant provenance in workflow systems”. In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 1366–1369.

Borkin, M. A., C. S. Yeh, M. Boyd, P. Macko, K. Z. Gajos, M. I. Seltzer, and H. Pfister. (2013). “Evaluation of Filesystem Provenance Visualization Tools”. *IEEE Trans. Vis. Comput. Graph.* 19(12): 2476–2485. DOI: 10.1109/TVCG.2013.155. URL: https://doi.org/10.1109/TVCG.2013.155.

Bourhis, P., D. Deutch, and Y. Moskovitch. (2020). “Equivalence-Invariant Algebraic Provenance for Hyperplane Update Queries”. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference Portland, OR, USA, June 14-19, 2020*. Ed. by D. Maier, R. Pottinger, A. Doan, W.-C. Tan, A. Alawini, and H. Q. Ngo. ACM. 415–429. ISBN: 978-1-4503-6735-6. DOI: 10.1145/3318464.3380578. URL: https://doi.org/10.1145/3318464.3380578.

Brachmann, M., C. Bautista, S. Castelo, S. Feng, J. Freire, B. Glavic, O. Kennedy, H. Müller, R. Rampin, W. Spoth, and Y. Yang. (2019). “Data Debugging and Exploration with Vizier”. In: *Proceedings of the 44th International Conference on Management of Data (Demonstration Track)*.

Brain, M., M. Gebser, J. Pührer, T. Schaub, H. Tompits, and S. Woltran. (2007). "Debugging ASP Programs by Means of ASP". In: *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*. Ed. by C. Baral, G. Brewka, and J. S. Schlipf. Vol. 4483. *Lecture Notes in Computer Science*. Springer. 31–43. ISBN: 978-3-540-72199-4. DOI: 10.1007/978-3-540-72200-7_5. URL: https://doi.org/10.1007/978-3-540-72200-7%5C_5.

Brain, M. and M. D. Vos. (2005). "Debugging Logic Programs under the Answer Set Semantics". In: *Answer Set Programming, Advances in Theory and Implementation, Proceedings of the 3rd Intl. ASP'05 Workshop, Bath, UK, September 27-29, 2005*. Ed. by M. D. Vos and A. Provetti. Vol. 142. *CEUR Workshop Proceedings*. CEUR-WS.org. URL: http://ceur-ws.org/Vol-142/page141.pdf.

Buneman, P., J. Cheney, and S. Vansummeren. (2008). "On the Expressiveness of Implicit Provenance in Query and Update Languages". *ACM Transactions on Database Systems (TODS)*. 33(4): 1–47.

Buneman, P., S. Khanna, and W.-C. Tan. (2001). "Why and Where: A Characterization of Data Provenance". In: *ICDT*. 316–330.

Buneman, P., S. Khanna, and W.-C. Tan. (2002). "On Propagation of Deletions and Annotations through Views". In: *PODS '02: Proceedings of the 21th Symposium on Principles of Database Systems*. 150–158.

Buneman, P., E. V. Kostylev, and S. Vansummeren. (2013). "Annotations are relative". In: *Proceedings of the 16th International Conference on Database Theory*. ACM. 177–188.

Caballero, R., Y. Garcia-Ruiz, and F. Saenz-Perez. (2015). "Debugging of Wrong and Missing Answers for Datalog Programs with Constraint Handling Rules". In: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming. PPDP '15*. Siena, Italy: ACM. 55–66. ISBN: 978-1-4503-3516-4.

Callahan, S. P., J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. (2006). "Managing the evolution of dataflows with vistrails". In: *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*. IEEE. 71–71.

Cate, B. ten, C. Civili, E. Sherkhonov, and W.-C. Tan. (2015). “High-level why-not explanations using ontologies”. In: *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM. 31–43.

Ceri, S., G. Gottlob, and L. Tanca. (1989). “What you always wanted to know about Datalog(and never dared to ask)”. *IEEE Transactions on Knowledge and Data Engineering*. 1(1): 146–166.

Chandra, A. K. and D. Harel. (1985). “Horn Clauses Queries and Generalizations”. *J. Log. Program.* 2(1): 1–15. DOI: 10.1016/0743-1066(85)90002-0. URL: https://doi.org/10.1016/0743-1066(85)90002-0.

Chapman, A. and H. V. Jagadish. (2009). “Why Not?” In: *SIGMOD ’09: Proceedings of the 35th SIGMOD International Conference on Management of Data*. 523–534.

Chapman, A., H. V. Jagadish, and P. Ramanan. (2008). “Efficient Provenance Storage”. In: *SIGMOD ’08: Proceedings of the 35th SIGMOD International Conference on Management of Data*. 993–1006.

Chavan, A., S. Huang, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. (2015). “Towards a Unified Query Language for Provenance and Versioning”. In: *7th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2015, Edinburgh, Scotland, UK, July 8-9, 2015*. Ed. by P. Missier and J. Zhao. USENIX Association. URL: https://www.usenix.org/conference/tapp15/workshop-program/presentation/chavan.

Cheney, J. (2007). “Program Slicing and Data Provenance”. *IEEE Data Engineering Bulletin*. 30(4): 22–28.

Cheney, J., A. Ahmed, and U. A. Acar. (2014). “Database Queries that Explain their Work”. In: *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*. Ed. by O. Chitil, A. King, and O. Danvy. ACM. 271–282. ISBN: 978-1-4503-2947-7. DOI: 10.1145/2643135.2643143. URL: https://doi.org/10.1145/2643135.2643143.

Cheney, J., L. Chiticariu, and W.-C. Tan. (2009). "Provenance in Databases: Why, How, and Where". *Foundations and Trends in Databases*. 1(4): 379–474.

Chirigati, F. S., D. Shasha, and J. Freire. (2013). "ReproZip: Using Provenance to Support Computational Reproducibility." In: *TaPP*.

Chiticariu, L. and W.-C. Tan. (2006). "Debugging Schema Mappings with Routes". In: *VLDB '06: Proceedings of the 32th International Conference on Very Large Data Bases*. 79–90.

Chiticariu, L., W.-C. Tan, and G. Vijayvargiya. (2005). "DBNotes: a Post-it System for Relational Databases based on Provenance". In: *SIGMOD '05: Proceedings of the 31th SIGMOD International Conference on Management of Data*. 942–944.

