Formal Foundation of Contribution Semantics and Provenance Computation through Query Rewrite in TRAMP

Boris Glavic

March 8, 2010

Contents

Chapter 1

Introduction

In this report we present the theoretical foundation of *TRAMP*. *TRAMP* is a schema mapping debugging system that uses provides provenance and query support as debugging functality for schema mappings scenarios. *TRAMP* is an extension of *Perm*, a relational provenance management system developed at University of Zurich. In this report we are not focussing on the debugging functionality added by *TRAMP*, but instead focus on the theoretical foundation of the provenance types provided by the system. In chapter 2 we present the contribution semantics for *data* provenance, *transformation* provenance, and *mapping* provenance used by *TRAMP*. Contribution semantics define which parts of the input (in case of *data* provenance) and which operators of a transformation (in case of *transformation* provenance) belong to the provenance of an output of a transformation. Thus, contribution semantics define "what provenance actually is". Based on the presented contribution semantics we demonstrate in chapter 3 how provenance according to these provenance types can be computed using algebraic rewrite techniques and proof the correctness and completeness of the algorithms used to compute provenance.

Chapter 2

Contribution Semantics

In this chapter we formally define *Perm-Influence-Contribution-Semantics* (*PI-CS*) the *contribution semantics* developed for the *Perm* system and prove several important properties of the provenance generated by this *CS* type. *PI-CS* is a type of *I-CS* based on *Lineage-CS*. We decided to develop our own type of *CS*, because the representation used by *Lineage-CS* (and also other *CS* types) is not suited for our approach to implement a "purely relational" provenance management system. Furthermore, as we will demonstrate in this chapter, *Lineage-CS*, in contrast to *PI-CS*, does not extend to queries with nested sub-queries (*sublinks*) and queries with negation. We define *transformation* provenance *CS* for the use in *TRAMP* as extensions of *PI-CS*.

First, we introduce an extended relational algebra which allows for a natural algebraic representation of SQL queries. Afterwards, we introduce *PI-CS*, demonstrate its applicability to the operators of the algebra, and study the relationship between this *CS* type and *Lineage-CS*. Finally, we present *transformation* provenance. Note that in this chapter we are only discussing the semantics of provenance and do not develop algorithms for generating provenance according to this semantics. Provenance computation is discussed in chapter 3.

2.1 Algebra Definition

In this section we introduce notational preliminaries and the relational algebra that are needed for the theoretical foundation of *Perm*. The algebra is defined in such a way that SQL queries have natural counterpart algebra expressions and it is easy to translate between the SQL and algebra representation of a query. This property is important, because it is not feasible to build a formal framework of provenance based on SQL, but, to be able to integrate provenance computation into a DBMS, the results established for algebra expressions have to be translated to SQL. As usual a relational database D is modeled as a *database schema* S and a *database instance* I. A database schema is a set of relation schemas: $S = \{\mathbf{R_1}, \ldots, \mathbf{R_n}\}$. Each relation schema is a function from a finite set $A \subset \mathscr{A}$ to the set of attribute domains $\mathscr{D}$ (we refer to the elements of this set as *data types*) where $\mathscr{A}$ is the set of possible attribute names. $\mathscr{S}$ is used to denote the set of all possible relation schemas. Every attribute domain is expected to contain the special value null: ε. Each relation schema is associated with a name by a function $Name : \mathscr{S} \rightarrow \mathscr{N}$ that assigns each relation schema in a database schema to an unique name. We assume a total order on the attribute names of a relation schema. I.e., a function $pos_{\mathbf{R}} : A \rightarrow \mathbb{N}$ that assigns each attribute from schema $\mathbf{R}$ to a unique position from the set $1, \ldots, |A|$. We use $\mathbf{R}(a_1 : d_1, \ldots, a_n : d_n)$ as a notational shortcut for a relation schema with attributes a_1 to a_n, name R, attribute order $a_1 : 1, \ldots, a_n : n$, and domains $d_1, \ldots, d_n$.

A *database instance* $I = \{R_1, \ldots, R_n\}$ of a database schema S is a set of *relations* that contains one relation R for each relational schema $\mathbf{R}$ in S. A relation R for a relation schema $\mathbf{R}$ is a subset of $RI = d_1 \times \ldots \times d_n$ and a function $mult : RI \rightarrow \mathbb{N}$. Each element $t = (v_1, \ldots, v_n)$ in RI is called a *tuple* and the value m assigned by *mult* to t is called the multiplicity of t. This means we are using the so-called *bag-* or *multiset-*semantics where every tuple is allowed to occur more than once in a relation. The elements $v_1, \ldots, v_n$ of a tuple are called attribute values. For convenience we use t^m to denote a tuple t with multiplicity m and t as

a shortcut for t[1].

If q is an algebra expression, then $\mathbf{Q}$ denotes the schema of the result relation produced by evaluating q[1]. We use $[[q]](I)$ to denote the result of evaluating algebra expression q over the database instance I. The database instance is omitted if it is clear from the context or irrelevant to the discussion. Q is used as a shortcut for $[[q]]$. The *Perm* algebra includes the standard operators of relational algebra. The evaluations of all algebra operators is presented in Figure 2.1. To simplify the definition of some operators negative or zero multiplicities of tuples indicate that a tuple does not belong to a relation.

Nullary Operators: A relation access is denoted by the name of the accessed relation. We allow for construction of singleton relations containing only a constant tuple t denoted by t.

Unary Operators: **Duplicate removal** $\delta(q_1)$ eliminates duplicates from its input (in other words it sets the multiplicity of every tuple to one). **Selection** $\sigma_C(q_1)$ returns all tuples t from Q_1 that fulfill the selection condition C (written as $t \models C$). A selection condition is an expression build from attributes, constants, comparisons (e.g., equality, less than, ...), function calls, and logical operators ($\neg, \wedge, \vee, \ldots$). C is restricted to return a boolean result. In addition we allow for conditional expressions: $if\ (e_1)\ then\ (e_2)\ else\ (e_3)$ evaluates to e_2 if e_1 evaluates to true. Otherwise it evaluates to e_3. This is similar to the *CASE* construct in SQL. The algebra defines two versions of **Projection**. One duplicate preserving version (Π^B: the superscript B stands for *bag*) and one duplicate removing version (Π^S: the superscript S stands for *set*). The duplicate preserving version ${\Pi^B}_A(q_1)$ returns the results of evaluating all projection expressions from the list $A = (a_1, \ldots, a_m)$ for each tuple in Q_1. Projection expressions are similar to selection conditions except that they are not restricted to return a boolean result and that the outermost construct in a projection expression can be a renaming $e \rightarrow a$ that causes the attribute which stores expression e to be named a in the result schema. The duplicate removing version of projection (${\Pi^S}_A(q_1)$) is defined as the application of the duplicate removal operator to the result of the duplicate preserving projection. Sometimes we use Π to denote the duplicate preserving version of projection. **Aggregation** $\alpha_{G,agg}$ groups its input on a list of group-by attributes and computes the aggregation functions from the list agg of aggregation functions for each group. One output tuple is produced for each group that contains the values of the group-by attributes for this group and the results of the aggregation functions [2]. In the definition presented in Figure 2.1 agg_i is one aggregation function from the list agg and B_i is the attribute used as input to aggregation function agg_i [3].

Join Operators: The *Perm* algebra includes several join operators. The **Cross product** $q_1 \times q_2$ is defined as in standard relational algebra. In the definition (t_1, t_2) denotes the concatenation of tuples t_1 and t_2. **Inner Join** $q_1 \bowtie_C q_2$ is a shortcut for applying a selection with condition C to the result of the cross product between q_1 and q_2. Three outer join types are defined in the algebra: **Left outer join** (⟕), **Right outer join** (⟖), and **Full outer join** (⟗). The outer join types are based on the inner join, but preserve tuples that are not joined with any other tuple. As the names indicate left outer join preserves only tuples from its left input, right outer join preserves only tuples from its right input, and full outer join preserves tuples from both inputs. $null(\mathbf{Q})$ denotes a tuple with schema $\mathbf{Q}$ and all attributes values set to *null*.

Set Operators: The algebra supports the three standard set operations **union** ($\cup$), **intersection** ($\cap$), and **set difference** ($-$). Like the projection operator, set operations are provided as a duplicate preserving and duplicate removing version (denoted by S and B).

[1] Defining $\mathbf{Q}$ independent of an database instance is valid, because the result schema of an algebra expression only depends on the database schema over which it is defined.

[2] We define the semantics of the standard aggregation functions $sum, avg, count, \ldots$ as in SQL. I. e., applying count to an empty relation returns zero and applying the other aggregation functions to an empty relations returns *null*

[3] Note that allowing only a single attribute as input of an aggregation function and only group-by attributes instead of group-by expressions does not limit the expressive power of the algebra. Expressions like $\alpha_{c*d,sum(a+b)}(q_1)$ can be written as $\alpha_{g_1,sum(agg_1)}(\Pi_{c*d \rightarrow g_1, a+b \rightarrow agg_1}(q_1)$ in our algebra. For brevity, we will use the first notation when appropriate.

Nullary Operators

$$[[t]] = \{t\}$$
$$[[R]] = \{t^n \mid t^n \in R\}$$

Unary Operators

$$[[\delta(q_1)]] = \{t \mid t^n \in Q_1\}$$
$$\left[\left[\Pi^B{}_A(q_1)\right]\right] = \{t' = (v_1,\ldots,v_m)^{sum} \mid sum = \sum_{t^n \in Q_1, t.A = t'} (n)\} \text{ for } A = (A_1,\ldots,A_m)$$
$$\left[\left[\Pi^S{}_A(q_1)\right]\right] = \delta(\Pi^B{}_A(q_1)) = \{a = (a_1,\ldots,a_n)^1 \mid t^m \in Q_1 \wedge t.A = a\} \text{ for } A = (A_1,\ldots,A_m)$$
$$[[\sigma_C(q_1)]] = \{t^n \mid t^n \in Q_1 \wedge t \models C\}$$
$$[[\alpha_{G,agg}(q_1)]] = \{(t.G, res_1,\ldots,res_n)^1 \mid t \in Q_1 \wedge \forall i \in \{1,n\} : res_i = agg_i(\Pi^B{}_{B_i}(\sigma_{G=t.G}(q_1))\}$$

Join Operators

$$[[q_1 \times q_2]] = \{(t_1,t_2)^{n\times m} \mid {t_1}^n \in Q_1 \wedge {t_2}^m \in Q_2\}$$
$$[[q_1 \bowtie_C q_2]] = \{t^{n\times m} \mid t^{n\times m} \in q_1 \times q_2 \wedge t \models C\}$$
$$[[q_1 \text{⟕}_C q_2]] = \{(t_1,t_2)^{n\times m} \mid (t_1,t_2)^{n\times m} \in [[q_1 \bowtie_C q_2]]\}$$
$$\cup \{(t_1, null(\mathbf{Q_2}))^n \mid {t_1}^n \in Q_1 \wedge (\nexists t_2 \in Q_2 : (t_1,t_2) \models C)\}$$
$$[[q_1 \text{⟖}_C q_2]] = \{(t_1,t_2)^{n\times m} \mid (t_1,t_2)^{n\times m} \in [[q_1 \bowtie_C q_2]]\}$$
$$\cup \{(null(\mathbf{Q_1}),t_2)^n \mid {t_2}^n \in Q_2 \wedge (\nexists t_1 \in Q_1 : (t_1,t_2) \models C)\}$$
$$[[q_1 \text{⟗}_C q_2]] = \{(t_1,t_2)^{n\times m} \mid (t_1,t_2)^{n\times m} \in [[q_1 \bowtie_C q_2]]\}$$
$$\cup \{(null(\mathbf{Q_1}),t_2)^n \mid {t_2}^n \in Q_2 \wedge (\nexists t_1 \in Q_1 : (t_1,t_2) \models C)\}$$
$$\cup \{(t_1, null(\mathbf{Q_2}))^n \mid {t_1}^n \in Q_1 \wedge (\nexists t_2 \in Q_2 : (t_1,t_2) \models C)\}$$

Set Operators

$$\left[[q_1 \cup^S q_2]\right] = \{t \mid t^n \in Q_1 \vee t^m \in Q_2\}$$
$$\left[[q_1 \cap^S q_2]\right] = \{t \mid t^n \in Q_1 \wedge t^m \in Q_2\}$$
$$\left[[q_1 -^S q_2]\right] = \{t \mid t^n \in Q_1 \wedge t^m \notin Q_2\}$$
$$\left[[q_1 \cup^B q_2]\right] = \{t^{n+m} \mid t^n \in Q_1 \wedge t^m \in Q_2\}$$
$$\left[[q_1 \cap^B q_2]\right] = \{t^{min(n,m)} \mid t^n \in Q_1 \wedge t^m \in Q_2\}$$
$$\left[[q_1 -^B q_2]\right] = \{t^{n-m} \mid t^n \in Q_1 \wedge t^m \in Q_2\}$$

Sublink Expressions

$$[[e\ IN\ q_{sub}]] = \exists t \in Q_{sub} : t = e \qquad [[e\ NOTIN\ q_{sub}]] = \neg\exists t \in Q_{sub} : t = e$$
$$[[e\ op\ ANY\ q_{sub}]] = \exists t \in Q_{sub} : e\ op\ t \qquad [[q_{sub}]] = Q_{sub}$$
$$[[e\ op\ ALL\ q_{sub}]] = \forall t \in Q_{sub} : e\ op\ t \qquad [[\ EXISTS\ q_{sub}]] = \exists t \in Q_{sub}$$

Figure 2.1: Perm Relational Algebra

Description	Shortcut
Relation	$R, S, T \ldots$
Attribute	$a, b, c \ldots$
List or Set of Attributes	$A, B, \ldots$
Renaming attribute a to b	$a \rightarrow b$
Shortcut for comparing all attributes from a list $A = (a_1, \ldots, a_n)$ with attributes from a list $B = (b_1, \ldots, b_n)$	$A = B := a_1 = b_1 \wedge \ldots \wedge a_n = b_n$
Shortcut for renaming all attributes from a list $A = (a_1, \ldots, a_n)$ to attributes from a list $B = (b_1, \ldots, b_n)$	$A \rightarrow B := a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots a_n \rightarrow b_n$
Concatenation of two tuples t_1 and t_2	(t_1, t_2)
Algebra expression	q
Result relation generated by evaluation of algebra expression q	Q or $[[q]]$
Schema of an relation R or of the result of evaluating algebra expression q	$\mathbf{R}$ and $\mathbf{Q}$
Null-value	ε

Figure 2.2: Notational Conventions for the Relational Model and *Perm* Algebra

Sublink Expressions: SQL allows for nested sub-queries in, e.g., the *WHERE* clause. To be able to represent such sub-queries (which we refer to as *sublinks*) in the *Perm* algebra we introduce nesting expressions that resemble the nesting constructs of SQL (*ALL*, *ANY*, *IN*, *EXISTS*, and *scalar sublink*). Similar approaches have been presented in [2, 1]. The $EXISTS\ q_{sub}$ expression evaluates to true, iff Q_{sub} contains at least one tuple. If an algebra expression q_{sub} is directly applied in an projection expressions or selection predicate, we call it a *scalar sublink*. A *scalar sublink* q_{sub} evaluates to the result of evaluating algebra expression $([[q_{sub}]])$. This kind of sublink is only defined if q_{sub} returns at most one tuple and $\mathbf{Q_{sub}}$ contains only a single attribute. If Q_{sub} returns the empty set, then this nested expression evaluates to ε. The sublink expression $e\ op\ ANY\ q_{sub}$ evaluates to true if for at least one tuple t from Q_{sub} the expression $e\ op\ t$ evaluates to true. Here op represents an arbitrary comparison operator. This nested expression is only defined if Q_{sub} contains a single attribute with a data type that is comparable to the result type of expression e [4]. The counterpart of the *ANY*-expression is the *ALL*-expression. $e\ op\ ALL\ q_{sub}$ evaluates to true, iff every tuple t from Q_{sub} fulfills the condition $e\ op\ t$. An ALL-sublink expression evaluates to true if Q_{sub} is the empty set. An ANY-sublink expression evaluates to false if $Q_{sub} = \emptyset$. Two additional nested expressions are provided for convenience: $e\ IN\ q_{sub}$ which is equivalent to $e = ANY\ q_{sub}$ and $e\ NOT IN\ q_{sub}$ which is equivalent to $\neg(e = ANY\ q_{sub})$.

As in SQL we allow for correlations between *sublinks* and the algebra expression they are used in (called outer expression or regular input of an operator). A *correlation* is a reference to an attribute of the outer expression from inside the *sublink*. For instance, in the expression $\sigma_{EXISTS\ \sigma_{S.b=R.a}(S)}(R)$ the attribute reference $R.a$ is a correlation, because it references an attribute from relation R that is the regular input of the selection. Sublinks with correlations are evaluated using so-called *nested iteration*. Nested iteration evaluates the sublink expression separately for each tuple from the regular input of the operator. We call a sublink expression *correlated* if it contains correlations and *uncorrelated* otherwise. A sublinks expression that contains another sublink expression is called *nested*. In spite of the fact that SQL supports sublinks in all clauses we limit the use of sublinks to projection and selection to simplify the provenance computation for these expressions and because this restriction does not limit the expressive power of the algebra. E.g., the following algebra expression $R \bowtie_{a\ IN\ \Pi_C(T)} S$ is equivalent to $\sigma_{a\ IN\ \Pi_C(T)}(R \times S)$ and $\alpha_{EXISTS\ (S),sum(a)}(R)$ is equivalent to $\alpha_{new,sum(a)}(\Pi^B{}_{EXISTS\ (S)\rightarrow new,a}(R))$.

Figure 2.2 shows notational conventions for algebra expressions that we will use throughout this thesis. Most of these shortcut have already been used in the definition of the algebra.

[4] In principle a type system would be needed to decide if two data types are comparable. We do not formally define such a type system for the *Perm* algebra because it is not needed in the definition and discussion of contribution semantics and would needlessly increase the complexity of the algebra.

person

SSN	name
1-1	Peter Peterson
2-4	Jens Jensen
5-6	Knut Knutsen

newspaper

newsId	name	publisher
1	NZZ	IEEE
2	20 Minuten	Springer

reads

pSSN	nNewsId
1-1	1
1-1	2
2-4	1

$$q_1 = \sigma_{\neg\, EXISTS\,(q_{sub})}(person) \qquad q_{sub} = \sigma_{\neg\, EXISTS\,(\sigma_{pSSN=SSN \wedge newsId=nNewsId}(reads))}(newspaper)$$

$$q_2 = \alpha_{name,count(*)}(reads \bowtie_{pSSN=SSN} person)$$

$$q_3 = \Pi^S{}_{person.name \rightarrow person, newspaper.name \rightarrow paper}(person \rtimes_{SSN=pSSN} (reads \bowtie_{nNewsId=newsId} newspaper))$$

```
q1 = SELECT * FROM person
     WHERE NOT EXISTS
          (SELECT * FROM newspaper
               WHERE NOT EXISTS
                    (SELECT * FROM reads WHERE pSSN = SSN AND nNewsId = newsId));

q2 = SELECT name, count(*) FROM reads, person WHERE pSSN = SSN GROUP BY name;

q3 = SELECT p.name a AS person, n.name AS paper
     FROM person p LEFT JOIN
          (reads r JOIN newspaper n ON (nNewsId = newsId)) ON (SSN = pSSN);
```

Q_1

SSN	name
1-1	Peter Peterson

Q_2

name	count
Peter Peterson	2
Jens Jensen	1

Q_3

person	paper
Peter Peterson	NZZ
Peter Peterson	20 Minuten
Jens Jensen	NZZ
Knut Knutsen	*NULL*

Figure 2.3: Example Algebra Expressions and Evaluations

Example 2.1. *Figure 2.3 presents some example algebra expressions, equivalent formulations in SQL, and the results of evaluating them over an example database instance. The example database models newspapers, persons, and which person reads which newspapers. Query q_1 from the example returns the persons that are reading all newspapers stored in the database. This query can be expressed in SQL as a nested NOT EXISTS: Return all persons for whom no newspaper exists that is not read by this person. In the algebra this query is expressed using a nested EXISTS sublink expression in the condition of the selection operator. Query q_2 returns the number of newspapers read by each person. In SQL this query is expressed using the standard aggregation function count grouping on the person relation's name attribute. Hence, the equivalent algebra expression uses the aggregation operator. Query q_3 returns all persons and the newspapers they are reading. An outer join is used to also return persons that do not read any newspapers. The algebra version of q_3 is an example for the application of renaming in projection expressions.*

2.2 Influence Data Provenance Contribution Semantics

In this section we formally define the *data* provenance *CS* types implemented in *Perm*. We first introduce the *I-CS* type of our system that is based on *Lineage-CS*, but extends this *CS* type for all algebra operators presented in the last section and addresses several shortcomings of *Lineage-CS* regarding provenance representation and applicability to algebra expressions with sublinks. Afterwards we compare the expressiveness of *Lineage-CS* and *PI-CS*.

2.2.1 Lineage Contribution Semantics

We base the *I-CS* definition used in *Perm* on *Lineage-CS* presented in [5], because this definition has several advantages over alternative *I-CS* types. First, users tend to intentionally express queries in a certain way and, therefore, the strong dependency of *Lineage-CS* on the syntactical structure of a query is an advantage. Second, this *CS* type is defined for a larger set of algebra operators than other approaches. Third, provenance is defined for single algebra operators which allows easy extension to new algebra operators. We provide a formal definition of *Lineage-CS*. The definition below is taken from [5] with the notation adapted to our conventions:

Definition 2.1 (Lineage-CS). *For an algebra operator op with inputs $Q_1,\ldots,Q_n$ from a database instance I and a tuple $t \in op(Q_1,\ldots,Q_n)$ a set $\mathscr{W}(op,I,t) =< Q_1^*,\ldots,Q_n^* >$ with $Q_i^* \subseteq Q_i$ is the* witness set *of t if it fulfills the following conditions:*

$$[[op(\mathscr{W}(op,I,t))]] = \{t^x\} \quad \textbf{(1)}$$

$$\forall i,t' \in Q_i^* : \left[\left[op(< Q_1^*,\ldots,Q_{i-1}^*,\{t'\},Q_{i+1}^*,\ldots >)\right]\right] \neq \emptyset \quad \textbf{(2)}$$

$$\neg\exists \mathscr{W}' \subset < Q_1,\ldots,Q_n >: \mathscr{W}' \supset \mathscr{W}(op,I,t) \wedge \mathscr{W}' \models (1),(2) \quad \textbf{(3)}$$

The first condition **(1)** in Definition 2.1 checks that the witness set produces exactly t and nothing else by evaluating operator op over the witness set. The second condition **(2)** checks that each tuple t' in the witness set contributes to t and, therefore, guarantees that no superficial tuples are included in the witness set. Finally, the third condition **(3)** checks that the witness set is the maximal list with these properties, meaning that no tuples that contribute to t are left out. Note that in condition 3 two lists of sets are compared according to their subsumption relationship ($\supset$). Below we formalize the notion of subsumption for lists:

Definition 2.2 (List Subsumption). *A list of sets U subsumes of a list of sets V ($U \supset V$), iff both lists have the same length ($|U| = |V|$), each set in U contains the elements from the corresponding set in V, and at least one set from U contains an element that is not included in the corresponding set from V:*

$$U \supset V \Leftrightarrow (|U| = |V|) \wedge (\forall i \in \{1,\ldots,|V|\} : U_i \supseteq V_i) \wedge (\exists i \in \{1,\ldots,|V|\} : U_i \supset V_i)$$

We omit the instance I for a witness set if it is clear from the context. *Lineage-CS* was originally studied for selection, projection, join, cross product, aggregation, union and set difference. In our discussion we include also intersection and outer joins. Note that the definition presented here is defined for set semantics and under this semantics it was proven that $\mathscr{W}(q,t)$ is unique. We postpone the discussion of bag semantics to later in this section.

