14 Data Provenance

The majority of this book has focused on how to take data from a plethora of sources and formats and integrate it into a single homogeneous view — such that it becomes indistinguishable from data with other origins. Sometimes, however, we would still like to be able to take a tuple from an integrated schema and determine *where it came from* and how it came to be.

This motivates a topic of study called *data provenance*, or sometimes *data lineage* or *data pedigree*. A data item's provenance is a record of how it came to be. In the broadest sense, this provenance may include a huge number of factors, e.g., who created the initial data, when they created them, or what equipment they used. Typically, however, in the database community we model provenance of derived data strictly in terms of how it was derived from the original base tuples in the original source databases. We leave it to individual applications to keep records of any additional information appropriate for the problem domain, like how and when the original base tuples were created, by whom, and so forth.

Example 14.1

Consider a situation where a scientific data warehouse imports data from multiple sources S_1, S_2. If a data conflict arises, the users and administrators of the data warehouse would like to know which source provided which data, and also which mappings were used to convert the data. If one source is known to be more authoritative than the other, then perhaps the conflict can be resolved simply by knowing the sources. Likewise, if a schema mapping is known to be error-prone, it would be useful to identify which tuples it produced.

We start in Section 14.1 by describing two different viewpoints on what provenance is: a set of annotations describing where data came from, or a set of relationships among data. Each viewpoint is very natural in certain settings. Section 14.2 provides an overview of the many applications of data provenance. Section 14.3 discusses a formal model for data provenance, that of *provenance semirings*, which captures the full detail of select-project-join-union queries. We briefly discuss how provenance can be stored in a database system in Section 14.4.

Principles of Data Integration, DOI: 10.1016/B978-0-12-416044-6.00014-4

14.1 The Two Views of Provenance

Provenance, as the means of describing how data are derived, can be thought of in two different, complementary ways. Typically one thinks of provenance in the first representation when reasoning about the existence or value of particular data items, whereas one thinks of provenance in the second representation when *modeling* it with respect to an entire database. Both views are equivalent, and one can convert from one to the other as convenient. Most systems that manipulate data provenance use both representations.

14.1.1 Provenance as Annotations on Data

Perhaps the most natural way to conceptualize provenance is as a series of annotations describing how each data item was produced. These annotations can be placed on different portions of the data. In the relational model, this means tuples or fields within a tuple may be annotated. For instance, the provenance of tuple t might be the set of tuples t_1, t_2, t_3 that were joined together to produce t. In a more complex model such as that of XML, provenance annotations might appear on (sub)trees. Such annotations could describe the tuple in terms of the transformations (mappings and/or query operations) and sources of the data.

Example 14.2

Refer to Figure 14.1 for a sample database instance, with two base data instances (tables R and S). Here we have a view V_1 computed using the relational algebra expression $R \bowtie S \cup S \bowtie S$. We can annotate each tuple in V_1 with an "explanation" of how it was produced; see the "directly derivable by" column in the figure. For instance, $V_1(1,3)$ is derivable in two alternative ways, where the first is the join of $R(1,2)$ with $S(2,3)$ and the second is the join of $R(1,4)$ with $S(4,3)$.

Relation R

A	B
1	2
1	4

Relation S

B	C
2	3
3	2
4	3

View $V_1 := R \bowtie S \cup S \bowtie S$

A	C	*directly derivable by...*
1	3	$R(1,2) \bowtie S(2,3) \cup R(1,4) \bowtie S(4,3)$
2	2	$S(2,3) \bowtie \rho_{B \to A, C \to B}\, S(3,2)$
3	3	$S(3,2) \bowtie \rho_{B \to A, C \to B}\, S(2,3)$

FIGURE 14.1 Two example base relation instances and a view instance (with definition shown in relational algebra).

This view of provenance is extremely convenient when one wishes to use the provenance to assign some sort of score or confidence assessment to a tuple. In essence, the provenance annotation gets treated as an expression whose value is the score.

As we shall see shortly, the notion of annotations can even be generalized to support recursive dependencies, e.g., due to a recursive view or cyclic set of schema mappings.

