Why and Where: A Characterization of Data Provenance*

Peter Buneman, Sanjeev Khanna**, and Wang-Chiew Tan

University of Pennsylvania
Department of Computer and Information Science
200 South 33rd Street, Philadelphia, PA 19104, USA
{peter,sanjeev,wctan}@saul.cis.upenn.edu

Abstract. With the proliferation of database views and curated databases, the issue of *data provenance* – where a piece of data came from and the process by which it arrived in the database – is becoming increasingly important, especially in scientific databases where understanding provenance is crucial to the accuracy and currency of data. In this paper we describe an approach to computing provenance when the data of interest has been created by a database query. We adopt a syntactic approach and present results for a general data model that applies to relational databases as well as to hierarchical data such as XML. A novel aspect of our work is a distinction between "why" provenance (refers to the source data that had some influence on the existence of the data) and "where" provenance (refers to the location(s) in the source databases from which the data was extracted).

1 Introduction

Data provenance — sometimes called "lineage" or "pedigree" — is the description of the origins of a piece of data and the process by which it arrived in a database. The field of molecular biology, for example, supports some 500 public databases [1], but only a handful of these are "source" data in the sense that they receive experimental data. All the other databases are in some sense *views* either of the source data or of other views. In fact, some of them are views of each other, which sounds nonsensical until one understands that the individual databases are not simply computed by queries, but also have added value in the form of corrections and annotations by experts (they are "curated"). A serious problem confronting the user of one of these databases is knowing the provenance of a given piece of data. This information is essential to anyone interested in the accuracy and timeliness of the data.

Understanding provenance and the process by which one records it is a complex issue. In this paper we address an important part of the general problem. Suppose a database (a view) $V = Q(D)$ is constructed by a query Q applied to

* This work was partly supported by a Digital Libraries 2 grant DL-2 IIS 98-17444
** Supported in part by an Alfred P. Sloan Research Fellowship.

databases D and we ask for the provenance of some piece of data d in $Q(D)$: what parts of the database D contributed to d? The problem has been addressed by [7, 2] for relational databases. In particular [7] considers the question : given a *tuple* in $Q(D)$ what tuples in D contributed to it. The crucial question here is what is meant by "contributed to". By examining provenance in a more general setting we draw a distinction between "where-provenance" – where does a given piece of data come from and – "why-provenance" – why is it in the database. Consider the following example:

```
SELECT name, telephone
FROM employee
WHERE salary > SELECT AVERAGE salary FROM employee
```

If one sees the tuple `("John Doe",1234)` in the output one could argue that every tuple in contributed to it, for modifying any tuple in the `employee` relation could affect the presence of `("John Doe",1234)` in the result. This is the why-provenance and it is what is studied in [7] as the set of contributing tuples. On the other hand, suppose one asks *where* the telephone number `1234` in the tuple `("John Doe",1234)` comes from, the answer is apparently much simpler: from the telephone field "John Doe" tuple in the input. This statement presupposes that `name` is a key for the `employee` relation; if it is not we need some other means of identifying the tuple in the source, for SQL does *not* eliminate duplicates. (Had we used `SELECT UNIQUE` the answer would be a set of locations.) The point is that where-provenance requires us to identify locations in the source data. Where-provenance is important for understanding the source of errors in data (what source data should John Doe investigate if he discovers that his telephone number is incorrect in the view.) It is also important for carrying annotations through database queries. Therefore as a basis for describing where-provenance, we use the data model proposed in [6] in which there is an explicit notion of location. The model has the advantage that it allows us to study provenance in a more general context than the relational model. Existing work on provenance considers only the relational model.

Outline. In the next section we describe the *deterministic* model in [6]. We then give a *syntactic* characterization of why-provenance and show that it is invariant under query rewriting. To this end, in Section 3 we describe a natural normal form for queries and give a strong normalization result for query rewriting. The normal form is useful because it also gives us a reasonable basis for defining where-provenance which turns out to be problematic and cannot, in general be expected to be invariant under query rewriting. We discuss a possible restriction for which where-provenance has a satisfactory characterization.

Related work. Why-provenance has been studied for relations in [2, 7]. To our knowledge no-one has studied where-provenance. A definition of why-provenance for relational views is given in [7], which also shows how to compute why-provenance for queries in the relational algebra. There, a semantic characterization of provenance is given which, when restricted to SPJU, has the expected properties such as invariance under query rewriting. In fact, the syntactic techniques developed in this paper, when restricted to a natural interpretation of the relational model, yield identical results to those in [7]. We do not know whether

there is semantic characterization for where-provenance nor do we know whether there is a semantic characterization of why-provenance that is well behaved on anything beyond than SPJU queries.

Expressing the why-provenance for a query is loosely related to the view maintenance problem [17]. It is apparently simpler in (a) that in why-provenance we are not interested in what is not in the view (view maintenance needs to account for additions to the database) and (b) that we are not asking how to reconstruct a view under a change to the source. Conversely, there is a loose connection between where-provenance and the view update problem. (If I want to update a data element in the output, what elements in the input need to be changed.) Recently, [10] has proposed using the deterministic model described here for view maintenance in scientific databases.

