Data Provenance for Attributes: Attribute Lineage

Dennis Dosso
University of Padua

Susan B. Davidson
University of Pennsylvania

Gianmaria Silvello
University of Padua

Abstract

In this paper we define a new kind of data provenance for database management systems, called *attribute lineage* for SPJRU queries, building on previous works on data provenance for tuples.

We take inspiration from the classical lineage, metadata that enables users to discover which tuples in the input are used to produce a tuple in the output. Attribute lineage is instead defined as the set of all cells in the input database that are used by the query to produce one cell in the output.

It is shown that attribute lineage is more informative than simple lineage and we discuss potential new applications for this new metadata.

1 Introduction

In the past, data was stored in curated databases or in other trusted sources of information kept under centralized control [2]. With the advent of the Internet, this assumption is no longer valid [5]. Data are today created, shared, copied, cited, reported, moved around, and combined indiscriminately.

On the other hand, data management is growing in complexity [6] also thanks to new algorithms, applications, and larger storage capacity.

In such an environment, it becomes more and more difficult to keep track of the origins, the reliability, and the process of elaboration of data used in research. One way to face such challenges is the deployment of *data provenance* [2].

Data provenance is information attached to data that describes its origin and the process which created it. It can also be seen as metadata pertaining to the derivation history of the data. It is particularly useful to help users to understand where data are coming from, and the process they went through.

Data provenance has been widely studied in different areas of data management. In this paper, we focus on provenance in the database management systems environment. For further details on data provenance, please refer to surveys like [2] and [6].

Many different notions of provenance have been proposed in the literature for data in database management systems [1, 3, 4], describing different kinds of relationships between data in the input and the output of a query. As reported in [2], these provenances, beyond the intrinsic information on how queries work, have been used in a number of applications, as the study of annotation propagation and view update.

In this paper, we focus on one of the earlier notions of provenance: *lineage*. First defined in [3], the *data lineage problem* consists in determining the source data which produced the output of a query and its process. Some applications of lineage are: (i) finding the data that originated specific views in scientific databases; (ii) finding faulty data in a huge dump generated from network monitors in the context of diagnosis systems; (iii) helping to translate view updates into the corresponding updates performed on the base data.

Lineage is defined in different ways. Here, we refer to the definition given in [2]. Given a database instance I, a query Q, and a tuple $t \in Q(I)$, the lineage of t is the set of all and only the tuples in I that were used in Q to produce it. We call these the *relevant* tuples.

In this paper, we propose a different kind of provenance called *attribute lineage*, which takes inspiration from lineage. To make a clear distinction, the classical lineage is *tuple based* that is, it tells the provenance of one tuple of the output. We also call it *tuple lineage*. Attribute lineage, on the other hand, is *attribute based*: it tells the provenance of one attribute in one output tuple. Informally, we define the attribute lineage of an output cell (a value in one output tuple) as the set of all and only the cells in I that were used to produce it.

This new lineage is more informative than tuple lineage since it not only indicates the relevant tuples, but also the relevant attributes inside these tuples. As such, it allows for an even higher understanding of the origin of data. This understanding can be useful in the tasks where a higher degree of granularity is required in the study of the provenance of data. One example is the identification of "hotspots" in a relational database. Given a set of queries over the same database instance, the more an attribute is used, the more frequently it

will appear in the lineages. It can, therefore, be considered a hotspot of the database, i.e. information that is used frequently and is therefore particularly important in the context of these queries.

The paper is organized as follows: in Section 2 we report the related works; Section 3 presents definitions that are used in this paper; Section 4 defines the attribute lineage and shows how it can be connected to the classical tuple lineage; finally, Section 5 reports our conclusions and the future works.

2 Related Work

In [2] four main data provenance types for database management systems are discussed: *lineage* [3], *why-provenance* [1], *how-provenance* [4] and *where-provenance* [1].

Why-, how- and where- provenance are all tuple-based: Given a database instance I, a query Q and the result $Q(D)$, consider one tuple t of the output. The provenance of t is information about the generation of this tuple using the provenance of the input tuples used by Q. Different types of provenance convey different levels of information.

Lineage is defined as the set of all (and only) tuples that are used in the query, i.e. the tuples that are *relevant* to its generation.

Why-provenance is based on the notion of *witness set*. A witness is a set of relevant tuples that *guarantee* the existence of t in $Q(D)$. Lineage is therefore an example of witness. The why-provenance of a tuple t is a particular set of witnesses – described in [1] – that are computed from the query, called the *witness basis*. A witness basis may consist of more than one witness. Therefore, why-provenance contains more information than lineage, since it describes alternative ways in which the same output may be generated.