14.1.2 Provenance as a Graph of Data Relationships

There are other situations where it is more natural to think about provenance as a means of connecting source and derived tuples via particular mappings: in other words, to think of provenance as a sort of graph. Such a graph is a very natural way to visualize provenance and in fact also to store provenance information.

More specifically, we model provenance as a hypergraph, with tuples as vertices. Each direct derivation of a tuple from a set of source tuples is a *hyperedge* connecting the source and derived tuples. In datalog parlance, a direct derivation of one tuple from a set of source tuples is called an *immediate consequent* of a particular rule.

Example 14.3

Refer again to Figure 14.1. We can restate the relational algebra expression for V_1 in datalog:

$$V_1(x,z) \text{:-} \quad R(x,y), S(y,z)$$
$$V_1(x,x) \text{:-} \quad S(x,y), S(y,x)$$

Each tuple in V_1's view instance is derivable as an immediate consequent from other tuples, i.e., it is directly derived by applying the datalog rule to those tuples. We see in the figure that the first tuple of V_1 can actually be derived by joining two different combinations of tuples from R and S. Meanwhile, the second and third tuples in V_1's instance are derived simply by doing a self-join on S. We can visualize the relationships between the tuples using the hypergraph of Figure 14.2.

In this example, the name of the derivation (V_1) and the relation for the resulting tuple ($V_1(a,b)$) are the same. In more general data integration settings, however, we may have multiple mappings or rules expressing constraints between source and target instances. Here,

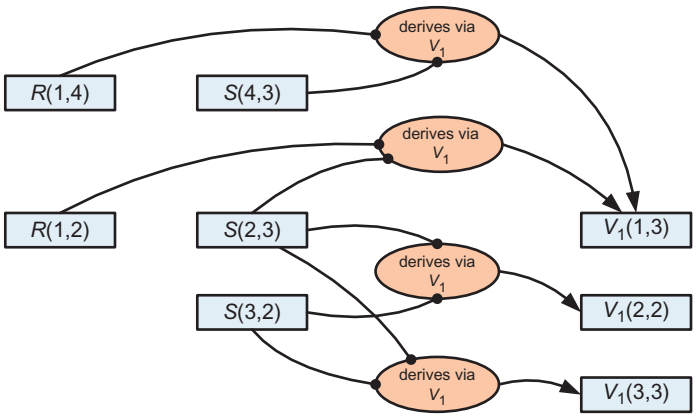

FIGURE 14.2 Example hypergraph of relationships between tuples in a database (and view) instance, where tuples are indicated by rectangular nodes and derivation hyperedges are represented by ellipses with incoming and outgoing arcs.

the rules or mappings may each receive a unique name, which is distinct from the output relation name.

14.1.3 Interchangeability of the Two Views

It is worth noting that any provenance graph can be equivalently represented using annotations on tuples, with a slight generalization of the form discussed in the previous section. In particular, if we want to support recursive derivations as in the graph, then we assign to each tuple a corresponding *provenance token*. For base tuples, this is a *base token* representing a known base value. For all other cases, we use a *result token* whose (derived) value is computed through a *provenance expression* over other provenance tokens. The set of all provenance tokens then forms a system of equations, whose solution may have an infinite expansion in the presence of recursion. Fortunately, as we shall discuss in Section 14.3, if we assign the right semantics to the mathematical structure in which the equations are represented, then we can manipulate and reason about the annotations in a tractable way.

Example 14.4

We illustrate the intuition of the system of equations in Figure 14.3. This is Figure 14.1 redrawn so all tuples are annotated with provenance tokens, which represent either base values or expressions over other provenance tokens.

Relation R

A	B	*annotation*
1	2	r_1
1	4	r_2

Relation S

B	C	*annotation*
2	3	s_1
3	2	s_2
4	3	s_3

View V_1

A	C	*annotation*
1	3	$v_1 = r_1 \bowtie s_1 \cup r_2 \bowtie s_3$
2	2	$v_2 = s_1 \bowtie s_2$
3	3	$v_3 = s_2 \bowtie s_1$

FIGURE 14.3 Figure 14.1 with provenance as a system of annotation equations over provenance tokens. Each tuple is annotated with a token, whose value is an expression over other tokens. We use join and union informally here and provide a formal framework in Section 14.3.