Chockler, H. and J. Halpern. (2004). "Responsibility and blame: A structural-model approach". *Journal of Artificial Intelligence Research*. 22(1): 93–115. ISSN: 1076-9757.

Chockler, H., J. Halpern, and O. Kupferman. (2008). "What causes a system to satisfy a specification?" *ACM Transactions on Computational Logic (TOCL)*. 9(3): 1–26. ISSN: 1529-3785.

Chomsky, N. and M. P. Schützenberger. (1959). "The algebraic theory of context-free languages". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 26. Elsevier. 118–161.

Chu, S., B. Murphy, J. Roesch, A. Cheung, and D. Suciu. (2018). "Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries". *Proceedings of the VLDB Endowment*. 11(11): 1482–1495.

Comini, M., G. Levi, and G. Vitiello. (1995). "Efficient Detection of Incompleteness Errors in the Abstract Debugging of Logic Programs". In: *Proceedings of the Second International Workshop on Automated Debugging, AADEBUG 1995, Saint Malo, France, May 22-24, 1995*. 159–174.

Cong, G., W. Fan, F. Geerts, J. Li, and J. Luo. (2012). "On the Complexity of Annotation Propagation and View Update Analyses". *IEEE Transactions on Knowledge and Data Engineering*.

Cui, Y. and J. Widom. (2000a). "Practical Lineage Tracing in Data Warehouses". In: *ICDE*. 367–378.

Cui, Y. and J. Widom. (2000b). "Storing Auxiliary Data for Efficient Maintenance and Lineage Tracing of Complex Views". In: *DMDW '00: Proceedings of the 2th International Workshop on Design and Management of Data Warehouses.*

Cui, Y. and J. Widom. (2001). "Run-time Translation of View Tuple Deletions using Data Lineage". *Tech. rep.* Stanford University.

Cui, Y., J. Widom, and J. L. Wiener. (2000). "Tracing the Lineage of View Data in a Warehousing Environment". *TODS.* 25(2): 179–227.

Damásio, C. V., A. Analyti, and G. Antoniou. (2013). "Justifications for logic programming". In: *Logic Programming and Nonmonotonic Reasoning.* Springer. 530–542.

Davison, A. P., M. Mattioni, D. Samarkanov, and B. Telenczuk. (2014). "Sumatra: A Toolkit for Reproducible Research". *Implementing Reproducible Research*: 57.

Dean, J. and S. Ghemawat. (2004). "MapReduce: simplified data processing on large clusters". In: *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6. OSDI'04.* San Francisco, CA.

Deutch, D., Z. Ives, T. Milo, and V. Tannen. (2013a). "Caravan: Provisioning for What-If Analysis". *CIDR '13.*

Deutch, D. and N. Frost. (2019). "Constraints-based explanations of classifications". In: *2019 IEEE 35th International Conference on Data Engineering (ICDE).* IEEE. 530–541.

Deutch, D., N. Frost, and A. Gilad. (2017). "Provenance for Natural Language Queries". *Proceedings of the VLDB Endowment.* 10(5).

Deutch, D., N. Frost, and A. Gilad. (2020). "Explaining Natural Language Query Results". *VLDB J.* 29(1): 485–508. DOI: 10.1007/s00778-019-00584-7. URL: https://doi.org/10.1007/s00778-019-00584-7.

Deutch, D., N. Frost, A. Gilad, and T. Haimovich. (2018a). "Nlprovenans: Natural Language Provenance for Non-Answers". *Proc. VLDB Endow.* 11(12): 1986–1989. DOI: 10.14778/3229863.3236241. URL: https://doi.org/10.14778/3229863.3236241.

Deutch, D. and A. Gilad. (2016). “QPlain: Query by explanation”. In: *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. 1358–1361. DOI: 10.1109/ICDE.2016.7498344. URL: https://doi.org/10.1109/ICDE.2016.7498344.

Deutch, D. and A. Gilad. (2019). “Reverse-Engineering Conjunctive Queries from Provenance Examples”. In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. 277–288. DOI: 10.5441/002/edbt.2019.25. URL: https://doi.org/10.5441/002/edbt.2019.25.

Deutch, D., A. Gilad, and Y. Moskovitch. (2015a). “Selective Provenance for Datalog Programs Using Top-K Queries”. *Proceedings of the VLDB Endowment*. 8(12).

Deutch, D., A. Gilad, and Y. Moskovitch. (2015b). “selP: Selective tracking and presentation of data provenance”. In: *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE. 1484–1487.

Deutch, D., A. Gilad, and Y. Moskovitch. (2018b). “Efficient Provenance Tracking for Datalog Using Top-K Queries”. *VLDB J.* 27(2): 245–269. DOI: 10.1007/s00778-018-0496-7. URL: https://doi.org/10.1007/s00778-018-0496-7.

Deutch, D., T. Milo, S. Roy, and V. Tannen. (2014). “Circuits for Datalog Provenance”. In: *ICDT*. 201–212.

Deutch, D., Y. Moskovitch, and V. Tannen. (2013b). “PROPOLIS: Provisioned Analysis of Data-Centric Processes”. *Proceedings of the VLDB Endowment*. 6(12).

Diestelkämper, R. and M. Herschel. (2020). “Tracing nested data with structural provenance for big data analytics”. In: *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*. Ed. by A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher, A. Khan, and B. Yang. OpenProceedings.org. 253–264. ISBN: 978-3-89318-083-7. DOI: 10.5441/002/edbt.2020.23. URL: https://doi.org/10.5441/002/edbt.2020.23.

Dietrich, B. and T. Grust. (2015). “A SQL Debugger Built from Spare Parts: Turning a SQL: 1999 Database System into Its Own Debugger”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 865–870.

Dignös, A., B. Glavic, X. Niu, M. H. Böhlen, and J. Gamper. (2019). “Snapshot Semantics for Temporal Multiset Relations”. *Proceedings of the VLDB Endowment*. 12(6): 639–652.

Dong, X. L. and D. Srivastava. (2013). “Compact explanation of data fusion decisions”. In: *Proceedings of the 22nd international conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 379–390.

Drabent, W., S. Nadjm-Tehrani, and J. Maluszynski. (1988). “Algorithmic Debugging with Assertions”. In: *Meta-Programming in Logic Programming, Workshop on Meta-Programming in Logic, META 1988, University of Bristol, 22-24 June, 1988*. 501–521.