How-provenance takes the form of a polynomial, called the *provenance polynomial*, where variables are taken from the set of identifiers of the tuples and coefficients are taken from $\mathbb{N}$. As suggested by the name, this provenance also conveys information on how the input tuples are used in Q. For example, when two tuples are combined in a join, their provenance is also combined in the polynomial using the $\cdot$ operator. When two or more tuples become equivalent in a union or a projection, the corresponding monomials are combined using the $+$ operator. It has been shown in [2] that how-provenance is the most general and informative of the three, and contains both why- and where-provenance.

Where-provenance is attribute-based. Given a tuple t and an attribute A of $Q(I)$, the where-provenance of the value $t \bullet A$ is the set of cells in I from which $t \bullet A$ has been copied. In this sense, where-provenance describes from *where* an attribute is coming.

The *attribute lineage* presented in this paper is attribute-based as in where-provenance, and gives the set of all and only the relevant cells used in the input I to build one cell $t \cdot A$ of the output.

3 Preliminaries

The notation that we use here is largely taken from [2]. Let $\mathbf{D}$ be a finite domain of data values $\{d_1, \ldots, d_n\}$ and $\mathcal{U}$ a collection of *field names* (also called attribute names). We use the symbols U, V to indicate finite subsets of $\mathcal{U}$.

A tuple t is a function $U \mapsto \mathbf{D}$, written as $(A_1 : d_1, \ldots, A_n : d_n)$. A tuple assigning values to each field name in U is called U-tuple. We write *Tuple* for the set of all tuples, *U-Tuple* for the set of all U-tuples. We write $t \bullet A$ for the value of the A-field of t and $t[U]$ for the restriction of tuple t over $U \subseteq V$ to field names in U. We write $t[A \mapsto B]$ for the result of renaming field A to B in t (assuming B is not already present in t).

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

We call *tuple location* a tuple tagged with its relation name, written (R,t). $TupleLoc = \mathcal{R} \times Tuple$ is the set of all tuple locations. A database instance I can equivalently be seen as a finite set of tuple locations $\{(R,t) | t \in I(R)\}$.

Similarly, a *field location*, or *cell*, refers to a particular field of a tagged tuple. Such a field is a triple $(R,t,A) \in \mathcal{R} \times Tuples \times U$. $FieldLoc$ is the name of the set of all field locations.

In our notation for relational algebra queries we use the selection σ_θ to filter a relation by retaining tuples satisfying some predicate θ. The form of predicates is left unspecified. Typically it includes quality tests $(A = B, A = d)$ or inequality tests $(A > d)$. We write $A \in \theta$ as a way to indicate that the attribute A is used in some test of θ.

Special Operators Taking inspiration from what is done in [2], we define attribute lineage as a function which maps a field location to a set of input field locations or to a special constant $\perp$, meaning *undefined*. In particular, we need to take into consideration two possible scenarios:

1. A cell has *empty* lineage provided it is present in the output but it was constructed by the query, e.g. using a constant expression.

2. A cell has *no* lineage provided it is not present in the output of the query.

The symbol $\emptyset$ denotes *empty lineage*, and $\perp$ denotes *no lineage* (or undefined).

The possibility of no lineage ($\perp$) means that we need to be careful when combining lineages in join, union and projection.

First, consider a query $Q_1 \bowtie Q_2$. A cell $c = (Q_1 \bowtie Q_2, t, A)$ has as lineage the union of set of attributes, as we shall see. These sets are computed from the information contained in

the tuples $t[U_1] \in Q_1$ and $t[U_2] \in Q_2$. However, if one of these two tuples is actually undefined ($\perp$), then c cannot be in the result, so its lineage should also be $\perp$.

We handle joins using a *strict union* operation [2], defined as follows:

$$\begin{aligned} \perp \cup_S X = \quad & X \cup_S \perp \quad = \perp \\ & X \cup_S Y \quad = X \cup Y \quad (X \neq \perp \neq Y) \end{aligned}$$

For the union operation, consider a query $Q_1 \cup Q_2$. If a tuple t is in both Q_1 and Q_2 then the lineage of $c = (Q_1 \cup Q_2, t, A)$ is the union of the lineages in the subqueries (and the union with other sets, as we shall show). If t is defined only in one subquery, then the lineage of c is derived only from the subquery in which t is defined. The lineage of c is undefined only if t is undefined in both subqueries. To handle this behavior, we use a *lazy union* operation [2]:

$$\begin{aligned} \perp \cup_L X = \quad & X \cup_L \perp \quad = X \\ & X \cup_L Y \quad = X \cup Y \quad (X \neq \perp \neq Y) \end{aligned}$$