Under this model, each product term, which is now expressed in terms of the tuples from which the tuple was directly derived, corresponds exactly to a hyperedge in the graph.

14.2 Applications of Data Provenance

Provenance, as a means of explaining answers, can be useful in many contexts. Motivations for storing and examining provenance can generally be split into three classes:

explaining a data item, scoring a data item, or reasoning about interactions among tuples.

EXPLANATIONS
There are a variety of contexts where an end user or data integrator may wish to see an "explanation" of a tuple in the output. Perhaps the tuple looks erroneous: we may need to see which mapping produced it in order to debug the mapping or clean the source. We may want an audit trail showing how the data came to be, including the operations applied. Alternatively, we may wish to see an explanation of where the data came from and how the tuple was produced in order to better assess the evidence in support of the answer. In such settings, our goal is probably to see a diagram, e.g., a directed graph, representing the dataflow and operations that produced a tuple.

SCORING
In a variety of cases it is possible to assign a base score to raw data, to mappings, and to computations like joins or unions. The base score can represent a confidence level, author-authoritativeness score, likelihood of landing on a node in a random walk, or similarity metric. In these cases, given provenance and a scheme for combining scores, we can automatically derive a score for the derived data values, in the form of an annotation. For instance, we can automatically derive probabilistic event expressions (describing the conditions under which a joint event is true) or negative log likelihood scores (describing the logarithm of the likelihood a joint event is true) from provenance information. A closely related kind of score might be an access level: if source data are known to have a particular set of access privileges associated with them, we may be able to automatically determine that derived data should have *at least* the same amount of protection.

REASONING ABOUT INTERACTIONS
Sometimes there is a need not merely to look at the provenance of a tuple, but to understand the relationship between different tuples. For instance, we might want to know whether two different tuples are independently derivable, whether one tuple is dependent on the presence of another, or whether two tuples share a mapping. Here we may want to see a set of data items for which the relationship holds, or possibly a graphical representation of the connections between the data items.

14.3 Provenance Semirings

Since the early 2000s, many different formulations of data provenance have been developed, each with the ability to capture certain levels of detail in explaining a tuple. The culmination of this work was the provenance semirings formalism, which is the most general representation of the relationships among tuples derived using the core relational algebra (select, project, join, union). Provenance semirings also provide two very useful

properties: (1) different, algebraically equivalent relational expressions[1] produce algebraically equivalent provenance; (2) one can use the same formalism to compute a wide variety of score types to a derived result in a materialized view simply based on its provenance annotation, i.e., without recomputing the result. Recent research has extended the provenance semiring model to richer query languages (e.g., with support for aggregation), but we focus on the core relational algebra here.

14.3.1 The Semiring Formal Model

If one looks at a tuple produced by a query in the relational algebra, there are two basic ways tuples get combined: through join or Cartesian product, resulting in a "joint" tuple, and via duplicate elimination in a projection or union operation, resulting in a tuple that was derived in more than one "alternative" way. Moreover, there are certain equivalences that hold over these operations, thanks to the relational algebra. For instance, union is commutative, and join distributes through union. The semiring model is designed to capture these equivalences, such that algebraically equivalent query expressions produce equivalent provenance annotations. Put in more dramatic terms, the semiring captures cases in which the query optimizer can choose different plans for evaluating a query, without affecting the provenance.

The basic provenance semiring formalism consists of the following:

- A set of *provenance tokens* or tuple identifiers K, which uniquely identify the relation and value for each tuple. Consider, for example, a relation ID and key as a means of obtaining a provenance token. Tokens are divided into base tokens, representing base values, and result tokens, representing the results of provenance expressions over other tokens.
- An abstract *sum* operator, $\oplus$, which is associative and commutative, and with an identity element $\mathbf{0}$ ($a \oplus \mathbf{0} \equiv a \equiv \mathbf{0} \oplus a$).
- An abstract *product* operator, $\otimes$, which is associative and commutative, which distributes through the sum operator, and for which there is an identity element $\mathbf{1}$ such that $a \otimes \mathbf{1} \equiv \mathbf{1} \otimes a \equiv a$. Moreover, $a \otimes \mathbf{0} \equiv \mathbf{0} \otimes a \equiv 0$.