Example 2.2. *As an example for Lineage-CS provenance consider the algebra expression* $q = \Pi^{B}{}_{a}(R)$ *over the relation presented below.*

R

a	b
1	2
2	3

Q

a
1
2

The Lineage-CS provenance of the result tuple $t = (1)$ *from q is as follows:*

$$\mathscr{W}(q,(1)) =< \{(1,2)\} >$$

The first condition of definition 2.1 is obviously fulfilled. The result of evaluating q over the set $\{(1,2)\}$ *is a relation that only contains tuple* (1)*. The second condition is trivially fulfilled, because the provenance contains only a single tuple. For the third condition we have to check that no super set of* $\mathscr{W}$ *fulfills conditions 1 and 2. In this case* $\mathscr{W}' = \{(1,2),(2,3)\}$ *the only super-set of* $\mathscr{W}$ *does not fulfill condition 1, because the result of applying q to* $\mathscr{W}'$ *contains tuple* $(2) \neq t$*.*

2.2.1.1 Transitivity and Sets of Output Tuples

Lineage-CS defines provenance to be transitive. I.e., if tuple t is in the provenance of tuple t' according to an operator op_1 and t' is in the provenance of t'' according to some operator op_2, then t belongs to the provenance of t'' according to $q = op_2(op_1)$. Therefore, the witness set of an algebra expression q is computed by recursively applying Definition 2.1 to each operator in q. An advantage of *Lineage I-CS* is that the focus on a single operator leads to a simple evaluation strategy and the witness set of each operator can be studied independently of the witness set of other operators. [5] also defines the provenance of a set T of result tuples according to *Lineage-CS* as:

$$\mathscr{W}(q,T) = \bigsqcup_{t \in T} \mathscr{W}(q,t)$$

Here $\bigsqcup$ stands for the element-wise union of two lists. E.g.:

$$< \{a\},\{b\} > \sqcup < \{c\},\{d\} >=< \{a,c\},\{b,d\} >$$

2.2.1.2 Bag Semantics

Lineage-CS was also extended for bag semantics. Under bag semantics two duplicates of a tuple cannot be distinguished, therefore it is impossible to determine from which duplicate a result tuple is derived. Consider the query $q = R -^{B} S$ over the relations $R = \{(1)^2\}$ and $S = \{(1)\}$. The result tuple (1) could be either derived from the first or the second tuple in R. Furthermore, if the result of an algebra expression contains a tuple t with a multiplicity greater than one, each duplicate of t might have been derived from different input tuples and its not clear which duplicate of t should be associated with which input tuple. As an example for this problem consider expression $q = \Pi^{B}{}_{a}(R)$ over relation $R = \{(1,2),(1,3)\}$ with schema $\mathbf{R} = (\mathbf{a},\mathbf{b})$. If definition 2.1 is applied to this query, then there are two sets $\mathscr{W}(q,(1))_1 =< \{(1,2)\} >$ and $\mathscr{W}(q,(1))_2 =< \{(1,3)\} >$ that fulfill the conditions of the definition. Cui et al. present two solutions to this problem. One called the *derivation set* is the set of all possible witness sets $\mathscr{W}(q,t)$ and the second one called the *derivation pool* is generated by computing the bag union of the individual Q_i^* elements of all possible witness sets $\mathscr{W}(q,t)$. The derivation set of the example query above would be $\{\mathscr{W}(q,(1))_1, \mathscr{W}(q,(1))_2\}$ and the derivation pool would be $< \{(1,2),(1,3)\} >$. The derivation set has the disadvantage that its size is proportional to the number of different derivations of a result tuple. Therefore, we use only the derivation pool. The definition we have given for *Lineage-CS* generates the derivation pool.

$$
\begin{aligned}
\mathscr{W}(R,t) &= < \{u^n \mid u^n \in R \wedge u = t\} > \\
\mathscr{W}(\sigma_C(q_1),t) &= < \{u^n \mid u^n \in Q_1 \wedge u = t\} > \\
\mathscr{W}(\Pi_A(q_1),t) &= < \{u^n \mid u^n \in Q_1 \wedge u.A = t\} > \\
\mathscr{W}(\alpha_{G,agg}(q_1),t) &= < \{u^n \mid u^n \in Q_1 \wedge t.G = u.G\} > \\
\mathscr{W}(q_1 \bowtie_C q_2,t) &= < \{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, \{u^n \mid u^n \in Q_2 \wedge u = t.\mathbf{Q_2}\} > \\
\mathscr{W}(q_1 \rtimes_C q_2,t) &= \begin{cases} < \{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, Q_2 > & \text{if } t \not\models C \\ < \{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, \{u^n \mid u^n \in Q_2 \wedge u = t.\mathbf{Q_2}\} > & \text{else} \end{cases} \\
\mathscr{W}(q_1 \ltimes_C q_2,t) &= \begin{cases} < Q_1, \{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_2}\} > & \text{if } t \not\models C \\ < \{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, \{u^n \mid u^n \in Q_2 \wedge u = t.\mathbf{Q_2}\} > & \text{else} \end{cases} \\
\mathscr{W}(q_1 \rtimes\ltimes_C q_2,t) &= \begin{cases} < \{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, Q_2 > & \text{if } t \not\models C \wedge t.\mathbf{Q_2} \text{ is } \varepsilon \\ < Q_1, \{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_2}\} > & \text{if } t \not\models C \wedge t.\mathbf{Q_1} \text{ is } \varepsilon \\ < \{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, \{u^n \mid u^n \in Q_2 \wedge u = t.\mathbf{Q_2}\} > & \text{else} \end{cases} \\
\mathscr{W}(q_1 \cup q_2,t) &= < \{u^n \mid u^n \in Q_1 \wedge u = t\}, \{u^n \mid u^n \in Q_2 \wedge u = t\} > \\
\mathscr{W}(q_1 \cap q_2,t) &= < \{u^n \mid u^n \in Q_1 \wedge u = t\}, \{u^n \mid u^n \in Q_2 \wedge u = t\} > \\
\mathscr{W}(q_1 - q_2,t) &= < \{u^n \mid u^n \in Q_1 \wedge u = t\}, \{u^n \mid u^n \in Q_2 \wedge u \neq t\} >
\end{aligned}
$$

Figure 2.4: Compositional Semantics for *Lineage-CS*

2.2.1.3 Compositional Semantics of *Lineage-CS*

To determine the provenance of an algebra expression using the conditions of definition 2.1 can be cumbersome, because the definition only states which conditions have to be fulfilled by the provenance, but not how to construct the provenance. Cui et. al presented how to generate *Lineage-CS* provenance using a construction based on the syntactical structure of an algebra expression. In the following we refer to the conditions of definition 2.4 as the *declarative semantics* of *Lineage-CS* and the semantics defined by the set construction as the *compositional semantics* of *Lineage-CS*. The construction rules for the *compositional semantics* are presented in Figure 2.4.

Example 2.3. *For example, the witness set of an output tuple t of a selection is always the singleton set containing t, because selection outputs unmodified input tuples. An output tuple t from an aggregation is derived from a set of input tuples that belong to the same group (have the same grouping attribute values as t).*

Example 2.4. *Figure 2.5 presents some examples of provenance according to Lineage-CS. Query q_a is an example for the representation of duplicate removal. The result tuple (1) from this query was generated from two result tuples of the inner join between R and S. Query q_b demonstrates the inclusion of the complete right input of the left join in the provenance for tuples that do not have join partners. An example for aggregation is given with query q_c. Note that all tuples from relation R with the same grouping attribute value 1 are in the provenance of the result tuple $(1,5)$. Both queries q_d and q_e illustrate the provenance for set difference operations. Query q_d is similar to an example from [4] that was used to show that tuples from the right input of a set difference can contribute to the result of this operator. Tuple (2) from relation T belongs to the provenance of result tuple $u = (2)$, because it indirectly contributed to u by removing tuple (2) from the result of $(S - T)$. If t had not been in relation T, then u would not be in the result of q_d. Query q_e is a counterexample to this form of reasoning. According to Lineage-CS tuple $v = (2)$ from relation U is not in the provenance of result tuple $x = (2)$, but if we apply the same reasoning as used for query q_d it contributes to x.*

As apparent form the example presented above, the problem with *Lineage-CS* for set difference is that it is defined for single algebra operators. But to distinguish between the cases presented with query q_e and

the one of query q_f information about the context a operator is used in is needed. In spite of the fact that *Why-CS* is defined for an algebra expression it does not solve this problem, because it generates the same provenance for both cases.

Cui et. al. proved that the *compositional semantics* of *Lineage-CS* is equivalent to the *declarative semantics* from definition 2.1. Recall that *Lineage-CS* was originally not studied for intersection and outer join types. Therefore, we now prove the equivalence of provenance according to definition 2.1 and the compositional semantics for these operators. The interested reader is referred to [4] for the proofs for the remaining operators.

Theorem 2.1 (Equivalence of Declarative and Compositional Semantics of *Lineage-CS*)**.** *The compositional and declarative semantics of Lineage-CS are equivalent.*

Proof. We only prove this theorem for the cases not handled in [4]: outer joins and intersection. Let $\mathscr{W}(op,t)$ be the witness set produced by the declarative semantics and $\mathscr{O}(op,t)$ be the witness set produced by the compositional semantics. We have to show that $\mathscr{W}(op,t) = \mathscr{O}(op,t)$ holds for $op \in \{⟕, ⟖, ⟗, \cap\}$ and all $t \in [[op]]$. This proposition can be proven by proving that $\mathscr{O}(op,t)$ fulfills conditions 1 to 3 from definition 2.1 and, thus, is indeed equal to $\mathscr{W}(op,t)$.

Case $⟕$:

We present the proof for the case $t \not\models C$ because if $t \models C$ holds then the behavior and provenance of the left outer join is the same as for the inner join. Thus, $\mathscr{O}(q_1 ⟕_C q_2, t) =< \{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, Q_2 >$.

Condition **(1)**: From the definition of the left outer join we know that there is no tuple t' in Q_2 for which $(t.\mathbf{Q_1}, t') \models C$ holds, because otherwise $t \models C$ would hold. Therefore, $[[\{u^n\} ⟕_C Q_2]] = \{t^x\}$.

Condition **(2)**: Both $\left[\left[\{u^1\} ⟕_C Q_2\right]\right] \neq \emptyset$ and $[[\{u^n\} ⟕_C \{t'\}]] \neq \emptyset$ with $t' \in Q_2$ trivially holds because for a non empty left hand side input the left join operator never produces an empty result.

Condition **(3)**: We prove the maximality of $\mathscr{O}(q_1 ⟕_C q_2)$ by contradiction. Assume a list $\mathscr{O}' \supset \mathscr{O}$ exists that fulfills conditions 1 and 2 from the definition. Since ${Q_2}^*$ cannot be extended, ${Q_1}^*$ from $\mathscr{O}'$ has to contain at least one tuple t' that is not in $\mathscr{O}$ with $u \neq t'$. From the semantics of the left outer join follows that $[[\{t'\} ⟕_C Q_2]]$ produces either a tuple $x = (t', t'')$ with t'' from Q_2 or a tuple $x' = (t', null(\mathbf{Q_2}))$ depending on the existence of an join partner for t' in Q_2. Because $t' \neq u$ we know that neither $t = x$ nor $t = x'$ holds, and, therefore condition 1 is violated. Hence, we conclude that, since no such $\mathscr{O}'$ can exists, $\mathscr{O}$ has to be maximal.

Case $⟖$:

The proof for right outer join is analog to the proof for left outer join.

Case $⟗$:

For full outer join we have to distinguish two cases: ($t \not\models C \wedge t.\mathbf{Q_1}$ is ε) and ($t \not\models C \wedge t.\mathbf{Q_2}$ is ε). Both cases can be proven analog to the proof for left join.

Case $\cap$:

According to the compositional semantics $\mathscr{O}(q_1 \cap q_2, t) =< \{u^n \mid u^n \in Q_1 \wedge u = t\}, \{u^n \mid u^n \in Q_2 \wedge u = t\} >$.

Condition **(1)**: We have to prove $[[\{u^n\} \cap \{u^m\}]] = \{t^x\}$ for some x.

$$[[\{u^n\} \cap \{u^m\}]] = [[\{t^n\} \cap \{t^m\}]] = \{t^{min(n,m)}\} \qquad \text{(definition of } \cap\text{)}$$

Condition **(2)**: Since intersection is symmetric it suffices to show that $\left[\left[\{u^n\} \cap \{u^1\}\right]\right] \neq \emptyset$ which trivially holds:

$$\left[\left[\{u^n\} \cap \{u^1\}\right]\right] = \{t^1\} \neq \emptyset$$

Condition **(3)**: We prove the maximality of $\mathscr{O}(q_1 \cap q_2, t)$ by contradiction. Assume a super-set $\mathscr{O}'$ of $\mathscr{O}$ exists that fulfills conditions 1 and 2 from the definition. Then $\mathscr{O}'$ has to contain a tuple $t' \neq t$ that is not in $\mathscr{O}$. w.l.o.g. assume $t' \in Q_1^*$. Then for condition 2 to hold $[[\{t'\} \cap Q_2^*]] \neq \emptyset$ has to be true. Either Q_2^* contains a tuple t'' that is equal to t', then condition 2 is fulfilled, but condition 1 is no longer fulfilled because $[[Q_1^* \cap Q_2^*]] \neq \{t\}$. Or Q_2^* does not contain such an tuple and, therefore, $[[\{t'\} \cap Q_2^*]] = \emptyset$ would hold which contradicts condition 2. Using the same reasoning as for the left outer join case we conclude that $\mathscr{O}$ has to be maximal. □

R

a	b
1	2
1	3
2	3
2	5

S

c
2
3

T

d
2
5

U

e
2
5
6

Q_a

a
1
2

Q_b

a	b	c
1	2	2
1	3	3
2	3	3
2	5	ε

Q_c

a	sum(b)
1	5
2	8

Q_d

b
2
5

Q_e

c
2
3

Q_f

a
1
2

$q_a = \Pi^S{}_a(R \rtimes_{b=c} S)$ $\quad q_b = R \ltimes\!\rtimes_{b=c} S$ $\quad q_c = \alpha_{a,sum(b)}(R)$

$q_d = \Pi^S{}_b(R) - (S - T)$ $\quad q_e = S - (T - U)$ $\quad q_f = \Pi^S{}_a(R \ltimes\!\rtimes_{b=c} S)$

$\mathscr{W}(q_a,(1)) =< \{(1,2),(1,3)\},\{(2),(3)\} >$

$\mathscr{W}(q_b,(2,3,3)) =< \{(2,3)\},\{(3)\} >$ $\quad \mathscr{W}(q_b,(2,5,\varepsilon)) =< \{(2,5)\},\{(2),(3)\} >$

$\mathscr{W}(q_c,(1,5)) =< \{(1,2),(1,3)\} >$

$\mathscr{W}(q_d,(2)) =< \{(2,3),(2,5)\},\{(3)\},\{(2),(5)\} >$

$\mathscr{W}(q_e,(2)) =< \{(2)\},\{\},\{\} >$

$\mathscr{W}(q_f,(2)) =< \{(2,3),(2,5)\},\{(2),(3)\} >$

$\mathscr{L}(q_a,(1)) = \{< (1,2),(2) >,< (1,3),(3) >\}$

$\mathscr{L}(q_b,(2,3,3)) = \{< (2,3),(3) >\}$ $\quad \mathscr{L}(q_b,(2,5,\varepsilon)) = \{< (2,5),(2) >,< (2,5),(3) >\}$

$\mathscr{L}(q_c,(1,5)) = \{< (1,2) >,< (1,3) >\}$

$\mathscr{L}(q_d,(2)) = \{< (2,3),(3),(2) >,< (2,3),(3),(5) >,< (2,5),(3),(2) >,< (2,5),(3),(5) >\}$

$\mathscr{L}(q_e,(2)) = \{< (2),\perp,\perp >\}$

$\mathscr{L}(q_f,(2)) = \{< (2,3),(2) >,< (2,5),(2) >,< (2,5),(3) >\}$

$\mathscr{DD}(q_a,(1)) = \{< (1,2),(2) >,< (1,3),(3) >\}$

$\mathscr{DD}(q_b,(2,3,3)) = \{< (2,3),(3) >\}$ $\quad \mathscr{DD}(q_b,(2,5,\varepsilon)) = \{< (2,5),\perp >\}$

$\mathscr{DD}(q_c,(1,5)) = \{< (1,2) >,< (1,3) >\}$

$\mathscr{DD}(q_d,(2)) = \{< (2,3),\perp,\perp >\}$

$\mathscr{DD}(q_e,(2)) = \{< (2),\perp,\perp >\}$

$\mathscr{DD}(q_f,(2)) = \{< (2,3),(2) >,< (2,5),\perp >\}$

Figure 2.5: Provenance According to *Lineage-CS*, *WL-CS*, and *PI-CS*

2.2.2 Perm Influence Contribution Semantics

Definition 2.1 generates useful provenance information and is defined for a larger subset of relational algebra than the *Lineage-CS*, but there are some issues with this definition that limit its usefulness:

1. **Representation**: Modeling provenance as independent sets of tuples has the disadvantage that the information about which input tuples were combined to produce a result tuple is not modeled and that in the provenance of a set of result tuples it is not clear to which result tuple a part of the provenance belongs too.
2. **Negation**: The maximization condition (**2**) is problematic if operations with negation or non-existence checks including set difference and outer joins are involved.
3. **Sublinks**: *Lineage-CS* is not unique and produces false positives for queries that use sublink expressions. We do not discuss this problem in this report. An approach to overcome the problems of *Lineage-CS* with sublinks is presented in [7].

Representation: As an example of the first problem consider query q_a from Fig. 2.5. The tuple $t = (1)$ from the result of q_a is derived from two tuples from relations R and S (all tuples from R and S that were joined and have an a-attribute value of 1). Which tuples contributed to t is apparent from *Lineage-CS* ($\mathscr{W}(q_a,t)$), but the information about which tuple from R was joined with which tuple from S is not modeled. To record this information we change the provenance representation from a list of subsets of the input relations to a set of *witness lists*. A witness list w is an element from $(Q_1^\varepsilon \times \ldots \times Q_n^\varepsilon)$ with $Q_i^\varepsilon = Q_i \cup \bot$. Thus, a witness list w contains a tuple from each input of an operator or the special value $\bot$. The value $\bot$ at position i in a witness list w indicates that no tuple from the ith input relation belongs to w (and, therefore, is useful in modeling outer joins and unions). Each witness list represents one combination of input relation tuples that were used together to derive a tuple.

Definition 2.3 (Witness List)**.** *For an algebra operator op with inputs $Q_1,\ldots,Q_n$ each element w from $(Q_1^\varepsilon \times \ldots \times Q_n^\varepsilon)$ with $Q_i^\varepsilon = Q_i \cup \bot$ is called a potential witness list of op. We use $w[i]$ to denote the ith component (tuple) of a witness list w and $w[i-j]$ to denote a sub-list containing only the ith till the jth component of w.*

Example 2.5. *Consider the following witness list: $w =< (1),(3),(5) >$. For example the third component of w is $w[3] = (5)$ and a list containing only the last two element of w is denoted by $w[2-3] =< (3),(5) >$.*

The modified version of Definition 2.1 using the witness list representation is presented below. We call this new contribution semantics type *Witness-List-CS* or short *WL-CS*. Note that in condition 1 of the *WL-CS* definition we use the evaluation of an operator over a set $\mathscr{L}$ of witness lists. We define this evaluation as the evaluation of the operator over reconstructed subsets of the original input relations. The reconstructed input relation subsets will contain all the tuples contained in the witness lists from $\mathscr{L}$. The notation ${Q_i}^R$ is used for the reconstructed subset of input relation Q_i.

Definition 2.4 (Operator Evaluation over Witness Lists)**.** *The evaluation of an operator op with inputs $Q_1,\ldots,Q_n$ over a set of witness lists W for this operator is defined as:*

$$[[op(W)]] = \left[\left[op({Q_1}^R,\ldots,{Q_n}^R)\right]\right]$$
$${Q_i}^R = \{t^n \mid t^n \in Q_i \wedge \exists w \in \mathscr{L} : w[i] = t\}$$

${Q_i}^R$ contains all tuples with their original multiplicity that are mentioned by at least one witness list. E.g., for a query $q = R \bowtie_{a=b} S$ over relations $R = \{(1)^2\}$ and $S = \{(1)\}$, and witness lists $w_1 =< (1),(1) >$ and $w_2 =< (1),(1) >$ the result of $q(\mathscr{L}(q,(1,1))$ would be $Q = \{(1,1)^2\}$.

Definition 2.5 (Witness-List-CS (WL-CS))**.** *For an algebra operator op with inputs $Q_1,\ldots,Q_n$, and a tuple $t \in op(Q_1,\ldots,Q_n)$ a set $\mathscr{L}(op,t) \subseteq (Q_1^\varepsilon \times \ldots \times Q_n^\varepsilon)$ with $Q_i^\varepsilon = Q_i \cup \bot$ is the set of witness lists of t according to WL-CS if it fulfills the following conditions:*

$$[[op(\mathscr{L}(op,t))]] = \{t^x\} \quad \textbf{(1)}$$

$$\forall w \in \mathscr{L}(op,t) : [[op(w)]] \neq \emptyset \quad \textbf{(2)}$$

$$\neg\exists \mathscr{L}' \subseteq (Q_1^\varepsilon \times \ldots \times Q_n^\varepsilon) : \mathscr{L}' \supset \mathscr{L}(op,t) \wedge \mathscr{D}\mathscr{D}' \models (1),(2) \quad \textbf{(3)}$$

Example 2.6. *Some examples for WL-CS are shown in Fig. 2.5. For instance, the provenance of q_a demonstrates that under WL-CS we preserve the information that tuples $(1,2)$ and (2) were joined. This fact cannot be deduced from the Lineage-CS provenance of q_a. Note that besides the representation of provenance as witness-lists WL-CS bears some similarity with Lineage-CS. For example, under Lineage-CS the provenance of tuple (1) from query q_a would be $\{\{(1,2),(2)\},\{(1,3),(3)\}\}$. Thus, for queries like this Lineage-CS also captures which tuples were used together to derive a result tuple. In contrast to Lineage-CS, our definition also captures this information for, e.g., queries that combine aggregation with set projection . For instance, the Lineage-CS provenance of tuple $t = (1)$ from the result of query $\alpha_{a,sum(b)}(R \rtimes_{b=c} S)$ would be a single witness, thus, the information which tuples were joined is lost in this representation.*

Negation: As an example of the problems that arise with operators that use some form of non-existence check, consider query q_b from Figure 2.5. According to Definition 2.1, the witness list of the result tuple $t = (2,5,\varepsilon)$ contains all tuples from relation S, but in fact none of them contributed to t. Definition 2.5 does not solve this problem - the *WL-CS* provenance includes witness lists which contain the tuple $(2,5)$ paired with every tuple in S. Thus, both *Lineage-CS* and *WL-CS* do not capture an intuitive notion of influence when an operator includes negation. We believe a better semantics for the provenance of tuple $t = (2,5,\varepsilon)$ from the result of q_b would be a witness list $< (2,5), \bot >$. This indicates that $(2,5)$ paired with no tuples from S contributed to t (rather than saying that *every* value of S is in the provenance of this tuple). To achieve this semantics, we extend the *WL-CS* provenance definition with an additional condition **(4)**. This condition states that we will exclude a witness list w from the provenance, if there is a "smaller" witness list w' in the provenance that subsumes w. A witness list w is subsumed by a witness list w' (denoted by $w \prec w'$) iff w' can be derived from w by replacing some input tuples from w with $\bot$.

