Using SQL for Efficient Generation and Querying of Provenance Information

Boris Glavic[1], Renée J. Miller[2], and Gustavo Alonso[3]

[1] Illinois Institute of Technology
bglavic@iit.edu
[2] University of Toronto
miller@cs.toronto.edu
[3] ETH Zurich
alonso@inf.ethz.ch

Abstract. In applications such as data warehousing or data exchange, the ability to efficiently generate and query provenance information is crucial to understand the origin of data. In this chapter, we review some of the main contributions of Perm, a DBMS that generates different types of provenance information for complex SQL queries (including nested and correlated subqueries and aggregation). The two key ideas behind Perm are representing data and its provenance together in a single relation and relying on query rewrites to generate this representation. Through this, Perm supports fully integrated, on-demand provenance generation and querying using SQL. Since Perm rewrites a query requesting provenance into a regular SQL query and generates easily optimizable SQL code, its performance greatly benefits from the query optimization techniques provided by the underlying DBMS.

1 Introduction

Peter Buneman was one of the first to recognize the importance of data provenance. With co-authors Khanna and Tan, he introduced two seminal models of *Why*- and *Where*-provenance [7]. Provenance, information about the creation process or the origin of data, can be used to debug queries and clean data in data warehouses, to understand and correct complex data integration transformations, for auditing, and to understand the value of data in curated databases. Provenance generation has also been used as a supporting technology for exchanging updates between heterogeneous databases [21], to provide access control based on the origin of data [31], and in modeling uncertainty in databases [35].

While provenance has many applications, these applications often place very high requirements on a provenance management system to be useful in practice. In this chapter, we overview the contributions of the *Perm* provenance management system [17]. *Perm* was designed as a scalable system for the generation and querying of provenance information over relational data. To understand the requirements for such a system, we begin with an example and then consider the foundations in provenance research on which *Perm* builds.

Customer

	SSN	name	age
u_1	1	Gert	34
u_2	2	Waltraud	65
u_3	3	Joe	19

Purchase

	month	desc	amount	creditc	import
p_1	Jan	starbucks	12	4059	1
p_2	Jan	grandson	3100	1234	1
p_3	Jan	rent	7000	1235	1
p_4	Feb	rent	7000	1235	1
p_5	Feb	tvshop	399	9999	2
p_6	Feb	starbucks	5	9999	2

Creditcard

	number	company	owner	limit
c_1	4059	VISA	1	4000
c_2	3066	MASTER	2	2000
c_3	1234	VISA	2	3000
c_4	1235	VISA	3	10000
c_5	9999	AE	3	400

Imports

	id	employee	company	date
i_1	1	Daniel	VISA	10.06.2000
i_2	2	Petra	AE	06.06.2000

Fig. 1. Example Database

Example 1 (Running Example). The example database shown in Figure 1 stores credit card information: customers, their credit cards, purchases made with credit cards (`Purchase`), and from which external database (recorded in the `company` attribute) a batch of purchase tuples was imported, when and by whom (`Imports`). For convenience, we show an identifier for each tuple in the instance (e.g., p_2). The query q shown in Figure 2 returns the months during which customers with at least two credit cards exceeded their credit limit on some card. To understand from which inputs of q the result tuple t_2 `(Joe,Feb)` is derived, a user needs access to the data provenance of the query and the ability to query this information. For example, a user may be interested in knowing if some of these over-drafts are caused by suspiciously low credit card limits. This question can be answered by running a query over the provenance of q to retrieve tuples in the result of q that depend on credit card tuples with low limit values (i.e., these credit card tuples belong to the data provenance of the tuples to be returned). Alternatively, if the user realizes that some names are spelled incorrectly in the query result, she needs to understand where the name attribute values in the query result have been copied from to trace this error. This requires access to a different type of provenance that tracks the copying of information instead of which inputs caused a tuple to appear in the query result.