2 A Deterministic Model

We describe the data model in [6] where the *location* of any piece of data can be uniquely described by a *path*. This model uses a variation of existing edge-labeled tree models for semistructured data [14, 13]. It is more restrictive in that the out-edges of each node have distinct labels; it is less restrictive because these labels may themselves be pieces of semistructured data[1]. Figure 1 shows how certain common data structures can be expressed in this "deterministic" model of semistructured data. Here, any node in the deterministic tree is uniquely determined by a path of edge labels from root node to that node. These paths are analogous to l-values in programming language terminology. We will describe shortly how relations can be cast in this model by using the keys as edge labels. Any object-oriented or semistructured database with *persistent* object identifiers for all structures can also be expressed. There is also a variety of hierarchical data formats that implicitly conform to this model. Notably ACeDB [9], a lightweight DBMS originally developed as a database for genetic data conforms rather closely to this model and also supports certain operations such as "deep union" which are essential to the techniques developed in this paper.

2.1 Syntax and Operations

Values. We use the notation $x{:}y$ to denote a pair whose label is x and value is y. We can think of x as the edge label and y as the subtree under it. We use the notation $\{x_1{:}y_1,...,x_n{:}y_n\}$ to denote a set of such pairs. Since the edge-labels $x_1, ..., x_n$ are distinct, this notation describes a *finite partial function* from values to values. A set of values $\{s_1, ..., s_n\}$ can always be described in our model by mapping each element in the set to some standard constant (`c` in Figure 1). The last example shows how edge labels can be themselves pieces of semi-structured data. Value equality can be computed inductively.

[1] For the purposes of normal forms, these pieces of semistructured data are required to be "linear".

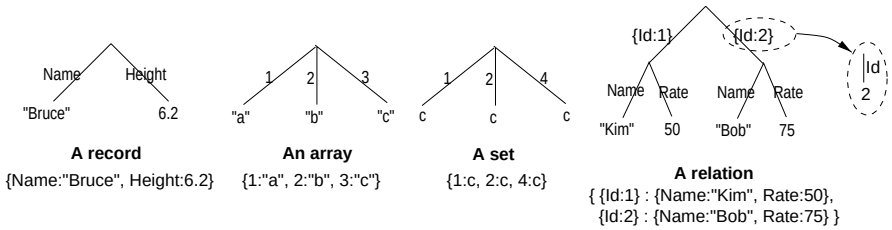


Fig. 1. Examples of data structures represented in our syntax.

Paths. We use the notation $x_1.x_2.\ldots.x_n$ for paths. In the last example of Figure 1, the path {`Id:1`} identifies the value {`Name:"Kim", Rate:50`} and the path {`Id:1`}`.Rate` identifies the value `50`.

Abbreviation. We use $e_1.e_2.....e_{n-1}{:}e_n$ as a shorthand for $\{e_1{:}\{e_2{:}$ $\{...\{e_{n-1}{:}e_n\}...\}\}\}$. We can think of $e_1.e_2.....e_{n-1}$ as the path leading to the value e_n. For example, {`Id:1`}`.Name:Kim` is an abbreviation for {{`Id:1`}`:`{`Name:Kim`}}.

Traversal. We use $v(p)$ to denote the subtree identified by a path p in value v. If path p does not occur in v, then $v(p)$ is undefined. For example: {`a:1,b:2`}`(c)` is undefined while {{`c:3`}`:1,b:2`}({`c:3`}) is `1`.

Path representation. Observe that any value in our model can be described by specifying the set of all paths to the constants at the terminal nodes. We call this the *path representation* of v. For example, the path representation of {`a:`{`1:c,3:d`}} is {(`a.1,c`),(`a.3,d`)}.

Definition 1. (Substructure) w is a *substructure* of v, denoted as $w \sqsubseteq v$, if the path representation of w is a subset of the path representation of v. □

Example. `a:`{`1:c,3:d`}$\sqsubseteq$ `a:`{`1:c,2:b,3:d`} but `a:`{`1:c,3:d`}$\not\sqsubseteq$ `b.a:`{`1:c,3:d`}. It is easy to see that since our model is deterministic, if $w \sqsubseteq v$ then w occurs as a part of v in a unique way.

Definition 2. (Deep Union) The *deep union* of v_1 with v_2, written as $v_1 \sqcup v_2$ is the value whose path representation is the union of the path representations of v_1 and v_2. Note that the result may not be a partial function in which case the deep union is undefined. □

Example. The deep union of {`a:1,b.c:2`} and {`b.d:4,e:5`} is {`a:1,b:`{`c:2, d:4`}`,e:5`} while the deep union of {`a:1,b.c:2`} and {`b.c:3,e:5`} is undefined.

2.2 An Encoding of Relations

We can encode relations as follows. Each relation name forms the label of an outgoing edge from the root node which is in turn mapped to the set of keys from that relation. Each key of a relation is then mapped to the corresponding tuple it identifies in the relation. If there is no key, the tuples are modeled as a

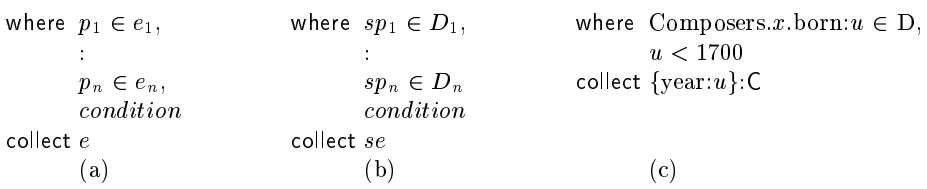


Fig. 2. (a) General form and (b) normal form of a query fragment. (c) An example.

set, that is, the entire tuple becomes an edge label. As an example, suppose we have two relations Composers and Works as shown below. The key for Composers is name and Works has a compound key (name, opus). The figure below also shows the encoding of the relations into our model. We see that keys of a tuple are placed on an edge in our model. If a tuple contains a compound key, we could model the entire compound key as a "linear" piece of semistructured data on the edge. That is, each key is placed one after another on the same edge. It does not matter which order we serialize the keys so long as this is done in a consistent manner.