For projection, if t is not in the query result, or if $A \notin U$, the lineage of the field location $c = (Q(I), t, A)$ is $\perp$. On the contrary, if multiple tuples t project to $t[U]$ and $A \in U$, then we want to combine their lineages in a lazy way. The lazy flattening operation is defined as follows:

$$\bigcup_L \emptyset = \perp$$

$$\bigcup_L \{X\} = X$$

$$\bigcup_L (X \cup Y) = \bigcup_L X \cup_L \bigcup_L Y$$

Similarly, the strict flattening operation is defined as follows:

$$\bigcup_S \emptyset = \perp$$

$$\bigcup_S \{X\} = \begin{cases} X & \text{if } X \neq \perp,\ \forall X \\ \perp & \text{otherwise} \end{cases}$$

$$\bigcup_S (X \cup Y) = \bigcup_S X \cup_S \bigcup_S Y$$

Sometimes we also write $\bigcup^L$ instead of $\bigcup_L$ when the subscript is already occupied by some condition. Now that we have adopted these auxiliary definitions, we are ready to define the lineage for one relational operator.

4 Attribute Lineage

In this section, an operational definition of attribute lineage for *one* relational operator is first given. Then, the same definition for a more complex query composed by a series of SPJRU queries is proposed. This is an operational definition due to the fact that the lineage is defined via annotation-propagation behavior of each relational operator independently.

We extend the definition of lineage given in [2] by taking into consideration the fact that attribute lineage needs to contain all the attributes *used* by a query. Each relational operator uses the attributes in a different way.

Definition 4.1. Attribute Lineage for a relational operator Given a database instance I, a query Q, a tuple $t{:}U \in Q(I)$ and a field location $c = (Q, t, A)$, $A \in V$, we say that the lineage of c, written as $Lin(Q, t, A)$, is an element of the set $\mathcal{P}(FieldLoc)$, defined as:

1. $Lin(\{u\}, t, A) = \begin{cases} \emptyset & \text{if } t = u \\ \perp & \text{otherwise} \end{cases}$
2. $Lin(R, t, A) = \begin{cases} \{(R, t, A)\} & \text{if } t \in I(R) \\ \perp & \text{otherwise} \end{cases}$
3. $Lin(\sigma_\theta(R), t, A) = \begin{cases} \bigcup_{B \in \theta}^S Lin(R, t, B) & \text{if } \theta(t) \\ \perp & \text{otherwise} \end{cases}$
4. $Lin(\pi_U(R), t, A) = \bigcup_{t' \in R(I), t'[U] = t}^L Lin(R, t', A)$
5. $Lin(\rho_{A \mapsto B}(R), t, C) = Lin(R, t[B \mapsto A], C[B \mapsto A])$
6. $Lin(R_1 \bowtie R_2, t, A) =$ $Lin(R_1, t[U_1], A') \cup_S Lin(R_2, t[U_2], A')$, where U_1, U_2 are the sets of attributes of R_1, R_2 respectively, and $A' \in U_1 \cap U_2$ (A' is the attribute used for the natural join).
7. $Lin(R_1 \cup R_2, t, A) = Lin(R_1, t, A) \cup_L Lin(R_2, t, A)$.

In the case $A \notin V$, then $Lin(Q(I), t, A) = \perp$.

This definition works for only one operator and represents the base definition for the lineage of an attribute. We now define the notion of lineage for an attribute in a view obtained by a combination of queries.

In the next definition, we use the where-provenance (defined in [1]) as an auxiliary function for our definition of Attribute Lineage. Here we report a compositional definition of where-provenance adapted from the one given in [2]. The difference is that while [2] considers all the attributes of an output tuple, in our definition we limit to consider only one attribute at the time.

Definition 4.2. Where-Provenance Given a database instance I, an SPJRU query Q', a tuple $t{:}U \in Q'(I)$ and an attribute $A \in U$, the where-provenance of the field location c in t is defined as follows:

1. $Where(\{u\}, t, A) = \begin{cases} \emptyset & \text{if } t = u \\ \perp & \text{otherwise} \end{cases}$
2. $Where(R, t, A) = \begin{cases} \{(R, t, A)\} & \text{if } t \in I(R) \\ \perp & \text{otherwise} \end{cases}$

3. $Where(\sigma_\theta(Q),t,A) = \begin{cases} Where(Q(I),t,A) & \text{if } \theta(t) \\ \bot & \text{otherwise} \end{cases}$

4. $Where(\pi_U(Q),t,A) = \bigcup^L_{c'=(Q,t',A)|t'[U]=t} Where(c')$.

5. $Where(\rho_{A\mapsto B}(Q),t,C[A \mapsto B]) = Where(Q,t[B \mapsto A],C[B \mapsto A])$.

