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 With the advent of the Internet, this assumption is no longer valid [5].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 diagnostic 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


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 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 \cdot A is the set of cells in I from which t \cdot 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 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 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 \cdot 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 \mathcal{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 instance I is a function I : (R_1 : U_1, \ldots, R_n : U_n) mapping each R_i : U_i \in \mathcal{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) \mid 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). We call FieldLoc = \mathcal{R} \times Tuple \times U. FieldLoc is the name of the set of all field locations.

In our notation for relational algebra queries we use the selection \sigma_\theta to filter a relation by retaining tuples satisfying some predicate \theta. 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 \theta.

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 \bot, 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 \bot denotes no lineage (or undefined).

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

First, consider a query Q_1 \times Q_2. A cell c = (Q_1 \times 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 ($\bot$), then $c$ cannot be in the result, so its lineage should also be $\bot$.

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

$$\bot \cup_L X = \begin{cases} X \cup_L \bot & = \bot \\ X \cup_L Y & = X \cup Y \ (X \neq \bot \neq Y) \end{cases}$$

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]:

$$\bot \cup_S X = \begin{cases} X \cup_S \bot & = \bot \\ X \cup_S Y & = X \cup Y \ (X \neq \bot \neq Y) \end{cases}$$

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 $\bot$. 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:

$$\cup_L \emptyset = \bot$$

$$\cup_L \{X\} = X$$

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

Similarly, the strict flattening operation is defined as follows:

$$\cup_S \emptyset = \bot$$

$$\cup_S \{X\} = \begin{cases} \begin{cases} X \quad & \text{if } X \neq \bot, \forall X \\
\bot \quad & \text{otherwise} \end{cases} \\ \bot \quad \text{otherwise} \end{cases}$$

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

Sometimes we also write $\cup^L$ instead of $\cup_L$ 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. $Lin(\{u\}, t, A) = \begin{cases} \emptyset & \text{if } t = u \\
\bot & \text{otherwise} \end{cases}$

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

3. $Lin(\sigma_\theta(R(t), A)) = \begin{cases} \cup_{B \in \theta} Lin(R(t), B) & \text{if } \theta(t) \\
\bot & \text{otherwise} \end{cases}$

4. $Lin(\pi_U(R(t), A)) = \cup_{t' \in I(R)} Lin(R(t'), A)$

5. $Lin(\rho_{A \rightarrow B}(R(t), C)) = Lin(R(t), B \rightarrow A), C | B \rightarrow A)$

6. $Lin(R_1 \times R_2, t, A) = Lin(R_1, t[U], A') \cup_S Lin(R_2, t[U], A')$, 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 $Lin(Q(I), t, A) = \bot$.

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. $Where(\{u\}, t, A) = \begin{cases} \emptyset & \text{if } t = u \\
\bot & \text{otherwise} \end{cases}$

2. $Where(R(t), A) = \begin{cases} \{\langle (R(t), A) \rangle \} & \text{if } t \in I(R) \\
\bot & \text{otherwise} \end{cases}$
3. Where(σ₀(Q), t, A) = Where(Q(I), t, A) if θ(t) otherwise

4. Where(πₜ(U), t, A) = \bigcup_{c=(Q', t, A) | U = Where(c')} Where(c')

5. Where(ρₜ→ₜₚ(Q), t, C[A \rightarrow B]) = Where(Q, t[B \rightarrow A], C[B \rightarrow A])

6. Where(\prod_{t \in A \rightarrow B} A)(Q, t, A) = Where(Q, t[U][A] \cup_s Where(Q, t[U][A])

7. Where(\prod_{t \in A \rightarrow B} A)(Q, t, A) = Where(Q, t[U][A] \cup_s Where(Q, t[U][A])

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

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 base case, where Q is composed by only one relational operator:

1. h(\prod_{t \in A \rightarrow B} A)(Q, t, A) = h(\prod_{t \in A \rightarrow B} A)(Q, t, A)

2. h(\piₜ(U), t, A) = h(\piₜ(U), t, A)

3. h(\piₜ(U), t, A) = h(\piₜ(U), t, A)

4. h(\prod_{t \in A \rightarrow B} A)(Q, t, A) = h(\prod_{t \in A \rightarrow B} A)(Q, t, A)

5. h(\prod_{t \in A \rightarrow B} A)(Q, t, A) = h(\prod_{t \in A \rightarrow B} A)(Q, t, A)

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

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]:

\[ K_{Lin} = \{ \mathcal{P}(TupleLoc) \}_1^\perp, 1, 0, \cup_L, \cup_S \} \]

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

\[ K'_{Lin} = \{ \mathcal{P}(FieldLoc) \}_1^\perp, 1, 0, \cup_L, \cup_S \} \]

Construct a homomorphism \( h : K_{Lin} \rightarrow K'_{Lin} \) such that:

\[
\begin{align*}
    h(\perp) &= \perp \\
    h(\emptyset) &= \emptyset \\
    h(x \cup_L y) &= h(x) \cup_L h(y) \\
    h(x \cup_S y) &= h(x) \cup_S h(y) \\
    h(\{ (R, t, A) \}) &= \{ (R, t) \} \\
    h(\text{Where}(c)) &= \emptyset \forall c
\end{align*}
\]

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 \( t \cdot A \) 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 \( h(Lin(Q, t, A)) = Lin(Q, t, I) \).