Definition 2.6 (Witness List Subsumption)**.** *Let w and w' be two witness lists from $\mathscr{L}(q,t)$. We define the subsumption relationship between w and w' (written as $w \prec w'$) as follows:*

$$w \prec w' \Leftrightarrow (\forall i : w[i] = w'[i] \vee w'[i] = \bot) \wedge (\exists i : w[i] \neq \bot \wedge w'[i] = \bot)$$

We use the definition of subsumption between witness lists to define a *CS* type with the desired negation semantics which we call *Perm-Influence-CS* (*PI-CS*).

Definition 2.7 (Perm-Influence-CS (PI-CS))**.** *For an algebra operator op with inputs $Q_1,\ldots,Q_n$, and a tuple $t \in op(Q_1,\ldots,Q_n)$ a set $\mathscr{D}\mathscr{D}(op,t) \subseteq (Q_1^\varepsilon \times \ldots \times Q_n^\varepsilon)$ where $Q_i^\varepsilon = Q_i \cup \bot$ is the set of witness lists of t according to PI-CS if it fulfills the following conditions:*

$$[[op(\mathscr{D}\mathscr{D}(op,t))]] = \{t^x\} \quad \textbf{(1)}$$

$$\forall w \in \mathscr{D}\mathscr{D}(op,t) : [[op(w)]] \neq \emptyset \quad \textbf{(2)}$$

$$\neg\exists \mathscr{D}\mathscr{D}' \subseteq (Q_1^\varepsilon \times \ldots \times Q_n^\varepsilon) : \mathscr{D}\mathscr{D}' \supset \mathscr{D}\mathscr{D}(op,t) \wedge \mathscr{D}\mathscr{D}' \models (1),(2),(4) \quad \textbf{(3)}$$

$$\forall w,w' \in \mathscr{D}\mathscr{D}(op,t) : w \prec w' \Rightarrow w \notin \mathscr{D}\mathscr{D}(op,t) \quad \textbf{(4)}$$

Condition **(4)** removes superfluous witness lists from the provenance of queries with outer joins and other forms of negation. However, it changes the semantics for the $\cup$ operator from that of Definition 2.1.

Under Definition 2.1, the provenance of a union result tuple would be a single witness list $< t_1, t_2 >$ if the result of the union is generated from a tuple t_1 from the left input and a tuple t_2 from the right input. We feel this is a bit misleading as it indicates that these two tuples *used together* influence t, when in fact each, independently, influences t. *PI-CS* captures this intuition by defining the provenance as $\{< t_1, \perp >, < \perp, t_2 >\}$. If the union semantics of Definition 2.1 is desired, we can easily achieve it with a simple post-processing rule to "repair" the provenance for unions. We define a operation $+$ for two witness lists w_1 and w_2 that combines them into a new witness list w by taking an input tuple from w_1 if w_2 is $\perp$ on this input and vice versa. If both w_1 and w_2 are not $\perp$ on at least one input, the operation is undefined. Notice that this post-processing does not influence the provenance of outer joins as it is defined on the provenance of a single tuple t.

$$\forall w, w' \in \mathscr{DD}(q,t) : w + w' = w'' \Rightarrow w'' \in \mathscr{DD}(q,t) \wedge w, w' \notin \mathscr{DD}(q,t) \quad \textbf{(U)}$$

For some cases the provenance of set difference under *WL-CS* represents the semantics of this operation more accurately than *PI-CS* (For instance, query q_d from Figure 2.5). Therefore, we implement both *PI-CS* data provenance after definition 2.7, and an alternative semantics which uses the *WL-CS* definition for union and set difference operators.

2.2.2.1 Transitivity

We define the *PI-CS* provenance of an algebra expression q containing more than one operator as recursively substituting tuples in a witness list for one of the operator in q with the witness lists for this tuple. I.e., a witness list w of an operator op_1 in q contains tuples from the input

Example 2.7. *The provenance of an algebra expression $q = \sigma_{a=b}(R \times S)$ is computed by first computing the provenance of $q' = \sigma_{a=b}([[R \times S]])$. Assume that $\mathscr{W}(q', (1,2)) = w_1 =< (1,2) >$. $(1,2)$ is a tuple from the result of the cross product. We proceed by computing the provenance of $w_2 = (1,2)$ in $R \times S$. The provenance of this tuple is the witness list $< (1), (2) >$. Now $(1,2)$ in w_1 is replaced by the contents of w_2 resulting in the witness list $< (1), (2) >$ for $\mathscr{DD}(q, (1,2))$.*

Definition 2.8 stated below defined the provenance of an algebra expression according to *PI-CS*.

Definition 2.8 (*PI-CS* for Algebra Expressions)**.** *The provenance according to PI-CS for an algebra expression $q = un(q_1)$ or $q = q_1$ bin q_2 where un is an unary operator and bin is a binary operator is defined as:*

If $q = un(q_1)$:

$$\begin{aligned}\mathscr{DD}(q,t) = &\{w =< v_1, \ldots, v_n >^p \mid \exists w' =< u >^m \in \mathscr{DD}(un(Q_1), t) \wedge u \in Q_1 \wedge w^p \in \mathscr{DD}(q_1, u)\} \\ &\cup \{w =< \perp, \ldots, \perp >^m \mid \exists w' =< \perp >^m \in \mathscr{DD}(un(Q_1), t)\}\end{aligned}$$

If $q = q_1$ bin q_2:

$$\begin{aligned}\mathscr{DD}(q,t) = \{w =< u_1, \ldots, u_n, v_1, \ldots, v_m >^{q \times r} \mid & w' =< u, v >^p \in \mathscr{DD}(Q_1 \text{ bin } Q_2, t) \\ & \wedge u \in Q_1 \wedge w' =< u_1, \ldots, u_n >^q \in \mathscr{DD}(q_1, u) \\ & \wedge v \in Q_2 \wedge w'' =< v_1, \ldots, v_m >^r \in \mathscr{DD}(q_2, v)\} \\ \cup \{w =< u_1, \ldots, u_n, \perp, \ldots, \perp >^q \mid & w' =< u, \perp >^p \in \mathscr{DD}(Q_1 \text{ bin } Q_2, t) \\ & \wedge u \in Q_1 \wedge w' =< u_1, \ldots, u_n >^q \in \mathscr{DD}(q_1, u)\} \\ \cup \{w =< \perp, \ldots, \perp, v_1, \ldots, v_m >^q \mid & w' =< \perp, v >^p \in \mathscr{DD}(Q_1 \text{ bin } Q_2, t) \\ & \wedge v \in Q_2 \wedge w'' =< v_1, \ldots, v_m >^q \in \mathscr{DD}(q_2, v)\} \\ \cup \{w =< \perp, \ldots, \perp >^1 \mid & w' =< \perp, \perp >^1 \in \mathscr{DD}(Q_1 \text{ bin } Q_2, t)\}\end{aligned}$$

$$
\begin{aligned}
\mathscr{DD}(R,t) &= \{<u>^n \mid u^n \in R \wedge u = t\} \\
\mathscr{DD}(\sigma_C(q_1),t) &= \{<u>^n \mid u^n \in Q_1 \wedge u = t\} \\
\mathscr{DD}(\Pi_A(q_1),t) &= \{<u>^n \mid u^n \in Q_1 \wedge u.A = t\} \\
\mathscr{DD}(\alpha_{G,agg}(q_1),t) &= \{<u>^n \mid u^n \in Q_1 \wedge u.G = t.G\} \cup \{<\perp> \mid Q_1 = \emptyset \wedge \mid G \mid = 0\} \\
\mathscr{DD}(q_1 \bowtie_C q_2,t) &= \{<u,v>^{n\times m} \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1} \wedge v^m \in Q_2 \wedge v = t.\mathbf{Q_2}\} \\
\mathscr{DD}(q_1 \rtimes_C q_2,t) &= \begin{cases} \{<u,\perp>^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\} & \text{if } \neg t \models C \\ \{<u,v>^{n\times m} \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1} \wedge v^m \in Q_2 \wedge v = t.\mathbf{Q_2}\} & \text{else} \end{cases} \\
\mathscr{DD}(q_1 \ltimes_C q_2,t) &= \begin{cases} \{<\perp,u>^n \mid u^n \in Q_2 \wedge u = t.\mathbf{Q_2}\} & \text{if } \neg t \models C \\ \{<u,v>^{n\times m} \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1} \wedge v^m \in Q_2 \wedge v = t.\mathbf{Q_2}\} & \text{else} \end{cases} \\
\mathscr{DD}(q_1 \rtimes\ltimes_C q_2,t) &= \begin{cases} \{<u,\perp>^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\} & \text{if } \neg t \models C \wedge t.\mathbf{Q_2} \text{ is } \varepsilon \\ \{<\perp,u>^n \mid u^n \in Q_2 \wedge u = t.\mathbf{Q_2}\} & \text{if } \neg t \models C \wedge t.\mathbf{Q_1} \text{ is } \varepsilon \\ \{<u,v>^{n\times m} \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1} \wedge v^m \in Q_2 \wedge v = t.\mathbf{Q_2}\} & \text{else} \end{cases} \\
\mathscr{DD}(q_1 \cup q_2,t) &= \{<u,\perp>^n \mid u^n \in Q_1 \wedge u = t\} \cup \{<\perp,u>^n \mid u^n \in Q_2 \wedge u = t\} \\
\mathscr{DD}(q_1 \cap q_2,t) &= \{<u,v>^{n\times m} \mid u^n \in Q_1 \wedge u = t \wedge v^m \in Q_2 \wedge v = t\} \\
\mathscr{DD}(q_1 - q_2,t) &= \{<u,\perp>^n \mid u^n \in Q_1 \wedge u = t\}
\end{aligned}
$$

Figure 2.6: Compositional Semantics for *PI-CS* for Single Operators

Though it might appear to be quite complex, this definition simply states that the witness list of an algebra expression q is created by replacing tuples in each witness list w of the outmost operator of q with the content of witness lists for these tuples (or $\perp$, if w contains $\perp$).

2.2.2.2 Compositional Semantics

Like for *Lineage-CS* we present an compositional semantics for *PI-CS* and prove its equivalence to the declarative semantics defined by Definition 2.7. The compositional semantics of *PI-CS* for each algebra operator are shown in Figure 2.6.

Example 2.8. *The PI-CS provenance $\mathscr{DD}$ of the queries from the running example is presented in Figure 2.5. Note the difference between WL-CS and PI-CS for queries with outer joins and set difference (q_b,q_d,q_e, and q_f).*

Theorem 2.2 (Equivalence of Compositional and Declarative Semantics of *PI-CS*)**.** *The compositional and declarative semantics of PI-CS are equivalent.*

Proof. Let $\mathscr{I}(op,t)$ be the witness set produced by the declarative semantics and $\mathscr{DD}(op,t)$ be the witness set produced by the compositional semantics. We have to show that $\mathscr{I}(op,t) = \mathscr{DD}(op,t)$ holds. This proposition can be proven by showing that $\mathscr{DD}(op,t)$ fulfills conditions 1 to 4 from definition 2.7 and, thus, is indeed equal to $\mathscr{I}(op,t)$.
Case $q = R$:
Obvious from the definition of R.
Case $q = \sigma_C(q_1)$:
Condition **(1)**: Because $u = t$ and t is in the result of q we know that $t \models C$. Therefore, $[[\sigma_C(u^n)]] = \{t^n\}$ and condition 1 holds.
Condition **(2)**: Using the same reasoning as for condition 1 we deduce that $[[\sigma_c(\{u\})]] = \{u\} \neq \emptyset$.

Condition **(3)**: We prove that $\mathscr{DD}(q,t)$ is maximal by contradiction. Assume a set $\mathscr{O} \supset \mathscr{DD}$ exists that fulfills conditions 1, 2, and 4. Then $\mathscr{O}$ contains a witness $w =< t' >$ with $t' \neq t$. If $t' \models C$ then $\mathscr{O}$ does not fulfill condition 1. Else $t' \not\models C$ holds. Hence, $[[\sigma_C(\{t'\})]] = \emptyset$ and condition 2 is not fulfilled.
Condition **(4)**: No witness w from $\mathscr{DD}$ contains $\perp$ and, therefore, condition 4 trivially holds.
Case $q = \Pi^{S/B}{}_A(q_1)$:
Condition **(1)**: Because $u.A = t$ we conclude that $\left[\left[\Pi^S{}_A(\{u^n\})\right]\right] = \{t^1\}$ and $\left[\left[\Pi^B{}_A(\{u^n\})\right]\right] = \{t^n\}$ holds. Thus, condition 1 is fulfilled.
Condition **(2)**: Using the same reasoning as in the proof of condition 1 we conclude that $\left[\left[\Pi^{S/B}{}_A(\{u\})\right]\right] = \{t\} \neq \emptyset$.
Condition **(3)**: Assume a set $\mathscr{O} \supset \mathscr{DD}$ exists that fulfills conditions 1,2, and 4. Then $\mathscr{O}$ contains a witness $w =< t' >$ with $t' \notin \mathscr{DD}$. From the compositional semantics of Π we know that $t'.A \neq t.A$, because otherwise t' would be contained in $\mathscr{DD}$. Therefore, $\left[\left[\Pi^{S/B}{}_A(\{t'\})\right]\right] \neq \{t^x\}$ and condition 1 is not fulfilled by $\mathscr{O}$.
Condition **(4)**: No witness w from $\mathscr{DD}$ contains $\perp$ and, therefore, condition 4 trivially holds.
Case $\alpha_{G,agg}(q_1)$:
Condition **(1)**: Assume $\left[\left[\alpha_{G,agg}(Q_1{}^R)\right]\right] \neq \{t\}$. Then $\left[\left[\alpha_{G,agg}(Q_1{}^R)\right]\right]$ contains a tuple $t' \neq t$. We know $t'.G = t.G$ because of the definition of aggregation and the compositional semantics of aggregation. Therefore, $t.agg \neq t'.agg$ holds. At least for one aggregation function agg_i from agg the tuples t and t' contain a different result: $t.agg_i \neq t'.agg_i$. But from

$$t.agg_i = agg_i(\Pi^B{}_{B_i}(\sigma_{G=t.G}(q_1))) = agg_i(\Pi^B{}_{B_i}(\{u \mid u.G = t.G \wedge u \in Q_1\})) = t'.agg_i$$

follows $t = t'$.
Condition **(2)**: From the definition of the aggregation operator we know that the result of this operator is never the empty set. Thus, condition 2 holds.
Condition **(3)**: Assume a set $\mathscr{O} \supset \mathscr{DD}$ exists that fulfills conditions 1, 2, and 4. Then $\mathscr{O}$ contains a witness list $w =< t' >$ with $t' \notin \mathscr{DD}$. If $t'.G \neq t''.G$ for $t'' \in \mathscr{DD}$ holds, then condition 1 is not fulfilled, because t' belongs to another group than t'' which by definition belongs to the same group as t. If t' belongs to the same group as t'' then it would be included in $\mathscr{DD}$.
Condition **(4)**: No witness list w from $\mathscr{DD}$ contains $\perp$ and, therefore, condition 4 trivially holds.
Case $q_1 \bowtie_C q_2$:
Condition **(1)**: From the definition of the compositional semantics of *PI-CS* provenance for the join operator we know that $t = (u,v)$ holds for each witness list $w =< u,v >$ in $\mathscr{DD}$. Applying the definition of the join operator we get $[[\{u\} \bowtie_C \{v\}]] = \{t\}$ and according to the semantics attached to computing an operator over a set of witnesses: $[[op(\mathscr{DD})]] = \{t^{n \times m}\}$.
Condition **(2)**: From the proof of condition 1 we know that $[[\{u\} \bowtie_C \{v\}]] = \{t\} \neq \emptyset$ and, therefore, condition 2 holds.
Condition **(3)**: Assume a set $\mathscr{O} \supset \mathscr{DD}$ exists that fulfills conditions 1,2, and 4. Then $\mathscr{O}$ contains a witness $w =< u',v' > \notin \mathscr{DD}$. We know that $(u',v') \neq t$. Either $(u',v') \models C$ which breaks condition 1 or $(u',v') \not\models C$ which breaks condition 2.
Condition **(4)**: No witness w from $\mathscr{DD}$ contains $\perp$ and, therefore, condition 4 trivially holds.
Case $q_1 \overrightarrow{\bowtie} q_2$:
We present only the proof for the case $t \not\models C$ because if $t \models C$ holds then the behavior and provenance of the left outer join is the same as for the inner join.
Condition **(1)**: For each witness list $w =< u, \perp > \in \mathscr{DD}$: $[[\{u\} \overrightarrow{\bowtie}_C \emptyset]] = \{t\}$. Therefore, $[[op(\mathscr{DD})]] = \{t^x\}$ and condition 1 holds.
Condition **(2)**: The left join operator never produces the empty set for an non empty left input. Thus, because each witness list in $\mathscr{DD}$ is of the form $< u, \perp >$ we can deduce that condition 2 holds.
Condition **(3)**: We prove the maximality of $\mathscr{DD}$ by contradiction. Assume a set $\mathscr{O} \supset \mathscr{DD}$ exists that fulfills conditions 1, 2, and 4 from the *PI-CS* definition. Then $\mathscr{O}$ contains a witness list $w' =< u',v' > \notin \mathscr{DD}$. $u' = t.\mathbf{Q_1}$ has to hold because else condition 1 would not be fulfilled. It follows that $v' \neq \perp$, because $< u', \perp > \in \mathscr{DD}$. Clearly, $w' \prec < u', \perp >$, which breaks condition 4.
Condition **(4)**: Each witness in $\mathscr{DD}$ is of the form $< u, \perp >$. Thus, for two witnesses w and w' the condition $w \prec w'$ can never be fulfilled. If follows that condition 4 holds.

Case $q_1 \rtimes\!\!\!\subset q_2$:
The proof for right outer join is analog to the proof for left outer join.
Case $q_1 \supset\!\!\!\rtimes\!\!\!\subset q_2$:
For full outer join we have to distinguish two cases: ($t \not\models C \wedge t.\mathbf{Q_1}$ is ε) and ($t \not\models C \wedge t.\mathbf{Q_2}$ is ε). Both cases can be proven analog to the proof for left outer join.
Case $q_1 \cup q_2$:
Condition **(1)**: Each witness list w from $\mathscr{D}\mathscr{D}$ is either $< u, \bot >$ or $< \bot, u >$ with $u = t$. Applying the definition of the union operator we get $[[\{u\} \cup \emptyset]] = [[\emptyset \cup \{u\}]] = \{t\}$. Since $[[op(\mathscr{D}\mathscr{D})]]$ is defined as the applying op to the union of the input to $[[op(w)]]$ for $w \in \mathscr{D}\mathscr{D}$, condition 1 holds.
Condition **(2)**: Using the fact $[[\{u\} \cup \emptyset]] = [[\emptyset \cup \{u\}]] = \{t\}$ established in the proof of condition 1 we conclude that condition 2 is fulfilled.
Condition **(3)**: Assume a set $\mathscr{O} \supset \mathscr{D}\mathscr{D}$ exists that fulfills conditions 1,2, and 4. Then $\mathscr{O}$ contains a witness list $w =< u', v' >$ with $w \notin \mathscr{D}\mathscr{D}$. If either u' or v' are neither $\bot$ nor equal to t, then condition 1 is not fulfilled. If only one of u' and v' is equal to $\bot$ then w would be in $\mathscr{D}\mathscr{D}$. For the remaining two cases ($w =< \bot, \bot >$ and $w =< t, t >$) either condition 2 or condition 4 is not fulfilled.
Condition **(4)**: All witness lists from $\mathscr{D}\mathscr{D}$ are either of the form $< u, \bot >$ or $< \bot, u >$. Thus, there are no two witness lists w and w' from $\mathscr{D}\mathscr{D}$ for which the precondition $w \prec w'$ from condition 4 is fulfilled and condition 4 holds.
Case $q_1 \cap q_2$:
Condition **(1)**: Each witness list w from $\mathscr{D}\mathscr{D}$ is of the form $w =< u, v >$ with $u = v = t$. Since $[[\{t\} \cap \{t\}]] = \{t\}$ condition 1 holds.
Condition **(2)**: Since intersection is symmetric it suffices to show that $\left[\left[\{u^n\} \cap \{u^1\}\right]\right] \neq \emptyset$ which trivially holds.
Condition **(3)**: We prove the maximality of $\mathscr{D}\mathscr{D}$ by contradiction. Assume a super-set $\mathscr{O}$ of $\mathscr{D}\mathscr{D}$ exists that fulfills conditions 1,2 and 4 from the definition. Then $\mathscr{O}$ has to contain a witness list that includes a tuple $t' \neq t$ that is not in $\mathscr{D}\mathscr{D}$. w.l.o.g. assume $t' \in {Q_1}^R$. Then for condition 2 to hold $\left[\left[\{t'\} \cap {Q_2}^R\right]\right] \neq \emptyset$ has to be true. Either ${Q_2}^R$ contains a tuple t'' that is equal to t', then condition 2 is fulfilled, but condition 1 is no longer fulfilled because $\left[\left[{Q_1}^R \cap {Q_2}^R\right]\right] \neq \{t^x\}$. Or ${Q_2}^R$ does not contain such an tuple and, therefore, $\left[\left[\{t'\} \cap {Q_2}^R\right]\right] = \emptyset$ would hold which contradicts condition 2. Hence, $\mathscr{D}\mathscr{D}$ is maximal.
Condition **(4)**: No witness list w from $\mathscr{D}\mathscr{D}$ contains $\bot$ and, therefore, condition 4 trivially holds.
Case $q_1 - q_2$:
Condition **(1)**: Each witness list in $\mathscr{D}\mathscr{D}$ is of the form $< u, \bot >$. Since $[[\{u\} - \emptyset]] = \{u\}$ condition 1 holds.
Condition **(2)**: Follows from the proof of condition 1.
Condition **(3)**: Assume a set $\mathscr{O} \supset \mathscr{D}\mathscr{D}$ exists that fulfills conditions 1,2, and 4. Then $\mathscr{O}$ contains a witness $w =< u', v' >$ with $w \notin \mathscr{D}\mathscr{D}$. $u' = t$ has to hold otherwise condition 1 would break. If $v' = \bot$ then w would be in $\mathscr{D}\mathscr{D}$. Else, a $w' =< t, \bot > \in \mathscr{D}\mathscr{D}$ subsumes w which would break condition 4.
Condition **(4)**: Every witness list in $\mathscr{D}\mathscr{D}$ is of the form $< u, \bot >$. Thus, there are no two witness lists w and w' from $\mathscr{D}\mathscr{D}$ for which the precondition $w \prec w'$ from condition 4 is fulfilled and condition 4 holds. □

2.2.3 Comparison of the Expressiveness of *Lineage-CS* and *PI-CS*

Having iteratively refined the definitions of *Lineage-CS* and *PI-CS* to produce meaningful provenance for a wide set of algebra expressions we now compare the expressiveness of the two contribution semantics. In detail, we answer the question: Does the provenance generated by both contribution semantics contain the same information? Of course this question can only be answered for the operators on which both semantics are meant to produce the same provenance. For instance, the provenance for left outer joins is different for *Lineage-CS* and *PI-CS*.