6. $Where(Q_1 \bowtie Q_2,t,A) = Where(Q_1,t[U_1],A) \cup_S Where(Q_2,t[U_2],A)$.

7. $Where(Q_1 \cup Q_2,t,A) = Where(Q_1,t[U_1],A) \cup_L Where(Q_2,t[U_2],A)$.

If $A \notin U$, then $Where(Q',t,A) = \bot$.

Now, we can use this definition of where-provenance as a function in the next definition.

Definition 4.3. Attribute Lineage for arbitrary queries Given a database I, a SPJRU query Q' over I, a tuple $t{:}U \in Q(I)$ and an attribute location $c = (Q',t,A)$ with $A \in U$, then the *lineage* of c according to Q' and I, denoted as $Lin(Q',t,A)$ is an element of $\mathcal{P}(FieldLoc)$, defined as follows:

1. $Lin(\sigma_\theta(Q),t,A) =$
$= \begin{cases} \bigcup^S_{c'=(Q,t,B)|B\in\theta}(Where(c') \cup_S Lin(c')) & \text{if } \theta(t) \\ \bot & \text{otherwise} \end{cases}$

2. $Lin(\pi_U(Q),t,A) =$
$= \bigcup^L_{c'=(Q,t',A)|t'[U]=t} (Where(c') \cup_S Lin(c'))$

3. $Lin(\rho_{A\mapsto B}(Q),t,C) = Lin(Q(I),t[B \mapsto A],C[B \mapsto A]) \cup_S Where(Q,t[B \mapsto A],A)$.

4. $Lin(Q_1 \bowtie Q_2,t,A) =$
$Where(c_1) \cup_S Lin(c_1) \cup_S Where(c_2) \cup_S Lin(c_2)$, where $c_1 = (Q_1(I),t[U_1],A'), c_2 = (Q_2(I),t[U_2],A')$.

5. $Lin(Q_1 \cup Q_2,t,A) =$
$(Where(c_1) \cup_S Lin(c_1)) \cup_L (Where(c_2) \cup_S Lin(c_2))$, where $c_1 = (Q_1(I),t,A), c_2 = (Q_2(I),t,A)$.

If $A \notin U$, then $Lin(Q',t,A) = \bot$.

4.1 Relating the Two Lineages

We now show that attribute provenance contains more information than tuple-oriented provenance, and how the latter can be derived from the former.

Consider the algebraic structure defined in [2]:

$$K_{Lin} = \{\mathcal{P}(TupleLoc)_\bot, \bot, \emptyset, \cup_L, \cup_S\}$$

Consider also the following semiring structure defined for the attribute lineage:

$$K'_{Lin} = \{\mathcal{P}(FieldLoc)_\bot, \bot, \emptyset, \cup_L, \cup_S\}$$

Construct a homomorphism $h : K_{Lin} \mapsto K'_{Lin}$ such that:

$$\begin{array}{cc} h(\bot) = \bot & h(\emptyset) = \emptyset \\ h(x \cup_L y) = h(x) \cup_L h(y) & h(x \cup_S y) = h(x) \cup_S h(y) \\ h(\{(R,t,A)\}) = \{R,t\} & h(Where(c)) = \emptyset \ \forall c \end{array}$$

Note that the last operation maps an attribute location to the corresponding tuple location that contains it. This homomorphism simply discards the information provided by where-provenance.

We show in the next proposition that this function provides a mapping from the elements of the attribute lineage of a cell $t \bullet A$ to the elements of the tuple lineage of the tuple t. In this way, given the attribute lineage, it is always possible to build the corresponding tuple lineage.

Proposition 4.1. Let Q be an SPJRU query over the database instance I, $t{:}U \in Q(I)$, and the field location $c = (Q,t,A)$, with $A \in U$. Then $h(Lin(Q,t,A)) = Lin(Q,I,t)$.

Proof. The proof is done by induction over Q. We use the definitions of lineage for tuples given in [2]. We start from the base case, where Q is composed by only one relational operator:

1. $h(Lin(\{u\},t,A)) =$
$\begin{cases} h(\emptyset) & \text{if } t = u \\ h(\bot) & \text{otherwise} \end{cases} =$
$\begin{cases} \emptyset & \text{if } t = u \\ \bot & \text{otherwise} \end{cases} = Lin(Q,I,\{u\})$

2. $h(Lin(R,t,A)) =$
$\begin{cases} h(\{(R,t,A)\}) & \text{if } t \in I(R) \\ h(\bot) & \text{otherwise} \end{cases} =$
$\begin{cases} \{(R,t)\} & \text{if } t \in I(R) \\ \bot & \text{otherwise} \end{cases} = Lin(R,I,t)$