Composers

name	born	period
"J.S. Bach"	1685	"baroque"
"G.F Handel"	1685	"baroque"
"W.A Mozart"	1756	"classical"

Works

name	opus	title
"J.S. Bach"	"BMV82"	"I have enough."
"J.S. Bach"	"BMV552"	NULL
"G.F Handel"	"HMV19"	"Art thou troubled?"

```
{ Composers:
   {{name:"J.S. Bach"}: {born:1685, period:"baroque"},
    {name:"G.F. Handel"}: {born:1685, period:"baroque"},
    {name:"W.A. Mozart"}: {born:1756, period:"classical"}},
  Works: {{{name:"J.S. Bach"}.opus:"BMV82"}:
                          { title: "I have enough." },
 {{name:"J.S. Bach"}.opus:"BMV552"}:
                          { title: "-" },
 {{name:"G.F Handel"}.opus:"HMV19"}:
                          { title: "Art thou troubled?" }} }
```

2.3 XML

At first sight, XML does not conform to a deterministic model. Insofar as some formal model for XML has been developed in the Document Object Model (DOM) [15] it is that of a *node*-labeled graph in which child labels may be repeated. The fact that it is node labeled is a minor irritant. Uniqueness is more serious. However, in the absence of any system of keys (see [16]) we can still fall back on the property, specified by the DOM, that child nodes can be uniquely identified by their positions and attribute nodes by their names. We defer the details of the translation of XML and a query language such as XML-QL [8] into our deterministic model and query language to the full version of this paper.

3 A Query Language

Query languages for semistructured data [3] are based on a general syntactic form shown in Figure 2(a). The p_is are patterns whose syntax follows the syntax for data (as defined in the previous section) augmented with variables[2]. Expressions $e, e_1, ..., e_n$ are essentially the same as patterns but may contain "where ...

[2] In semistructured query languages, patterns can also include regular expressions on the edge labels. We will not deal with such patterns in this paper.

collect..." expressions (nested queries). *condition* is simply a boolean predicate on the variables of the query.

The interpretation of a "where ... collect..." expression is as follows: consider each assignment of the variables in the expression that makes each pattern p_i a substructure of the corresponding expression e_i. For each such assignment, evaluate the *condition*. If it is true, add the (instantiated) value of e to the output. Finally "union" together the output values. This interpretation is quite general, but to make it precise we must (a) define what we mean by "union" and (b) say what values a variable can bind to (a constant, an arbitrary value, or something in between). Languages, see [13, 5] vary in their choice of the "union" operation. In our case, we use the deep union operation. Thus the output of the query in Figure 2(c) is {`year:1685`}:C even though this value is emitted twice. A consequence of this is that the result of a query maybe undefined.

We add the deep union operation to our language, and the general syntax can be summarized by the following grammar:

$e ::=$ where $p \in e, \ldots, p \in e$, *condition* collect $e \mid e \sqcup e \mid \{e : e\} \mid c \mid x$

where c ranges over constants, x over variables, p over patterns and *condition* over conditions. Note that $\{e_1 : e'_1, \ldots, e_n : e'_n\}$ is in fact a shorthand for $\{e_1 : e'_1\} \sqcup \ldots \sqcup \{e_n : e'_n\}$. We refer to this query language, for want of a better term, as DQL (Deterministic QL). The syntax of the query language is quite general, but its interpretation is limited by the model. In order to set up the machinery to analyze provenance, we will make some restrictions both on the syntax and interpretation of queries for the soundness of our rewrite rules. First, we impose some syntactic restrictions.

Definition 3. (Well-Formed Query) A query Q is said to be *well-formed* if (a) no pattern p_i is a single variable, (b) each expression e_i is either a (nested) query or an expression that does not involve a query, and (c) each comparison is between variables or between variables and constants only. □

Conditions (a) and (b) are required for the soundness of our rewrite rules. Condition (c) restricts our queries to the "conjunctive" fragment for which containment of queries can be easily determined. In addition to well-formedness, we say a query is *well-defined* if it is not undefined on any input. For the rest of the paper, we consider only queries that are both well-formed and well-defined. The next restriction we place is on the interpretation of a query. For this, we need the notion of a *singular* expression, which consists of a single path terminated by a constant or variable.

Definition 4. (Singular expression) A expression e is *singular* if $e \neq (e_1 \sqcup e_2)$ for any non-empty and distinct expressions e_1 and e_2. □

Our restriction on the interpretation is that variables may only bind to singular values. At first sight, this seems very restrictive and the interpretation of a query is unusual. Consider the query (a) in Figure 3. It binds singular values to y, and the output is {{`name:"J.S. Bach"`}.`born:1685`, {`name:"G.F. Handel"`}.`born:1685`}. This is probably not the expected output for someone

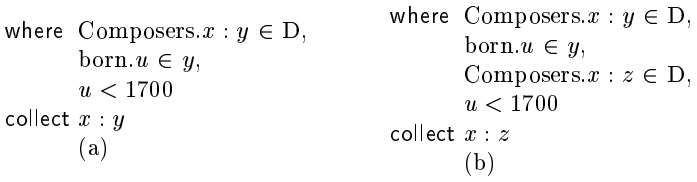


Fig. 3. More example DQL queries. C denotes some constant.

familiar with, say, XML-QL in which variables bind to complete subtrees. However there is an easy translation, illustrated in query (b) from the XML-QL interpretation[3] into DQL. Note that the deep union reconstructs the subtree.