Intuitively, it is clear that *PI-CS* provenance contains information that is not modeled by *Lineage-CS*, because, for instance, for a set of result tuples of a join the representation used by *Lineage-CS* does not model which tuples were used together by the join. On the other hand, the representation of *PI-CS* does not model the original multiplicity of input tuples. Formally, this intuitions can be proven, by showing that every function that translates between these two representations cannot have an inverse.

Theorem 2.3 (Non Equivalence of *PI-CS* and *Lineage-CS*)**.** *Let function H be a function that maps a witness set to a set of witness lists with the property $H(\mathscr{W}(q,I,t)) = \mathscr{DD}(q,I,t)$. Let H' be a function that maps a set of witness lists to a witness set with $H'(\mathscr{DD}(q,I,t)) = \mathscr{W}(q,I,t)$. It follows that both H and H' are not invertible and, therefore, none of them can exist.*

Proof.
Case H:

If H has an inverse, then H has to be injective and surjective. Thus, if we find two queries q_1 and q_2,database instance I_1 and I_2, and tuples $t_1 \in Q_1$ and $t_2 \in Q_2$ for which $\mathscr{W}(q_1,I_1,t_1) \neq \mathscr{W}(q_2,I_2,t_2)$ and $\mathscr{DD}(q_1,I_1,t_1) = \mathscr{DD}(q_2,I_2,t_2)$ hold, we have proven our claim by demonstrating that H is not injective (given that either $q_1 \neq q_2$, $I_1 \neq I_2$, or $t_1 \neq t_2$). Consider the following queries, database instances, and result tuples:

$$\begin{aligned} q_1 &= R \rtimes_{a=b} S & \qquad q_2 &= R \ltimes_{a=b} S \\ t_1 &= (1,1) & \qquad t_2 &= (1,1) \\ I_1 &= \{R = \{(1)\}, S = \{(1)^2\}\} & \qquad I_2 &= \{R = \{(1)^2\}, S = \{(1)\}\} \end{aligned}$$

As shown below for this parameter combinations no injective function H can exists that fulfills the condition $H(\mathscr{W}(q,I,t)) = \mathscr{DD}(q,I,t)$:

$$\begin{aligned} \mathscr{W}(q_1,I_1,t_1) &= <\{(1)\},\{(1)^2\}> & \neq & & <\{(1)^2\},\{(1)\}> &= \mathscr{W}(q_2,I_2,t_2) \\ \mathscr{DD}(q_1,I_1,t_1) &= \{<(1),(1)>,<(1),(1)>\} & = & \{<(1),(1)>,<(1),(1)>\} &= \mathscr{DD}(q_2,I_2,t_2) \end{aligned}$$

Case H':

We present an example for which $\mathscr{DD}(q_1,t_1) \neq \mathscr{DD}(q_2,t_2)$ and $\mathscr{W}(q_1,t_1) = \mathscr{W}(q_2,t_2)$ hold to prove that H' has no inverse:

$$\begin{aligned} q_1 &= \Pi^S{}_a(R \rtimes_{b=c} S) & \qquad q_2 &= \Pi^S{}_a(R \ltimes_{b\neq c} S) \\ t_1 &= (1) & \qquad t_2 &= (1) \\ I_1 &= \{R = \{(1,1),(1,2)\}, S = \{(1),(2)\}\} & \qquad I_2 &= \{R = \{(1,1),(1,2)\}, S = \{(1),(2)\}\} \end{aligned}$$

$$\begin{aligned} \mathscr{W}(q_1,I_1,t_1) &= <\{(1,1),(1,2)\},\{(1),(2)\}> & = & <\{(1,1),(1,2)\},\{(1),(2)\}> &= \mathscr{W}(q_2,I_2,t_2) \\ \mathscr{DD}(q_1,I_1,t_1) &= \{<(1,1),(1)>,<(1,2),(2)>\} & \neq & \{<(1,1),(2)>,<(1,2),(1)>\} &= \mathscr{DD}(q_2,I_2,t_2) \end{aligned}$$

□

Despite the fact that *PI-CS* and *Lineage-CS* are not equivalent which we have proven above, these *CS* types are nonetheless related to each other in the sense that they consider the same input relation tuples to belong to the provenance (with the obvious exception of operators such as left outer join for which *PI-CS* was deliberately defined to generate a different provenance than *Lineage-CS*). We prove this claim by presenting a third representation of provenance and functions H and H' that translate from the *PI-CS* and *Lineage-CS* representations into this representation. If we are able to define these functions in a way that $H(\mathscr{W}(q,I,t)) = H'(\mathscr{DD}(q,I,t))$ then we have shown that regarding the information stored in this new representation both *CS* types are equivalent. The reduced representation we use is the same as the one of *Lineage-CS* except that the $Q_i{}^*$ subsets of the input relations are sets instead of bags. Therefore, the translation from *Lineage-CS* to the new representation is trivial:

$$H(\mathscr{W}(q,I,t)) = <\{t \mid t^x \in Q_1{}^*\}, \ldots, \{t \mid t^x \in Q_n{}^*\}>$$

The definition of H' is straightforward too:

$$H'(\mathscr{DD}(Q,I,t)) = <\{t \mid \exists w \in \mathscr{DD} \wedge w[1] = t\}, \ldots, \{t \mid \exists w \in \mathscr{DD} \wedge w[n] = t\}>$$

Theorem 2.4 (Equivalence of the Reduced Representation of *Lineage-CS* and *PI-CS*)**.** *The reduced representations of Lineage-CS and PI-CS are equivalent for algebra expressions containing only the following operators:* $\Pi, \sigma, \ltimes, \cup, \cap, \alpha$*. Thus, for the translation functions H and H′ the following holds:*

$$H(\mathscr{W}(q,t)) = H'(\mathscr{DD}(q,t))$$

Proof. We prove this theorem by induction over the structure of an algebra expression q and for an arbitrary result tuple t of q:
Base Case:
$\underline{q = \sigma_C(q_1)}$:

$$\begin{aligned}
&H(\mathscr{W}(q,t))\\
=&H(< \{u^n \mid u^n \in Q_1 \wedge u = t\} >)\\
=&< \{t\} >\\
=&H'(\{< u >^n \mid u^n \in Q_1 \wedge u = t\})\\
=&H'(\mathscr{DD}(q,t))
\end{aligned}$$

$\underline{q = \Pi_A(q_1)}$:

$$\begin{aligned}
&H(\mathscr{W}(q,t))\\
=&H(< \{u^n \mid u^n \in Q_1 \wedge u.A = t\} >)\\
=&< \{u \mid u^n \in Q_1 \wedge u.A = t\} >\\
=&H'(\{< u >^n \mid u^n \in Q_1 \wedge u.A = t\})\\
=&H'(\mathscr{DD}(q,t))
\end{aligned}$$

$\underline{q = \alpha_{G,agg}(q_1)}$:

$$\begin{aligned}
&H(\mathscr{W}(q,t))\\
=&H(< \{u^n \mid u^n \in Q_1 \wedge u.G = t.G\} >)\\
=&< \{u \mid u^n \in Q_1 \wedge u.G = t.G\} >\\
=&H'(\{< u >^n \mid u^n \in Q_1 \wedge u.G = t.G\})\\
=&H'(\mathscr{DD}(q,t))
\end{aligned}$$

$\underline{q = q_1 \ltimes_C q_2}$

$$\begin{aligned}
&H(\mathscr{W}(q,t))\\
=&H(< \{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, \{v^m \mid v^m \in Q_2 \wedge v = t.\mathbf{Q_2}\} >)\\
=&< \{u \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, \{v \mid v^m \in Q_2 \wedge v = t.\mathbf{Q_2}\} >\\
=&H'(\{< u, v >^{n \times m} \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1} \wedge v^m \in Q_2 \wedge v = t.\mathbf{Q_2}\})\\
=&H'(\mathscr{DD}(q,t))
\end{aligned}$$

$\underline{q = q_1 \cup q_2}$

$$\begin{aligned}
&H(\mathscr{W}(q,t))\\
=&H(< \{u^n \mid u^n \in Q_1 \wedge u = t\}, \{v^m \mid v^m \in Q_2 \wedge v = t\} >)\\
=&< \{u \mid u^n \in Q_1 \wedge u = t\}, \{v \mid v^m \in Q_2 \wedge v = t >\\
=&H'(\{< u, \perp >^n \mid u^n \in Q_1 \wedge u = t\} \cup \{< \perp, v >^m \mid v^m \in Q_2 \wedge v = t\})\\
=&H'(\mathscr{DD}(q,t))
\end{aligned}$$

$\underline{q = q_1 \cap q_2}$

$$
\begin{aligned}
& H(\mathscr{W}(q,t)) \\
=& H(<\{u^n \mid u^n \in Q_1 \wedge u = t\}, \{v^m \mid v^m \in Q_2 \wedge v = t\}>) \\
=& <\{u \mid u^n \in Q_1 \wedge u = t\}, \{v \mid v^m \in Q_2 \wedge v = t\}> \\
=& H'(\{<u,v>^{n\times m} \mid u^n \in Q_1 \wedge u = t \wedge v^m \in Q_2 \wedge v = t\}) \\
=& H'(\mathscr{D}\mathscr{D}(q,t))
\end{aligned}
$$

Induction step:
Follows from the transitivity of *Lineage-CS* and *PI-CS*. □

2.3 Transformation Provenance Contribution Semantics

In this section we present the contribution semantics for *transformation* provenance developed for *TRAMP* and based on *PI-CS*. *data* provenance relates output and input data, but does not provide any information about how data was processed by a transformation. More specifically, it does not contain information about which parts of a transformation were used to derive an output tuple. As an example, consider a transformation that uses the duplicate preserving *union* operator. Each output tuple of the union is produced from exactly one of the relations that are the inputs of the union. Recall that the notion *transformation* provenance is used to describe this type of provenance information.

Transformation provenance is similar in motivation to *how*-provenance[8] that models some transformation information by recording alternative and conjunctive use of tuples by a query. Unlike *how*-provenance, *transformation* provenance is *operator-centric*, describing the contribution of each operator in a transformation. This is also in contrast to other *data CS* types, which are in general *data-centric*. Our approach is more similar to provenance approaches for workflow-management systems, that traditionally have focused more on transformations [9]. *Transformation* provenance is extremely useful in understanding how data is processed by an algebra expression, because it allows us to understand which parts of the expression (that is, which operators) produced a result data item. Hence, we model *transformation* provenance as what parts of a query contributed to an output tuple. This approach bears some similarities with *Why-not* provenance presented in [3], so we will discuss the relation to this model and the superior evaluation strategy developed for our model while introducing transformation provenance.

We model the transformation provenance of a query q using an annotated algebra tree for q. For an output tuple t and a witness list w in the *PI-CS* data provenance of t, the transformation provenance will include 1 and 0 annotations on the operators of the transformation q. A 1 indicates this operator on w influences t, a 0 indicates it does not.

Example 2.9. *Consider query q_a in Fig. 2.7. The data provenance of output tuple (2) according to PI-CS contains two witness lists. The transformation provenance of (2) for the first witness list is a tree with every node annotated by a 1 (the left tree presented in Figure 2.7). The transformation provenance of (2) with the second witness is a tree with every node annotated by a 1, except the node for the base relation S which does not contribute and hence would have an annotation of 0 (the right tree presented in Figure 2.7).*

We now formalize annotations for algebra trees and then define *transformation* provenance based on the notion of *data* provenance according to *PI-CS*.

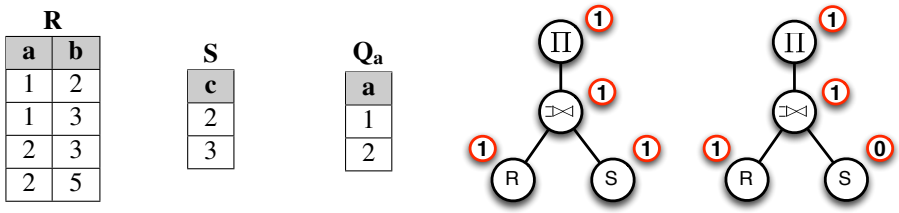
Definition 2.9 (Algebra Tree)**.** *An algebra tree $Tree_q = (V,E)$ for a query q is a tree that contains a node for each algebra operator used in q (including the base relation accesses as leaves). In such tree there is a parent-child relationship between two nodes n_1 and n_2, iff the algebra operator represented by n_2 is an input of the algebra operator represented by n_1. We define a pre-order on the nodes to give each node an identifier (and to order the children of binary operators)[a].*

[a] This is necessary, because for non commutative operators like left outer join the order of inputs matters.

Given an algebra expression q and an operator op used in this expression, we denote the subtree under op by sub_{op}. We use $sub_{op}(w)$ to denote the evaluation of sub_{op} over the witness list w. Based on the concept of the algebra tree of an algebra expression we define annotated algebra trees and will use them to represent transformation provenance information.

Definition 2.10 (Annotated Algebra Tree)**.** *An $\mathscr{A}$-annotated algebra tree for a transformation q is a pair $(Tree_q, \theta)$ where $\theta : V \in Tree_q \rightarrow Pow(\mathscr{A})$ is a function that associates each operator in the tree with a set of annotations from a domain $\mathscr{A}$.*

For transformation provenance, the annotations sets will be singleton sets from the domain $\mathscr{A} = \{0,1\}$ and we assign these annotations specific semantics. However, we include the more general definition as



$$q_a = \Pi_a(R \rtimes\!\ltimes_{b=c} S)$$

$$\mathscr{DD}(q_a,(1)) = \{<(1,2),(2)>,<(1,3),(3)>\} \qquad \mathscr{DD}(q_a,(2)) = \{<(2,3),(3)>,<(2,5),\perp>\}$$

$\mathscr{T}(q_c,(1))$:

$$\theta_{<(1,2),(2)>}(op) = 1 \qquad \theta_{<(1,3),(3)>}(op) = 1$$

$\mathscr{T}(q_c,(2))$:

$$\theta_{<(2,3),(3)>}(op) = 1 \qquad \theta_{<(2,5),\perp>}(op) = \begin{cases} 0 \text{ if } op = \mathbf{S} \\ 1 \text{ else} \end{cases}$$

Figure 2.7: Transformation Provenance Example

annotations could be used in a more general way to represent other provenance information (the developer who last checked-in a query, the origin of a transformation).

This representation is conceptually similar to the one use in [3] to model *Why-not* provenance. Recall that *Why-not* provenance models why a certain input (represented as a pattern) does not contribute to some result. This information is presented as so-called *picky* parts of a query, which means the parts of a query where the input of interest "got lost". In our representation the picky operations would be labeled with 0 annotations. In contrast to their work we do not require a user to come up with a certain input that got lost, but define the annotations based solely on *data* provenance. Transformation provenance can be computed efficiently without the need to compute *data* provenance which is the way *Why-not* provenance is computed in [3].

We now have the necessary preliminaries to formally define transformation provenance based on *data* provenance. Intuitively, each witness list of the *data* provenance of a tuple t represents one evaluation of an algebra expression q. For each witness list, each part of the algebra expression has either contributed to the result of evaluating q on w or not. Therefore, we represent the transformation provenance as a set of annotated algebra trees of q with one member per witness list w. We use *data* provenance to decide whether an operator op in q should get a 0 or a 1 annotation. Basically, if evaluating the subtree sub_{op} under op on w results in the empty set, then op has contributed nothing to the result t and should not be included in the transformation provenance.

Definition 2.11 (Transformation Provenance Contribution Semantics)**.** *The transformation provenance of an output tuple t of q is a set $\mathscr{T}(q,t)$ of $\{0,1\}$-annotated-trees defined as follows:*

$$\mathscr{T}(q,t) = \{(Tree_q, \theta_w) \mid w \in \mathscr{DD}(q,t)\}$$

$$\theta_w(op) = \begin{cases} 0 \text{ if } [[sub_{op}(w)]] = \emptyset \\ 1 \text{ else} \end{cases}$$

Example 2.10. *Fig. 2.7 shows the PI-CS data and transformation provenance for both result tuples of query q_a. The PI-CS provenance of tuple $t_1 = (1)$ contains two witness lists $w_1 =< (1,2),(2) >$ and $w_2 =< (1,3),(3) >$. For both witness lists the transformation provenance annotation function θ_w annotates each operator of q with 1. We can verify that this is correct by computing $[[sub_{op}(w_1)]]$ and $[[sub_{op}(w_2)]]$ for each operator in the query. For the second result tuple $t_2 = (2)$ there are two witness lists $w_3 =< (2,3),(3) >$ and $w_4 =< (2,5),\perp >$. The annotation function θ_{w_4} for witness list w_4 annotates S with 0 and every other operator in q with 1. S is not contained in the transformation provenance, because no tuple from S was joined with the tuple $(2,5)$ from R to produce tuple t_2 and, therefore, the access to relation S does not contribute to t_2 according to w_4.*

(a)

$$q_a = \Pi^{2}_{SK_1(name),first(name),last(name),SK_2(name)}(A^3)$$
$$\cup^{1} \Pi^{4}_{SK_3(A.name,I.name),first(A.name),last(A.name),I.name}(A^6 \bowtie^{5}_{inst=instId} I^7)$$
$$q_b = \Pi^{1}_{SK_1(A.name,I.name),first(A.name),last(A.name),I.name}(A^3 \overline{\bowtie}^{2}_{inst=instId} I^4)$$

(b)

$$\mathscr{M}(q_b,t_1) = \{\{M_2\}\}$$
$$\mathscr{M}(q_b,t_2) = \{\{M_1\}\}$$
$$\mathscr{M}(q_b,t_3) = \{\{M_2\}\}$$

(c)

q_a

$$\mu_{M_1}(op) = \begin{cases} 1 & \forall op \in \{1,2,3\} \\ 0 & \forall op \in \{4,5,6,7\} \end{cases}$$
$$\mu_{M_2}(op) = \begin{cases} 1 & \forall op \in \{1,4,5,6,7\} \\ 0 & \forall op \in \{2,3\} \end{cases}$$

q_b

$$\mu_{M_1}(op) = \begin{cases} 1 & \forall op \in \{1,2,3\} \\ 0 & \forall op \in \{4\} \end{cases}$$
$$\mu_{M_2}(op) = 1$$

(d)

Author

name	inst
Peter Smith	1
Helga Hauser	ε
Hilde Schneider	2

Institute

instId	name	location
1	A	Denver
2	B	Dallas

Q_b

authId	firstName	lastName	affiliation
$SK_1(PeterSchmith,A)$	Peter	Smith	A
$SK_1(HelgaHauser,\varepsilon)$	Helga	Hauser	ε
$SK_1(HildeSchneider,B)$	Hilde	Schneider	B

Figure 2.8: Mapping Provenance Example

2.4 Mapping Provenance Contribution Semantics

Since *TRAMP* extends *Perm* with schema mapping debugging functionality, it was necessary to introduce a new type of transformation provenance that takes the mappings into account from which a transformation is dervied from. In a mapping scenario, transformations are derived from a set of declarative schema mappings. For example, consider the mappings $\mathbf{M_1}$ and $\mathbf{M_2}$ from the motivation. Two possible implementations of these mappings are presented in Figure 2.8(a) (we use A and I as a shortcut for the *Author* and *Institute* relations). In our example, SK_1, SK_2, and SK_3 are skolem functions which may be generated by a mapping system or user-defined functions that are used to fill in values for un-mapped attributes. Notice that both q_a and q_e implement M_1 and M_2, but they are not equivalent.

They may produce different target data for the same source instance. Both actually produce *universal solutions* [**?**], and so may be generated by mapping tools that are based on data exchange theory [**?**, **?**, and others]. M_2 may produce a less redundant target, but both these and other transformations may be generated by a mapping tool. Hence, in debugging mappings and transformations, we would like to know not only what parts of a transformation produced a target tuple t (the *transformation* provenance of t), but also from what mappings these transformations (or operators withing a transformation) were derived. Hence, we define mapping provenance based on transformation provenance and the correspondences between transformations and mappings (the $\mathscr{A}$ relation of a mapping scenario MS).

Example 2.11. *For the first transformation q_a, the correspondence between mappings and transformations is quite clear. The union and the left input of the union corresponds to mapping* $\mathbf{M_1}$ *in the sense that they process tuples to create target data as specified by* $\mathbf{M_1}$*. The union and its right input corresponds to mapping* $\mathbf{M_2}$*. For the second transformation q_b, the entire transformation implements mapping* $\mathbf{M_2}$*, while* $\mathbf{M_1}$ *is implemented by the join with only its left input (A) and the final projection which provides values for unmapped target elements and implements functions specified in both mappings.*

The *mapping* provenance of an tuple t that was generated by some transformation q should contain all the mappings that contributed to t indirectly by corresponding to a part of the transformation that is in the transformation provenance of t. Therefore, mapping provenance can be defined on-top of transformation provenance.

A correspondence between a mapping and a part of a transformation is modeled by adding additional annotations (specifically, mapping identifiers) to the algebra tree for a transformation. For an algebra tree, $Tree_q = (V,E)$, we introduce one new annotation function, μ_M, per mapping $M \in \Sigma_{st}$. The function μ_M is 1 for each operator that implements this mapping, 0 otherwise.