3. $h(Lin(\sigma_\theta(R),t,A)) =$
$\begin{cases} h(\bigcup^S_{B\in\theta} Lin(R,t,B)) & \text{if } \theta(t) \\ h(\bot) & \text{otherwise} \end{cases} =$
$\begin{cases} \bigcup^S_{B\in\theta} h(Lin(R,t,B)) & \text{if } \theta(t) \\ \bot & \text{otherwise} \end{cases} =$
$\begin{cases} \bigcup^S_{B\in\theta} Lin(Q,I,t) & \text{if } \theta(t) \\ \bot & \text{otherwise} \end{cases} =$
$\begin{cases} Lin(Q,I,t) & \text{if } \theta(t) \\ \bot & \text{otherwise} \end{cases} = Lin(\sigma_\theta(R),t,A)$

4. $h(Lin(\pi_U(R),t,A)) =$
$h(\bigcup^L_{t'\in R(I),t'[U]=t} Lin(R,t',A)) =$
$\bigcup^L_{t'\in R(I),t'[U]=t} h(Lin(R,t',A)) =$
$\bigcup^L_{t'\in R(I),t'[U]=t} Lin(R,I,t') = Lin(\pi_U(R),I,t)$

5. $h(Lin(\rho_{A\mapsto B}(R),t,C)) =$
$h(Lin(R,t[B\mapsto A],C[B\mapsto A])) =$
$Lin(R,I,t[B\mapsto A]) = Lin(\rho_{A\mapsto B},I,t)$

6. $h(Lin(R_1 \bowtie R_2,t,A)) =$
$h(Lin(R_1,t[U_1],A')) \cup_S h(Lin(R_2,t[U_2],A')) =$
$Lin(R_2,t[U_1]) \cup_S Lin(R_2,t[U_2]) = Lin(R_1 \bowtie R_2,I,t)$

7. $h(Lin(R_1 \cup R_2,t,A)) =$
$h(Lin(R_1,t,A)) \cup_L h(Lin(R_2,t,A)) =$
$= Lin(R_1,I,t) \cup_L Lin(R_2,I,t) = Lin(R_1 \cup R_2,I,t)$

Where points 3, 4, 5, 6, and 6 use point 2. We can now go to the inductive step:

1. $h(Lin(\sigma_\theta(Q),t,A)) =$
$\begin{cases} h(\bigcup^S_{c'=(Q,t,B)\mid B\in\theta}(Where(c') \cup_S Lin(c')) & \text{if } \theta(t) \\ h(\bot) & \text{otherwise} \end{cases} =$
$\begin{cases} \bigcup^S_{c'=(Q,t,B)\mid B\in\theta} h(Where(c') \cup_S h(Lin(c') & \text{if } \theta(t) \\ \bot & \text{otherwise} \end{cases} =$
$\begin{cases} \bigcup^S_{c'=(Q,t,B)\mid B\in\theta} \emptyset \cup_S Lin(Q,I,t) & \text{if } \theta(t) \\ \bot & \text{otherwise} \end{cases} =$
$\begin{cases} Lin(Q,I,t) & \text{if } \theta(t) \\ \bot & \text{otherwise} \end{cases} = Lin(\sigma_\theta(Q),I,t)$

2. $h(Lin(\pi_U(Q),t,A)) =$
$h(\bigcup^L_{c'=(Q,t',A)\mid t'[U]=t}(Where(c') \cup_S Lin(c')) =$
$\bigcup^L_{c'=(Q,t',A)\mid t'[U]=t}(h(Where(c') \cup_S h(Lin(c')) =$
$\bigcup^L_{t'\in Q(U)\mid t'[U]=t} Lin(Q,I,t') =$
$Lin(\pi_U(Q),I,t)$

3. $h(Lin(\rho_{A\mapsto B}(Q),t,C) =$
$h(Lin(Q,t[B \mapsto A],C[B \mapsto A]) \cup_S Where(Q,t[B \mapsto A],A)) =$
$Lin(Q,R,t[B\mapsto A]) =$
$Lin(\rho_{A\mapsto B},I,t)$

4. $h(Lin(Q_1 \bowtie Q_2,t,A)) =$
$h(Where(Q_1,t[U_1],A')) \cup_S h(Lin(Q_1,t[U_1],A')) \cup_S$
$h(Where(Q_2,t[U_2],A')) \cup_S h(Lin(Q_2,t[U_2],A')) =$
$\emptyset \cup_S Lin(Q_1,I,t[U_1]) \cup_S \emptyset \cup_S Lin(Q_2,I,t[U_2]) =$
$Lin(Q_1 \bowtie Q_2,I,t)$