Restrictive as it may seem, DQL can capture positive (SPJU) relational queries and positive nested relational algebra ([12]). It is less expressive than XML-QL in that (a) it cannot express path patterns involving a Kleene-star(*), (b) it works only on hierarchical structures, and (c) the forms of Skolem function and nested query forms in XML-QL that can be simulated are limited. We omit the details in this paper.

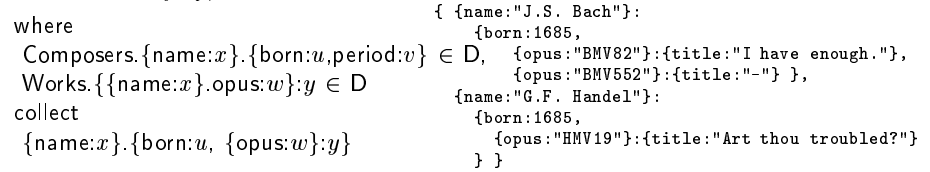
Definition 5. (Normal Form) A query Q is said to be in *normal form* if Q has the form $Q_1 \sqcup ... \sqcup Q_m$ where each Q_i is as shown in Figure 2(b). sp_i and se is a singular pattern and singular expression respectively. D_i is a database constant and *condition* is a boolean predicate on the variables of the query. □

Our main result in this section is that every well-formed query has an equivalent normal form which can be determined from our rewrite system $\mathcal{R}$. We omit the details of $\mathcal{R}$ and state the *strong normalization* result which says that starting from any well-formed query, any sequence of application of rewrite rules leads to a normal form in a finite number of steps.

Theorem 1. (Strong Normalization) The rewrite system $\mathcal{R}$ is strongly normalizing.

4 Two Meanings of Provenance

Equipped with a data model and query language, we are now in a position to formulate two meanings of provenance and to compute the provenance of a component d in a view $V = Q(D)$ where Q is as query and D is the source data. We will formulate the provenance of d as a query Q' that is completely determined by Q, D and d.



[3] We assume that the XML-QL interpretation contains a skolem function that groups by composer names.

The above query, say Q_1 expresses a join on components of the database described in Section 2.2. Consider the value referenced by {name:"G.F Handel"}.born. This value was generated by Q_1 as any instance of the "collect" expression in which the variable x was bound to "G.F Handel" and u to 1685. We now look at the patterns in the "where" clause to find what (simultaneous) matches of these patterns caused these bindings. In this case there is only one such match consisting of the patterns (after instantiating the variables):

```
Composers.{name:"G.F.Handel"}.{born:1685,period:"baroque"}
Works.{{name:"G.F. Handel"}.opus:"HMV19"}.title:"Art thou troubled?"
```

Moreover if we apply Q_1 to any database that contains these structures, we will obtain an output that contains {name:"G.F. Handel"}.born:1685. This is the rationale for calling these structures the why-provenance of the value referenced by {name:"G.F Handel"}.born. However, if we are interested in the where-provenance of {name:"G.F Handel"}.born, we only need to look at the pattern(s) that bind the variable u to determine that it came from the path Composers.{name:"G.F Handel"}.born.

Our example suggests that one natural approach to compute provenance is via syntactic analysis of the query and this is the approach that we take.

5 Why-Provenance

In the model-theoretic approach to datalog programs described in [4], these programs are viewed as a set of first-order sentences describing the desired answer. For example, if we have a datalog rule $R(u) : -R_1(u_1), ..., R_n(u_n)$, we could associate the logical sentence: $\forall x_1, ..., x_m \bigwedge_{i\in[1..n]} R_i(u_i) \rightarrow R(u)$ where $x_1, ..., x_m$ are variables occuring the in the rule. A DQL query $\{e \mid p_1 \in D, ..., p_n \in D, condition\}$ could be viewed as the following logical sentence: $\forall x_1, ..., x_m$ $(\bigwedge_{i\in[1..n]} p_i \in D$ and $condition) \rightarrow e$ is in the output. $x_1, ..., x_m$ are variables which occurs in the query. Therefore a value v is provable if there exists a valuation that will make the premise true and puts v in the output.

As discussed earlier, the structures in the why-provenance example correspond to a proof for {name:"G.F Handel"}.born:1685. We call the collection of values taken from D that proves an output, a *witness* for the output. More specifically, we say a value s is a *witness* for a value t with respect to a query Q and a database D, if $t \sqsubseteq Q(s)$ and $s \sqsubseteq D$. The value shown below is a witness for {name:"G.F Handel"}.born:1685.

```
{ Composers.{name:"G.F. Handel"}.{born:1685, period:"baroque"},
  Works.{{name:"G.F. Handel"}.opus:"HMV19"}.title:"Art thou troubled?" }
```

5.1 Witness Basis

We now refine the notion of witness as introduced above to be explicitly tied to the structure of a given query as well as an input database. Specifically, for a singular value t, we only consider witnesses that correspond to the deep union of values taken from D (at the leaves of a proof tree for t) with respect to

a query Q. For Q_1 and output {name:"G.F Handel"}.born:1685, the witness above corresponds to values at the leaves of the proof tree taken from D. The following is also a witness for the same value but it is not the result of deep union of values at the end of any proof tree for that value.

```
{ Composers.{{name:"G.F. Handel"}.{born:1685, period:"baroque"},
             {name:"W.A Mozart"}.{born:1756, period:"classical"}},
  Works.{{name:"G.F. Handel"}.opus:"HMV19"}.title:"Art thou troubled?" }
```

We describe next our notion of a *witness basis* which captures the set of all witnesses of the former type for any value t in $Q(D)$. Our definition closely follows the syntax of the query.