Example 2.12. *For example, consider transformation q_a in Figure 2.8(a) and the annotation functions for this transformation (Figure 2.8(c)). We use the superscript (red) preordering to refer to the individual operators of a query (these are actually the node identifiers for algebra tree nodes used for transformation provenance). The operator 3 (representing the relation A), implements M_1 and tuples from A may flow through every operator above 3 in the tree so μ_{M_1} annotates every operator on the path from 3 (A) to the root with a 1, and all other operators with a 0. Similarly, the operators 6 (A) and 7 (I) implement M_2, as does every operator above these two in the tree. So μ_{M_2} annotates each of the nodes 1,4,5,6,7 with a 1 and all other nodes with a 0.*

Figure 2.8(c) also presents the two mapping annotation functions for transformation q_b. Here, there is a single node for A (Node 3) which implements both M_1 and M_2. The node for I (Node 4) implements only M_2. Hence, μ_{M_1} assigns a 1 to operator 3 and all nodes above 3 in the tree, and μ_{M_2} assigns a 1 to operator 4 and all nodes above 4 in the tree.

Notice that the mapping annotation function will depend on the language used for mappings. We have implemented mapping annotation functions for source-to-target tgds, but of course this could be extended to other languages, including the visual mapping languages of some commercial tools. Below we formalize the notation of *mapping* provenance using the annotation functions μ_M (Recall that θ_w is the *transformation* provenance annotation function for witness list w).

Definition 2.12 (Mapping Provenance)**.** *The mapping provenance $\mathscr{M}(q,t)$ for a tuple t from the result of query q is defined using the mapping annotation functions μ_M over the transformation provenance $\mathscr{T}(q,t)$ as follows:*

$$\mathscr{M}(q,t) = \{\mathscr{M}_w \mid w \in \mathscr{DD}(q,t)\}$$
$$\mathscr{M}_w = \{M \mid \forall op \in V : \mu_M(op) = \theta_w(op)\}$$

Example 2.13. *As an example of mapping provenance consider Figure 2.8(d), which presents an instance of the Author and Institute relation, the result of applying q_b on this instance, and the mapping provenance (b) for each result tuple of q_b.*

2.5 Summary

In this chapter we presented the *contribution semantics* realized in *Perm* and its extension *TRAMP*. *CS* types are of immense importance for a provenance management system, because they define ”What provenance actually is”. We discussed how our definitions of *CS* relate to existing *CS* types and to each other, and how we extended the existing notion of *Lineage-CS* to overcome problems of the original definition and make it applicable to algebra expression with sublinks. Furthermore, for *TRAMP* we introduced a contribution semantics for transformation provenance based on the *PI-CS data* provenance contribution semantics. In summary, in this chapter we presented sound formal definitions of the semantics of provenance, but did not discuss how to compute provenance according to a certain *CS* type. Note that the compositional semantics can be used to compute provenance by recursively applying the constructions from this semantics for each operator in a query, but, as we will demonstrate in the next chapter, there are more practical and efficient approaches for computing provenance.

Chapter 3

Provenance Computation through Algebraic Rewrite

In the last chapter we introduced the *CS* types developed for *Perm* and prove several import properties of their compositional semantics and relationships to standard *CS* types. However, we did not discuss how provenance can be computed efficiently according to these *CS* types. In this chapter we present algorithms that allow the efficient computation of provenance by using algebraic rewrites. In detail, we demonstrate how to represent provenance information as normal relations and introduce rewrite rules that transform an algebra expression q into an algebra expression q^+ that computes the provenance of q in addition to its original result.

Out of the possible approaches to provenance computation we choose algebraic rewrite, because this approach has several important advantages over alternative approaches like, e.g., the inverse approach:

- **No Modification of Data Model**: Provenance information is modeled as normal relations which can be, e.g., stored in a standard DBMS, queried using SQL, and stored as a view.
- **No Modification of Execution Model**: The rewritten query q^+ that computes the provenance of a query q is expressed in the same algebra as q (To be more precise, the *Perm* algebra introduced in section 2.1). This will allow for the seamless integration of the rewrite rules into an existing DBMS and enables us to benefit from the advanced query optimizations applied by this system. Furthermore, provenance information can be queried using the same query language as for normal data.
- **Sound Theoretical Foundation**: The algebraic representation of provenance computation enables us to prove the correctness of the developed algorithms.

In the following we discuss how to represent provenance in the relational model in section 3.1 and demonstrate how we can transform a query q into a query q^+ that generates this representation for *PI-CS* provenance (section 3.2). In section 3.3 we present a relational representation of *transformation* provenance and rewrite rules for this type of provenance in section 3.4.

shop

name	numEmpl
Merdies	3
Joba	14

sales

sName	itemId
Merdies	1
Merdies	2
Merdies	2
Joba	3
Joba	3

items

id	price
1	100
2	10
3	25

Q_{ex}

name	sum(price)
Merdies	120
Joba	50

$$q_{ex} = \alpha_{name,sum(price)}(\sigma_{name=sName \wedge itemId=id}(shop \times sales \times items))$$

$$\mathscr{DD}(q_{ex},(Merdies,120)) = \{ <(Merdies,3),(Merdies,1),(1,100)>, \\ <(Merdies,3),(Merdies,2),(2,50)>^2\}$$
$$\mathscr{DD}(q_{ex},(Joba,50)) = \{ <(Joba,14),(Joba,3),(3,25)>^2\}$$

Figure 3.1: Running Example

3.1 Relational Representation of Data Provenance Information

The witness lists used by the *Perm CS* types for *data* provenance have a natural representation in the relational model. For instance, the set $\mathscr{DD}$ of witness lists generated by *PI-CS* can be represented as a single relation, because each witness list contains tuples with the same schema (We postpone the discussion of the representation of $\perp$ for now).

Example 3.1. *As an example of this representation consider the provenance of query q_{ex} presented in Figure 3.1 that computes the total profit for each shop over an example database of shops (with name and number of employees), items they are selling, and purchases (sales relation). The witness lists from $\mathscr{DD}(q_{ex},(Merdies,120))$ can be represented as the following relation:*

$$\{(Merdies,3,Merdies,1,1,100),(Merdies,3,Merdies,2,2,10)^2\}$$

If this representation is used to represent complete sets of witness lists according to some *CS* type the problem arises that it is no longer clear to which output tuple of a query a witness lists belongs to. Recall that we identified this association between original query results and provenance as one of the main requirements of a *PMS*. Therefore, we include the original query results in our representation of *data* provenance. The extended representation models each witness list w and the original result tuple it is associated to as a single tuple.

Example 3.2. *The PI-CS provenance of query q_{ex} would be represented as (The part of a tuple marked in red corresponds to the original result tuple) :*

$$\{(Merdies,120,Merdies,3,Merdies,1,1,100), \\ (Merdies,120,Merdies,3,Merdies,2,2,10)^2, \\ (Joba,50,Joba,14,Joba,3,3,25)^2,\}$$

Let us now consider the provenance representation for an arbitrary algebra statement q. We use Q^{PI} to denote the relational representation of the provenance of a query q according to *PI-CS*. To produce the provenance relation Q^{PI}, the original result relation Q is extended with all attributes from all base

$\mathbf{Q_{ex}^{PI}}$

name	sum(price)	$\mathscr{N}$(name)	$\mathscr{N}$(numEmpl)	$\mathscr{N}$(sName)	$\mathscr{N}$(itemId)	$\mathscr{N}$(id)	$\mathscr{N}$(price)
Merdies	120	Merdies	3	Merdies	1	1	100
Merdies	120	Merdies	3	Merdies	2	2	100
Merdies	120	Merdies	3	Merdies	2	2	100
Joba	50	Joba	14	Joba	3	3	25
Joba	50	Joba	14	Joba	3	3	25

Figure 3.2: Example Provenance Representation

relations accessed by q. Multiple references to a base relation are handled as separate relations. For each original result tuple t and witness list $w \in \mathscr{DD}(q,t)$ a tuple $(t, w[1], \ldots, w[n])$ is added to Q^{PI}. Hence, the original tuple has to be duplicated, if there is more than one witness list in the provenance of this tuple. The attribute names in the schema of Q^{PI} are used to indicated from which base relation attribute a result attribute is derived from. The attributes that correspond to the original result attributes of q are not renamed. To generate unique and predictable names for attributes storing provenance information we introduce a function $\mathscr{N}_q : \mathbb{N} \rightarrow \mathscr{A}$ that maps each attribute position in $\mathbf{Q^{PI}}$ to a unique name. The definition of $\mathscr{N}$ can be found in [6]. Here we just assume that $\mathscr{N}$ exists and present an attribute name that is generated by $\mathscr{N}$ from a base relation attribute a as $\mathscr{N}(a)$. For example, query q_{ex} accesses base relations *shop*, *sales* and *items*. In consequence, the schema of $Q_{ex}{}^{PI}$ according to the simplified representation is:

$$\mathbf{Q_{ex}^{PI}} = (name, sum(price), \mathscr{N}(name), \mathscr{N}(numEmpl), \mathscr{N}(sName), \mathscr{N}(itemId), \mathscr{N}(id), \mathscr{N}(price))$$

To be able to represent $\perp$ values in a witness list using the same schema as for regular tuple values we represent a $\perp$ at position i of witness list w as a tuple with schema $\mathscr{N}_q(\mathbf{R_i})$ and all attributes set to ε. For instance, a witness list $w =< (1), \perp >$ for tuple $t = (3,2)$ from the provenance of a query over relations R and S with schemas $\mathbf{R} = (a)$ and $\mathbf{S} = (b,c)$ would be represented as:

$$(3,4,1,\varepsilon,\varepsilon)$$

Below we present a formal definition of the relational representation of *data* provenance. This definition will be used in the correctness proofs of the query rewrite rules that generate queries to compute this representation.

Definition 3.1 (Relational PI-CS Data Provenance Representation)**.** *Let q be an algebra expression over base relations $R_1, \ldots, R_n$. The relational representation Q^{PI} of the provenance of q according to PI-CS is defined as:*

$$Q^{PI} = \{(t, w[1]', \ldots, w[n]')^m \mid t^p \in Q \wedge w^m \in \mathscr{DD}(q,t)\}$$

$$w[i]' = \begin{cases} w[i] & \text{if } w[i] \neq \perp \\ null(\mathbf{R_i}) & \text{else} \end{cases}$$

Example 3.3. *Figure 3.1 shows the provenance representation for query q_{ex} according to definition 3.1. For instance, the first tuple in $Q_{ex}{}^{PI}$ represents the original result tuple $t = (Merdies, 1)$ and the relational representation of witness list $w =< (Merdies, 3), (Merdies, 1), (1, 100) >$ from the provenance of t.*

This representation is quite verbose, but has several advantages over alternative representations:

- **Single Relation**: Provenance and original data is represented together in a single relation which can, e.g., be stored in as a view.

- **Association between Provenance and Normal Data**: Which witness list belongs to which result tuple is explicitly stored in the representation, because each tuple in Q^+ contains one original result tuple and one of its witness lists.
- **Provenance represented as Complete Tuples**: We deliberately choose to represent provenance as complete tuples instead of tuple identifiers to simplify the interpretation and querying of provenance.

We demonstrate the disadvantages of non-relational provenance representations on hand of the running example from Figure 3.1 and for the representation used for *Lineage-CS* (e.g., [4]). *Lineage-CS* provenance would represent the provenance of q_{ex} as the following list of relations[1]:

$$\begin{aligned}<&\{(Merdies,3),(Joba,14)\},\\ &\{(Merdies,1),(Merdies,2),(Merdies,2),(Joba,3),(Joba,3)\},\\ &\{(1,100),(2,10),(3,25)\}>\end{aligned}$$

This representation has two major disadvantages. First, a query having a list of relations as its result can not be expressed in relational algebra, because each algebra operator has only a single result relation. Thus, provenance queries and data are not in the same data model as the original data and queries. Second, the result only includes provenance data. There is no direct association between the original result and the contributing tuples. This is especially problematic if the provenance of a set of tuples is computed, because one would loose the information about which of the provenance tuples contributed to which of the original result tuples. This example presents an extreme case of this problem where the provenance is the complete original database instance. As demonstrated above, these shortcomings are avoided by the provenance representation used by *Perm*.

[1] The actual representation would be different because we are using bag semantics here.

3.2 Rewrite Rules for Perm-Influence Contribution Semantics

Having presented the provenance representation for which rewrites should be produced, we now present how a query q is transformed by the *Perm* approach into a query q^+ that generates the desired provenance result schema and propagates provenance according to *PI-CS*. In this section we limit the discussion to algebra expressions without sublinks. Rewrite rules for algebra expressions with sublinks can be found in [7]. The algebraic rewrites we use to propagate provenance from the base relations to the result of a query is defined for single algebra operators. The approach used by Cui in [5] for *Lineage-CS*, the *CS* from which *PI-CS* is derived from, is based on inversion of queries that trace the origin of a tuple (or set of tuples) from the result back to the source. A disadvantage of this approach is that it requires the instantiation of intermediate results for some operators like aggregation that are not invertible. The *Perm* approach omits the instantiation of intermediate results. The provenance computation for each operator in a query depends exclusively on the result relation of its rewritten inputs and is independent of the computation that generated this rewritten inputs. Thus, we do not have to keep earlier results to compute the current step. This approach has additional advantages if the provenance of only a part of a query should be computed.

The *Perm* algebraic rewrites are modeled as a function $+ : (\mathscr{E}, < \mathscr{A} >) \rightarrow (\mathscr{E}, < \mathscr{A} >)$ that transforms a query q into a provenance computing query q^+. Recall that $\mathscr{E}$ denotes the set of all possible algebra expressions and $< \mathscr{A} >$ denotes the set of all possible lists of attribute names. The list of attribute names (called provenance attribute list $\mathscr{P}$ of a query) is needed to define $+$ as rewrite rules for each algebra operator that are independent of each other. $\mathscr{P}$ is used to store the list of attributes of $\mathbf{Q}^+$ that are used to store the witness lists. The result of $+$ for a query q that contains multiple algebra operators is computed by recursively applying the rewrite rules for to each operator in the query. To be able to rewrite a query incrementally, the rewrite rules have to be applicable to rewritten inputs, i.e., a rewrite has to distinguish between normal and provenance attributes in its input (This is why $\mathscr{P}$ is needed).

Each rewrite rule is modeled as a structural modification (the $\mathscr{E} \rightarrow \mathscr{E}$ part of $+$) and a modification of the provenance attribute list (the $< \mathscr{A} > \rightarrow < \mathscr{A} >$ part of $+$). For two provenance attribute lists $\mathscr{P}_1 =< a_1, \ldots, a_n >$ and $\mathscr{P}_2 =< b_1, \ldots, b_n >$, the list concatenation operation $\blacktriangleright$ is defined as $\mathscr{P}_1 \blacktriangleright \mathscr{P}_2 =< a_1, \ldots, a_n, b_1, \ldots, b_n >$.

Definition 3.2 (Provenance Rewrite Meta-Operator)**.** *The provenance rewrite meta-operator* $+ : (\mathscr{E}, < \mathscr{A} >) \rightarrow (\mathscr{E}, < \mathscr{A} >)$ *maps a pair* $(q, <>)$ *to a pair* $q^+, \mathscr{P}(q^+)$. $+$ *is defined over the structure of q as rewrite rules for each algebra operator which are shown in Figure 3.3.*

We call $+$ a meta-operator, because, in contrast to algebra operator that transform relational data, it transforms algebra expressions.

3.2.1 Unary Operators Rewrite Rules

In Figure 3.3 the structural and provenance attribute list rewrites for each algebra operator are presented separately. We now discuss each rewrite rule in detail. The rewrite rule **(R1)** for base relation accesses duplicates the attributes from a base relation R and renames them according to the provenance attribute naming function $\mathscr{N}_q$[2]. The provenance attribute list of the rewritten base relation access contains the duplicated attributes.

Rewrite rule **(R2)**, the rule for $\Pi^{S/B}{}_A(q_1)$, rewrites a projection by adding the list of provenance attributes from ${q_1}^+$ to the projection list. For example, if q_1 is an access of base relation *items*, $\mathscr{P}(q^+)$ is $< \mathscr{N}(id), \mathscr{N}(price) >$. $(\Pi^{S/B}{}_A(item))^+$ preserves the complete tuples from relation *item* that were used to compute the result of $\Pi_A(item)$.

The result of applying $+$ to a selection (rewrite rule **(R3)**) is generated by applying the unmodified selection to its rewritten input, because adding the provenance attributes to the input of the selection does not change the result of the selection condition. Therefore, only tuples that are extended versions of original result tuples are in the result of the rewritten selection. The provenance attribute list of a rewritten selection is the provenance attribute list of its rewritten input.

[2]Here q is the complete query that is rewritten, because the attribute names depend on the structure of the complete query.

Structural Rewrite

Unary Operators

$$q^+ = R^+ = \Pi_{\mathbf{R},\mathbf{R}\to\mathscr{N}(\mathbf{R})}(R) \qquad \textbf{(R1)}$$

$$q^+ = (\sigma_C(q_1))^+ = \sigma_C({q_1}^+) \qquad \textbf{(R2)}$$

$$q^+ = ({\Pi^{S/B}}_A(q_1))^+ = {\Pi^B}_{A,\mathscr{P}(q^+)}({q_1}^+) \qquad \textbf{(R3)}$$

$$q^+ = (\alpha_{G,agg}(q_1))^+ = {\Pi^B}_{G,agg,\mathscr{P}(q^+)}(\alpha_{G,agg}(q_1) \leftouterjoin_{G=_nX} {\Pi^B}_{G\to X,\mathscr{P}({q_1}^+)}({q_1}^+)) \qquad \textbf{(R4)}$$

Join Operators

$$q^+ = (q_1 \times q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^+)}({q_1}^+ \times {q_2}^+) \qquad \textbf{(R5.a)}$$

$$q^+ = (q_1 \bowtie_C q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^+)}({q_1}^+ \bowtie_C {q_2}^+) \qquad \textbf{(R5.b)}$$

$$q^+ = (q_1 \leftouterjoin_C q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^+)}({q_1}^+ \leftouterjoin_C {q_2}^+) \qquad \textbf{(R5.c)}$$

$$q^+ = (q_1 \rightouterjoin_C q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^+)}({q_1}^+ \rightouterjoin_C {q_2}^+) \qquad \textbf{(R5.d)}$$

$$q^+ = (q_1 \fullouterjoin_C q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^+)}({q_1}^+ \fullouterjoin_C {q_2}^+) \qquad \textbf{(R5.e)}$$

Set Operations

$$q^+ = (q_1 \cup^{S/B} q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_1 \cup^S q_2 \leftouterjoin_{\mathbf{Q_1}=_nX} {\Pi^B}_{\mathbf{Q_1}\to X,\mathscr{P}({q_1}^+)}({q_1}^+) \leftouterjoin_{\mathbf{Q_1}=_nY} {\Pi^B}_{\mathbf{Q_2}\to Y,\mathscr{P}({q_2}^+)}({q_2}^+)) \qquad \textbf{(R6.a)}$$

$$q^+ = (q_1 \cup^{S/B} q_2)^+ = ({q_1}^+ \times null(\mathscr{P}({q_2}^+))) \cup^B ({\Pi^B}_{\mathbf{Q_1},\mathscr{P}(q^+)}({q_2}^+ \times null(\mathscr{P}({q_1}^+)))) \qquad \textbf{(R6.b)}$$

$$q^+ = (q_1 \cap^{S/B} q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_1 \cap^S q_2 \bowtie_{\mathbf{Q_1}=_nX} {\Pi^B}_{\mathbf{Q_1}\to X,\mathscr{P}({q_1}^+)}({q_1}^+) \bowtie_{\mathbf{Q_1}=_nY} {\Pi^B}_{\mathbf{Q_2}\to Y,\mathscr{P}({q_2}^+)}({q_2}^+)) \qquad \textbf{(R7)}$$

$$q^+ = (q_1 -^{S/B} q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathscr{P}(q^+)}({\Pi^S}_{\mathbf{Q_1}}(q_1 -^{S/B} q_2) \bowtie_{\mathbf{Q_1}=_nX} {\Pi^B}_{\mathbf{Q_1}\to X,\mathscr{P}({q_1}^+)}({q_1}^+) \leftouterjoin_{\mathbf{Q_1}\neq_n\mathbf{Q_2}} {q_2}^+) \qquad \textbf{(R8.a)}$$

$$q^+ = (q_1 -^{S/B} q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathscr{P}(q^+)}({\Pi^S}_{\mathbf{Q_1}}(q_1 -^{S/B} q_2) \bowtie_{\mathbf{Q_1}=_nX} {\Pi^B}_{\mathbf{Q_1}\to X,\mathscr{P}({q_1}^+)}({q_1}^+) \times null(\mathscr{P}({q_2}^+))) \qquad \textbf{(R8.b)}$$

Provenance Attribute List Rewrite

$$\mathscr{P}(q^+) = \begin{cases} \mathscr{P}(q^+) & \text{if } q = \sigma_C(q_1) \vee q = \Pi_A(q_1) \vee q = \alpha_{G,agg}(q_1) \\ \mathscr{P}({q_1}^+) \blacktriangleright \mathscr{P}({q_2}^+) & \text{if } q = (q_1 \diamond_C q_2) \vee q = (q_1 \cup^{S/B} q_2) \vee q = (q_1 \cap^{S/B} q_2) \vee q = (q_1 -^{S/B} q_2) \\ \mathscr{N}(R) & \text{if } q = R \end{cases}$$

Figure 3.3: *PI-CS* Algebraic Rewrite Rules for Queries without Sublinks

The rewrite rule **(R4)** rewrites an aggregation operation. We can not add additional tuples to the input of an aggregation or add additional attributes to its result schema without changing the values of the aggregation functions. This means we cannot propagate provenance information through an aggregation directly. Therefore, the rewritten query contains the original aggregation. The result of the original aggregation is joined with the rewritten version of q_1 using an equality condition on the grouping attributes. This is feasible because, according to the compositional semantics of *PI-CS*, all tuples with the same grouping attribute values as an result tuple t belong to the witness lists of t. We use an left outer join to handle the case of an aggregation with an empty input. According to the semantics of the aggregation operator the result of the aggregation is a single tuple in this case (with empty provenance). Note that the comparison operator $=_n$ instead of normal equality is used in the rewrite rule. This comparison operator is defined as:

$$a =_n b \Leftrightarrow a = b \vee (a \text{ is } \varepsilon \wedge b \text{ is } \varepsilon)$$

If the input of an aggregation contains tuples with *null* values in the group-by attributes, one output tuple is generated for this group of tuples. The $=_n$ comparison operator guarantees that the provenance of such a group is handled correctly. The provenance attribute list of a rewritten aggregation is the provenance attribute list of its rewritten input.

3.2.2 Join Operator Rewrite Rules

The rewrite rules **(R5.a)** to **(R5.e)** rewrite join operators.The provenance attribute list of a rewritten join operator is the concatenation of the provenance attribute lists of its rewritten inputs, because each witness list in the provenance of a join result contains a witness list for the left input and a witness list for a right input. $\diamond$ is used in the provenance attribute list construction as a placeholder for one of the join types of the *Perm* algebra. A join operator is rewritten by applying the join to the rewritten inputs and using a projection to produce the correct ordering of the result attributes (provenance attribute after normal attributes). This is possible, because adding provenance attributes to the input relations q_1 and q_2 does not change the result of the join condition.