El Gebaly, K., P. Agrawal, L. Golab, F. Korn, and D. Srivastava. (2014). “Interpretable and informative explanations of outcomes”. *Proceedings of the VLDB Endowment*. 8(1).

El Gebaly, K., G. Feng, L. Golab, F. Korn, and D. Srivastava. (2018). “Explanation Tables”. *Sat.* 5: 14.

Eltabakh, M. Y., M. Ouzzani, W. G. Aref, A. K. Elmagarmid, Y. Laura-Silva, M. U. Arshad, D. Salt, and I. Baxter. (2008). “Managing Biological Data using BDBMS”. In: *ICDE ’08: Proceedings of the 24th International Conference on Data Engineering (demonstration)*. 1600–1603.

Esparza, J., S. Kiefer, and M. Luttenberger. (2007). “On fixed point equations over commutative semirings”. In: *STACS 2007*. 296–307.

Esparza, J., M. Luttenberger, and M. Schlund. (2014). “FPsolve: A Generic Solver for Fixpoint Equations over Semirings”. In: *Implementation and Application of Automata*. 1–15.

Fabbri, D. and K. LeFevre. (2011). “Explanation-based auditing”. *Proceedings of the VLDB Endowment*. 5(1): 1–12.

Fagin, R., P. Kolaitis, L. Popa, and W. Tan. (2005a). “Composing schema mappings: Second-order dependencies to the rescue”. *ACM Transactions on Database Systems (TODS)*. 30(4): 994–1055.

Fagin, R., P. G. Kolaitis, R. J. Miller, and L. Popa. (2005b). “Data Exchange: Semantics and Query Answering”. *Theoretical Computer Science*. 336(1): 89–124.

Farnadi, G., B. Babaki, and L. Getoor. (2018). “Fairness in Relational Domains”. In: *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society, AIES 2018, New Orleans, LA, USA, February 02-03, 2018*. 108–114. DOI: 10.1145/3278721.3278733. URL: https://doi.org/10.1145/3278721.3278733.

Feng, S., A. Huber, B. Glavic, and O. Kennedy. (2019). “Uncertainty Annotated Databases - A Lightweight Approach for Approximating Certain Answers”. In: *Proceedings of the 44th International Conference on Management of Data*.

Fernandez, C., F. J. Provost, and X. Han. (2019). “Counterfactual Explanations for Data-Driven Decisions”. In: *Proceedings of the 40th International Conference on Information Systems, ICIS 2019, Munich, Germany, December 15-18, 2019*. URL: https://aisel.aisnet.org/icis2019/data%5C_science/data%5C_science/8.

Fink, R., L. Han, and D. Olteanu. (2012). “Aggregation in probabilistic databases via knowledge compilation”. *Proceedings of the VLDB Endowment*. 5(5): 490–501.

Flum, J., M. Kubierschky, and B. Ludäscher. (1997). “Total and partial well-founded datalog coincide”. In: *Database Theory—ICDT’97*. Springer. 113–124.

Foster, J. N., T. J. Green, and V. Tannen. (2008). “Annotated XML: Queries and Provenance”. In: *PODS ’08: Proceedings of the 27th Symposium on Principles of Database Systems*. 271–280.

Freire, C., W. Gatterbauer, N. Immerman, and A. Meliou. (2015). “The complexity of resilience and responsibility for self-join-free conjunctive queries”. *Proceedings of the VLDB Endowment*. 9(3): 180–191.

Freire, C., W. Gatterbauer, N. Immerman, and A. Meliou. (2020). "New Results for the Complexity of Resilience for Binary Conjunctive Queries with Self-Joins". In: *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*. 271–284. DOI: 10.1145/3375395.3387647. URL: https://doi.org/10.1145/3375395.3387647.

Freire, J., P. Bonnet, and D. Shasha. (2011). "Exploring the coming repositories of reproducible experiments: Challenges and opportunities". *Proc. VLDB Endow.* 4: 1494–1497.

Freire, J., P. Bonnet, and D. Shasha. (2012). "Computational reproducibility: state-of-the-art, challenges, and database research opportunities". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 593–596.

Freire, J. and F. Chirigati. (2018). "Provenance and the Different Flavors of Computational Reproducibility". *Data Engineering*: 15.

Freire, J. and C. T. Silva. (2012). "Making Computations and Publications Reproducible with VisTrails". *Computing in Science and Engineering*. 14(4): 18–25.

Gawlick, D. and V. Radhakrishnan. (2011). "Fine Grain Provenance Using Temporal Databases". In: *TaPP '11: 3rd USENIX Workshop on the Theory and Practice of Provenance*.

Geerts, F., G. Karvounarakis, V. Christophides, and I. Fundulaki. (2012). "Algebraic Structures for Capturing the Provenance of SPARQL Queries". In: *Proceedings of the 16th International Conference on Database Theory*.

Geerts, F., A. Kementsietsidis, and D. Milano. (2006). "MONDRIAN: Annotating and querying databases through colors and blocks". In: *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. IEEE. 82–82.

Geerts, F. and A. Poggi. (2010). "On database query languages for K-relations". *Journal of Applied Logic*. 8(2): 173–185.

Glavic, B. (2010). "Perm: Efficient Provenance Support for Relational Databases". *PhD thesis*. University of Zurich.

Glavic, B. and G. Alonso. (2009a). “Perm: Processing Provenance and Data on the same Data Model through Query Rewriting”. In: *Proceedings of the 25th IEEE International Conference on Data Engineering*. 174–185.

Glavic, B. and G. Alonso. (2009b). “Provenance for Nested Subqueries”. In: *Proceedings of the 12th International Conference on Extending Database Technology*. 982–993.

Glavic, B. and G. Alonso. (2009c). “The Perm Provenance Management System in Action”. In: *Proceedings of the 35th ACM SIGMOD International Conference on Management of Data (Demonstration Track)*. 1055–1058.

Glavic, B., G. Alonso, R. J. Miller, and L. M. Haas. (2010). “TRAMP: Understanding the Behavior of Schema Mappings through Provenance”. *Proceedings of the Very Large Data Bases Endowment*. 3(1): 1314–1325.

Glavic, B., J. Du, R. J. Miller, G. Alonso, and L. M. Haas. (2011). “Debugging Data Exchange with Vagabond”. *Proceedings of the VLDB Endowment (Demonstration Track)*. 4(12): 1383–1386.