Definition 6. (Witness Basis) Consider a normal form query Q. The *witness basis* for a singular value t with respect to Q and D, denoted as $W_{Q,D}(t)$, is:

(1) If Q is of the form $Q_1 \sqcup ... \sqcup Q_n$ then $W_{Q,D}(t) = W_{Q_1,D}(t) \cup ... \cup W_{Q_n,D}(t)$.
(2) If Q is of the form $\{e \mid p_0 \in e_0, ..., p_n \in e_n, condition\}$, let Ψ be the set of all valuations on the variables of Q such that "where" clause of Q holds under each valuation in Ψ. Then, $W_{Q,D}(t) = \{[\![p_0]\!]_\psi \sqcup ... \sqcup [\![p_n]\!]_\psi \mid \psi \in \Psi, t = [\![e]\!]_\psi\}$. Note that e_i $(0 \leq i \leq n)$ is a database constant since Q is in normal form.
(3) Otherwise, $W_{Q,D}(t) = \{\}$.

More generally, for any well-formed query Q, we can define the witness basis by extending (2) as follows. We partition the set of $p_i \in e_i$ in the "where" clause of Q into two parts: $S_1 = \{p_i \mid e_i$ is the database constant $D\}$ and $S_2 = \{(p_i, e_i) \mid p_i$ is a pattern matched against a query $e_i\}$. We use $p_0^1, ..., p_k^1$ to denote the members of S_1 and $(p_0^2, e_0^2), ..., (p_m^2, e_m^2)$ to denote the members of S_2. Let Ψ be the set of all valuations on the variables of Q such that for each valuation in Ψ, "where" clause of Q holds. Then $W_{Q,D}(t) = \{P_1 \sqcup P_2 \mid \psi \in \Psi, t \sqsubseteq [\![e]\!]_\psi, P_1 = [\![p_0^1]\!]_\psi \sqcup ... \sqcup [\![p_k^1]\!]_\psi, P_2 = w_1 \sqcup ... \sqcup w_m$ where $w_i \in W_{\psi(e_i^2),D}([\![p_i^2]\!]_\psi)\}$. For a compound value t, the witness basis is the product of individual witness basis of singular values making up t. That is, consider $t = t_1 \sqcup ... \sqcup t_m$ where each t_i is singular. Then $W_{Q,D}(t) = \{w_1 \sqcup ... \sqcup w_m \mid w_i \in W_{Q,D}(t_i)\}$. □

The general definition above looks for patterns which are matched against the database constant D and patterns which match against queries. The former is collected together as part of the witness under P. If the generator is a nested query, we inductively look for the witness basis of these patterns under the valuation and later combine the results together by taking the product. Next, we show that the witness basis of a well-formed query is in fact the same as the witness basis of its normal form.

Lemma 1. If $Q \rightsquigarrow Q'$ via the rewrite system $\mathcal{R}$, then for any value t in the output of $Q(D)$, $W_{Q,D}(t) = W_{Q',D}(t)$.

Computing a Witness Basis. We next show a procedure for finding $W_{Q,D}(t)$ where t is a singular value and Q is a query in normal form. That is, $Q = Q_1 \sqcup ... \sqcup Q_n$ and each $Q_i = \{e_i \mid p_{i1} \in D, ..., p_{ik_i} \in D, condition_i\}$. To look for members of the witness basis of t, we need to search for valuations on variables in

each Q_i that will produce t. For those valuations that produce t, the deep union of p_{i1} to p_{ik_i} under each valuation is returned as a result. However, instead of searching the witness basis directly, we produce a query Q'_i, which when evaluated, will generate the witness basis. The "where" clause of Q'_i is the same as Q_i and the "collect" clause contains an output expression which is the deep union of all patterns in the "where" clause of Q_i placed on an edge. This is to prevent inter-mixing with other members of the witness basis. The algorithm for generating Q'_i from Q_i is described below. $\text{Why}(t, Q, D)$ is simply $\text{Why}(t, Q_1, D) \sqcup ... \sqcup \text{Why}(t, Q_n, D)$. ψ is a valuation from e_i to t such that $\psi(e_i) = t$. This technique is sound and complete in the sense that the set of witnesses in $W_{Q,D}(t)$ is the same as the set of witnesses returned by $\text{Why}(t, Q, D)(D)$.

Algorithm: $\text{Why}(t, Q_i, D)$

Let Δ denote the "where" clause of Q_i.
Let Δ' denote the deep union of patterns in Δ.
if there is a valuation ψ from e_i to t **then**
  Return the query "where $\psi(\Delta)$ collect $\psi(\Delta')$:C"
  (For simplicity, we did not serialize the output expression on the edge.)
else
  No query is returned
end if

Theorem 2. (Soundness and Completeness) Let Q be a query in normal form and t be any singular value in the output of $Q(D)$. Then $W_{Q,D}(t) = \mathsf{Why}(t, Q, D)(D)$.