3.2.3 Set Operations Rewrite Rules

The provenance attribute list of a rewritten set operation is the concatenation of the provenance attribute lists of its rewritten inputs. Recall that we defined two contribution semantics for union. One that combines two tuples from both inputs into a single witness list if they contributed to the same result tuple (this is the behaviour of *Lineage-CS*) and one the would generate two separate witness lists for this case. Rewrite rule **(R6.a)** implements the first version. The desired combination of rewritten input tuples into a single output cannot be achieved by applying the union to the rewritten inputs. Thus, each input is rewritten separately and then joined with original union query on the complete set of original result attributes (Recall that according to the compositional semantics for union input tuples belong to a witness list of a result tuple t if they are equal to t). We have to use outer joins to preserve tuples that are derived from only one of the inputs. The projection that is applied to the output of the joins removes superficial attributes introduced by the joins.

Rewrite rule **(R6.b)** implements the other contribution semantics for union. This rewrite rule simply extends the tuples from both rewritten inputs with null values to make them union compatible. Each tuple in the result of the rewritten query is derived from either the left or the right input and has the provenance attributes of the other rewritten input set to *null*. Hence, each tuple models a witness list of type $< t, \perp >$ or $< \perp, t >$ which are the two types of witness lists produced by the second compositional semantics of union.

The rewrite of an intersection operator (rewrite rule **(R7)** is similar to rule (R6.a). It also uses joins to attach the provenance attributes from the rewritten inputs to the original result tuples of the intersection. In contrast to union it is not necessary to use outer joins, because each result tuple of an intersection is derived from tuples of both inputs.

Like for union, we proposed two contribution semantics for set difference. One that, like *Lineage-CS*, includes tuples from the right input of the set difference in the provenance and the other one that includes

Original Query	$q_{ex} = \alpha_{name,sum(price)}(\sigma_{name=sName \wedge itemid=id}(shop \times sales \times items))$
Step 1	$q_{ex}{}^{+} = \Pi_{name,sum(price),\mathscr{P}(q^{+})}(\alpha_{name,sum(price)}(q) \bowtie_{name=x} \Pi_{name \to x,\mathscr{P}(q^{+})}(q^{+}))$ $q = \sigma_{name=sName \wedge itemid=id}(shop \times sales \times items)$ $\mathscr{P}(q_{ex}{}^{+}) = \mathscr{P}(q^{+})$
Step 2	$q^{+} = \Pi_{name,numEmpl,sName,itemId,id,price,\mathscr{P}(q^{+})}(\sigma_{name=sName \wedge itemid=id}(q_{cross}{}^{+}))$ $q_{cross}{}^{+} = shop^{+} \times sales^{+} \times items^{+}$ $\mathscr{P}(q^{+}) = \mathscr{P}(shop^{+}) \blacktriangleright \mathscr{P}(sales^{+}) \blacktriangleright \mathscr{P}(items^{+})$
Step 3	$shop^{+} = \Pi_{name,numEmpl,name \to \mathscr{N}(name),numEmpl \to \mathscr{N}(numEmpl)}(shop)$ $\mathscr{P}(shop^{+}) = (\mathscr{N}(name), \mathscr{N}(numEmpl))$
Step 4	$sales^{+} = \Pi_{sName,itemId,sName \to \mathscr{N}(sName),itemId \to \mathscr{N}(itemid)}(sales)$ $\mathscr{P}(sales^{+}) = (\mathscr{N}(sName), \mathscr{N}(itemId))$
Step 5	$items^{+} = \Pi_{id,price,id \to \mathscr{N}(id),price \to \mathscr{N}(price)}(items)$ $\mathscr{P}(items^{+}) = (\mathscr{N}(id), \mathscr{N}(price))$

$Q_{ex}{}^{+}$

name	sum(price)	$\mathscr{N}$(name)	$\mathscr{N}$(numEmpl)	$\mathscr{N}$(sName)	$\mathscr{N}$(itemId)	$\mathscr{N}$(id)	$\mathscr{N}$(price)
Merdies	120	Merdies	3	Merdies	1	1	100
Merdies	120	Merdies	3	Merdies	2	2	100
Merdies	120	Merdies	3	Merdies	2	2	100
Joba	50	Joba	14	Joba	3	3	25
Joba	50	Joba	14	Joba	3	3	25

Figure 3.4: Example Application of the Provenance Rewrite Meta-Operator

only tuples from the left input. Rewrite rule **(R8.a)** implements the first semantics in a similar way as the union and intersection rewrite rules. Rewrite Rule **(R8.b)** only joins the rewritten left input and fills the provenance attributes of the rewritten right input with null values. Note that the joins applied by the set operation rewrite rules use the $=_n$ comparison operator to deal with input tuples that contain null values.

3.2.4 Example Query Rewrite

As an example, reconsider query q_{ex} from the running example. Figure 3.4 presents the application of $+$ to this query. The top level operator of q_{ex} is an aggregation operator. Applying rewrite rule (R4) we get $q_{ex}{}^{+}$ as shown in 3.4 (**step 1**). Rule (R4) states that the $\mathscr{P}$-list for $q_{ex}{}^{+}$ equals $\mathscr{P}(q^{+})$. At this point, q^{+} has not been computed, thus, $\mathscr{P}(q_{ex}{}^{+})$ is not expanded further in this step. The remaining sub-query q is a selection, which is left untouched by the rewrite (R2). The cross-product $q_{cross} = shop \times sales \times item$ is handled by rewrite rule (R5.a) (see 3.4 (**step 2**)). The $\mathscr{P}$-list of a rewritten cross product is the concatenation of the $\mathscr{P}$-lists of its rewritten inputs. In this case, the provenance attribute lists of rewritten base relations *shop*, *sales* and *items*. In Figure 3.4 (**steps 3-5**) rewrite rule (R1) is used to derive the rewritten base relations $shop^{+}$, $sales^{+}$ and $items^{+}$. The result of $q_{ex}{}^{+}$ is presented at the lower part of Figure 3.4. Note that $q_{ex}{}^{+}$ generates exactly the *PI-CS* provenance representation introduced in section 3.1.

If one takes a careful look at this example, it is obvious that if q_{ex} had been represented as an operator-tree, we would have computed the structural rewrite top-down and computed the $\mathscr{P}$-lists in a second bottom-up tree-traversal according to the sequence of operations applied in the example. A single bottom-up computation of a rewrite is possible as well, because the rewrite rules do not enforce a specific evaluation order. It seems that the bottom-up approach is better suited, because the $\mathscr{P}$-lists of rewritten sub-expressions of a query q needed to compute q^{+}'s $\mathscr{P}$-list are immediately available, but the bottom-up

approach has other disadvantages.

Using the set of rewrite rules, we are able to transform a algebra expression q into a *single* relational algebra expression q^+ generating the provenance of q. A major advantage of this approach is that provenance computation and normal queries are expressed with the same query language which enables provenance to be queried like normal data. For example, if a user needs to know which items where sold by shops with a total sales bigger than 100, this query can be represented as $q_1 = \Pi_{pId}(\sigma_{sum(price)>100}({q_{ex}}^+))$. Note that it is possible to write down the algebra expression of this query as a query solely on ${q_{ex}}^+$, because of the direct association between provenance and original data in ${Q_{ex}}^+$, i.e., we can use provenance and original attributes in conditions and projections.

3.2.5 Proof of Correctness and Completeness

As mentioned before computing provenance by evaluating algebra expressions allows us to prove the correctness of provenance computation. To show that the + meta-operator generates a query that computes provenance according to *PI-CS* we have to prove that the result Q^+ of the query q^+ generated by the algebraic rewrite rules is equal to Q^{PI}, the relational representation of provenance according to *PI-CS*.

Theorem 3.1 (Correctness and Completeness of the PI-CS Rewrite Rules)**.** *Let q be an algebra expression without sublink expressions. The result Q^+ of the algebra expression q^+ generated by applying the provenance rewrite meta-operator to $(q, <>)$ is equal to Q^{PI} and, thus, generates provenance according to PI-CS:*

$$Q^+ = Q^{PI}$$

Proof. We have to show that each tuple in Q^+ is of the form $(t, w[1]', \ldots, w[n]')$ for an original result tuple t and one of its witness lists w (soundness) and for each combination of $t \in Q$ and $w \in \mathscr{DD}(q,t)$ there is a corresponding tuple $(t, w[1]', \ldots, w[n]')$ in Q^+ (completeness). The proof of this theorem is twofold. First, we show that each tuple in Q^+ is an extension of an tuple from Q and that for each tuple in Q there is a corresponding extended tuple in Q^+. This proves that the tuples in Q^+ and Q^{PI} agree on the original result attributes. We call this property *result preservation*, because it states that all the original result tuples of q and nothing else is stored in the original result attributes of Q^+. Second, we prove that each extension $t^+ = (t, w[1]', \ldots, w[n]') \in Q^+$ of $t \in Q$ contains the relational representation of a witness list w from $\mathscr{DD}(q,t)$ and that Q^+ contains an extended tuple t^+ for each witness list in $\mathscr{DD}(q,t)$. We refer to this property as *witness list preservation*.

Result Preservation

The result preservation property is proven by showing that $\Pi^S_{\mathbf{Q}}(q^+) = \Pi^S_{\mathbf{Q}}(q)$. We prove this proposition by induction over the structure of an algebra expression. Assuming we have proven $\Pi^S_{\mathbf{Q}}(q^+) = \Pi^S_{\mathbf{Q}}(q)$ for all algebra expressions with maximal operator nesting depth i we have to show that $\Pi^S_{\mathbf{OP(Q)}}((op(q))^+) = \Pi^S_{\mathbf{OP(Q)}}(op(q))$ holds for every unary operator op and that $\Pi^S_{\mathbf{Q_1\ OP\ Q_2}}((q_1\ op\ q_2)^+) = \Pi^S_{\mathbf{Q_1\ OP\ Q_2}}(q_1\ op\ q_2)$ holds for every binary operator.

Induction Start:
The only algebra expression with nesting depth 0 is a base relation access R. The other nullary operator t is not of interest because its result is not derived from base data but instead generated by the query itself. Therefore, the provenance of this operator is empty.

$$\begin{aligned} & \Pi^S_{\mathbf{R}}(R^+) \\ =& \Pi^S_{\mathbf{R}}(\Pi^B_{\mathbf{R},\mathbf{R}\to\mathscr{N}(\mathbf{R})}(R)) && \text{(algebraic equivalences)} \\ =& \Pi^S_{\mathbf{R}}(R) \end{aligned}$$

Induction Step:

Case $q = \Pi^{S/B}{}_{A}(q_1)$:

$$
\begin{aligned}
&\Pi^{S}{}_{\mathbf{Q}}(q^{+}) \\
=&\Pi^{S}{}_{A}((\Pi^{S/B}{}_{A}(q_1))^{+}) \\
=&\Pi^{S}{}_{A}((\Pi^{S/B}{}_{A,\mathscr{P}(q_1{}^{+})}(q_1{}^{+}))) \\
=&\Pi^{S}{}_{A}(q_1{}^{+}) \\
=&\Pi^{S}{}_{A}(\Pi^{S}{}_{\mathbf{Q_1}}(q_1{}^{+})) && \text{(because expression A is defined solely over attributes from } \mathbf{Q_1}\text{)} \\
=&\Pi^{S}{}_{A}(\Pi^{S}{}_{\mathbf{Q_1}}(q_1)) && \text{(induction hypothesis)} \\
=&\Pi^{S}{}_{A}(\Pi^{S/B}{}_{A}(q_1)) \\
=&\Pi^{S}{}_{A}(q)
\end{aligned}
$$

Case $q = \sigma_C(q_1)$:

$$
\begin{aligned}
&\Pi^{S}{}_{\mathbf{Q}}(q^{+}) \\
=&\Pi^{S}{}_{\mathbf{Q}}(\sigma_C(q_1{}^{+})) \\
=&\Pi^{S}{}_{\mathbf{Q}}(\sigma_C(\Pi^{S}{}_{\mathbf{Q_1}}(q_1{}^{+}))) && \text{(since } C \text{ is defined solely over attributes from } \mathbf{Q} \text{ and } \mathbf{Q} = \mathbf{Q_1}\text{)} \\
=&\Pi^{S}{}_{\mathbf{Q}}(\sigma_C(\Pi^{S}{}_{\mathbf{Q_1}}(q_1))) && \text{(induction hypothesis)} \\
=&\Pi^{S}{}_{\mathbf{Q}}(\sigma_C(q_1)) \\
=&\Pi^{S}{}_{\mathbf{Q}}(q)
\end{aligned}
$$

Case $q = \alpha_{G,agg}(q_1)$:

$$
\begin{aligned}
&\Pi^{S}{}_{\mathbf{Q}}(q^{+}) \\
=&\Pi^{S}{}_{G,agg}(\alpha_{G,agg}(q_1) \rtimes_{G=_n X} \Pi^{B}{}_{G\to X,\mathscr{P}(q_1{}^{+})}(q_1{}^{+})) \\
=&\Pi^{S}{}_{G,agg}(\alpha_{G,agg}(q_1)) && \text{(semantics of left join and duplicate removal of projection)} \\
=&\Pi^{S}{}_{\mathbf{Q}}(q)
\end{aligned}
$$

Case $q = q_1 \diamond_C q_2$:

$$
\begin{aligned}
&\Pi^{S}{}_{\mathbf{Q}}(q^{+}) \\
=&\Pi^{S}{}_{\mathbf{Q_1},\mathbf{Q_2}}(q_1{}^{+} \diamond_C q_2{}^{+}) \\
=&\Pi^{S}{}_{\mathbf{Q_1},\mathbf{Q_2}}(\Pi^{B}{}_{\mathbf{Q_1}}(q_1{}^{+}) \diamond_C \Pi^{B}{}_{\mathbf{Q_2}}(q_2{}^{+})) && \text{(equivalence of } \Pi^{B}{}_{\mathbf{Q}}(q) \text{ with } q\text{)} \\
=&\Pi^{S}{}_{\mathbf{Q_1},\mathbf{Q_2}}(\Pi^{S}{}_{\mathbf{Q_1}}(q_1{}^{+}) \diamond_C \Pi^{S}{}_{\mathbf{Q_2}}(q_2{}^{+})) && \text{(pushing duplicate removal into the join)} \\
=&\Pi^{S}{}_{\mathbf{Q_1},\mathbf{Q_2}}(\Pi^{S}{}_{\mathbf{Q_1}}(q_1) \diamond_C \Pi^{S}{}_{\mathbf{Q_2}}(q_2)) && \text{(induction hypothesis)} \\
=&\Pi^{S}{}_{\mathbf{Q}}(q)
\end{aligned}
$$

Case $q = q_1 \cup^{S/B} q_2$ **(R6.a)**:

$$
\begin{aligned}
&\Pi^{S}{}_{\mathbf{Q}}(q^{+}) \\
=&\Pi^{S}{}_{\mathbf{Q_1}}(\Pi^{B}{}_{\mathbf{Q_1},\mathscr{P}(q^{+})}(q_1 \cup^{S/B} q_2 \rtimes_{\mathbf{Q_1}=_n X} \Pi^{B}{}_{\mathbf{Q_1}\to X,\mathscr{P}(q_1{}^{+})}(q_1{}^{+}) \rtimes_{\mathbf{Q_1}=_n Y} \Pi^{B}{}_{\mathbf{Q_2}\to Y,\mathscr{P}(q_2{}^{+})}(q_2{}^{+}))) \\
=&\Pi^{S}{}_{\mathbf{Q_1}}(\Pi^{B}{}_{\mathbf{Q_1}}(q_1 \cup^{S/B} q_2 \rtimes_{\mathbf{Q_1}=_n X} \Pi^{B}{}_{\mathbf{Q_1}\to X,\mathscr{P}(q_1{}^{+})}(q_1{}^{+}) \rtimes_{\mathbf{Q_1}=_n Y} \Pi^{B}{}_{\mathbf{Q_2}\to Y,\mathscr{P}(q_2{}^{+})}(q_2{}^{+}))) \\
=&\Pi^{S}{}_{\mathbf{Q_1}}(\Pi^{S}{}_{\mathbf{Q_1}}(q_1 \cup^{S/B} q_2)) && \text{(semantics of left outer join and duplicate removal of projection)} \\
=&\Pi^{S}{}_{\mathbf{Q}}(q)
\end{aligned}
$$

Case $q = q_1 \cup^{S/B} q_2$ **(R6.b)**:

$$
\begin{aligned}
&\Pi^S_{\mathbf{Q}}(q^+)\\
=&\Pi^S_{\mathbf{Q_1}}((q_1{}^+ \ltimes null(\mathscr{P}(q_2{}^+))) \cup^B (\Pi_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_2{}^+ \ltimes null(\mathscr{P}(q_1{}^+)))))\\
=&\Pi^S_{\mathbf{Q_1}}(\Pi^S_{\mathbf{Q_1}}((q_1{}^+ \ltimes null(\mathscr{P}(q_2{}^+)))) \cup^B \Pi^S_{\mathbf{Q_1}}((\Pi^B_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_2{}^+ \ltimes null(\mathscr{P}(q_1{}^+))))))
\end{aligned}
$$

Pushing projection into union:

$$
\begin{aligned}
=&\Pi^S_{\mathbf{Q_1}}(\Pi^S_{\mathbf{Q_1}}(q_1{}^+)) \cup^B \Pi^S_{\mathbf{Q_2}}(\Pi^B_{\mathbf{Q_2}}(q_2{}^+))\\
=&\Pi^S_{\mathbf{Q_1}}(\Pi^S_{\mathbf{Q_1}}(q_1)) \cup^B \Pi^S_{\mathbf{Q_2}}(\Pi^S_{\mathbf{Q_2}}(q_2)) && \text{(induction hypothesis)}\\
=&\Pi^S_{\mathbf{Q}}(q)
\end{aligned}
$$

Case $q = q_1 \cap^{S/B} q_2$:

$$
\begin{aligned}
&\Pi^S_{\mathbf{Q}}(q^+)\\
=&\Pi^S_{\mathbf{Q_1}}(\Pi^B_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_1 \cap^{S/B} q_2 \bowtie_{\mathbf{Q_1}=_n X} \Pi^B_{\mathbf{Q_1}\to X,\mathscr{P}(q_1{}^+)}(q_1{}^+) \bowtie_{\mathbf{Q_1}=_n Y} \Pi^B_{\mathbf{Q_2}\to Y,\mathscr{P}(q_2{}^+)}(q_2{}^+)))\\
=&\Pi^S_{\mathbf{Q_1}}(\Pi^B_{\mathbf{Q_1}}(q_1 \cap^{S/B} q_2 \bowtie_{\mathbf{Q_1}=_n X} \Pi^B_{\mathbf{Q_1}\to X,\mathscr{P}(q_1{}^+)}(q_1{}^+) \bowtie_{\mathbf{Q_1}=_n Y} \Pi^B_{\mathbf{Q_2}\to Y,\mathscr{P}(q_2{}^+)}(q_2{}^+)))
\end{aligned}
$$

From the semantics of intersection and the induction hypothesis we know that every tuple from $q_1 \cap q_2$ will find join partners in $\Pi^B_{\mathbf{Q_1}\to X,\mathscr{P}(q_1{}^+)}(q_1{}^+)$ and $\Pi^B_{\mathbf{Q_2}\to Y,\mathscr{P}(q_2{}^+)}(q_2{}^+)$.

$$
\begin{aligned}
=&\Pi^S_{\mathbf{Q_1}}(\Pi^B_{\mathbf{Q_1}}(q_1 \cap^{S/B} q_2))\\
=&\Pi^S_{\mathbf{Q}}(q)
\end{aligned}
$$

Case $q = q_1 -^{S/B} q_2$ **(R8.a)**:

$$
\begin{aligned}
&\Pi^S_{\mathbf{Q}}(q^+)\\
=&\Pi^S_{\mathbf{Q_1}}(\Pi^B_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_1 -^{S/B} q_2 \bowtie_{\mathbf{Q_1}=_n X} \Pi_{\mathbf{Q_1}\to X,\mathscr{P}(q_1{}^+)}(q_1{}^+) \rtimes_{\mathbf{Q_1}\neq_n \mathbf{Q_2}} q_2{}^+))\\
=&\Pi^S_{\mathbf{Q_1}}(\Pi^B_{\mathbf{Q_1}}(q_1 -^{S/B} q_2 \bowtie_{\mathbf{Q_1}=_n X} \Pi^B_{\mathbf{Q_1}\to X,\mathscr{P}(q_1{}^+)}(q_1{}^+) \rtimes_{\mathbf{Q_1}\neq_n \mathbf{Q_2}} q_2{}^+))
\end{aligned}
$$

From the semantics of set difference we know that every tuple from $q_1 -^{S/B} q_2$ will find at least on join partner in $\Pi_{\mathbf{Q_1}\to X,\mathscr{P}(q_1{}^+)}(q_1{}^+)$.

$$
\begin{aligned}
=&\Pi^S_{\mathbf{Q_1}}(\Pi^B_{\mathbf{Q_1}}(q_1 -^{S/B} q_2))\\
=&\Pi^S_{\mathbf{Q}}(q)
\end{aligned}
$$

Case $q = q_1 -^{S/B} q_2$ **(R8.b)**:

$$
\begin{aligned}
&\Pi^S_{\mathbf{Q}}(q^+)\\
=&\Pi^S_{\mathbf{Q_1}}(\Pi_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_1 -^{S/B} q_2 \bowtie_{\mathbf{Q_1}=_n X} \Pi_{\mathbf{Q_1}\to X,\mathscr{P}(q_1{}^+)}(q_1{}^+) \rtimes null(\mathscr{P}(q_2{}^+))))\\
=&\Pi^S_{\mathbf{Q_1}}(\Pi^B_{\mathbf{Q_1}}(q_1 -^{S/B} q_2)) && \text{(same argument as for rewrite rule \textbf{(R8.a)})}\\
=&\Pi^S_{\mathbf{Q}}(q)
\end{aligned}
$$

Witness List Preservation

To prove the *witness list preservation* property we have to show that each tuple in Q^+ is an extension of an original result tuple t with the relational representation of one of t's *PI-CS* witness lists. Let q be an algebra expression defined over base relations $R_1,\ldots,R_n$ then we have to proof the following equivalence:

$$Q^+ = Q^{PI}$$

This equality can be expressed as:

$$u = (t, v_1, \ldots, v_n) \in Q^+ \Leftrightarrow w \in \mathscr{DD}(q,t) \wedge w[i]' = v_i$$

These is equivalent to $Q^+ = Q^{PI}$ because we have already proven that q^+ fulfills the result preservation property. Otherwise we would have to include an additional condition $t \in Q$ on both sides of the equivalence. As for *Result Preservation* we proof this property by induction over the structure of an algebra expression.