1.1 Requirements for Provenance Systems

The example and discussion above motivates four requirements for relational provenance systems. **(Requirement 1)** Support different types of provenance with sound semantics. Information from different provenance types is often needed to best understand the data and how it has been transformed. We would consider a provenance type to have sound semantics, if it provably captures our intuitive understanding of provenance. For example, the provenance of a query

```
SELECT DISTINCT name, month
FROM
     (SELECT month, creditc, SUM(amount) AS total
     FROM purchase p
     GROUP BY month, creditc) AS monthly,
     customer c,
     creditcard cc
WHERE p.cc = cc.number
      AND cc.owner = c.id
      AND total > cc.limit
      AND c.id IN (SELECT cc2.owner
                   FROM creditcard cc2
                   GROUP BY cc2.id
                   HAVING count(*) > 1)
```

Query Result

	name	month
t_1	Waltraud	Jan
t_2	Joe	Feb

Fig. 2. Example Query

result should be sufficient to derive this result through the query. **(Requirement 2)** Support provenance generation for SQL including complex features such as nested subqueries and aggregation. The example query is relatively simple in comparison with queries used in data warehouse applications. A system must support a large subset of SQL to be useful in practice. **(Requirement 3)** Support complex queries over provenance information. Provenance is difficult to interpret without the ability to extract parts of interest. For instance, even identifying for which tuples the provenance is interesting requires query support to be feasible for large databases. Generally, users will want to use queries to specify the characteristics of what provenance they want to see. Many interesting questions that can be answered using provenance data require the use of advanced SQL-like features such as aggregation over the provenance. For example, *Which over-drafts are based on a large number of small purchases (high count, but low average amount)?* **(Requirement 4)** Support efficient generation and querying of provenance for large database instances. Provenance can easily outgrow the size of the database for complex queries. Unless a user explicitly requests *all* the provenance, the system should efficiently generate only provenance that satisfies the user's request (by combining provenance generation with a user's query over the provenance). In our example, if a user is only interested in over-drafts due to low credit limits (a query on provenance), then the system should not generate provenance for all over-drafts.

1.2 State of the Art

The tremendous amount of work on relational provenance brings us close, but not all the way, to achieve these requirements. We present the state-of-the-art along these four dimensions and discuss how Perm has contributed to each.

(1) Support for different types of provenance. The largest body of work on relational provenance is on semantics. We have a rich literature on different semantics, along with a rich literature comparing these semantics and analyzing when they are useful [10]. *Data provenance*, which represents dependencies between a query's output and input data, has been categorized based on the type of dependency that is modeled. *Why*-provenance, intuitively, models which input tuples are used to create an output tuple, though there are different ways to formalize this notion. Types of *Why*-provenance are the original *Why*-provenance as pioneered by Buneman et al. [7], *Lineage* proposed by Cui et al. [12], and *PI-CS* (Perm Influence contribution semantics) the original provenance semantics supported by Perm [16]. *Where*-provenance models where values in an output tuple are copied from. Types of *Where*-provenance include the *Where*-provenance introduced by Buneman et al. [7] and the *C-CS* semantics (Copy contribution semantics) of Perm [16]. *How*-provenance augments *Why*-provenance with information about how input tuples are used to create an output tuple. *Provenance polynomials* [22, 24] and later versions of the *Trio* [35] provenance model can be classified as *How*-provenance. Provenance polynomials are the most general form of annotation in the framework of Green et al. [22] that defines the positive relational algebra for relations annotated with elements from a semiring (called K-relations). Thus, provenance polynomials generalize all provenance semantics that can be modeled as semirings such as the original Why-provenance and the Trio-model [23]. Foster et al. [14] use the semiring model to annotate unordered XML data and compute provenance for XQuery. Through an XML encoding of annotated relations and XQuery encoding of relational algebra this approach provides a type of attribute granularity *Where*-provenance. The semiring model has been extended for aggregation [4] and several extensions for set difference have been proposed [3, 20, 15]. Recently, Kostylev et al. [29] studied data annotated with more than one type of annotation within this framework. Several other data provenance types have been presented in the literature that do not fall directly under these categories. For example, causality-based provenance [30, 9], types inspired by program analysis [1, 8], and *transformation provenance* [19] (which operators of a query contribute to a result). Most provenance systems implement one type of provenance. *DBNotes* [11, 5, 33] is an annotation management system that uses Where-provenance [7] to propagate annotations. *Trio* [35] is a database system with support for uncertainty and provenance. Boolean formulas over tuple variables are used as provenance. Lineage was implemented in the *WHIPS* data-warehouse prototype [12]. The update-exchange system *Orchestra* [21] uses provenance polynomials to record the provenance of updates exchanged between peers. In principle, Orchestra also supports Why-provenance and the model of Trio, because these provenance types can be extracted from provenance polynomials. Green provides a provenance hierarchy showing how this extraction can be achieved [23].