A Comparison. We point out here that our notion of witness basis coincides with the derivation of a tuple in [7] for SPJU queries where the general case of theta-join is considered. The details are deferred to the full version.

5.2 Minimal Witness Basis

Observe that a witness for a value is invariant under all equivalent queries but the witness basis is not. We show next that a subset of the witness basis, called *minimal witness basis*, is in fact invariant under queries with only equalities.

Definition 7. (Minimal Witness, Minimal Witness Basis) A value s is a *minimal witness* for a value t with respect to Q if $\forall s' \sqsubset s \;\; t \not\sqsubseteq Q(s')$. The *minimal witness basis* for a value t with respect to a query Q and database D, denoted as $M_{Q,D}(t)$, is a maximal subset of $W_{Q,D}(t)$ such that $\forall m \in M_{Q,D}(t)$ $\nexists w \in W_{Q,D}(t)$ such that $w \sqsubset m$. □

Example: According to the query Q_1, introduced earlier, the witness shown in Section 5 is a minimal witness for `{name:"G.F. Handel"}.born:1685` while the witness shown in Section 5.1 is not.

Theorem 3. (Invariance of Minimal Witness Basis under Equivalent queries) If Q and Q' are two equivalent well-formed queries with only equality conditions and t is contained in $Q(D)$ and $Q'(D)$, then $M_{Q,D}(t) = M_{Q',D}(t)$.

The proof of this theorem is based on a homomorphism theorem which shows that for the class of well-formed and well-defined queries (with equality conditions), query containment is equivalent to the existence of a homomorphism between the queries. Based on the ideas in [11], we can also extend this theorem to certain subclasses of queries with inequalities. Thus the invariance property of minimal witness basis in fact holds across this larger class of queries.

5.3 Cascaded Witnesses (Query Composition)

Suppose we have some data sources – a mixture of materialized views (V) and actual databases (D) – and a query written against these sources. We may choose to find the witness basis for a value with respect to these sources (our witnesses will therefore consist of values from both V and D) and subsequently finding the witness basis of those components taken from the views so that eventually, witnesses in the witness basis consist of only values from D. We show next that the witness basis obtained in this manner is the same as first "composing out" the views in the query using the composition rule in our rewrite system $\mathcal{R}$ and obtaining the witness basis according to the rewritten query. In fact, this result is an important special case of Lemma 1 where views are nested queries not sharing any variables with the outer query block.

Theorem 4. (Unnesting of Witnesses) Let D be a set of databases, V be a query written against D and Q be a query written against D and V. Then for a value t in $Q(D,V)$, $W_{Q',D}(t) = \{w \sqcup w' \mid (w \sqcup v') \in W_{Q,\{D,V(D)\}}(t), v'$ is the value taken from view $V(D)$, $w' \in W_{V,D}(v')\}$ where Q' is the rewritten query via our rewrite system $\mathcal{R}$ in which view V has been "composed out".

6 Where-Provenance

So far we have explored the issue of what pieces of input data validate the existence of an output value, for a given query. We now focus on identifying what pieces of input data helped create various values that appear in the output. The where-provenance of a specific value in the output is closely connected to the witnesses for the output in that only some parts of any witness are used to construct a specific output value. For instance, in the example described in Section 4, the output value "1685" in `{name:"G.F. Handel"}.born:1685` depends only on `Composers.{name:"G.F. Handel"}.born:1685` in the input. We refer to the path `Composers.{name:"G.F. Handel"}.born` in the input as the where-provenance of this output value. This informal description already suggests an intuitive procedure for determining the where-provenance of any specific value in the output: determine which output variable was bound to this specific value, and then identify the pieces of input data that were bound to this output variable. However, this intuition is fragile and there are many difficulties involved in formalizing this intuition as illustrated by the sequence of examples below. Consider the following two equivalent queries that look for employees with a salary of \$50K :

$$Q_1 = \textsf{where}\ \ \text{Emps.}\{\text{Id:}x\}.\text{salary:\$50K} \in \text{D},\ \ \textsf{collect}\ \{\text{Id:}x\}.\text{salary:\$50K}$$

$$Q_2 = \textsf{where}\ \ \text{Emps.}\{\text{Id:}x\}.\text{salary:}y \in \text{D},\ y = \$50K\ \ \textsf{collect}\ \{\text{Id:}x\}.\text{salary:}y$$

Suppose we wish to determine the where-provenance of \$50K in an output tuple. In case of query Q_1, there is no variable in the collect clause which the value \$50K can be identified with. The where-provenance of this value in Q_1 is the query itself since the value is hard-wired into the query output. For Q_2, the output variable y can be associated with the value \$50K and can be used to identify what contributed to this value. By convention, we will consider the where-provenance of a specific value in the output to be defined only if it can be associated with one or more variables in the output expression of a query. Otherwise, we will ascribe the where-provenance of a value to the query itself. This example illustrates that the notion of where-provenance is hard to keep invariant over equivalent queries in general.