More formally, $(K, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ form a *commutative semiring* and $(K, \oplus, \mathbf{0})$ and $(K, \otimes, \mathbf{1})$ form *commutative monoids*, where $\otimes$ is distributive over $\oplus$.

Intuitively, if each element of K is a tuple ID, then $\oplus$ captures settings where the same result is derived by unioning or projecting from multiple expressions. The $\otimes$ operator corresponds to joining the results of multiple expressions or applying a given mapping rule.

[1] Here, we refer to equivalent with the "standard" relational algebraic transformations used in an optimizer, hence the set of query plans that might be produced for any query. See the bibliographic notes for details on some subtleties.

Example 14.5

Suppose we assign a provenance token $X_{y,z}$ to each tuple of the form $X(y,z)$ in Figure 14.1. For instance, tuple $R(1,2)$ receives token $R_{1,2}$. Then the provenance of tuples in view V_1 would be as follows:

A	C
1	3
2	2
3	3

provenance expression

$V_{1,3} = R_{1,2} \otimes S_{2,3} \oplus R_{1,4} \otimes S_{4,3}$

$V_{2,2} = S_{2,3} \otimes S_{3,2}$

$V_{3,3} = S_{3,2} \otimes S_{2,3}$

EXTENSION: TOKENS FOR MAPPINGS

It is sometimes convenient to also assign a provenance token to each mapping or rule (or even *version* of a mapping or rule definition) in a derivation. This enables us to track not only the data items but also the mappings used to create a tuple. We might ultimately want to assign a score to each mapping, e.g., based on its likelihood of correctness, and to include the mapping's score in our computation of an annotation. Chapter 13 discusses how we can use *by-table* semantics to assign probabilistic scores to schema mappings.

Example 14.6

Let us expand Example 14.5 to a scenario where we are uncertain about the quality of view V_1. We assign a token v_1 to the view V_1, representing our confidence in the quality of the view's output. Then the provenance of tuples in view V_1 would be as follows:

A	C
1	3
2	2
3	3

provenance expression

$V_{1,3} = v_1 \otimes [R_{1,2} \otimes S_{2,3} \oplus R_{1,4} \otimes S_{4,3}]$

$V_{2,2} = v_1 \otimes [S_{2,3} \otimes S_{3,2}]$

$V_{3,3} = v_1 \otimes [S_{3,2} \otimes S_{2,3}]$

Note from the example that we define the provenance of a tuple as a polynomial expression over the provenance tokens of other tuples. This model is general enough to capture complex provenance relationships, including recursive relationships. There is a direct translation to and from the hypergraph representation of Figure 14.1, as well: each provenance token for a tuple is encoded as a tuple node in the hypergraph; each provenance token for a mapping becomes a hyperedge, connecting from the tuple nodes for the tokens being multiplied and connecting to the tuple node(s) for the resulting (output) provenance tokens.

14.3.2 Applications of the Semiring Model

One application of data provenance is simply to help the end user visualize how a tuple was derived, e.g., to perform debugging or to better understand the data. Figure 14.4 shows an example of this, with the provenance visualizer used in the ORCHESTRA system. The visualizer uses rectangles to represent tuples within relations. Derivations are indicated by diamonds, such as the derivation through mapping M5 (labeled as such). Additionally, insertions of source data are indicated with "+" nodes.

The true power of the provenance semiring formalism occurs in its ability to assign scores to tuples by giving an interpretation to the $\oplus$, $\otimes$ operators and a score to the base provenance tokens. By performing an evaluation over the provenance representation, we can automatically derive the score. Moreover, the same stored provenance representation can be directly used in all of the applications — we do not need a different stored representation for, say, visualizing provenance versus assigning scores or counts to tuples.