Perm's Contribution To the best of our knowledge, Perm is the first system to support a representative set of provenance semantics including the relational adaptation of the *Where*-provenance [10] as defined by Buneman et

al. [7], provenance polynomials [22], and new types of *Why*, *Where*, and *How* (defined further throughout this chapter) that include a new form of transformation provenance [19]. In contrast to Orchestra, generation of these provenance types is supported natively instead of deriving them from a more expressive provenance model. This enables us to use type-specific optimizations during provenance generation for more efficient execution. Perm also supports propagating user-defined annotations based on Why semantics.

(2) Support for provenance generation for complex SQL. DBNotes supports the SQL equivalent of unions of conjunctive queries (set-semantics) [11, 5]. WHIPS computes Lineage for ASPJ (aggregate-select-project-join queries) and set operations (union, intersection and set difference) [12]. Lineage was defined for set-semantics, but extensions for bag-semantics were discussed by the authors. Trio supports ASPJ queries with set operations though the released prototype has stricter limitations (e.g., single aggregation in a query) [35]. Orchestra supports union of SPJ queries and is the only approach to support recursion [21]. However, the semiring model used by Orchestra has also been extended for aggregation [4]. In contrast to the Lineage and Perm Why-provenance models, which only record provenance for each result tuple of an aggregation, this extension of the semirings model attaches provenance to each aggregated value. This has the advantage of enabling deletion propagation, but results in increased provenance size and a more complex provenance model.

Perm's Contribution Like WHIPS, Perm supports ASPJ queries and set operations. Perm is the first provenance system to support nested and correlated subqueries.

(3) Support for complex queries over provenance information. Most systems do not represent provenance relationally. To query provenance, they provide special query languages over their provenance data model. The query language of DBNotes, *pSQL* [11, 5], provides some support for querying annotations (provenance) which is equivalent to being able to pose SPJ queries with unions on the provenance. Orchestra supports *ProQL* [25], a query language for the graph representation of provenance polynomials for relations derived though schema mappings. ProQL queries return a subgraph of the input based on path expressions used in the query and optionally evaluate the provenance polynomial of a tuple in a certain semiring, i.e., change the type of annotations attached to tuples.[4] The language does not support aggregation directly. However, some types of aggregation can be simulated using semiring evaluations. *TriQL* [34], the query language of Trio, has a conditional language construct that evaluates to true if tuples from two specified relations are connected by lineage. WHIPS [12] does not introduce a new query language for provenance. SQL queries can be used to query provenance generated by the system. However, the system represents provenance as a list of relations which makes querying this information more complicated. WHIPS does not associate data with its provenance.

[4] This feature can be used to derive other provenance types from the polynomials.

Perm's Contribution Perm uses a relational representation for provenance that models the connection between a query result tuple and its provenance. Hence, Perm supports full SQL for querying data associated with provenance.

(4) Support for large databases. In DBNotes [5] provenance annotations for relations in a database are materialized. DBNotes generates provenance during the execution of a pSQL query. Such a query is translated into a single SQL query over a relational encoding of annotated relations. This allows the system to rely on a DBMS to optimize the execution. However, the SQL query results have to be post-processed to transform them into DBNotes's data model which introduces a potential performance bottleneck. A query in Orchestra's query language ProQL [25] is implemented by running several queries over a materialized relational encoding of a provenance graph. Orchestra produces provenance during update-exchange. Update-exchange and provenance generation is expressed in datalog extended with skolem functions and implemented in a Java middleware which evaluates the datalog rules over a relational DBMS. Even though some care is taked to avoid shipping data between Java and the DBMS, using several SQL queries to implement a single ProQL query and full materialization of provenance information limits the scalability of the approach. Trio [2] generates provenance eagerly during query execution. The system materializes the results of each query and creates a separate relation to store its provenance as a mapping between input and output tuple identifiers. Trio is implemented as a Python middleware and a set of PostgreSQL UDFs (user-defined functions). WHIPS [12] implements provenance generation as stored procedures that split a query q into subexpressions and execute one or more SQL queries to retrieve the Lineage of each segment. This separation into multiple queries limits the space of possible optimizations that the underlying DBMS can apply.

Perm's Contribution Provenance generation in Perm is on-demand, meaning that Perm supports simple SQL language extensions (SQL-PLE) to let a user specify when (and what) provenance to compute. In Perm, a query over provenance information would usually include a subquery that generates the provenance. Thus, provenance generation and querying are entangled within a single SQL-PLE query that is rewritten by the system into a single SQL query. This approach allows us to take full advantage of the optimizer of the underlying DBMS. For SQL queries without nesting, our experience shows that the optimizer can (and does) significantly improve the performance of provenance queries by, e.g., pushing selections over provenance data into the provenance generation. For nested subqueries, we present a set of novel un-nesting and de-correlation optimizations tailored for provenance generation.