Induction Start:

Case $q = R$:

$\Rightarrow$:

$$\begin{aligned} & u^m = (t, v_1) \in R^+ \\ \Rightarrow & v_1 = t \wedge t^m \in R && (R^+ \text{ duplicates the attribute values of attributes in } \mathbf{R}) \\ \Rightarrow & w^m \in \mathscr{DD}(q,t) \wedge w[1]' = v_1 && (\text{since } \mathscr{DD}(R,t) = \{< t >^m | \, t^m \in R\}) \end{aligned}$$

$\Leftarrow$:

$$\begin{aligned} & w^m \in \mathscr{DD}(q,t) \wedge w[1]' = v_1 \\ \Rightarrow & v_1 = t && (\text{compositional semantics of } \mathscr{DD}) \\ \Rightarrow & u = (t, v_1)^m \in R^+ && (R^+ \text{ duplicates values of attributes in } \mathbf{R}) \end{aligned}$$

Induction Step for Unary Operators:

Assuming that *witness list preservation* holds for algebra expressions with a maximal nesting depth i we have to show that this property holds for algebra expressions with maximal nesting depth $i+1$. Hence, for unary operators op we have to show that for an algebra expression $q = op(q_1)$ with nesting depth $i+1$ the following holds under the assumption that $(t, v_1, \ldots, v_n) \in {Q_1}^+ \Leftrightarrow w \in \mathscr{DD}(q_1,t) \wedge w[i]' = v_i$:

$$(t, v_1, \ldots, v_n)^m \in Q^+ \Leftrightarrow w^m \in \mathscr{DD}(q,t) \wedge w[i]' = v_i$$

Using the definition of transitivity for $\mathscr{DD}$ the right hand side can be transformed into:

$$\begin{aligned} & (w^x \in \mathscr{DD}(op(Q_1),t) \wedge w =< t' > \wedge {w_1}^m \in \mathscr{DD}(q_1,t') \wedge w_1[i]' = v_i) \\ & \vee (w^m \in \mathscr{DD}(op(Q_1),t) \wedge w =<\perp> \wedge m = 1 \wedge \forall i : v_i = (\varepsilon,\ldots,\varepsilon)) \end{aligned}$$

Substituting the induction hypothesis we get:

$$\begin{aligned} & (w^x \in \mathscr{DD}(op(Q_1),t) \wedge w =< t' > \wedge (t', v_1, \ldots, v_n)^m \in {Q_1}^+) \\ \vee & (w^m \in \mathscr{DD}(op(Q_1),t) \wedge w =<\perp> \wedge m = 1 \wedge \forall i : v_i = (\varepsilon,\ldots,\varepsilon)) \end{aligned}$$

Thus, we have to prove the following equivalence:

$$\begin{aligned} & (t, v_1, \ldots, v_n) \in Q^+ \\ \Leftrightarrow & \\ & (w^x \in \mathscr{DD}(op(Q_1),t) \wedge w =< t' > \wedge (t', v_1, \ldots, v_n)^m \in {Q_1}^+) \\ & \vee (w^m \in \mathscr{DD}(op(Q_1),t) \wedge w =<\perp> \wedge m = 1 \wedge \forall i : v_i = (\varepsilon,\ldots,\varepsilon)) \end{aligned}$$

This means q^+ produces correct results if it propagates witness list representation correctly. For a witness list representation w' from the rewritten input, q^+ has to attach w' to a tuple t in the output iff t is derived from a tuple t' from the input and in the rewritten input t' is attached to w'.

Case $q = \sigma_C(q_1)$:
Applying rewrite rule (R2) to q we get $q^+ = \sigma_C({q_1}^+)$. From the compositional semantics of *PI-CS* for selection we know that a witness list w in $\mathscr{DD}(\sigma_C(Q_1),t)$ will never contain $\perp$. Therefore, the right disjunct of the right hand side of the equivalence we have to prove is never fulfilled. It follows that we can simplify the right hand side to:

$$(w^x \in \mathscr{DD}(op(Q_1),t) \wedge w =< t' > \wedge (t', v_1, \ldots, v_n)^m \in {Q_1}^+)$$

Using the simplified equivalence we get:

$$\begin{aligned}
&(t, v_1, \ldots, v_n)^m \in Q^+ \\
\Leftrightarrow &(t', v_1, \ldots, v_n)^m \in {Q_1}^+ \wedge t' = t && \text{(definition of selection)} \\
\Leftrightarrow &(t', v_1, \ldots, v_n) \in {Q_1}^+ \wedge w^m =< t' > \in \mathscr{DD}(\sigma_C(Q_1),t) && \text{(compositional semantics of PI-CS)}
\end{aligned}$$

Case $q = \Pi^{S/B}{}_A(q_1)$:
Applying rewrite rule (R3) to q we get $q^+ = \Pi^B{}_{A,\mathscr{P}(q^+)}({q_1}^+) = \Pi^B{}_{A,\mathscr{P}({q_1}^+)}({q_1}^+)$. Using the same arguments as applied in the proof for selection we deduce that the simplified equivalence can be applied in the proof for projection too.

$$\begin{aligned}
&(t, v_1, \ldots, v_n)^m \in Q^+ \\
\Leftrightarrow &(t', v_1, \ldots, v_n)^m \in {Q_1}^+ \wedge t'.A = t && \text{(definition of projection)} \\
\Leftrightarrow &(t', v_1, \ldots, v_n)^m \in {Q_1}^+ \wedge w^m =< t' > \in \mathscr{DD}(\Pi^{S/B}{}_A(Q_1),t) && \text{(compositional semantics of PI-CS)}
\end{aligned}$$

Case $q = \alpha_{G,agg}(q_1)$:
Applying rewrite (R2) to q we get $q^+ = \Pi^B{}_{G,agg,\mathscr{P}(q^+)}(\alpha_{aggr,G}(q_1) \rtimes_{G=_n X} \Pi^B{}_{G\to X,\mathscr{P}({q_1}^+)}({q_1}^+))$ with $\mathscr{P}(q^+) = \mathscr{P}({q_1}^+)$. We distinguish between two cases:

1. $G = ()$ and $Q_1 = \emptyset$
2. else

Case 1: For the first case we know that $\mathscr{DD}(q,t) = \{<\perp>\}$. Therefore, the left disjunction of the right-hand side of the equivalence we have to prove can be removed. In the following let $v = (v_1, \ldots, v_n)$.

$$\begin{aligned}
&(t, v_1, \ldots, v_n)^m \in Q^+ \\
\Leftrightarrow &t \in Q \wedge (((u,v)^m \in Q' \wedge q' = \Pi^B{}_{G\to X,\mathscr{P}({q_1}^+)}({q_1}^+) \wedge u.X =_n t.G) \vee (v = \varepsilon, \ldots \varepsilon)) \\
\Leftrightarrow &t \in Q \wedge (v = \varepsilon, \ldots, \varepsilon) \in Q' \\
\Leftrightarrow &<\perp> \in \mathscr{DD}(\alpha_{G,agg}(Q_1),t)
\end{aligned}$$

Case 2: For the second case we know that every result tuple t in $\alpha_{G,agg}(q_1)$ will find a join partner in $\Pi^B{}_{G\to X,\mathscr{P}({q_1}^+)}({q_1}^+)$, because otherwise t would not in Q.

$$\begin{aligned}
&(t, v_1, \ldots, v_n)^m \in Q^+ \\
\Leftrightarrow &(u,v)^m \in Q' \wedge q' = \Pi^B{}_{G\to X,\mathscr{P}({q_1}^+)}({q_1}^+) \wedge u.X = t.G \wedge u.v = t.v && \text{(definition of left join)} \\
\Leftrightarrow &(t', v_1, \ldots, v_n)^m \in {Q_1}^+ \wedge t'.G = t.G \\
\Leftrightarrow &(t', v_1, \ldots, v_n)^m \in {Q_1}^+ \wedge w^x =< t' > \in \mathscr{DD}(\alpha_{G,agg}(Q_1),t)
\end{aligned}$$

Induction Step for Binary Operators:

To prove witness list preservation for binary operators we use the same approach as applied for unary operators. The difference is that applying the transitivity definition of *PI-CS* to $q = q_1\ op\ q_2$ generates a more complex equivalence:

$$(t, v_1, \ldots, v_n, u_1, \ldots, u_m)^{p \times r} \in Q^+ \Leftrightarrow case_1 \vee case_2 \vee case_3 \vee case_4$$

where

$$\begin{aligned}
case_1 =& (w^x \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< t', t'' > \\
& \wedge w[q_1]^p \in \mathscr{DD}(q_1, t') \wedge w[i]' = v_i \wedge w[q_2]^r \in \mathscr{DD}(q_2, t'') \wedge w[i+n]' = u_i) \\
case_2 =& (w^x \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< t', \perp > \\
& \wedge w[q_1]^p \in \mathscr{DD}(q_1, t') \wedge w[i]' = v_i \wedge \forall i : u_i = (\varepsilon, \ldots, \varepsilon) \wedge r = 1) \\
case_3 =& (w^x \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< \perp, t' > \\
& \wedge \forall i : v_i = (\varepsilon, \ldots, \varepsilon) \wedge p = 1 \wedge w[q_2]^r \in \mathscr{DD}(q_2, t') \wedge w[n+i]' = u_i) \\
case_4 =& (w \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< \perp, \perp > \wedge \forall i : v_i = (\varepsilon, \ldots, \varepsilon) \wedge \forall i : u_i = (\varepsilon, \ldots, \varepsilon) \wedge p = r = 1)
\end{aligned}$$

Substituting the induction hypothesis for each of these cases produces the following equivalent formulation:

$$\begin{aligned}
case_1 =& (w^x \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< t', t'' > \\
& \wedge (t', v_1, \ldots, v_n)^p \in {Q_1}^+ \wedge (t', u_1, \ldots, u_m)^p \in {Q_2}^+) \\
case_2 =& (w^x \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< t', \perp > \\
& \wedge (t', v_1, \ldots, v_n)^p \in {Q_1}^+ \wedge \forall i : u_i = (\varepsilon, \ldots, \varepsilon) \wedge r = 1) \\
case_3 =& (w^x \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< \perp, t' > \\
& \wedge \forall i : v_i = (\varepsilon, \ldots, \varepsilon) \wedge (t', u_1, \ldots, u_m)^p \in {Q_2}^+) \\
case_4 =& (w \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< \perp, \perp > \wedge \forall i : v_i = (\varepsilon, \ldots, \varepsilon) \wedge \forall i : u_i = (\varepsilon, \ldots, \varepsilon) \wedge p = r = 1)
\end{aligned}$$

We prove this equivalence by identifying under which pre-conditions each of the cases is fulfilled (the individual case are non-overlapping) and then under assumption of these pre-conditions prove the equivalence of the left hand side with this case. Note that some of these cases can be precluded for several of the binary operators, because the provenance of these operators never contains witness lists of the requested format.

Case $q = q_1 \times q_2$:

According to the compositional semantics of *PI-CS* for cross product each witness list is of the form $w =< u, v >$ where u respective v are tuples from Q_1 respective Q_2. Therefore, all cases except $case_1$ can be excluded.

$$\begin{aligned}
& (t, v_1, \ldots, v_n, u_1, \ldots, u_m)^{p \times r} \in Q^+ \\
\Leftrightarrow & (t.\mathbf{Q_1}, v_1, \ldots, v_n)^p \in {Q_1}^+ \wedge (t.\mathbf{Q_2}, u_1, \ldots, u_m)^q \in {Q_2}^+ && \text{(semantics of projection and cross product)} \\
\Leftrightarrow & (t', v_1, \ldots, v_n)^p \in {Q_1}^+ \wedge (t'', u_1, \ldots, u_m)^q \in {Q_2}^+ \wedge t' = t.\mathbf{Q_1} \wedge t'' = t.\mathbf{Q_2} \\
\Leftrightarrow & w^x =< t', t'' > \in \mathscr{DD}(Q_1 \times Q_2, t) \wedge (t', v_1, \ldots, v_n)^p \in {Q_1}^+ \wedge (t'', u_1, \ldots, u_m)^q \in {Q_2}^+ \\
& && \text{(compositional semantics for cross product)}
\end{aligned}$$

Case $q = q_1 \bowtie_C q_2$:

According to the compositional semantics of *PI-CS* for join each witness list is of the form $w =< u, v >$ where u respective v are tuples from Q_1 respective Q_2. Therefore, all cases except case 1 can be excluded.

$$\begin{aligned}
&(t, v_1, \ldots, v_n, u_1, \ldots, u_m)^{p \times r} \in Q^+ \\
\Leftrightarrow &(t.\mathbf{Q_1}, v_1, \ldots, v_n)^p \in {Q_1}^+ \wedge (t.\mathbf{Q_2}, u_1, \ldots, u_m)^q \in {Q_2}^+ && \text{(semantics of projection and join)} \\
\Leftrightarrow &(t', v_1, \ldots, v_n)^p \in {Q_1}^+ \wedge (t'', u_1, \ldots, u_m)^q \in {Q_2}^+ \wedge t' = t.\mathbf{Q_1} \wedge t'' = t.\mathbf{Q_2} \\
\Leftrightarrow &w^x =< t', t'' >^x \in \mathscr{DD}(Q_1 \bowtie_C Q_2, t) \wedge (t', v_1, \ldots, v_n)^p \in {Q_1}^+ \wedge (t'', u_1, \ldots, u_m)^q \in {Q_2}^+ \\
& && \text{(compositional semantics for join)}
\end{aligned}$$

Case $q = q_1 \text{⟕}_C q_2$:

Each witness list w in the *PI-CS* provenance for left outer join is either of the form $< u, v >$ with u from Q_1 and v from Q_2 or of form $< u, \bot >$ with u from Q_1. Hence, only cases 1 and 2 have to be considered.
Case 1: According to the compositional semantics for left outer join case 1 applies if $t \models C$. In this case the semantics of left outer join, its compositional semantics and rewrite rule coincide with the join case.
Case 2: Case 2 applies if $t = (t', \varepsilon, \ldots, \varepsilon)$ where $t' \in Q_1$. This means t is generated from a left hand side input tuple that does not have a join partner in Q_2.

$$\begin{aligned}
&(t, v_1, \ldots, v_n, \varepsilon, \ldots, \varepsilon)^p \in Q^+ \\
\Leftrightarrow &(t', v_1, \ldots, v_n)^p \in {Q_1}^+ \wedge t = (t', \varepsilon, \ldots, \varepsilon) \\
\Leftrightarrow &(t', v_1, \ldots, v_n)^p \in {Q_1}^+ \wedge w^x =< t', \bot >^x \in \mathscr{DD}(Q_1 \text{⟕}_C Q_2, t)
\end{aligned}$$

Case $q = q_1 \bowtie\!\text{⟖}_C q_2$:

For right outer join only cases 1 and 3 apply. The proves for both cases are analog to the proves for left outer join.
Case $q = q_1 \text{⟗}_C q_2$:

For full outer join case 1 to 3 apply and are proven as for the other outer join types.
Case $q = q_1 \cup^{S/B} q_2$ (*PI* semantics):

All witness lists in the provenance of a union are either $< u, \bot >$ or $< \bot, v >$ with $u \in Q_1$ and $v \in Q_2$ if *PI-CS* semantics are applied. This means only cases 2 and 3 apply.
Case 2:

$$\begin{aligned}
&(t, v_1, \ldots, v_n, \varepsilon, \ldots, \varepsilon)^p \in Q^+ \\
\Leftrightarrow &(t, v_1, \ldots, v_n)^p \in {Q_1}^+ \\
\Leftrightarrow &(t, v_1, \ldots, v_n)^p \in {Q_1}^+ \wedge w^x \in \mathscr{DD}(Q_1 \cup Q_2) \wedge w =< t, \bot >
\end{aligned}$$

Case 3: Is symmetric to the proof of case 2.
Case $q = q_1 \cup^{S/B} q_2$ (alternative semantics):

For the alternative semantics of union provenance (rewrite rule **6.a**) also case 1 applies.
Case 1:

$$\begin{aligned}
&(t, v_1, \ldots, v_n, u_1, \ldots, u_m)^{p \times r} \in Q^+ \\
\Leftrightarrow &(t, v_1, \ldots, v_n)^p \in {Q_1}^+ \wedge (t, u_1, \ldots, u_m)^r \in {Q_2}^+ && \text{(semantics of left outer join)} \\
\Leftrightarrow &(t, v_1, \ldots, v_n)^p \in {Q_1}^+ \wedge (t, u_1, \ldots, u_m)^r \in {Q_2}^+ \wedge w^x \in \mathscr{DD}(Q_1 \cup^{S/B} Q_2) \wedge w =< t, t >
\end{aligned}$$

Case 2:

$$\begin{aligned}&(t,v_1,\ldots,v_n,\varepsilon,\ldots,\varepsilon)^p \in Q^+\\ \Leftrightarrow&(t,v_1,\ldots,v_n)^p \in {Q_1}^+\\ \Leftrightarrow&(t,v_1,\ldots,v_n)^p \in {Q_1}^+ \wedge w^x \in \mathscr{D}\mathscr{D}(Q_1 \cup^{S/B} Q_2) \wedge w = <t,\bot>\end{aligned}$$

Case 3: Is symmetric to the proof of case 2.
Case $q = q_1 \cap^{S/B} q_2$:

For intersection all witness lists are of the form $<u,v>$ with $u \in Q_1$ and $v \in Q_2$. Only case 1 applies.

$$\begin{aligned}&(t,v_1,\ldots,v_n,u_1,\ldots,u_m)^{p\times r} \in Q^+\\ \Leftrightarrow&(t,v_1,\ldots,v_n)^p \in {Q_1}^+ \wedge (t,u_1,\ldots,u_m)^r \in {Q_2}^+ && \text{(semantics of join)}\\ \Leftrightarrow&(t,v_1,\ldots,v_n)^p \in {Q_1}^+ \wedge (t,u_1,\ldots,u_m)^r \in {Q_2}^+ \wedge <t,t>^x \in \mathscr{D}\mathscr{D}(Q_1 \cap^{S/B} Q_2)\end{aligned}$$

Case $q = q_1 -^{S/B} q_2$ (*PI-CS* semantics):

All witness lists for set difference are of the form $<u,\bot>$ with $u \in Q_1$. Therefore, only case 2 applies

$$\begin{aligned}&(t,v_1,\ldots,v_n,\varepsilon,\ldots,\varepsilon)^p \in Q^+\\ \Leftrightarrow&(t,v_1,\ldots,v_n)^p \in {Q_1}^+\\ \Leftrightarrow&(t,v_1,\ldots,v_n)^p \in {Q_1}^+ \wedge <t,\bot>^x \in \mathscr{D}\mathscr{D}(Q_1 -^{S/B} Q_2)\end{aligned}$$

Case $q = q_1 -^{S/B} q_2$ (alternative semantics):

Under the alternative semantics a witness lists from $\mathscr{D}\mathscr{D}(q,t)$ is either of form $<t,v>$ with $t \in Q_1$ and $v \in Q_2$ if Q_2 contains tuples that are not equal to t or of form $<t,\bot>$ otherwise. This means cases 1 and 2 apply.
Case 1:

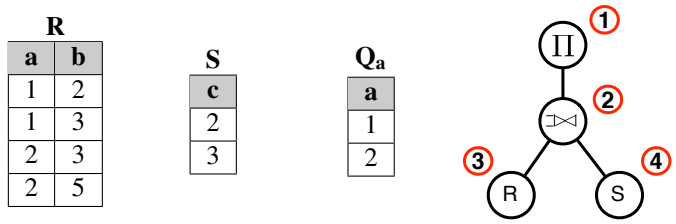
$$\begin{aligned}&(t,v_1,\ldots,v_n,u_1,\ldots,u_m)^{p\times r} \in Q^+\\ \Leftrightarrow&(t,v_1,\ldots,v_n)^p \in {Q_1}^+ \wedge (t',u_1,\ldots,u_m)^r \in {Q_2}^+ \wedge t \neq t'\\ \Leftrightarrow&(t,v_1,\ldots,v_n)^p \in {Q_1}^+ \wedge (t',u_1,\ldots,u_m)^r \in {Q_2}^+ \wedge <t,t'>^x \in \mathscr{D}\mathscr{D}(Q_1 -^{S/B} Q_2,t)\end{aligned}$$

Case 2:

$$\begin{aligned}&(t,v_1,\ldots,v_n,\varepsilon,\ldots,\varepsilon)^p \in Q^+\\ \Leftrightarrow&(t,v_1,\ldots,v_n)^p \in {Q_1}^+ \wedge \nexists(t',u_1,\ldots,u_m) \in {Q_2}^+ : t \neq t'\\ \Leftrightarrow&(t,v_1,\ldots,v_n)^p \in {Q_1}^+ \wedge <t,\bot>^x \in \mathscr{D}\mathscr{D}(Q_1 -^{S/B} Q_2,t)\end{aligned}$$

□

Having proven theorem 3.1 we established an very important property. Given a query q (without sublinks) we know how to transform it into a query q^+ by applying the meta-operator $+$. q^+ is guaranteed to compute the provenance of q according to *PI-CS* alongside with the original results of q. Furthermore, in the result of Q^+ each original tuple t is extended with the relational representations of its *PI-CS* witness lists. The correctness and completeness of $+$ was proven in two steps. First, we demonstrated that q^+ preserves the original result tuples of q (*result preservation*). I.e., if Q contains a tuples t then Q^+ will contain extended versions of t and all tuples (t,v) in Q^+ are extended versions of tuples from Q. Second, we have proven that each extended version of a tuple t in Q^+ is generated by attaching the relational representation of a witness list from $\mathscr{D}\mathscr{D}(q,t)$ to t (*witness list preservation*). This means only relational representations of witness lists from $\mathscr{D}\mathscr{D}(q,t)$ are included in Q^+ and q^+ generates the correct associations between witness lists and original result tuples as requested by the relational representation Q^{PI} of *PI-CS* provenance. In the next section we extend the rewrite rules for algebra expressions with sublinks and prove the correctness and completeness of these extensions.



R

a	b
1	2
1	3
2	3
2	5

S

c
2
3

Q_a

a
1
2

$$q_a = \Pi^S{}_a(R \rtimes\!\ltimes_{b=c} S)$$

$$\mathscr{T}(q_c,(1)) : \theta_{<(1,2),(2)>}(op) = 1 \qquad \theta_{<(1,3),(3)>}(op) = 1$$

$$\mathscr{T}(q_c,(2)) : \theta_{<(2,3),(3)>}(op) = 1 \qquad \theta_{<(2,5),\perp>}(op) = \begin{cases} 0 \text{ if } op = \mathbf{S} \\ 1 \text{ else} \end{cases}$$

$$\Theta_{<(1,2),(2)>} = \{1,2,3,4\} \qquad \Theta_{<(1,3),(3)>} = \{1,2,3,4\}$$

$$\Theta_{<(2,3),(3)>} = \{1,2,3,4\} \qquad \Theta_{<(2,5),\perp>} = \{1,2,3\}$$

$Q_a{}^{\text{Trans}}$

a	$\mathscr{T}$
1	{1,2,3,4}
1	{1,2,3,4}
2	{1,2,3,4}
2	{1,2,3}

Figure 3.5: Transformation Provenance Representation Example

3.3 Relational Representation of Transformation Provenance Information

In this section we introduce a simplistic relational representation of *transformation* provenance. More user-friendly representations are not discussed in this report. Recall that the *transformation* provenance of an algebra expression q contains one annotated algebra tree for each witness list in the *PI-CS data* provenance of q. These annotated trees all have the same nodes and edges; they only differ in their annotations functions θ_w. Therefore, we factor out the static part (that is the tree) in the relational representation of *transformation* provenance and only represent the annotation functions. Each annotation function θ_w is represented as the set of nodes from the algebra tree for which θ_w evaluates to 1. To simplify this representation identifiers for the nodes in an algebra tree are created by a pre-order traversal of the tree. We call the set representation of an annotation function θ_w the *annotation set* Θ_w.