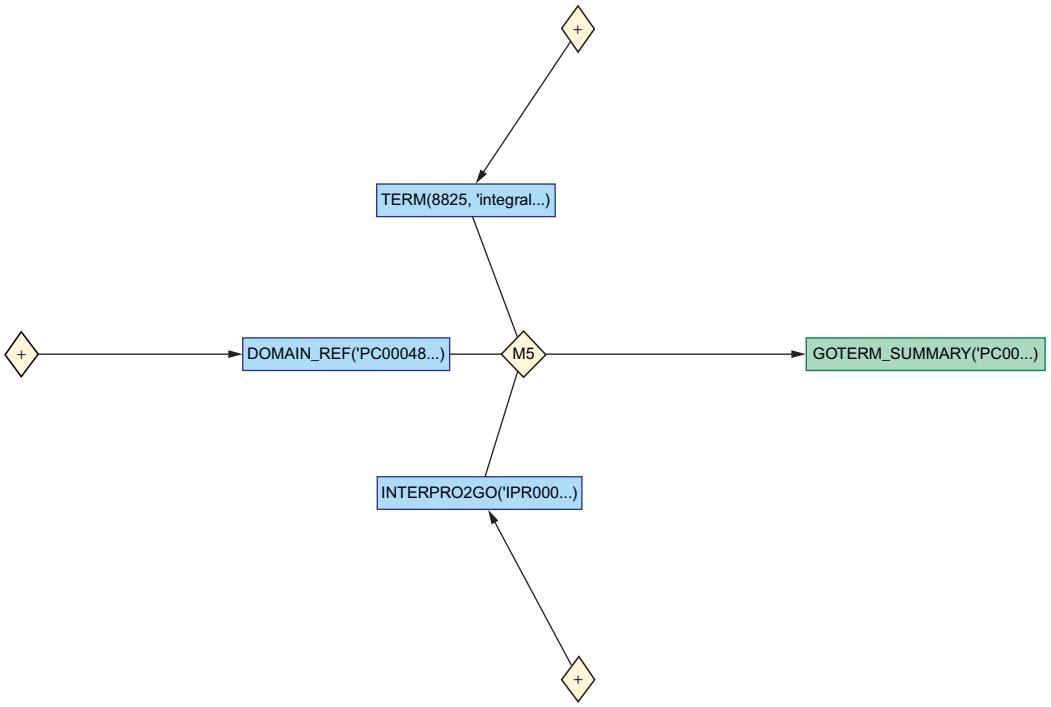

FIGURE 14.4 Example of provenance visualization as a hypergraph, from the ORCHESTRA system. "+" nodes represent insertions of source data. Rectangles represent tuples within relations, and diamonds, such as the one labeled "M5," represent derivations through schema mappings.

Example 14.7

Suppose we have a simple scoring model in which the score of a tuple is computed based on the number of times it is derived. Let the value of each base tuple $R_{x,y}$ or $S_{y,z}$ be the integer 1, let $\otimes$ be arithmetic multiplication, and let $\oplus$ be arithmetic sum. Assign the **1** element to be the integer value 1 and the **0** element to be the integer value 0. Finally, let the value of v_1 be the integer value 1.

Then we get the following for V_1:

A	C	*evaluation of provenance expression*
1	3	$V_{1,3} = 1 \cdot 1 \cdot 1 + 1 \cdot 1 \cdot 1 = 2$
2	2	$V_{2,2} = 1 \cdot 1 \cdot 1 = 1$
3	3	$V_{3,3} = 1 \cdot 1 \cdot 1 = 1$

Example 14.8

Suppose we have a slightly more complex scoring model that is commonly used in information retrieval: each tuple receives a score that is its *negative log likelihood*, i.e., the negation of the log of its probability. Assuming independence of all tuples, the negative log likelihood of a join result is the sum of the negative log likelihoods of the expressions being joined. When a tuple has two different derivations, we will pick the most likely of the cases (i.e., the smallest of the negative log likelihoods).[2]

We can compute negative log likelihood annotations for each tuple as follows. Let the value of each base tuple $R_{x,y}$ or $S_{y,z}$ be the negative log likelihood of the tuple; let $\otimes$ be the arithmetic sum, and let $\oplus$ be the the operator *min*. Assign the **1** element to be the real value 1 and the **0** element to be the real value 0. Finally, let the value of v_1 be the negative log likelihood of the view being correct. As we combine these components, we will receive a single score. In many cases, we may wish to return only the *top k-scoring* tuples in the view (See Section 16.2 for more details).