In summary, given the maturity of data provenance models, with Perm we sought to build upon the state-of-the-art in provenance systems to provide a complete relational provenance management system that supports efficient querying and generation of provenance. Our approach focuses on robust SQL support (including correlated subqueries) and full support for querying provenance using SQL. Approaches that generate and store the complete provenance of a query during execution incur large storage costs and runtime overheads, and, thus

Semantics	Category	Granularity
PI-CS	Data (Why)	Tuple
C-CS	Data (Where)	Tuple
Transformation Provenance	Transformation	Algebra Operator
Where	Data (Where)	Attribute Value
Polynomials	Data (How)	Tuple

Fig. 3. Supported Provenance Types

may not be applicable to large databases and/or complex queries. We call such approaches *exhaustive* to distinguish them from approaches that only generate provenance *on-demand*. The main innovation of Perm is to represent a query's result and provenance in a single relation which is generated on-demand by rewriting the original query into a query producing this representation. In the remainder of this chapter, we overview how this simple idea enabled the development of a robust relational provenance system that achieves the advances towards all four requirements we have presented.

We give an end-to-end overview of our approach in Section 2. Afterwards, we present three of the provenance types supported by Perm in detail and discuss how they were implemented within the system (Sections 3 to 5). For each provenance type, we present the formal definition, the algebraic (and SQL) rewrites used to generate its relational representation, and present some of the optimizations that can be applied in a provenance system like Perm.

2 The Perm Approach

We now present an overview of the Perm system focusing on its relational provenance representation (Section 2.1), query rewrite techniques (Section 2.2), and SQL language extensions (Section 2.3). Perm represents provenance information as relations generated and queried on-demand using standard SQL queries. If the users requests one of the provenance types supported by Perm for a query q using the SQL-PLE language extension, the system transforms q into an SQL query that returns the provenance of q in addition to the regular results of q. Perm supports the provenance types shown in Figure 3. The initial version supporting *PI-CS* provenance (*Perm Influence contribution semantics*, a form of *Why*-provenance) for ASPJ queries and set operations was introduced by Glavic and Alonso [17] and later extended for nested and correlated subqueries [18]. *Transformation* provenance, provenance that models which operators of a query influence a query results, was introduced in *TRAMP* [19], an extension of Perm for debugging data exchange scenarios. In this chapter, we also present *Copy contribution semantics*, a *Where*-provenance type supported by Perm, and several optimizations for PI-CS [16]. To demonstrate the flexibility of our approach we have also implemented the original Where-provenance [7] and provenance polynomials [22] in Perm. As mentioned in the introduction, provenance polyno-

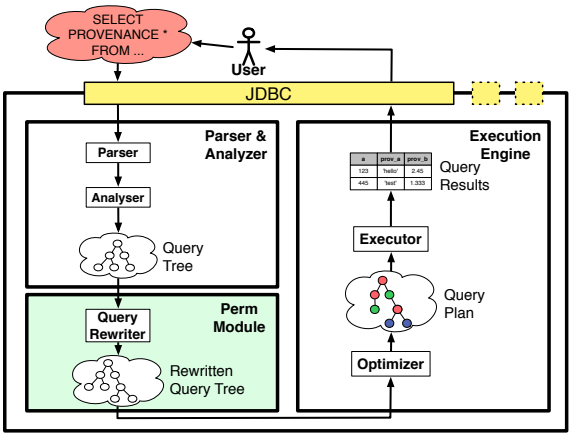

Fig. 4. *Perm* Architecture

mials generalize several other provenance semantics. We will discuss how PI-CS relates to this model in Section 3.6.

Perm is implemented as a modified PostgreSQL engine, extending its SQL dialect with provenance features. Provenance generation in Perm is light-weight and lazy: no provenance is generated unless explicitly requested. Thus, if the provenance features of Perm are not used, the system behaves like a normal Postgres server - clients will observe no overhead in runtime [5] or storage space. Figure 4 shows the architecture of the system. The parser and analyzer module of PostgreSQL (extended to recognize SQL-PLE) parse incoming SQL queries and transform them into an internal tree representation. The output of the analyzer module is passed to the *Perm* rewrite module. This module implements the query rewrite rules as transformations on query trees. The rewritten query tree produced by the *Perm* module is handed over to the original Postgres optimizer. From the optimizer's point of view the input it retrieves is a regular SQL query.