Glavic, B., K. S. Esmaili, P. M. Fischer, and N. Tatbul. (2013a). “Ariadne: Managing Fine-Grained Provenance on Data Streams”. In: *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*. 291–320.

Glavic, B., S. Köhler, S. Riddle, and B. Ludäscher. (2015). “Towards Constraint-based Explanations for Answers and Non-Answers”. In: *Proceedings of the 7th USENIX Workshop on the Theory and Practice of Provenance*.

Glavic, B., A. Meliou, and S. Roy. (2021). “Trends in Explanations”. *Foundations and Trends® in Databases*: to appear.

Glavic, B., R. J. Miller, and G. Alonso. (2013b). “Using SQL for Efficient Generation and Querying of Provenance Information”. *In search of elegance in the theory and practice of computation: a Festschrift in honour of Peter Buneman*: 291–320.

Gondran, M. and M. Minoux. (2008). *Graphs, dioids and semirings: new models and algorithms*. Vol. 41. Springer Science & Business Media.

Grädel, E. and V. Tannen. (2017). “Semiring Provenance for First-Order Model Checking”. *arXiv preprint arXiv:1712.01980*.

Grädel, E. and V. Tannen. (2020). “Provenance analysis for logic and games”. *Moscow Journal of Combinatorics and Number Theory*. 9(3): 203–228.

Green, T. (2011). “Containment of conjunctive queries on annotated relations”. *Theory of Computing Systems*. 49(2): 429–459.

Green, T., G. Karvounarakis, and Z. Tannen. (2010). “Provenance in ORCHESTRA”.

Green, T. J. and V. Tannen. (2017). “The Semiring Framework for Database Provenance”. In: *PODS*. 93–99.

Green, T. J. (2009). “Containment of Conjunctive Queries on Annotated Relations”. In: *ICDT ’09: Proceedings of the 16th International Conference on Database Theory*. 296–309.

Green, T. J., Z. G. Ives, and V. Tannen. (2009). “Reconcilable Differences”. In: *ICDT ’09: Proceedings of the 16th International Conference on Database Theory*. Saint Petersburg, Russia. 212–224.

Green, T. J., G. Karvounarakis, and V. Tannen. (2007a). “Provenance Semirings”. In: *PODS*. 31–40.

Green, T. J., G. Karvounarakis, N. E. Taylor, O. Biton, Z. G. Ives, and V. Tannen. (2007b). “ORCHESTRA: Facilitating Collaborative Data Sharing”. In: *SIGMOD ’07: Proceedings of the 33th SIGMOD International Conference on Management of Data*.

Grust, T., F. Kliebhan, J. Rittinger, and T. Schreiber. (2011). “True language-level SQL debugging”. In: *Proceedings of the 14th International Conference on Extending Database Technology*. ACM. 562–565.

Gulzar, M. A., M. Interlandi, T. Condie, and M. Kim. (2017a). “Debugging Big Data Analytics in Spark with BigDebug”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM. 1627–1630.

Gulzar, M. A., M. Interlandi, X. Han, M. Li, T. Condie, and M. Kim. (2017b). "Automated debugging in data-intensive scalable computing". In: *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM. 520–534. ISBN: 978-1-4503-5028-0. DOI: 10.1145/3127479.3131624. URL: https://doi.org/10.1145/3127479.3131624.

Guo, P. J., S. Kandel, J. M. Hellerstein, and J. Heer. (2011). "Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts". In: *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM. 65–74.

Gupta, A., I. S. Mumick, *et al.* (1995). "Maintenance of materialized views: Problems, techniques, and applications". *IEEE Data Eng. Bull.* 18(2): 3–18.

Halpern, J. (2000). "Axiomatizing causal reasoning". *Arxiv preprint cs/0005030*.

Halpern, J. and J. Pearl. (2005). "Causes and explanations: A structural-model approach. Part I: Causes". *The British journal for the philosophy of science*. 56(4): 843. ISSN: 0007-0882.

Heinis, T. and G. Alonso. (2008). "Efficient Lineage Tracking for Scientific Workflows". In: *SIGMOD '08: Proceedings of the 34th SIGMOD International Conference on Management of Data*. 1007–1018.

Hellerstein, J. M., V. Sreekanti, J. E. Gonzalez, J. Dalton, A. Dey, S. Nag, K. Ramachandran, S. Arora, A. Bhattacharyya, S. Das, M. Donsky, G. Fierro, C. She, C. Steinbach, V. Subramanian, and E. Sun. (2017). "Ground: A Data Context Service." In: *CIDR*.

Herschel, M. and M. Hernandez. (2010). "Explaining Missing Answers to SPJUA Queries". *PVLDB*. 3(1): 185–196.

Herschel, M. (2013). "Wondering why data are missing from query results?: ask conseil why-not". In: *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*. ACM. 2213–2218.

Herschel, M., R. Diestelkämper, and H. B. Lahmar. (2017). "A survey on provenance: What for? What form? What from?" *The VLDB Journal*: 1–26.

Herschel, M., M. A. Hernández, and W.-C. Tan. (2009). "Artemis: A System for Analyzing Missing Answers". In: *VLDB '09: Proceedings of the 35th International Conference on Very Large Data Bases (demonstration)*. 1550–1553.

Hoekstra, R. and P. Groth. (2014). "PROV-O-Viz - Understanding the Role of Activities in Provenance". In: *Provenance and Annotation of Data and Processes - 5th International Provenance and Annotation Workshop, IPAW 2014, Cologne, Germany, June 9-13, 2014. Revised Selected Papers*. Ed. by B. Ludäscher and B. Plale. Vol. 8628. *Lecture Notes in Computer Science*. Springer. 215–220. ISBN: 978-3-319-16461-8. DOI: 10.1007/978-3-319-16462-5_18. URL: https://doi.org/10.1007/978-3-319-16462-5%5C_18.

Huang, J., T. Chen, A. Doan, and J. F. Naughton. (2008). "On the Provenance of Non-answers to Queries over Extracted Data". *PVLDB: Proceedings of the VLDB Endowment archive*. 1(1): 736–747.

Hull, R. and M. Yoshikawa. (1990). "ILOG: Declarative Creation and Manipulation of Object Identifiers". In: *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*. Morgan Kaufmann. 455–468. ISBN: 1-55860-149-X. URL: http://www.vldb.org/conf/1990/P455.PDF.

Ibrahim, K., X. Du, and M. Eltabakh. (2015). "Proactive Annotation Management in Relational Databases". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2017–2030.