Example 3.4. *In Figure 3.5 we reconsider the transformation provenance example from chapter 2. The algebra tree presented on the top right of this figure shows the generated node identifiers. These node identifiers are used in the set representations of the θ_w annotations functions for the transformation provenance of example query q_a. For instance, $\Theta_{<(2,5),\perp>}$ contains the identifiers for all nodes except the one for the base relation access S, because $\theta_{<(2,5),\perp>}$ evaluates to 0 for this node.*

Similar to the relational representation of *data* provenance we represent *transformation* provenance and the original result data in a single relation Q^{trans}. The annotation sets for the witness lists of a original result tuple are stored in a single additional attribute $\mathscr{T}$. Each tuple in Q^{trans} stores one original result tuple and one annotation set Θ_w for a $w \in \mathscr{WD}(q,t)$.

Definition 3.3 (Relational Transformation Provenance Representation)**.** *Let q be an algebra expression. The relational representation Q^{Trans} of the provenance of q according to the transformation provenance CS is defined as:*

$$Q^{Trans} = \{(t, \Theta_w)^{m \times p} \mid t^p \in Q \wedge w^m \in \mathscr{D}\mathscr{D}(q,t)\}$$

Example 3.5. *The relational representation ${Q_a}^{trans}$ of the transformation provenance of example query q_a is shown at the bottom of Figure 3.5. For instance, the last tuple in this relation represents the original result tuple (2) and the annotation set $\Theta_{<(2,5),\perp>}$ and, therefore, the set stored in the $\mathscr{T}$ attribute of this tuple includes all node identifiers except the one for the access to base relation S.*

Structural Rewrite

$$q^T = (R)^T = \Pi^B{}_{\mathbf{R},\mathscr{T}(q^T)\to\mathscr{T}}(R) \quad \textbf{(T1)}$$

$$q^T = (\sigma_C(q_1))^T = \Pi^B{}_{\mathbf{Q_1},\mathscr{T}(q^T)\to\mathscr{T}}(\sigma_C({q_1}^T)) \quad \textbf{(T2)}$$

$$q^T = (\Pi^{S/B}{}_A(q_1))^T = \Pi^B{}_{A,\mathscr{T}(q^T)\to\mathscr{T}}({q_1}^T) \quad \textbf{(T3)}$$

$$q^T = (\alpha_{G,agg}(q_1))^T = \Pi^B{}_{G,agg,\mathscr{T}(q^T)\to\mathscr{T}}(\alpha_{G,agg}(q_1) \rtimes_{G=_n X} \Pi^B{}_{G\to X,\mathscr{T}}({q_1}^T)) \quad \textbf{(T4)}$$

$$q^T = (q_1 \diamond_C q_2)^T = \Pi^B{}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{T}(q^T)\to\mathscr{T}}({q_1}^T \diamond_C {q_2}^T) \quad \textbf{(T5)}$$

$$q^T = (q_1 \cup^{S/B} q_2)^T = \Pi^B{}_{\mathbf{Q_1},\mathscr{T}(q^T)\to\mathscr{T}}({q_1}^T \cup^{S/B} {q_2}^T) \quad \textbf{(T6)}$$

$$q^T = (q_1 \cap^{S/B} q_2)^T = \Pi^B{}_{\mathbf{Q_1},\mathscr{T}(q^T)\to\mathscr{T}}(q_1 \cap^{S/B} q_2 \rtimes_{\mathbf{Q_1}=_n X} \Pi^B{}_{\mathbf{Q_1}\to X,\mathscr{T}}({q_1}^T) \rtimes_{\mathbf{Q_1}=_n \mathbf{Q_2}} {q_2}^T) \quad \textbf{(T7)}$$

$$q^T = (q_1 -^{S/B} q_2)^T = \Pi^B{}_{\mathbf{Q_1},\mathscr{T}(q^T)\to\mathscr{T}}(q_1 -^{S/B} q_2 \rtimes_{\mathbf{Q_1}=_n X} \Pi^B{}_{\mathbf{Q_1}\to X,\mathscr{T}}({q_1}^T)) \quad \textbf{(T8)}$$

Transformation Provenance Attribute Rewrite

$$\mathscr{T}(R^T) = \{R\}$$

$$\mathscr{T}((\sigma_C(q_1))^T) = \{\sigma_c(q_1)\} \cup \mathbf{Q_1}.\mathscr{T}$$

$$\mathscr{T}((\Pi^{S/B}{}_A(q_1))^T) = \{\Pi^{S/B}{}_A(q_1)\} \cup \mathbf{Q_1}.\mathscr{T}$$

$$\mathscr{T}((\alpha_{G,agg}(q_1))^T) = \{\alpha_{G,agg}(q_1)\} \cup \mathbf{Q_1}.\mathscr{T}$$

$$\mathscr{T}((q_1 \diamond_C q_2)^T) = \{q_1 \diamond_C q_2\} \cup \mathbf{Q_1}.\mathscr{T} \cup \mathbf{Q_2}.\mathscr{T}$$

$$\mathscr{T}((q_1 \cup^{S/B} q_2)^T) = \{q_1 \cup^{S/B} q_2\} \cup \mathbf{Q_1}.\mathscr{T}$$

$$\mathscr{T}((q_1 \cap^{S/B} q_2)^T) = \{q_1 \cap^{S/B} q_2\} \cup \mathbf{Q_1}.\mathscr{T} \cup \mathbf{Q_2}.\mathscr{T}$$

$$\mathscr{T}((q_1 -^{S/B} q_2)^T) = \{q_1 -^{S/B} q_2\} \cup \mathbf{Q_1}.\mathscr{T}$$

Figure 3.6: Transformation Provenance Rewrite Rules

3.4 Rewrite Rules for Transformation Provenance

We now present a meta-operator T for *transformation* provenance that transforms an algebra expression q into an algebra expression q^T that computes the relational representation Q^{Trans} of the *transformation* provenance of q. Like for *data* provenance these meta-operator is defined inductively over the structure of an algebra expression as rewrite rules for each algebra operator.

Definition 3.4 (Transformation Provenance Rewrite Meta-Operator)**.** *The transformation provenance rewrite meta-operator $T : \mathscr{E} \to \mathscr{E}$ is defined inductively over the structure of an input algebra expression q by applying the rewrite rules presented in Figure 3.6 to each operator in q.*

Fig. 3.6 presents the rewrite rules that implement T. For each rewrite rules the structural modification of the algebra expression and the computation of the new transformation provenance attribute $\mathscr{T}$ is presented separately. For the transformation provenance attribute rewrite we use the following notational conventions: $\{q\}$ denotes the node identifier of the top operator of q in the algebra tree of q. E.g., for the algebra expression $\sigma_C(R \rtimes_{a=b} S)$ the expression $\{R \rtimes_{a=b} S\}$ represents the node identifier 2. The $\cup$ used in the definition of the annotation sets is the normal set union operation except that we define $\mathscr{T} \cup \varepsilon = \mathscr{T}$.

The rewrite rule **(T1)** for a base relation access adds the singleton annotation set containing the node identifier of the base relation access $\{R\}$ as the value for attribute $\mathscr{T}$ to each generated result tuple. A selection is rewritten by rule **(T2)** by applying the unmodified selection to ${q_1}^T$ and adding an outermost projection that simply adds the node identifier of the selection operator to the annotation set of the rewritten input ${q_1}^T$. **(T3)**, the rewrite rule for projection, works analogously. An aggregation is rewritten **(T4)** by joining the rewritten input ${q_1}^T$ with the original aggregation and using a projection to add the node identifier for the aggregation to the annotation set of q^T.

$$q_a = \Pi^{S}{}_{a}(R \rtimes\!\lhd_{b=c} S)$$

$$q_a{}^{T} = \Pi^{B}{}_{a,\{1\}\cup\mathscr{T}\to\mathscr{T}}(\Pi^{B}{}_{a,b,c,\{2\}\cup\mathbf{R}^{+}.\mathscr{T}\cup\mathbf{S}^{+}.\mathscr{T}\to\mathscr{T}}(\Pi^{B}{}_{a,b,\{3\}\to\mathscr{T}}(R) \rtimes\!\lhd_{b=c} \Pi^{B}{}_{c,\{4\}\to\mathscr{T}}(S)))$$

$\mathbf{Q_a}^{\mathbf{T}}$

a	$\mathscr{T}$
1	$\{1,2,3,4\}$
1	$\{1,2,3,4\}$
2	$\{1,2,3,4\}$
2	$\{1,2,3\}$

Figure 3.7: Transformation Provenance Rewrite Example

The rewrite rule for join operators **(T5)** (here $\diamond$ denotes one of the algebra join operators) unions the annotation sets of the rewritten inputs and add the node identifier of the join to the result. Note that this is correct behavior for outer joins, because we have defined the union of a annotation set with ε as $\mathscr{T} \cup \varepsilon = \mathscr{T}$.

Rewrite rule **(T6)** for the union operator unions the rewritten inputs and uses a projection to union the annotation sets of both rewritten inputs with the node identifier of the union operator. In the transformation provenance attribute rewrite the $\mathscr{T}$ attribute of the input is referenced without using a qualification (e.g., $\mathbf{S}^{+}.\mathscr{T}$), because the result schema of the union operator is the schema of its the left hand input. **(T7)**, the rewrite rule for intersection works in a similar way as the *PI-CS* rewrite rule for this operator: the original intersection is joined with the rewritten left and right input on the original result attributes. The applied projection unions the annotation sets from both rewritten inputs with the node identifier of the intersection operator. A set difference is rewritten by **(T8)** using the same approach. For set difference the right input is not rewritten, because the *PI-CS* provenance of the right input is $\perp$. Hence, the annotation set of the right input is always the empty set.

Example 3.6. *Figure 3.7 shows the application of the T meta-operator to the example query q_a from Figure 3.5. In $q_a{}^{T}$ the node identifiers of the individual operators are added to the intermediate annotation set produced by the the rewritten input of the operator. Note that some node identifiers (1,2, and 3) are guaranteed to be contained in each annotation set produced by $q_a{}^{T}$. Thus, the rewritten query could be simplified to:*

$$\Pi^{B}{}_{a,\{1,2,3\}\cup\mathbf{S}^{+}.\mathscr{T}\to\mathscr{T}}(R \rtimes\!\lhd_{b=c} \Pi^{B}{}_{c,\{4\}\to\mathscr{T}}(S))$$

Note that in the simplified query R is not rewritten at all. This kind of simplification is not specific to the example, but can be applied to a wide range of algebra expressions. We now prove the correctness and completeness of the rewrite rules and then discuss the simplification in detail.

Theorem 3.2 (Correctness and Completeness of the Transformation Provenance Rewrite Rules)**.** *For a algebra expression q the transformation provenance rewrite rules as presented in Figure 3.6 compute the relational representation of transformation provenance as defined in Definition 3.3:*

$$Q^{Trans} = Q^{T}$$

Proof.
To prove this theorem we use a modified version of the transformation provenance rewrite rules (denoted by q^{T+} that in addition to the annotation sets also propagate *PI-CS* witness list representations. The meta-operator implemented by the modified rewrite rules is called $T+$. The transformation provenance rewrite

rules use the same structural rewrites as the *PI-CS*. Therefore, the modified rewrites can be derived from the original $\mathscr{T}$ meta-operator rewrite rules by simply adding the provenance attribute list $\mathscr{P}$ to each outermost projection of an rewritten operator. E.g., the modified rule for projection is:

$$q^{T+} = (\Pi^{S/B}{}_A(q_1))^{T+} = \Pi^{B}{}_{A,\mathscr{T}(q^{T+})\rightarrow\mathscr{T},\mathscr{P}(q^{+})}({q_1}^{T+})$$

The modified versions allow us to reason over witness list over which a $\mathscr{T}$ set is defined. Instead of the original equivalence $Q^{Trans} = Q^T$ we proof the equivalence $Q^{TransPI} = Q^{T+}$ where $Q^{TransPI}$ is defined as

$$Q^{TransPI} = \{(t, \Theta_w, w')^{m\times p} \mid t^p \in Q \wedge w^m \in \mathscr{D}\mathscr{D}(q,t)\}$$

The only difference between Q^{Trans} and $Q^{TransPI}$ respective Q^T and Q^{T+} is that, in addition to the annotation sets, also the witness list from which the annotation set is derived from is represented. Therefore, the equivalence $Q^{Trans} = Q^T$ follows from $Q^{TransPI} = Q^{T+}$. Similar to the proof for *PI-CS*, the equivalence $Q^{TransPI} = Q^{T+}$ is proven in two steps. First the *result preservation* property of $T+$ is proven. Afterwards, we prove that for tuple $(t, w', x) \in Q^{T+}$, the set x is the annotation set Θ_w derived for witness list w (correctness), and that for every witness list $w \in \mathscr{D}\mathscr{D}(q,t)$ the tuple (t, w', Θ_w) is in Q^{T+} (completeness). We refer to this property as *Annotation Set Preservation*). From the proof of *PI-CS* we know that for a witness list $w \in \mathscr{D}\mathscr{D}(q,t)$ a tuple (t, w', x) is in Q^{T+}. Hence, only the correctness part of the *annotation set preservation* has to be proven.

Result Preservation

The transformation provenance rewrite rules apply the same structural rewrites as the *PI-CS* rewrite rules. For the *PI-CS* rewrite rules we have proven that they fulfill the *result preservation* property. Therefore, this property is also fulfilled for the *transformation* rewrite rules.

Annotation Set Preservation

We prove the *annotation set preservation* by induction over the structure of an algebra expression q. We have to show that each result tuple in Q^{T+} is of the form (t, w', Θ_w).

Induction Start:

For $q = R$ each tuple in Q^{T+} is of form (t, w', T) with $w' = t$ and $T = \{R\}$. We have to show that $\{R\} = \Theta_w$. $\{R\}$ is contained in Θ_w if $R(w) = R(< t >) \neq \emptyset$ which is trivially fulfilled, because $R(< t >) = \{t\}$.

Induction Step: Given that the transformation provenance rewrite rules produce correct annotation sets for algebra expressions with a maximal nesting depth of n we have to prove that the same holds for algebra expressions with nesting depth $n+1$. Let op be an unary operator and q_1 be an algebra expression with maximal nesting depth $n+1$. Then we have to show that for (t, w', T) in $(op(q_1))^{T+}$ the following holds: $T = \Theta_w$.

Case $\sigma_C(q_1)$:

The rewrite rule for selection applies the unmodified selection and adds the annotation for the selection to the annotation set from ${q_1}^{T+}$. Hence, each tuple from q^{T+} is of form $(t, w', \{q\} \cup \Theta_v))$ where Θ_v is the annotation set of the witness list v in $\mathscr{D}\mathscr{D}(q_1, t')$ from which w is derived from. To prove that $\{q\} \cup \Theta_v = \Theta_w$ we have to show that (1) Θ_w contains $\{q\}$ and that (2) $\Theta_w \cap OP(q_1)$ agrees with $\Theta_v \cap OP(q_1)$. Here $OP(q_1)$ denotes the node identifiers for all operators in q_1. If w is a witness list for q then $q(w) \neq \emptyset$. Thus, the first property is fulfilled. From the compositional semantics of *PI-CS* we know that $w = v$. Therefore, the property (2) is fulfilled, because each $sub_{op}(w) = sub_{op}(v)$ for op in q_1.

Case $\Pi^{S/B}{}_A(q_1)$:

Case $\alpha_{G,agg}(q_1)$:

As for selection the rewrite rules for projection and aggregation produce tuples of the following form $(t, w', \{q\} \cup \Theta_v)$. Property (1) and (2) holds for the same reason as for selection.

For binary operators we can use the same reasoning as for unary operators to prove property (1) (the node identifier of op is included in the annotation set of op). For the proof of property (2) for binary

operators the following cases have to be considered (based on the three cases in the definition of transitivity for *PI-CS*):

1. The witness list $w = v_1 \blacktriangleright v_2$ for v_1 and v_2 being the witness lists for q_1 respective q_2 from which w is derived trough transitivity.
2. The witness list $w = v_1 \blacktriangleright <\bot,\ldots,\bot>$ for v_1 being the witness lists for q_1 from which w is derived trough transitivity.
3. The witness list $w = <\bot,\ldots,\bot> \blacktriangleright v_2$ for v_2 being the witness lists for q_2 from which w is derived trough transitivity.

Case $q_1 \times q_2$:
Case $q_1 \bowtie_C q_2$:
Case $q_1 \cap^{S/B} q_2$:

For cross product, join, and intersection only the first case applies and the proof is analog to the proof for unary operators, because each of these operators combine the Θ_{v_1} and Θ_{v_2} sets.

Case $q_1 \text{⟕}_C q_2$:

For left outer join the first and the second case apply. The first case applies for tuples which fulfill the join condition. Therefore, the annotation set construction $(\{q_1 \text{⟕} q_2\} \cup \mathbf{Q_1}.\mathscr{T} \cup \mathbf{Q_2}.\mathscr{T})$ applied by the rewrite rule generates the correct set. The second case applies for tuples t that do not fulfill the join condition. In this case all parts of w that correspond to q_2 are set to $\emptyset$. It follows that $[[q_2(w)]] = \emptyset$. This means in Θ_w does not include any node identifiers from $OP(q_2)$. This correctly modeled in the rewrite rule, because t does not fulfill the join condition and, therefore, attribute $\mathbf{Q_2}.\mathscr{T}$ is null (Recall that we use ε is an alternative representation of the empty set).

Case $q_1 \bowtie\!\text{⟖}_C q_2$:
Case $q_1 \text{⟕}\!\text{⟖}_C q_2$:
Case $q_1 \cup^{S/B} q_2$:

The proof for right outer join, full outer join, and union are analog to the proof of left outer join.

Case $q_1 -^{S/B} q_2$:

For set difference only the second case applies. The rewrite rule discards the annotation set for q_2. Hence, no node identifiers from $OP(q_2)$ are included in the resulting annotation set.

□

3.4.1 Rewrite Rules Simplification

Recall that we presented a simplification of the rewritten example query ${q_a}^T$ that uses the fact that some node identifiers are guaranteed to be in the annotation sets of ${q_a}^T$. In general a node identifier for an operator op is guaranteed to be in the annotation set of a rewritten query if computing $sub_{op}(w)$ for a witness list $w \in \mathscr{DD}(q,t)$ for $t \in Q$ never returns the empty set. We use this fact to identify criteria for applying the presented simplification. According to the compositional semantics of *PI-CS* the only operators that include $\bot$ in witness lists are the outer joins, union and set difference. According to the transitivity of *PI-CS* the only possibility $\bot$ can occur in a witness list of a query q is that one of the aforementioned operators is used in q. From the definition of *PI-CS* (condition 2) and the definition of the algebra operators we can deduce that the evaluation of q over a witness list will never result in the empty set if none of the $\bot$ generating operators is used in q. This means queries involving only operators that never include $\bot$ in their witness lists (which we refer to as *transformation static*) are rewritten by simply adding an projection to the query that generates the static annotation set for this query. By static we mean that the annotation set is independent of the input data. Even more, if a query includes non-static operators, all static operators that are in an sub-tree for which the root operator is non-static can be rewritten by adding the annotation set trough a projection.

Proposition 3.1 (Transformation Provenance Rewrite Simplification)**.** *For an algebra expression q, all operators that are static and do not have a non-static operator as their ancestor in the algebra tree of q or are only in the left/right sub-tree of an left/right outer join have static transformation provenance.*

3.5 Managing Mapping Provenance

3.5.1 Mapping Provenance Integration

To support *mapping* provenance the transformation provenance computation is modified to use the mapping annotation functions. We represent the mapping annotation function μ_M for each mapping $\mathbf{M}$ as a set that contains a operator identifiers for which $\mu_M(I(op)) = 1$ for the operator op with this identifier ($I(op)$. Recall that a mapping belongs to the mapping provenance iff its mapping function μ_M agrees with θ_w on every input ($\mu_M(op) = \theta_w(op)$). Translated set representations of μ_M and θ_w this is an equality-check on these sets.

3.6 Summary

In this chapter we introduced relational representations for provenance according to the *contribution semantics* presented in chapter 2 and demonstrated how to generate these representations by evaluating rewritten algebra expressions. Several meta-operators were discussed that transform an algebra expression q into a rewritten form that computes the relational representation of a type of provenance for q. For each of the meta-operators we proved that the rewritten algebra expressions generated by this operator computes the corresponding relational representation of provenance. For *PI-CS* provenance of algebra expressions with sublinks we have introduced several rewrite strategies that utilize un-nesting and de-correlation techniques to rewrite these queries. Furthermore, for *transformation* provenance we presented simplifications for the rewrites and demonstrated when they can be applied. In summary, we have theoretically sound algorithms for computing relational provenance representations according to several *CS* types. The relational representations allow us to store provenance in a relational database and query it using *SQL*. Which is a huge advantage over existing approaches that do not support querying of provenance information at all or supply only very limited query capabilities. Modeling provenance computation as algebraic rewrites has the intrinsic advantage that provenance computations can be seamlessly integrated into *SQL* which is not the case for other provenance approaches, because they usually develop a new language for provenance computation and querying. Note that though these languages may be implemented as rewrites too, this in general does not mean that they can be easily integrated in SQL. This is due to the fact that the applied provenance representation is not relational and is produced by post-processing the result of the rewritten queries. In the next chapter we will demonstrate how to integrate the *Perm* provenance representation and computation into a relational *DBMS*.

Bibliography

[1] Michael O. Akinde and Michael H. Böhlen. Efficient Computation of Subqueries in Complex OLAP. *ICDE '03: Proceedings of the 19th International Conference on Data Engineering*, pages 163–174, 2003.

[2] Lars Baekgaard and Leo Mark. Incremental Computation of Nested Relational Query Expressions. *ACM Transactions on Database Systems (TODS)*, 20(2):111–148, 1995.

[3] Adriane Chapman and H. V. Jagadish. Why Not? In *SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 523–534, 2009.

[4] Yingwei Cui. *Lineage Tracing in Data Warehouses*. PhD thesis, Stanford University, 2002.

[5] Yingwei Cui and Jennifer Widom. Lineage Tracing in a Data Warehousing System. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering (demonstration)*, pages 683–684, 2000.

[6] Boris Glavic and Gustavo Alonso. Perm: Processing Provenance and Data on the same Data Model through Query Rewriting. In *ICDE '09: Proceedings of the 25th International Conference on Data Engineering*, pages 174–185, 2009.

[7] Boris Glavic and Gustavo Alonso. Provenance for Nested Subqueries. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 982–993, 2009.

[8] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance Semirings. In *PODS '07: Proceedings of the 26th Symposium on Principles of Database Systems*, pages 31–40, 2007.

[9] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A Survey of Data Provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.