2.1 Provenance Representation

Perm represents the provenance of a query q as a single relation that contains both the original query results of q and its provenance. Provenance information is attached to a query result tuple by extending the tuple with additional attributes that are used to store provenance information. Regular result tuples are duplicated if necessary to represent the complete provenance.

Data Provenance: PI-CS and C-CS, the two data provenance semantics developed for Perm, represent provenance as so-called witness lists. A witness-list for a query is a list of input tuples that were used together to derive an output tuple; one from each input relation of the query (leaves of the algebra tree) or the special value $\perp$ which indicates that no tuple from the relation at this leaf of

[5] Except for an additional traversal of the query tree to search for SQL-PLE constructs.

the tree contributed to the output tuple. The relational representation of PI-CS and C-CS appends all attributes from the relations accessed by the query to the query's result schema. The additional attributes in the provenance representation are used to extend a result tuple with all tuples from one of its witness lists. Thus, tuples with more than one witness list in their provenance are duplicated and each duplicate is paired with the relational encoding of one witness list. To distinguish between regular result attributes and provenance attributes, the later are identified by a prefix and the name of the relation they are derived from (adding a distinguishing identifier for relations that are accessed more than once by the query). The special value $\perp$ used in witness lists is modeled as `NULL` values in the representation.

Transformation Provenance: Transformation provenance models which parts of a query (that is, which operator) contributed to an output tuple. Provenance is represented as a single attribute of either type text or XML that stores the SQL string of the query (or an XML representation thereof) with the transformation provenance modeled as tags (`<NOT>...</NOT>`) that surround parts of the query that did not contribute to a result tuple. We introduced the XML representation to enable query access to transformation provenance (using the XSLT support of PostgreSQL).

Example 2 (Provenance Representation). Consider the query shown in Figure 5 evaluated over the example database and its PI-CS and transformation provenance. The provenance attribute names for PI-CS are given in a separate table to simplify the exposition. Tuple t_2 in the result of the query was derived by joining tuple u_2 with tuples c_2 and c_3. Thus, the PI-CS provenance of tuple t_2 consists of two witness lists $< u_2, c_2, \perp >$ and $< u_2, c_3, \perp >$. These are represented as two tuples in the relational representation by duplicating t_2 and pairing each duplicate with the tuples from one of the witness lists. Tuple t_1 is derived from the left input of the union without any influence from its right input. Therefore, the right input is enclosed in a `NOT` tag in the transformation provenance of t_1.

The provenance representation used in Perm has several advantages. (1) Provenance is represented as a standard relation, that can be stored as a view or queried using SQL. Even more important, the system can often avoid generating provenance that will be filtered out in later stages of a query using the DBMS optimizer (see Section 2.2). (2) Representing data provenance as complete tuples and directly associating a query's regular result data with its provenance allows a user to understand how they relate to each other and enables queries that make use of this information.

However, these advantages come at the price of verbosity and in some cases loosing the ability to run queries over the regular results. The verbosity is usually unproblematic, because the user can run queries over this information to extract parts of interest and instruct the system to only use certain attributes as provenance instead of complete input tuples. The duplication of regular result tuples is necessary to be able to pair them with their complete provenance, but it may restrict the execution of normal queries over this relation (i.e., result tuple multiplicities may be different from the multiplicities of the original

```
(SELECT name
FROM customer c JOIN creditcard cc
     ON (c.ssn = cc.owner))
UNION
(SELECT employee FROM imports);
```

Query Result

	name
t_1	Gert
t_2	Waltraud
t_3	Joe
t_4	Daniel
t_5	Petra

PI-CS Provenance

		customer			creditcard				imports			
	name	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}
t_1	Gert	1	Gert	34	4059	VISA	1	4000				
t_2	Waltraud	2	Waltraud	65	3066	MASTER	2	2000				
t_2	Waltraud	2	Waltraud	65	1234	VISA	2	3000				
t_3	Joe	3	Joe	19	1235	VISA	3	10000				
t_3	Joe	3	Joe	19	9999	AE	3	400				
t_4	Daniel								1	Daniel	VISA	10.06.2000
t_5	Petra								2	Petra	AE	06.06.2000

Attribute Names

alias	attribute name
p_1	prov_customer_ssn
p_2	prov_customer_name
p_3	prov_customer_age
p_4	prov_creditcard_number
p_5	prov_creditcard_company
p_6	prov_creditcard_owner
p_7	prov_creditcard_limit
p_8	prov_imports_id
p_9	prov_imports_employee
p_{10}	prov_imports_company
p_{11}	prov_imports_added

Transformation Provenance

	name	trans_prov
t_1	Gert	`SELECT c.name FROM customer c JOIN creditcard cc ON c.ssn = cc.owner UNION <NOT>SELECT imports.employee FROM imports</NOT>`

Fig. 5. Provenance Representation

query). However, the original result multiplicities can be reconstructed from the provenance and input multiplicities if needed.