Our next example shows that when multiple pieces of data may simultaneously contribute to a specific value in the output, it may be difficult to identify all the pieces.

$$Q_3 = \textsf{where}\ \ \text{Emps.}\{\text{Id:}x\}.\text{salary:}y \in \text{D},\ \text{Emps.}\{\text{Id:}x\}.\text{bonus:}y \in \text{D}\ \ \textsf{collect}\ \{\text{Id:}x\}.\text{new\_salary:}y$$

$$Q_4 = \textsf{where}\ \ \text{Emps.}\{\text{Id:}x\}.\text{salary:}y \in \text{D},\ \text{Emps.}\{\text{Id:}x\}.\text{salary:}z \in \text{D},\ \text{Emps.}\{\text{Id:}x\}.\text{bonus:}z \in \text{D}\ \ \textsf{collect}\ \{\text{Id:}x\}.\text{new\_salary:y}$$

In case of Q_3, the value associated with any new_salary component in the output originated from both the salary and bonus components of the corresponding employee. This is easily identified by tracking the output variable y through the query. But in Q_4, which is equivalent to Q_3, on any input data where salary and bonus are atomic values, one needs to recognize that z is always forced to agree with y and hence where-provenance is determined by y and z together. This suggests that in general the syntactic structure of a query may not suffice for identifying the where-provenance. Even in cases where syntactic analysis alone may work, this issue becomes rather difficult to handle once we consider nested queries. Consider the following two equivalent queries:

$$Q_5 = \textsf{where}\ \ \text{R}.x.y : z \in \text{D},\ \text{S}.x.y : z \in \text{D}\ \ \textsf{collect}\ x.y : z$$

$$Q_6 = \textsf{where}\ \ \text{R}.x.y : z \in \text{D},\ \text{S}.t.u \in \text{D},\ t : u \in (\textsf{where}\ \ \text{R}.x.y : z \in \text{D}\ \ \textsf{collect}\ x.y : z),\ \ \textsf{collect}\ \{x.y : z, t : u\}$$

When applied to an input database {`R.1.2:3,S.1.2:3`}, these queries produce as output `1.2:3`. The where-provenance of value `3` in the output is {`R.1:2, S.1:2`} in case of query Q_5. In contrast, where-provenance of the same value with respect to Q_6 requires one to identify that u binds to $y : z$ via the nested query. Then, the where-provenance is given by {`R.1:2,S.1:2`} in this case as well.

A Syntactic Approach. The examples above highlight that for general queries, where-provenance is not invariant over the space of equivalent queries, and that

a purely syntactic characterization of where-provenance is unlikely to yield a complete description of the where-provenance. However, we use the syntactic approach and identify a restricted class of queries referred to as *traceable queries*, for which where-provenance is preserved under rewriting. Our approach is based on formalizing our initial intuition of using variables in the output expression of a query as a means of identifying the where-provenance of a value. Specifically, for each successful valuation of the query, we systematically explore the pieces of input data contributing to the identified output variable; and we refer to this as the *derivation basis* of the output value. To determine where-provenance of a value resulting from a traceable query, it suffices to work with the normal form of the query. Once a query is in normal form, a straightforward procedure can be used to compute the derivation basis of a given value.

Paths. To identify the where-provenance of a value in our tree of values, we need to extend our notion of paths. We augment our syntax for paths with "%". For example, to refer to the value {`name:"J.S. Bach"`} which is a value on the edge of Composers relation, we could use the path `Composers.{name:"J.S. Bach"}%`. To refer to the value `"J.S. Bach"`, we could use the path `Composers.{name:"J.S. Bach"}%name`.

We show next the definition of derivation basis (where-provenance) for queries in normal form. Informally, the derivation basis for $l{:}v$ finds a variable x in the output expression that will generate v. This can be done by partially matching $l{:}v$ against the output expression e. All the paths to x in the patterns of Q are then determined. Then, for any valuation that satisfies the "where" clause, the valuation of the patterns in the "where" clause will form the witness, and the valuation of the paths that point to x will be the where-provenance of $l{:}v$ with respect to this witness. Altogether, they form the derivation basis of $l{:}v$. We refer to the procedure that computes the derivation basis of $l{:}v$ as Where($l{:}v$,Q,D). It is similar to Why(t, Q, D) in that we generate a query which when applied to D will produce the derivation basis. The "where" clause of the generated query is the "where" clause of Q and the "collect" clause of the generated query emits two things: the patterns and the paths pointing to x in the "where" clause of Q.

Definition 8. (Derivation Basis) Consider a normal form query Q. The *derivation basis* for $l{:}v$ where v is an atomic value, denoted as $\Gamma_{Q,D}(l : v)$ with respect to Q and D, is defined as below:

(1) If $Q = Q_1 \sqcup ... \sqcup Q_n$ then $\Gamma_{Q,D}(l : v) = \Gamma_{Q_1,D}(l : v) \cup ... \cup \Gamma_{Q_n,D}(l : v)$.

(2) If Q has the form $\{e \mid p_0 \in e_0, ..., p_n \in e_n, condition\}$, let Ψ be the set of valuations on the variables of Q such that the "where" clause of Q holds under each valuation and $\psi(e)$ contains $l{:}v$. For each $\psi \in \Psi$, let p_{x_ψ} denote the path in e that points to a variable x_ψ such that there exists p' and p'' so that $l = p'.p''$ and $\psi(p_{x_\psi}) = p'$ and $\psi(x_\psi)(p'') = v$. Then, $\Gamma_{Q,D}(l : v) = \{([\![p_0]\!]_\psi \sqcup ... \sqcup [\![p_n]\!]_\psi, S) \mid \psi \in \Psi, S = \{\psi(p'_i).p'' \mid p'_i$ is the path that points to variable x_ψ in pattern p_i, $0 \leq i \leq n\}\}$.

(3) Otherwise, $\Gamma_{Q,D}(l : v) = \{\}$.