5. $h(Lin(Q_1 \cup Q_2,t,A)) =$
$[h(Where(Q_1(I),t,A)) \cup_S h(Lin(Q_1(I),t,A))] \cup_L$
$[h(Where(Q_2(I),t,A)) \cup_S h(Lin(Q_2(I),t,A))] =$
$= \emptyset \cup_L Lin(Q_1,I,t) \cup_L \emptyset \cup_L Lin(Q_1,I,t) =$
$Lin(Q_1 \cup Q_2,I,t)$

□

5 Conclusions and Future Work

In this paper we presented attribute lineage, a new kind of provenance for data at the attribute level that enables users to know which cells in the input database were used to produce an output cell. Attribute lineage is more informative than tuple-oriented lineage since it not only gives information about which tuples are relevant to the creation of the cell but also which attributes inside these tuples were used in the query. We also gave a homomorphism between attribute lineage and tuple lineage, showing that the former is more general than the letter.

In future work, we will follow two directions. first, building on the ideas in this paper for attribute lineage (where-provenance), we will propose two *new attribute provenances*: *attribute why-provenance* and *attribute how-provenance*. We will show how these provenances are respectively more general than their tuple-based counterparts, and how the attribute how-provenance is more general than both the attribute why-provenance and the attribute lineage. Second, we will study the computational complexity of computing attribute provenance, considering both time and space. To do so, we will consider synthetic datasets of different sizes and compute attribute provenances of tuples produced by different queries. In this way, we will record the required time to compute these lineages and the space required to store them.

References

[1] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *Database Theory - ICDT 2001, 8th International Conference*, pages 316–330, 2001.

[2] J. Cheney, L. Chiticariu, and W.C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[3] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[4] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40. ACM, 2007.

[5] C. A. Lynch. When documents deceive: Trust and provenance as new factors for information retrieval in a tangled web. *Journal of the American Society for Information Science and Technology*, 52(1):12–17, 2001.

[6] Y. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.

Appendix

A Example

In this section we report an example to describe how attribute lineage works. Figure 1 reports a simple database composed by two tables, originally taken from [2]. The first table is called *Agencies*, the second one *External Tours*. Every tuple and every cell present a unique identifier.

tId	Name	BasedIn	phone
a_1	BayTours ($a_{1,1}$)	San Francisco ($a_{1,2}$)	415-1200 ($a_{1,3}$)
a_2	HarborCruz ($a_{2,1}$)	Santa Cruz ($a_{2,2}$)	831-3000 ($a_{2,3}$)

tId	Name	Destination	type	price
e_1	BayTours ($e_{1,1}$)	San Francisco ($e_{1,2}$)	car ($e_{1,3}$)	50 ($e_{1,4}$)
e_2	BayTours ($e_{2,1}$)	Santa Cruz ($e_{2,2}$)	bus ($e_{2,3}$)	100 ($e_{2,4}$)
e_3	BayTours ($e_{3,1}$)	Santa Cruz ($e_{3,2}$)	boat ($e_{3,3}$)	250 ($e_{3,4}$)
e_4	BayTours ($e_{4,1}$)	Monterey ($e_{4,2}$)	boat ($e_{4,3}$)	400 ($e_{4,4}$)
e_5	HarborCruz ($e_{5,1}$)	Monterey ($e_{5,2}$)	boat ($e_{5,3}$)	200 ($e_{5,4}$)
e_6	HarborCruz ($e_{6,1}$)	Carmel ($e_{6,2}$)	train ($e_{6,3}$)	90 ($e_{6,4}$)

Figure 1: A simple online travel database composed by two relations: Agencies (above) and ExternalTours (below). Every tuple and attribute is uniquely identified by a unique id.

Consider now Figure 2. The query Q_1 asks for all the agencies in the database that perform boat tours and their phone numbers. The output of the query, called R_3, is also shown in Figure 2. We report two tables. The first one, table a, is the one actually obtained from the query Q_1. The second, table b, is obtained if we do not use the `DISTINCT` operator.

We focus on the first tuple of table a, o_1, and on the value of the attribute *phone*, $415-1200$. We call this field location $o_{1,2}$. Its coordinates are $o_{1,2} = (R_3, o_1, phone)$. Note that, if we use set semantic, $o_{1,2}$ is actually the product of a merge between two other attributes, highlighted in table b, due to the projection. We want to know the attribute lineage of this attribute.

To do this, we consider, step by step, the different operations performed by the query, going backward from the output relation R_3.