Ikeda, R., H. Park, and J. Widom. (2011a). "Provenance for generalized map and reduce workflows". In: *CIDR*. 273–283.

Ikeda, R., S. Salihoglu, and J. Widom. (2011b). "Provenance-based refresh in data-oriented workflows". In: *Proceedings of the 20th ACM international conference on Information and knowledge management. CIKM '11*. Glasgow, Scotland, UK: ACM. 1659–1668. ISBN: 978-1-4503-0717-8.

Ikeda, R. and J. Widom. (2010). "Panda: A System for Provenance and Data". In: *TaPP '10*. Stanford InfoLab.

Imieliński, T. and W. Lipski Jr. (1984). "Incomplete Information in Relational Databases". *Journal of the ACM (JACM)*. 31(4): 761–791.

Interlandi, M., A. Ekmekji, K. Shah, M. A. Gulzar, S. D. Tetali, M. Kim, T. D. Millstein, and T. Condie. (2018). "Adding data provenance support to Apache Spark". *VLDB J.* 27(5): 595–615. DOI: 10.1007/s00778-017-0474-5. URL: https://doi.org/10.1007/s00778-017-0474-5.

Interlandi, M., K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie. (2016). "Titian: Data Provenance Support in Spark". *PVLDB*. 9(3).

Ives, Z., A. Haeberlen, T. Feng, and W. Gatterbauer. (2012). "Querying provenance for ranking and recommending".

Ives, Z. G., T. J. Green, G. Karvounarakis, N. E. Taylor, V. Tannen, P. P. Talukdar, M. Jacob, and F. Pereira. (2008). "The ORCHESTRA Collaborative Data Sharing System". *SIGMOD Record*. 37(2): 26–32.

Ives, Z. G., N. Khandelwal, A. Kapur, and M. Cakir. (2005). "ORCHESTRA: Rapid, Collaborative Sharing of Dynamic Data". In: *CIDR '05: Proceedings of the 2th Conference on Innovative Data Systems Research*.

Jagadish, H. V., F. Bonchi, T. Eliassi-Rad, L. Getoor, K. P. Gummadi, and J. Stoyanovich. (2019). "The Responsibility Challenge for Data". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska. ACM. 412–414. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3314327. URL: https://doi.org/10.1145/3299869.3314327.

Janin, Y., C. Vincent, and R. Duraffort. (2014). “CARE, the comprehensive archiver for reproducible execution”. In: *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering, TRUST 2014, Edinburgh, United Kingdom, June 9-11, 2014*. Ed. by G. Fursin, B. R. Childers, A. K. Jones, and D. Mossé. ACM. 1:1–1:7. ISBN: 978-1-4503-2951-4. DOI: 10.1145/2618137.2618138. URL: https://doi.org/10.1145/2618137.2618138.

Jensen, C. and R. Snodgrass. (1999). “Temporal Data Management”. *IEEE Transactions on Knowledge and Data Engineering*. 11(1): 36–44.

Joglekar, M., H. Garcia-Molina, and A. G. Parameswaran. (2019). “Interactive Data Exploration With Smart Drill-Down”. *IEEE Trans. Knowl. Data Eng.* 31(1): 46–60. DOI: 10.1109/TKDE.2017.2685998. URL: https://doi.org/10.1109/TKDE.2017.2685998.

Kandel, S., A. Paepcke, J. Hellerstein, and J. Heer. (2011). “Wrangler: Interactive visual specification of data transformation scripts”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 3363–3372.

Karabeg, D. and V. Vianu. (1991). “Simplification Rules and Complete Axiomatization for Relational Update Transactions”. *ACM Trans. Database Syst.* 16(3): 439–475. DOI: 10.1145/111197.111208. URL: https://doi.org/10.1145/111197.111208.

Karvounarakis, G. (2009). “Provenance in collaborative data sharing”. *PhD thesis*. University of Pennsylvania.

Karvounarakis, G. and T. Green. (2012). “Semiring-Annotated Data: Queries and Provenance”. *SIGMOD Record*. 41(3): 5–14.

Karvounarakis, G., Z. Ives, and V. Tannen. (2010). “Querying data provenance”. In: *Proceedings of the 2010 international conference on Management of data*. ACM. 951–962.

Karvounarakis, G., T. J. Green, Z. G. Ives, and V. Tannen. (2013). “Collaborative data sharing via update exchange and provenance”. *ACM Transactions on Database Systems (TODS)*. 38(3): 19.

Kaushik, R., Y. Fu, and R. Ramamurthy. (2013). “On scaling up sensitive data auditing”. In: *Proceedings of the 39th international conference on Very Large Data Bases*. VLDB Endowment. 313–324.

Kaushik, R. and R. Ramamurthy. (2011). "Efficient auditing for complex SQL queries". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. 697–708. DOI: 10.1145/1989323.1989396. URL: https://doi.org/10.1145/1989323.1989396.

Kemp, D. B., D. Srivastava, and P. J. Stuckey. (1995). "Bottom-Up Evaluation and Query Optimization of Well-Founded Models". *Theor. Comput. Sci.* 146(1&2): 145–184. DOI: 10.1016/0304-3975(94)00153-A. URL: https://doi.org/10.1016/0304-3975(94)00153-A.

Koh, P. W. and P. Liang. (2017). "Understanding Black-box Predictions via Influence Functions". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by D. Precup and Y. W. Teh. Vol. 70. *Proceedings of Machine Learning Research*. PMLR. 1885–1894. URL: http://proceedings.mlr.press/v70/koh17a.html.

Köhler, S., B. Ludäscher, and Y. Smaragdakis. (2012). "Declarative datalog debugging for mere mortals". *Datalog in Academia and Industry*: 111–122.

Köhler, S., B. Ludäscher, and D. Zinn. (2013). "First-Order Provenance Games". In: *In Search of Elegance in the Theory and Practice of Computation*. Springer. 382–399.

Kolaitis, P. G. and C. H. Papadimitriou. (1988). "Why Not Negation by Fixpoint?" In: *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 21-23, 1988, Austin, Texas, USA*. Ed. by C. Edmondson-Yurkanan and M. Yannakakis. ACM. 231–239. ISBN: 0-89791-263-2. DOI: 10.1145/308386.308446. URL: https://doi.org/10.1145/308386.308446.