Beyond these simple examples, there are a wide variety of commutative semirings that have been shown to be useful. See Table 14.1 for a list of the operators and base value assignments. The derivability semiring assigns true to all base tuples and determines whether a tuple (whose annotation must also be true) can be derived from them. Trust is very similar, except that we must check each base (EDB) tuple to see whether it is trusted, annotating it with true or false. Moreover, each mapping may be associated with the

[2]Note that this does not compute the probabilistically correct answer. However, it is a tractable approximation often used in machine learning, since computing the true probabilities is a #P-complete problem.

Table 14.1 Some Useful Assignments of Base Values and Operations in Evaluating Provenance Expressions to Compute Scores

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

The term "lineage" actually has three distinct meanings, as described in the bibliographic notes. Here we refer to the original definition.

multiplicative identity value true, not affecting the value of the other terms with which it is multiplied; or may be associated with the untrusted value false, returning false when multiplied with any term. Any derived tuples with annotation true are trusted. The confidentiality level semiring assigns a confidentiality access level to a tuple derived by joining multiple source tuples. For any join, it assigns the highest (most secure) level of any input tuple to the result; for any union, it assigns the lowest (least secure) level required. The weight/cost semiring, used in the previous example, is useful in ranked models where output tuples are given a cost, evaluating to the sums of the individual scores or weights of atoms joined (and to the lowest cost of different alternatives in a union). This semiring can be used to produce ranked results in a keyword search or to assess data quality. The probability semiring represents probabilistic event expressions that can be used for query answering in probabilistic databases.[3] The lineage semiring corresponds to the set of all base tuples contributing to some derivation of a tuple. The number of derivations semiring, used in our first example, counts the number of ways each tuple is derived, as in the bag relational model.

Semiring provenance has also been extended to support nested data, such as XML (see the bibliographic notes).

14.4 Storing Provenance

The hypergraph representation of provenance semirings can be very naturally encoded using relations. Suppose we have a set of relational tuples and want to also encode their provenance. Each tuple node in the graph can be encoded, of course, by a tuple in a table within the database. Each type of provenance hyperedge (conjunctive rule within a view)

[3] Computing actual probabilities from these event expressions is in general a #P-complete problem; the provenance semiring model does not change this.

has its own schema, comprising a set of typed input and output tuple IDs that are foreign keys to tuples in the DBMS; each instance of this hyperedge type can be stored as a tuple within the database.

Example 14.9

For the running example in this chapter we can create two provenance tables, $P_{V1-1}(R.A, R.B, S.B, S.C, V1.A, V1.C)$ and $P_{V1-2}(S.B, S.C, S.B', S.C', V1.A, V1.C)$, representing the first and second conjunctive queries within V_1, with the following content to describe the specific hyperedges comprising derivations through V_1

P_{V1-1}

R.A	R.B	S.B	S.C	V1.A	V1.C
1	2	2	3	1	3
1	4	4	3	1	3

P_{V1-2}

S.B	S.C	S.B′	S.C′	V1.A	V1.C
2	3	3	2	2	2
3	2	2	3	3	3

One can in fact further optimize this storage scheme by observing that in many cases, the various relations in a derivation have overlapping (shared) attributes. For instance, most of the output attributes should be derived from input attributes; and most likely, we only combine input tuples based on equality of certain attributes. A provenance table only needs to contain one copy of each of these attributes.

Example 14.10

We refine the previous schemas to $P_{V1-1}(A, B, C)$ and $P_{V1-2}(B, C, C')$ with the following content:

P_{V1-1}

A	B	C
1	2	3
1	4	3

P_{V1-2}

B	C	C′
2	3	2
3	2	3

where we know that in the rule of V_1, $R.A = V.A = A$, $R.B = S.B = B$, and $S.C = V_1.C = C$; and in the second rule of V_1, for the first atom $S.B$ is equal to $V_1.A$ and column B, $S.C$ in the first atom matches $S.B$ in the second atom and has value C, and for the second atom $S.C = V_1.C = C'$.

Bibliographic Notes

As of the writing of this book, data provenance remains an area of intensive research in the database community. A major point of active work is capturing more expressive queries (e.g., negation, aggregation) for querying and indexing provenance storage and for understanding data provenance's relationship with the problem of encoding provenance of workflow systems. (In the latter case, a workflow system is generally a pipeline of stages in which little is known about their semantics.)