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

$R_3 = \pi_{Name,Phone}(R_2)$

tId	Name	Phone
o_1	BayTours ($o_{1,1}$)	415-1200 ($o_{1,2}$)
o_2	HarborCruz ($o_{2,1}$)	831-3000 ($o_{2,2}$)

a

tId	Name	Phone
o_1	BayTours ($o_{1,1}$)	415-1200 ($o_{1,2}$)
o'_1	BayTours ($o'_{1,1}$)	415-1200 ($o'_{1,2}$)
o_2	HarborCruz ($o_{2,1}$)	831-3000 ($o_{2,2}$)

b

Figure 2: The query Q_1 and its result, table R_3 – represented with and without the set semantics – with identifiers for each cell.

Considering Q_1, we can decompose it in three operations: join, selection and projection, performed in this order. Starting backward, R_3 is the output of the last operation, a projection, performed on a table R_2, presented in Figure 3.

$R_2 = \sigma_{type='boat'}(R_1)$

tId	Name	BasedIn	phone	Destination	type	price
$\dot{o}_1$	BayTours ($a_{1,1}, e_{3,1}$)	San Francisco ($a_{1,2}$)	415-1200 ($a_{1,3}$)	Santa Cruz ($e_{3,2}$)	boat ($e_{3,3}$)	250 ($e_{3,4}$)
$\dot{o}_2$	BayTours ($a_{1,1}, e_{4,1}$)	San Francisco ($a_{1,2}$)	415-1200 ($a_{1,3}$)	Monterey ($e_{4,2}$)	boat ($e_{4,3}$)	400 ($e_{4,4}$)
$\dot{o}_3$	HarborCruz ($a_{2,1}, e_{5,1}$)	Santa Cruz ($a_{2,2}$)	831-3000 ($a_{2,3}$)	Monterey ($e_{5,2}$)	boat ($e_{5,3}$)	200 ($e_{5,4}$)

Figure 3: Intermediate step reporting R_2, before the projection and after the selection. Every cell is annotated with its where-provenance.

Now the question is "how did the cell $o_{1,2} = (R_3, o_1, phone)$ end up in R_3?". This was possible thanks to the presence of the two cells highlighted in Figure 3. The projection was performed on the attributes *name* and *phone*, but since we are asking the lineage of $o_{1,2}$, only the attribute phone is relevant. By definition, the lineage of $o_{1,2}$ is given by the union of the lineages of the two cells highlighted in R_2 in purple with the where-provenance of those same two cells, as per formula reported in Figure 4.

$$\begin{aligned} Lineage(R_3, o_1, Phone) &= Where(R_2, \ddot{o}_1, Phone) \cup Lineage(R_2, \ddot{o}_1, Phone) \\ &\cup Where(R_2, \ddot{o}_2, Phone) \cup Lineage(R_2, \ddot{o}_2, Phone) \\ &= \{a_{1,3},\} \cup Lineage(R_2, \ddot{o}_1, Phone) \cup \{a_{1,3}\} \cup Lineage(R_2, \ddot{o}_2, Phone) \end{aligned}$$

Figure 4: Formulas for the lineage of $o_{1,2}$ computed at the level of the table R_3.

The where-provenance tells us which cells from I are used, while the recursive application on the lineage formula is necessary to unfold the whole process that brought $o_{1,2}$ in its place. We call the two used cells $c_1 = (R_2, \ddot{o}_1, Phone)$ and $c_2 = (R_2, \ddot{o}_2, Phone)$. We now proceed recursively for the lineage of these two.

$R_1 = Agencies \bowtie ExternalTours$

tId	Name	BasedIn	phone	Destination	type	price
$\dot{o}_1$	BayTours ($a_{1,1}; e_{1,1}$)	San Francisco ($a_{1,2}$)	415-1200 ($a_{1,3}$)	SanFrancisco ($e_{1,2}$)	car ($e_{1,3}$)	50 ($e_{1,4}$)
$\dot{o}_2$	BayTours ($a_{1,1}; e_{2,1}$)	San Francisco ($a_{1,2}$)	415-1200 ($a_{1,3}$)	Santa Cruz ($e_{2,2}$)	bus ($e_{2,3}$)	100 ($e_{2,4}$)
$\dot{o}_3$	BayTours ($a_{1,1}; e_{3,1}$)	San Francisco ($a_{1,2}$)	415-1200 ($a_{1,3}$)	Santa Cruz ($e_{3,2}$)	boat ($e_{3,3}$)	250 ($e_{3,4}$)
$\dot{o}_4$	BayTours ($a_{1,1}; e_{4,1}$)	San Francisco ($a_{1,2}$)	415-1200 ($a_{1,3}$)	Monterey ($e_{4,2}$)	boat ($e_{4,3}$)	400 ($e_{4,4}$)
$\dot{o}_5$	HarborCruz ($a_{2,1}; e_{5,1}$)	Santa Cruz ($a_{2,2}$)	831-3000 ($a_{2,3}$)	Monterey ($e_{5,2}$)	boat ($e_{5,3}$)	200 ($e_{5,4}$)
$\dot{o}_6$	HarborCruz ($a_{2,1}; e_{6,1}$)	Santa Cruz ($a_{2,2}$)	831-3000 ($a_{2,3}$)	Carmel ($e_{6,2}$)	train ($e_{6,3}$)	90 ($e_{6,4}$)