2.2 On-Demand Provenance Generation Using Query Rewrites

The research underlying Perm has demonstrated that SQL is powerful enough to express the computation of provenance for a large subset of queries expressible in SQL. The approach supports aggregations, set operations, nested or correlated subqueries, and user-defined functions. We do not support non-deterministic functions that return different results for the same input in the scope of one query. For example, a random number generator is a non-deterministic function.

Requesting the provenance of a query q through the system's SQL extensions (see Section 2.3) instructs Perm to rewrite q into a standard SQL query that returns one type of provenance for q using the provenance representation introduced in Section 2.1. The query rewrites for each provenance type were developed following the process shown in Figure 6. (1) We state a provenance type's semantics as a declarative definition and define a relational representation. This approach was chosen because correctness criteria one would intuitively expect to hold for provenance are easily stated declaratively. For instance, for data prove-

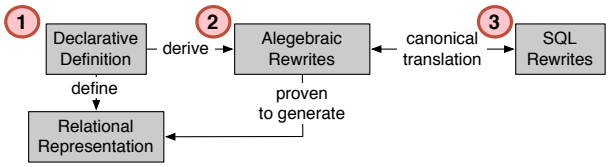

Fig. 6. SQL Rewrite Development Process

nance, the provenance of a tuple t from the result of a query q should contain sufficient information to produce the tuple t. (2) From the declarative definition we derive algebraic rewrites which transform a query into a provenance-generating query and prove their correctness. (3) A canonical translation is applied to translate the algebraic rewrites into SQL rewrites.

The seamless integration of provenance generation as an SQL language feature has many advantages. We can provide full SQL query support for provenance information (Requirement 3). The rewrite rules are unaware of how the provenance attributes of their input were produced. Thus, they can be used to propagate provenance information that was created manually or by another provenance management system. A query over provenance data is implemented as a regular SQL query with a subquery that implements the provenance generation. Thus, we fully utilize the DBMS optimizer to speed up provenance computation by, e.g., pushing selections and projections applied by a query into the provenance generation (Requirement 4). Since optimizing provenance generation is still in its infancy, this is a feasible approach for efficient provenance generation and querying (e.g., we can efficiently compute the PI-CS provenance of the TCP-H benchmark queries for a 1GB TCP-H instance [16]).

2.3 SQL Language Extension

The *provenance language extension* (SQL-PLE) of Perm enriches SQL with additional keywords to request provenance, control how far to trace provenance, and to inform the system about existing provenance information. The keyword `PROVENANCE` is employed in the `SELECT` clause of a query q to instruct *Perm* to compute the provenance of q. An optional `ON CONTRIBUTION` modifier is used to choose the provenance type that is produced (PI-CS is the default). For example, the query below returns the PI-CS provenance of the query from Figure 2.

```
SELECT PROVENANCE DISTINCT name, month
FROM (SELECT month, creditc, SUM(amount) AS total
      ...
```

Note that all original SQL features provided by PostgreSQL are not affected by the language extension, and even more important, they can be used in combination with provenance computation. Given the provenance representation of Perm this enables complex queries that filter provenance based on properties of the input tuples in the provenance, the results of the query, or both. This type of query functionality generalizes what has been called *backward* (track the

provenance of an output) and *forward* (which outputs have a certain input in their provenance) provenance queries in related work [6, 27, 26].

Example 3 (Querying Provenance). Assume the user expected the running example query to return less credit over-drafts. Her assumption is that some over-drafts are caused by credit card limits which have been recorded too low. The user runs the following query to determine which over-drafts are caused by (have tuples in their provenance with) suspiciously low credit card limits (say $500):

```
SELECT *
FROM (SELECT PROVENANCE DISTINCT name, month
      ...
      HAVING count(*) > 1)) AS orig
WHERE prov_creditcard_limit < 500;
```

The default behavior is to generate the provenance of a complete query by tracing which tuples in a query's output are affected by which tuples in the query's input. Perm also supports limiting the provenance generation to parts of a query to trace the effect of intermediate query results instead of the input relations. The keyword `BASERELATION` is appended to an item in the `FROM` clause to limit how far back the provenance is traced.