Koop, D. (2016). "Versioning Version Trees: The Provenance of Actions that Affect Multiple Versions". In: *Provenance and Annotation of Data and Processes - 6th International Provenance and Annotation Workshop, IPAW 2016, McLean, VA, USA, June 7-8, 2016, Proceedings*. Ed. by M. Mattoso and B. Glavic. Vol. 9672. *Lecture Notes in Computer Science*. Springer. 109–121. ISBN: 978-3-319-40592-6. DOI: 10.1007/978-3-319-40593-3_9. URL: https://doi.org/10.1007/978-3-319-40593-3%5C_9.

Kostylev, E. V. and P. Buneman. (2012). "Combining dependent annotations for relational algebra". In: *Proceedings of the 15th International Conference on Database Theory*. ACM. 196–207.

Kostylev, E. V., J. L. Reutter, and A. Z. Salamon. (2013). "Classification of Annotation Semirings over Containment of Conjunctive Queries". *TODS*.

Kunde, M., H. Bergmeyer, and A. Schreiber. (2008). "Requirements for a Provenance Visualization Component". In: *Provenance and Annotation of Data and Processes, Second International Provenance and Annotation Workshop, IPAW 2008, Salt Lake City, UT, USA, June 17-18, 2008. Revised Selected Papers*. Ed. by J. Freire, D. Koop, and L. Moreau. Vol. 5272. *Lecture Notes in Computer Science*. Springer. 241–252. ISBN: 978-3-540-89964-8. DOI: 10.1007/978-3-540-89965-5_25. URL: https://doi.org/10.1007/978-3-540-89965-5%5C_25.

Lee, S., S. Köhler, B. Ludäscher, and B. Glavic. (2017). "A SQL-Middleware Unifying Why and Why-Not Provenance for First-Order Queries". In: *Proceedings of the 33rd IEEE International Conference on Data Engineering*. 485–496.

Lee, S., B. Ludäscher, and B. Glavic. (2018). "PUG: a framework and practical implementation for why and why-not provenance". *The VLDB Journal*. 28(1): 47–71. ISSN: 0949-877X. DOI: 10.1007/s00778-018-0518-5.

Lee, S., B. Ludäscher, and B. Glavic. (2020). "Approximate Summaries for Why and Why-not Provenance". *Proceedings of the VLDB Endowment*. 13(6): 912–924.

Logothetis, D., S. De, and K. Yocum. (2013). "Scalable lineage capture for debugging DISC analytics". In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 17.

Luttenberger, M. and M. Schlund. (2013). "Convergence of Newton's Method over Commutative Semirings". In: *Language and Automata Theory and Applications - 7th International Conference, LATA 2013, Bilbao, Spain, April 2-5, 2013. Proceedings*. 407–418. DOI: 10.1007/978-3-642-37064-9_36. URL: https://doi.org/10.1007/978-3-642-37064-9%5C_36.

Luttenberger, M. and M. Schlund. (2014). “Regular Expressions for Provenance”. In: *TaPP*.

Meliou, A., W. Gatterbauer, K. Moore, and D. Suciu. (2010). “The Complexity of Causality and Responsibility for Query Answers and non-Answers”. *Proceedings of the VLDB Endowment*. 4(1): 34–45.

Meliou, A. and D. Suciu. (2012). “Tiresias: The database oracle for how-to queries”. In: *Proceedings of the 2012 international conference on Management of Data*. ACM. 337–348.

Meliou, A., W. Gatterbauer, S. Nath, and D. Suciu. (2011). “Tracing data errors with view-conditioned causality”. In: *SIGMOD Conference*. Ed. by T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis. ACM. 505–516. ISBN: 978-1-4503-0661-4.

Mohri, M. (2002). “Semiring Frameworks and Algorithms for Shortest-Distance Problems”. *J. Autom. Lang. Comb.* 7(3): 321–350.

Moreau, L., B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowskag, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. den Bussche. (2011). “The open provenance model core specification (v1. 1)”. *Future Generation Computer Systems*. 27(6): 743–756.

Moreau, L., J. Freire, J. Futrelle, R. E. McGrath, J. Myers, and P. Paulson. (2008). “The Open Provenance Model: An Overview”. In: *IPAW ’08: International Provenance and Annotation Workshop*. 323–326.

Moreau, L., J. Freire, J. Myers, J. Futrelle, and P. Paulson. (2007). “The Open Provenance Model”.

Moreau, L. and P. Groth. (2013). “Provenance: An introduction to prov”. *Synthesis Lectures on the Semantic Web: Theory and Technology*. 3(4): 1–129.

Moreau, L. and P. Missier. (2013a). *http://www.w3.org/TR/prov-overview/*.

Moreau, L. and P. Missier. (2013b). “Prov-dm: The prov data model”. *http://www.w3.org/TR/2013/REC-prov-dm-20130430/*.

Moura, L. M. de and N. Bjørner. (2011). “Satisfiability Modulo Theories: Introduction and Applications”. *Commun. ACM*. 54(9): 69–77. DOI: 10.1145/1995376.1995394. URL: https://doi.org/10.1145/1995376.1995394.

Müller, T., B. Dietrich, and T. Grust. (2018). "You say what, i hear where and why: (mis-)interpreting SQL to derive fine-grained provenance". *Proceedings of the VLDB Endowment*. 11(11): 1536–1549.

Müller, T. and T. Grust. (2015). "Provenance for SQL through Abstract Interpretation: Value-less, but Worthwhile". *Proceedings of the VLDB Endowment*. 8(12).

Nguyen, D., J. Park, and R. Sandhu. (2013). "A provenance-based access control model for dynamic separation of duties". In: *Privacy, Security and Trust (PST), 2013 Eleventh Annual International Conference on*. IEEE. 247–256.

Niu, F., C. Zhang, C. Ré, and J. W. Shavlik. (2012). "DeepDive: Web-scale Knowledge-base Construction using Statistical Learning and Inference". In: *Proceedings of the Second International Workshop on Searching and Integrating New Web Data Sources, Istanbul, Turkey, August 31, 2012*. 25–28. URL: http://ceur-ws.org/Vol-884/VLDS2012%5C_p25%5C_Niu.pdf.

Niu, X., B. Glavic, S. Lee, B. Arab, D. Gawlick, Z. H. Liu, V. Krishnaswamy, S. Feng, and X. Zou. (2017a). "Debugging Transactions and Tracking their Provenance with Reenactment". *Proceedings of the VLDB Endowment (Demonstration Track)*. 10(12): 1857–1860.