More generally, the derivation basis of $l{:}v$ where v is a compound value is defined to be the derivation basis of all possible (path,value) pairs $p'{:}v'$ such that

$p'{:}v'$ points to a value in v. The derivation basis for multiple (path,value) pairs is defined to be the product of the derivation basis of individual (path,value) pairs. That is, $\Gamma_{Q,D}(p_1{:}v_1, p_2{:}v_2) = \Gamma_{Q,D}(p_1{:}v_1) * \Gamma_{Q,D}(p_2{:}v_2) = \{(w_1 \sqcup w_2, P_1 \cup P_2) \mid (w_1, P_1) \in \Gamma_{Q,D}(p_1{:}v_1), (w_2, P_2) \in \Gamma_{Q,D}(p_2{:}v_2)\}$. □

We omit the definition for queries in the general form and remark that the main difference is that it looks for the derviation basis inductively for patterns matched against nested queries. We show next that in dealing with the derivation basis for the class of traceable queries, we can restrict our attention to the derivation basis corresponding to their normal forms.

Definition 9. (Traceable Queries) A well-defined query Q is *traceable* if (a) each pattern in the query matches either against some database constant or against a subquery, (b) every subquery in Q is a view which does not share any variables with the outer scope (c) only a singular pattern is allowed to match against a subquery and (d) this pattern and output expression of the subquery consist of a sequence of distinct variables (variables do not repeat) and have the same length. □

Example: The first query below is not traceable because the variable u is being used in the inner query (this violates condition (b)). The second query is not traceable because an expression $\{y{:}w\}$ is used in the pattern sequence (this violates condition (d) where each expression in the sequence can only be a variable).

$$\begin{array}{l}\mathsf{where}\ \ x : u \in D,\\ \qquad y : z \in \begin{pmatrix}\mathsf{where}\ \ u : v \in D\\ \mathsf{collect}\ u : v\end{pmatrix},\\ \mathsf{collect}\ x : z\end{array} \qquad \begin{array}{l}\mathsf{where}\ \ x : y \in D,\\ \qquad \{y : w\} : z \in \begin{pmatrix}\mathsf{where}\ \ u : v \in D\\ \mathsf{collect}\ u : v\end{pmatrix},\\ \mathsf{collect}\ x : y\end{array}$$

Proposition 1. If Q is a traceable query and $Q \rightsquigarrow Q'$ via rewrite system $\mathcal{R}$, then Q' is a traceable query.

Proposition 2. For the class of traceable queries, if $Q \rightsquigarrow Q'$ via rewrite system $\mathcal{R}$, then for any $l{:}v$ in the output of $Q(D)$, $\Gamma_{Q,D}(l : v) = \Gamma_{Q',D}(l : v)$.

7 Conclusions

We have described a framework for both describing and understanding provenance of data in the context of SPJU queries and views. Data provenance is examined from two perspectives, namely (1) Why is a piece of data in the output?, and (2) Where did a piece of data come from?

We have taken a syntactic approach to understanding both notions of provenance, and we have described a system of rewrite rules in which why-provenance is preserved over the class of well-defined queries and where-provenance is preserved over the class of traceable queries.

One interesting direction for future work is to identify necessary and sufficient conditions for the class of well-defined queries. Another interesting direction

is to study how additional constraints on the input instances, e.g., functional dependencies, can help us obtain a more complete description of the where-provenance of a piece of data.

Acknowledgements. We thank Victor Vianu, Susan Davidson, Val Tannen and the Penn Database Group students for many useful exchanges; and the paper reviewers for many useful comments.

References

1. *INFOBIOGEN. DBCAT, The Public Catalog of Databases.* `http://www.infobiogen.fr/services/dbcat/`, cited 5 June 2000.
2. A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, pages 91–102, 1997.
3. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web. From Relations to Semistructured Data and XML.* Morgan Kaufman, 2000.
4. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison Wesley Publishing Co, 1995.
5. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. *Journal on Digital Libraries*, 1(1), 1996.
6. P. Buneman, A. Deutsch, and W. Tan. A Deterministic Model for Semistructured Data. In *Proc. of the Workshop On Query Processing for Semistructured Data and Non-standard Data Formats*, pages 14–19, 1999.
7. Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *ICDE*, pages 367–378, 2000.
8. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. *XML-QL: A Query Language for XML*, 1998. `http://www.w3.org/TR/NOTE-xml-ql`.
9. R. Durbin and J. T. Mieg. *ACeDB – A C. elegans Database: Syntactic definitions for the ACeDB data base manager*, 1992. `http://probe.nalusda.gov:8000/acedocs/syntax.html`.
10. H. Liefke and S. Davidson. Efficient View Maintenance in XML Data Warehouses. Technical Report MS-CIS-99-27, University of Pennsylvania, 1999.
11. A. Klug. On conjuncitve queries containing inequalities. *Journal of the ACM*, 1(1):146–160, 1988.
12. L. Wong. Normal Forms and Conservative Properties for Query Languages over Collection Types. In *PODS*, Washington, D.C., May 1993.
13. P. Buneman and S. Davidson and G. Hillebrand and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *SIGMOD*, pages 505–516, 1996.
14. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *ICDE*, 1996.
15. World Wide Web Consortium (W3C). *Document Object Model (DOM) Level 1 Specification*, 2000. `http://www.w3.org/TR/REC-DOM-Level-1`.
16. World Wide Web Consortium (W3C). *XML Schema Part 0: Primer*, 2000. `http://www.w3.org/TR/xmlschema-0/`.
17. Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327, 1995.