Example 4 (Limit Provenance Generation). Retrieving the full provenance of the running example query may return a large number of tuples, because each aggregated monthly amount (subquery `monthly`) can depend on a large number of individual purchases. Questions like the one from Example 3 can be answered without information about the influence of each individual purchase tuple. The user can mark the subquery `monthly` with the `BASERELATION` keyword to only investigate the effect of the aggregated monthly amounts.

```
SELECT PROVENANCE DISTINCT name, month
FROM (SELECT month, creditc, SUM(amount) AS total
      FROM purchase p
      GROUP BY month, creditc) BASERELATION AS monthly,
      ...
```

Perm can handle existing provenance information that was not produced by the system itself as long as (1) it is stored in additional attributes of tuples following the representation used by Perm and (2) the system is made aware of which attributes store provenance information (by appending the keyword `PROVENANCE` followed by a list of attribute names to the `FROM`-clause item).

Example 5 (External Provenance). The `imports` relation from the running example stores from which data sources each purchase tuple is imported. This is a type of provenance information for the purchase relation. Joining the imports relation with the purchase relation and using the `PROVENANCE` keyword in the `FROM` clause, the user makes Perm aware of the existence of the additional provenance data. The system will treat this provenance in the same way as provenance generated by the system itself. The modified example query is shown below.

```
SELECT PROVENANCE DISTINCT name, month
FROM (SELECT month, creditc, SUM(amount) AS total
      FROM (SELECT *
            FROM purchase, imports
            WHERE id = import
            ) PROVENANCE (employee, company, added) AS p
      GROUP BY month, creditc) AS monthly,
      ...
```

3 Perm Influence Contribution Semantics (PI-CS)

This and the following sections discuss the provenance types supported by Perm in more depth. Recall that we follow the process shown in Figure 6 to develop provenance semantics that are implemented as SQL query rewrites. The PI-CS provenance semantics was developed based on Lineage [12]. Lineage defines provenance for single operators declaratively. This definition is extended for queries with more than one operator by assuming transitivity. Lineage represents the provenance of a tuple t from the result of a query q as a list of relations; each element in the list is a subset of one input relation of the query. PI-CS also uses a declarative per-operator definition and transitivity, but represents provenance as witness-lists, defines a relational representation (see Section 2.1), and extends the declarative definition of the semantics with additional constraints to handle outer-joins, set difference, and nested subqueries correctly. For the proofs of the theorems we present in this section see Glavic [16].

3.1 Background and Notation

Before discussing the details of PI-CS, we present the relational algebra variant used in Perm and introduce notational conventions. The algebra (shown in Figure 7) is an extended relational algebra that operates on bags (multi-sets). We use t^n to denote that tuple t has the multiplicity n (number of duplicates) with the convention that a tuple with multiplicity smaller than one is not present in a relation. Let q be a query. We use $[[q]]$ (and sometimes Q) to denote the result of evaluating q and $\mathbf{Q}$ to denote its schema (the same notation is used for relations). The projection of a tuple t on a list A of attributes (or expressions) is denoted as $t.A$. Projection (Π) projects its input on a list of expressions over attributes, constants, functions and renaming (represented by $a \to b$). Selection (σ), joins ($\bowtie, \overline{\bowtie}, \ldots$), and set operations are defined as usual. Duplicate elimination (δ) returns the input relation with all tuple multiplicities set to one. Aggregation (α) groups its input on a list G of grouping expressions and computes the aggregation functions from list agg for each group. Here B_i denotes the list of input attributes for aggregation function agg_i. Each result tuple of an aggregation contains the grouping expression values and the aggregation function results (res_i) for one group. The value *null* is represented as ε and we write $null(q)$ for a tuple of null-values with schema $\mathbf{Q}$. Due to space limitations we do

$$[[\Pi_A(q)]] = \{t^n \mid n = \sum_{u^m \in Q \wedge u.A=t} m\} \qquad [[\sigma_C(q)]] = \{t^n \mid t^n \in Q \wedge t \models C\}$$

$$[[\alpha_{G,agg}(q)]] = \{(t.G, res_1, \ldots, res_m)^1 \mid t^n \in Q \wedge \forall_{i \in \{1,m\}} : res_i = agg_i(\Pi_{B_i}(\sigma_{G=t.G}(q)))\}$$

$$[[q_1 \bowtie_C q_2)]] = \{(t_1 \blacktriangleright t_2)^{n \times m} \mid {t_1}^n \in Q_1 \wedge {t_2}^m \in Q_2 \wedge (t_1 \blacktriangleright t_2) \models C\}$$