Niu, X., R. Kapoor, B. Glavic, D. Gawlick, Z. H. Liu, V. Krishnaswamy, and V. Radhakrishnan. (2017b). "Provenance-aware Query Optimization". In: *Proceedings of the 33rd IEEE International Conference on Data Engineering*. 473–484.

Niu, X., R. Kapoor, B. Glavic, D. Gawlick, Z. H. Liu, V. Krishnaswamy, and V. Radhakrishnan. (2018). "Heuristic and Cost-based Optimization for Diverse Provenance Tasks". *IEEE Transactions on Knowledge and Data Engineering*. DOI: 10.1109/TKDE.2018.2827074.

Olston, C. and B. Reed. (2011). "Inspector Gadget: A Framework for Custom Monitoring and Debugging of Distributed Dataflows". *PVLDB*. 4(12): 1237–1248. URL: http://www.vldb.org/pvldb/vol4/p1237-olston.pdf.

Olteanu, D. and M. Schleich. (2016). "Factorized Databases". *ACM SIGMOD Record*. 45(2): 5–16.

Olteanu, D. and J. Závodny. (2011). "On Factorisation of Provenance Polynomials". In: *TaPP '11: 3rd USENIX Workshop on the Theory and Practice of Provenance.*

Olteanu, D. and J. Závodný. (2015). "Size Bounds for Factorised Representations of Query Results". *ACM Transactions on Database Systems (TODS).* 40(1): 2.

Park, J., D. Nguyen, and R. Sandhu. (2012). "A provenance-based access control model". In: *Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on.* IEEE. 137–144.

Pasquier, T. F. J., X. Han, M. Goldstein, T. Moyer, D. M. Eyers, M. I. Seltzer, and J. Bacon. (2017). "Practical whole-system provenance capture". In: *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017.* ACM. 405–418. ISBN: 978-1-4503-5028-0. DOI: 10.1145/3127479.3129249. URL: https://doi.org/10.1145/3127479.3129249.

Pearl, J. (2000). *Causality: models, reasoning, and inference.* Cambridge Univ Pr. ISBN: 0521773628.

Pereira, L. M. (1986). "Rational Debugging in Logic Programming". In: *Third International Conference on Logic Programming, Imperial College of Science and Technology, London, United Kingdom, July 14-18, 1986, Proceedings.* 203–210. DOI: 10.1007/3-540-16492-8_76. URL: https://doi.org/10.1007/3-540-16492-8%5C_76.

Perera, R., U. Acar, J. Cheney, and P. Levy. (2012). "Functional programs that explain their work". In: *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming.* ACM. 365–376.

Pham, Q., T. Malik, and I. Foster. (2013). "Using provenance for repeatability". In: *Proceedings of the 5th USENIX conference on Theory and Practice of Provenance.* 2–2.

Pham, Q., T. Malik, B. Glavic, and I. Foster. (2015). "LDV: Lightweight Database Virtualization". In: *Proceedings of the 31st IEEE International Conference on Data Engineering.* 1179–1190.

Pimentel, J. F., L. Murta, V. Braganholo, and J. Freire. (2019). "A large-scale study about quality and reproducibility of jupyter notebooks". In: *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*. Ed. by M. D. Storey, B. Adams, and S. Haiduc. IEEE. 507–517. ISBN: 978-1-7281-3412-3. DOI: 10.1109/MSR.2019.00077. URL: https://doi.org/10.1109/MSR.2019.00077.

Psallidas, F. and E. Wu. (2018a). "Smoke: Fine-Grained Lineage At Interactive Speed". *Proc. VLDB Endow.* 11(6): 719–732. DOI: 10.14778/3184470.3184475. URL: https://doi.org/10.14778/3184470.3184475.

Psallidas, F. and E. Wu. (2018b). "Smoke: Fine-grained lineage at interactive speed". *Proceedings of the VLDB Endowment*. 11(6): 719–732.

Ramusat, Y., S. Maniu, and P. Senellart. (2018). "Semiring Provenance over Graph Databases". In: *10th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2018, London, UK, July 11-12, 2018*. Ed. by M. Herschel. USENIX Association. URL: https://www.usenix.org/conference/tapp2018/presentation/ramusat.

Rio, N. D. and P. P. da Silva. (2007). "Probe-It! Visualization Support for Provenance". In: *Advances in Visual Computing, Third International Symposium, ISVC 2007, Lake Tahoe, NV, USA, November 26-28, 2007, Proceedings, Part II*. Ed. by G. Bebis, R. D. Boyle, B. Parvin, D. Koracin, N. Paragios, T. F. Syeda-Mahmood, T. Ju, Z. Liu, S. Coquillart, C. Cruz-Neira, T. Müller, and T. Malzbender. Vol. 4842. *Lecture Notes in Computer Science*. Springer. 732–741. ISBN: 978-3-540-76855-5. DOI: 10.1007/978-3-540-76856-2_72. URL: https://doi.org/10.1007/978-3-540-76856-2%5C_72.

Roy, S. and D. Suciu. (2014). "A formal approach to finding explanations for database queries". In: *SIGMOD*.

Salimi, B., L. Bertossi, D. Suciu, and G. V. den Broeck. (2016). "Quantifying Causal Effects on Query Answering in Databases". In: *8th USENIX Workshop on the Theory and Practice of Provenance (TaPP 16)*. Washington, D.C.: USENIX Association.

Salimi, B., J. Gehrke, and D. Suciu. (2018). “Bias in OLAP Queries: Detection, Explanation, and Removal”. In: *Proceedings of the 2018 International Conference on Management of Data*. ACM. 1021–1035.

Scheidegger, C. E., H. Vo, D. Koop, J. Freire, and C. T. Silva. (2008). “Querying and Re-using Workflows with VisTrails”. In: *SIGMOD ’08: Proceedings of the 34th SIGMOD International Conference on Management of Data*. ACM. 1251–1254.

Schwartz, E. J., T. Avgerinos, and D. Brumley. (2010). “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”. In: *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society. 317–331. ISBN: 978-0-7695-4035-1. DOI: 10.1109/SP.2010.26. URL: https://doi.org/10.1109/SP.2010.26.

Senellart, P. (2017). “Provenance and Probabilities in Relational Databases: From Theory to Practice”. *SIGMOD record*.