Provenance (sometimes called pedigree or lineage) has been a topic of interest in the scientific research community for many years. Perhaps the earliest formalisms for provenance in the database community date back to the early 2000s. Cui and Widom first proposed a notion of *lineage* for data warehouses, where the lineage of a tuple in a view consisted of the IDs of the set of source tuples or *witnesses* [155]. Roughly concurrently, Buneman et al. proposed notions of *why-* and *where-provenance* in [98]. Why-provenance contains a *set of sets* of base tuple IDs; each "inner" set represents a combination of tuples that can derive the tuple of interest in the view, and the different sets represent different possible derivations. Where-provenance, rather than annotating tuples, instead drills down to annotate an individual field. Work on the Trio system developed a different formalism for lineage, approximately equivalent to bags of sets of source tuples, described in [67]. Finally, the lineage term has also recently been used to refer to probabilistic event expressions, as in [535]. See [267] for more details on the relationship among the majority of different provenance formalisms.

The provenance semiring formalism has seen extensive study in recent years. A major reason is that it preserves the commonly used relational algebra equivalences that hold over bag semantics. It should be noted, however, that there are certain relational algebra equivalences that hold in set semantics but not bag semantics, such as join and union idempotence — such properties do not hold in the semiring model. See [28, 267] for more details. While we do not expressly discuss it in this book, the provenance semiring formalism has been extended to support provenance annotations over hierarchical data, as in where-provenance or the XML data model [231]. Extensions have also been developed to support queries with aggregation [29].

A provenance model similar to that of semirings, but focused on supporting several other operators such as semijoins, is that of Perm [252, 253]. Building upon Perm, the TRAMP model [254] distinguishes among data, query, and mapping provenance.

Others have looked at a concept closely related to provenance — that of explaining why an answer is *not* produced or why an answer is ranked higher than others [117, 307, 372]. Finally, connections have been made to the notion of *causality* in determining *which* part of provenance "best explains" a result [424, 425].

Recent work on the Trio [67] and ORCHESTRA [324] projects motivated a new study of provenance with greater ability to capture alternative derivations. In Trio, probabilities are assigned to a derived tuple by first computing the *lineage* (expression over probabilistic events). In ORCHESTRA, trust levels or ranks are computed for output tuples depending on

how they are derived. See [67] and [231, 267, 269], respectively, for more detail. An excellent survey of data provenance as a field appears in [133].

Provenance has been studied within the database community for a more expressive set of operators within the context of scientific workflow systems. Here, a workflow consists of "black box" operators strung together in a pipeline (possibly with control structures such as iteration). In contrast to data provenance, which exploits our knowledge of relational operator semantics, workflow provenance must assume that few if any equivalences hold. Hence provenance is purely a graph representing connections between data inputs, tools, and data products. The workflow provenance community has been at the core of the development of the Open Provenance Model [470]. The scientific workflow system that has the most comprehensive integration with data provenance is VisTrails [58], which is a workflow system for producing scientific visualizations. VisTrails enables users to examine and query provenance [513, 522], and even to generalize it into workflows. Other workflow systems such as Taverna [468] and Kepler [400] also have support for provenance. An emerging topic in the workflow provenance space is the notion of *securing* or abstracting away portions of a provenance graph [87, 164]. Recent work has proposed a model for unifying database and workflow provenance [27].

Related forms of data provenance are also studied outside the database community. Two prominent areas include those of provenance for filesystems [448] and work on dependency analysis in programs [132]. Recently, work on declarative techniques for describing network protocols [397] has led to a notion of *network provenance* [595]. Building upon these ideas, new forms of provenance for recording dynamic events in a system [594], and for ensuring that distributed provenance is unforgeable [593], have been developed. Others have examined provenance's connections to the Semantic Web and Linked Open Data efforts [387, 436, 591].

Finally, given that provenance has its own data model (as a hypergraph or system of semiring equations) and operations (projecting portions of a graph, computing annotations) — as well as value to end users — a natural question is how to store and query it. An initial provenance query language and storage scheme was proposed in [343].