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:
5. \( h(\text{Lin}(\rho_{A \rightarrow B}(R), t, C)) = \)
\[
\begin{align*}
&h(\text{Lin}(R, t[B \mapsto A], C[B \mapsto A])) = \\
&\text{Lin}(R, t[B \mapsto A]) = \text{Lin}(\rho_{A \rightarrow B}, I, t)
\end{align*}
\]

6. \( h(\text{Lin}(R_1 \times R_2, t, A)) = \)
\[
\begin{align*}
&h(\text{Lin}(R_1, t[U_1], A')) \cup_S h(\text{Lin}(R_2, t[U_2], A')) = \\
&\text{Lin}(R_1 \times R_2, t[U_2]) = \text{Lin}(R_1 \times R_2, I, t)
\end{align*}
\]

7. \( h(\text{Lin}(R_1 \cup R_2, t, A)) = \)
\[
\begin{align*}
&h(\text{Lin}(R_1, t, A)) \cup_L h(\text{Lin}(R_2, t, A)) = \\
&\text{Lin}(R_1 \cup R_2, I, t) = \text{Lin}(R_1 \cup R_2, I, t)
\end{align*}
\]

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

1. \( h(\text{Lin}(\sigma_{\theta}(Q), t, A)) = \)
\[
\begin{align*}
&\begin{cases} 
\bigcup_{c' \in (Q \cup t)' \cap Q} (\text{Lin}(c')) & \text{if } \theta(t) \\
\bot & \text{otherwise}
\end{cases} \\
&h(\bigcup_{c' \in (Q \cup t)' \cap Q} (\text{Lin}(c'))) = \text{Lin}(\sigma_{\theta} Q), I, t)
\end{align*}
\]

2. \( h(\text{Lin}(\pi_{\theta}(Q), t, A)) = \)
\[
\begin{align*}
&\begin{cases} 
\bigcup_{c' \in (Q \cup t)' \cap Q} (\text{Lin}(c')) & \text{if } \theta(t) \\
\bot & \text{otherwise}
\end{cases} \\
&h(\bigcup_{c' \in (Q \cup t)' \cap Q} (\text{Lin}(c'))) = \text{Lin}(\pi_{\theta} Q), I, t)
\end{align*}
\]

3. \( h(\text{Lin}(\rho_{A \rightarrow B}(Q), t, C) = \)
\[
\begin{align*}
&\text{Lin}(Q, t[B \mapsto A], C[B \mapsto A]) \cup_S \text{Where}(Q, t[B \mapsto A], A') = \\
&\text{Lin}(Q, R, t[B \mapsto A]) = \text{Lin}(\rho_{A \rightarrow B}(Q), I, t)
\end{align*}
\]

4. \( h(\text{Lin}(Q_1 \times Q_2, t, A)) = \)
\[
\begin{align*}
&h(\text{Where}(Q_1, t[U_1], A')) \cup_S h(\text{Lin}(Q_1, t[U_1], A')) \cup_S \\
&h(\text{Where}(Q_2, t[U_2], A')) \cup_S h(\text{Lin}(Q_2, t[U_2], A')) = \\
&\emptyset \cup_S \text{Lin}(Q_1, t[U_1]) \cup_S \emptyset \cup_S \text{Lin}(Q_2, t[U_2]) = \\
&\text{Lin}(Q_1 \times Q_2, I, t)
\end{align*}
\]

5. \( h(\text{Lin}(Q_1 \cup Q_2, t, A)) = \)
\[
\begin{align*}
&\begin{cases} 
\bigcup_{c' \in (Q_1 \cup t)' \cap Q} (\text{Lin}(c')) & \text{if } \theta(t) \\
\text{Lin}(Q_1 \cup Q_2, I, t) & \text{otherwise}
\end{cases} \\
&\bigcup_{c' \in (Q_1 \cup t)' \cap Q} (\text{Lin}(c')) = \text{Lin}(Q_1 \cup Q_2, I, t)
\end{align*}
\]

---

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 latter.

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


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 ExternalTours. Every tuple and every cell present a unique identifier.

![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.](image)

Consider now Figure 2. The query Q1 asks for all the agencies in the database that perform boat tours and their phone numbers. The output of the query, called R3, is also shown in Figure 2. We report two tables. The first one, table a, is the one actually obtained from the query Q1. The second, table b, is obtained if we do not use the DISTINCT operator.

![Figure 2: The query Q1 and its result, table R3 – represented with and without the set semantics – with identifiers for each cell.](image)

To do this, we consider, step by step, the different operations performed by the query, going backward from the output relation R3.

\[
Q_1 : \text{SELECT DISTINCT a.name, a.phone} \\
\text{FROM Agencies a, ExternalTours e} \\
\text{WHERE a.name = e.name AND e.type = 'boat'}
\]

Figure 3: Intermediate step reporting R2, 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_{1,1}, \text{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.

![Figure 4: Formulas for the lineage of o_{1,2} computed at the level of the table R3.](image)

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, \bar{o}_{1,1}, \text{Phone}) and c_2 = (R_2, \bar{o}_{2,1}, \text{Phone}). We now proceed recursively for the lineage of these two.

![Figure 5: Table R1 obtained after the join, and the formula for the attribute lineage of (R_2, \bar{o}_{1,1}, \text{type}).](image)

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_3 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, o_{1,1}, \text{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 o_{1,1} and o_{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
It is easy to see how the lineage of \( o_{1,2} \) is thus the set \( \{a_{1,1}, a_{1,3}, e_{3,3}, e_{4,1}, e_{4,3}\} \). 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.