Senellart, P., L. Jachiet, S. Maniu, and Y. Ramusat. (2018). “ProvSQL: provenance and probability management in postgreSQL”. *Proceedings of the VLDB Endowment*. 11(12): 2034–2037.

Shu, H. (2000). “Using constraint satisfaction for view update”. *Journal of Intelligent Information Systems*. 15(2): 147–173. ISSN: 0925-9902.

Stonebraker, M., J. Chen, N. Nathan, C. Paxson, and J. Wu. (1993). “Tioga: Providing Data Management Support for Scientific Visualization Applications”. In: *VLDB*. 25–38.

Suciu, D. (2020). “Probabilistic Databases for All”. In: *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*. Ed. by D. Suciu, Y. Tao, and Z. Wei. ACM. 19–31. ISBN: 978-1-4503-7108-7. DOI: 10.1145/3375395.3389129. URL: https://doi.org/10.1145/3375395.3389129.

Suciu, D., D. Olteanu, C. Ré, and C. Koch. (2011). “Probabilistic databases”. *Synthesis Lectures on Data Management*. 3(2): 1–180.

Suriarachchi, I., Q. Zhou, and B. Plale. (2015). “Komadu: A Capture and Visualization System for Scientific Data Provenance”. *Journal of Open Research Software*. 3(1).

Tannen, V. (2017). “Provenance analysis for FOL model checking”. *ACM SIGLOG News*. 4(1): 24–36.

That, D. H. T., G. Fils, Z. Yuan, and T. Malik. (2017). “Sciunits: Reusable Research Objects”. In: *13th IEEE International Conference on e-Science, e-Science 2017, Auckland, New Zealand, October 24-27, 2017*. IEEE Computer Society. 374–383. ISBN: 978-1-5386-2686-3. DOI: 10.1109/eScience.2017.51. URL: https://doi.org/10.1109/eScience.2017.51.

Tran, Q. T. and C.-Y. Chan. (2010). “How to ConQueR why-not questions”. In: *SIGMOD ’10: Proceedings of the 2010 international conference on Management of data*. Indianapolis, Indiana, USA: ACM. 15–26. ISBN: 978-1-4503-0032-2.

Van den Broeck, G. and D. Suciu. (2017). “Query Processing on Probabilistic Data: A Survey”.

Van Gelder, A., K. A. Ross, and J. S. Schlipf. (1991). “The well-founded semantics for general logic programs”. *Journal of the ACM (JACM)*. 38(3): 619–649.

Vansummeren, S. and J. Cheney. (2007). “Recording Provenance for SQL Queries and Updates”. *IEEE Data Engineering Bulletin*. 30(4): 29–37.

Vollmer, M., L. Golab, K. Böhm, and D. Srivastava. (2019). “Informative Summarization of Numeric Data”. In: *Proceedings of the 31st International Conference on Scientific and Statistical Database Management, SSDBM 2019, Santa Cruz, CA, USA, July 23-25, 2019*. Ed. by C. Maltzahn and T. Malik. ACM. 97–108. ISBN: 978-1-4503-6216-0. DOI: 10.1145/3335783.3335797. URL: https://doi.org/10.1145/3335783.3335797.

Wang, Q., T. Yu, N. Li, J. Lobo, E. Bertino, K. Irwin, and J. Byun. (2007). “On the Correctness Criteria of Fine-Grained Access Control in Relational Databases”. In: *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. Ed. by C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C. Kanne, W. Klas, and E. J. Neuhold. ACM. 555–566. ISBN: 978-1-59593-649-3. URL: http://www.vldb.org/conf/2007/papers/research/p555-wang.pdf.

Wang, X., X. L. Dong, and A. Meliou. (2015). "Data X-Ray: A Diagnostic Tool for Data Errors". In: Sigmod.

Weiser, M. (1981). "Program slicing". *Proceedings of the 5th international conference on Software engineering*: 439–449.

Wu, E., S. Madden, and M. Stonebraker. (2012). "SubZero: A Fine-Grained Lineage System for Scientific Databases".

Wu, E. and S. Madden. (2013). "Scorpion: Explaining Away Outliers in Aggregate Queries". *PVLDB*. 6(8): 553–564.

Wu, Y., A. Haeberlen, W. Zhou, and B. T. Loo. (2013). "Answering why-not queries in software-defined networks with negative provenance". In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM. 3.

Wu, Y., M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. (2014). "Diagnosing missing events in distributed systems with negative provenance". In: *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM. 383–394.

Wu, Y., V. Tannen, and S. B. Davidson. (2020). "PrIU: A Provenance-Based Approach for Incrementally Updating Regression Models". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo. ACM. 447–462. ISBN: 978-1-4503-6735-6. DOI: 10.1145/3318464.3380571. URL: https://doi.org/10.1145/3318464.3380571.

Wylot, M., P. Cudré-Mauroux, and P. Groth. (2014). "TripleProv: Efficient Processing of Lineage Queries in a Native RDF Store". In: *Proceedings of the 24th international conference on World Wide Web (WWW)*.

Xu, J., W. Zhang, A. Alawini, and V. Tannen. (2018). "Provenance Analysis for Missing Answers and Integrity Repairs". *Data Engineering*: 39.

Yan, Z., V. Tannen, and Z. G. Ives. (2016). "Fine-grained Provenance for Linear Algebra Operators". In: *8th USENIX Workshop on the Theory and Practice of Provenance (TaPP 16)*. Washington, D.C.: USENIX Association.

Yang, Y., N. Meneghetti, R. Fehling, Z. H. Liu, and O. Kennedy. (2015). "Lenses: an on-demand approach to ETL". *Proceedings of the VLDB Endowment*. 8(12): 1578–1589.

Zaharia, M., M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. (2010). "Spark: cluster computing with working sets". In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. 10–10.

Zhang, J. and H. Jagadish. (2010). "Lost source provenance". In: *Proceedings of the 13th International Conference on Extending Database Technology*. ACM. 311–322.

Zhang, Z., E. R. Sparks, and M. J. Franklin. (2017). "Diagnosing machine learning pipelines with fine-grained lineage". In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM. 143–153.

Zheng, N., A. Alawini, and Z. G. Ives. (2019). "Fine-Grained Provenance for Matching & ETL". In: *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE. 184–195. ISBN: 978-1-5386-7474-1. DOI: 10.1109/ICDE.2019.00025. URL: https://doi.org/10.1109/ICDE.2019.00025.

Index