$$\begin{aligned} Lineage(R_2, \ddot{o}_1, Phone) &= Where(R_1, \dot{o}_3, type) \cup Lineage(R_1, \dot{o}_3, type) \\ &= \{e_{3,3},\} \cup Lineage(R_1, \dot{o}_3, type) \end{aligned}$$

Figure 5: Table R_1 obtained after the join, and the formula for the attribute lineage of $(R_2, \ddot{o}_1, type)$.

Focus now only on c_1, the purple cell (the lineage of c_2 is computed similarly). In Figure 5 we represent R_1, the table from which R_2 is created through the selection. "How that cell was able to end up in R_2?" It was because the previous operation was a selection on the attribute *type*. In particular, the cell that guaranteed the presence of c_1 was $c_3 = (R_1, \dot{o}_3, type)$ (the yellow one in Figure 5). The lineage of c_1 is therefore given by the why-provenance of c_3 united to its lineage, as reported in the formula of Figure 5.

The last step of the computation is represented in Figure 6. Here we are asking how c_3 was able to be present in R_1. It was because the cells $a_{1,1}$ and $e_{3,1}$, highlighted in the figure, were used in the natural join. The lineage of c_3 is therefore the union of the identifier of these two cells. Here we reached

tId	Name	BasedIn	phone
a_1	BayTours ($a_{1,1}$)	San Francisco ($a_{1,2}$)	415-1200 ($a_{1,3}$)
a_2	HarborCruz ($a_{2,1}$)	Santa Cruz ($a_{2,2}$)	831-3000 ($a_{2,3}$)

tId	Name	Destination	type	price
e_1	BayTours ($e_{1,1}$)	San Francisco ($e_{1,2}$)	car ($e_{1,3}$)	50 ($e_{1,4}$)
e_2	BayTours ($e_{2,1}$)	Santa Cruz ($e_{2,2}$)	bus ($e_{2,3}$)	100 ($e_{2,4}$)
e_3	BayTours ($e_{3,1}$)	Santa Cruz ($e_{3,2}$)	boat ($e_{3,3}$)	250 ($e_{3,4}$)
e_4	BayTours ($e_{4,1}$)	Monterey ($e_{4,2}$)	boat ($e_{4,3}$)	400 ($e_{4,4}$)
e_5	HarborCruz ($e_{5,1}$)	Monterey ($e_{5,2}$)	boat ($e_{5,3}$)	200 ($e_{5,4}$)
e_6	HarborCruz ($e_{6,1}$)	Carmel ($e_{6,2}$)	train ($e_{6,3}$)	90 ($e_{6,4}$)

$$Lineage(R_1, \dot{o}_3, type) = \{Activity, a_1, Name\} \cup \{ExternalTours, e_3, Name\} = \{a_{1,1}, e_{3,1}\}$$

Figure 6: The tables Activity and ExternalTours, before the join, and the formula for the lineage of $(R_1, \dot{o}_1, type)$.

the base case, with no more lineages to unfold. These sets thus discovered can be substituted on the previous equations, going up the recursive tree to compute the original lineage of c_0.

It is easy to see how the lineage of $o_{1,2}$ is thus the set $\{a_{1,1}, a_{1,3}, e_{3,1}, e_{3,3}, e_{4,1}, e_{4,3}\}$. With similar computations, it can be found that the attribute $o_{1,1}$ has attribute lineage $\{a_{1,1}, e_{3,1}, e_{3,3}, e_{4,1}, e_{4,3}\}$. The two lineages are different: the second one does not contain $a_{1,3}$, since it is never used by the projection to obtain $o_{1,1}$.

It is also easy to see how the lineage of the tuple o_1, where $o_{1,2}$ is contained, is the set $\{a_1, e_3, e_4\}$. As we can see, all the attributes in the attribute lineage belong to one tuple of the tuple lineage. A surjective function can be defined from the attribute lineage to the tuple lineage simply discarding the information about the column. Thus, intuitively, the attribute lineage "covers" the tuple lineage, since it only contains cells from tuples of the tuple lineage, and all the tuples of the tuple lineage are represented. This intuition is actually confirmed by Proposition 4.1.