$$[[q_1 \overline{\bowtie}_C q_2]] = \{(t_1 \blacktriangleright t_2)^{n \times m} \mid {t_1}^n \in Q_1 \wedge {t_2}^m \in Q_2\}$$
$$\cup \{(t_1 \blacktriangleright null(q_2))^n \mid {t_1}^n \in Q_1 \wedge (\nexists t_2 \in Q_2 : (t_1 \blacktriangleright t_2) \models C)\}$$

$$[[q_1 \cup q_2]] = \{t^{n+m} \mid t^n \in Q_1 \wedge t^m \in Q_2\} \quad [[q_1 \cap q_2]] = \{t^{min(n,m)} \mid t^n \in Q_1 \wedge t^m \in Q_2\}$$

$$[[q_1 - q_2]] = \{t^{n-m} \mid t^n \in Q_1 \wedge t^m \in Q_2\} \quad [[\delta(q)]] = \{t^1 \mid t^n \in Q\}$$

Fig. 7. Algebra

not include the algebra for nested subqueries [18], but instead present an example. The SQL query `SELECT * FROM R WHERE R.a IN (SELECT b FROM S)` can be written as $\sigma_{a\ IN\ \Pi_b(S)}(R)$. We use $< e_1, \ldots, e_n >$ to denote a list with elements e_1 to e_n and $l_1 \blacktriangleright l_2$ to denote the concatenation of lists l_1 and l_2.

3.2 Declarative Definition

We start by stating the properties of PI-CS as a declarative definition and define a relational representation for this provenance type. The declarative definition allows us to directly state the properties we expect to hold for PI-CS. The PI-CS provenance for a result tuple of a query q is a subset of the multiset of *potential witness lists* for q - a set with all possible combinations of input tuples from the query and the special value $\perp$. Recall from Section 2.1 that $\perp$ denotes that no tuple from a specified relation participates in a witness list.

Definition 1 (Potential Witness Lists). *For a query q with inputs $q_1, \ldots, q_n$ the bag $\mathcal{W}(q)$ of potential witness lists for q is defined as:*

$$\mathcal{W}(q) = \{< t_1, \ldots, t_n >^{m_1 \times \ldots \times m_n} \mid \forall_{i \in \{1,n\}} : t_i^{m_i} \in Q_i \vee (t_i = \perp \wedge m_i = 1)\}$$

We use $w[i]$ to denote the i^{th} component (tuple) of a witness list w. A witness list w' subsumes a witness list w ($w \prec w'$) iff w can be derived from w' by replacing some tuples with $\perp$: $(\forall i : w[i] = w'[i] \vee w[i] = \perp) \wedge (\exists i : w'[i] \neq \perp \wedge w[i] = \perp)$.

The declarative definition for PI-CS defines the provenance of a tuple t from the result of a single algebra operator *op* as a subset of $\mathcal{W}(op)$ that fulfills the following four conditions. (1) Evaluating *op* over the provenance of t returns t.[6] This guarantees that the provenance of t is sufficient to produce t. (2) Each witness list w in the provenance contributes to the result, that is, evaluating the operator over w returns a non-empty result. (3) Subsumed witness lists are excluded from the provenance. This condition is necessary to produce precise

[6] Glavic [16] defines a semantics for query evaluation over sets of witness lists.

provenance for outer-joins and set union. (4) The provenance is the maximal multi-set with these properties, meaning that no witness lists that contribute to t are left out. The provenance of a query is defined by recursively applying the per-operator definition to each operator of the query.

Definition 2 (Declarative Definition of PI-CS). *Let op be an algebra operator with inputs $q_1, \ldots, q_n$ and t a tuple in the result of op ($t^x \in [[op]]$). A multi-set $P \subseteq \mathcal{W}(op)$ is the PI-CS provenance $PI(op, t)$ of t iff:*

$$[[op(P)]] = \{t^x\} \tag{1}$$

$$\forall w \in P : [[op(w)]] \neq \emptyset \tag{2}$$

$$\neg\exists w, w' \in P : w \prec w' \tag{3}$$

$$\neg\exists P \supset P' \subseteq \mathcal{W}(q) : P' \models (1), (2), (3) \tag{4}$$

The PI-CS provenance $PI(q, t)$ of a tuple t from the result of a query q is defined by transitivity over $PI(op, t)$ for each operator op in q.

For simplicity, we left out additional conditions applied in the definition to handle nested subqueries and adapted the definition slightly (without changing its semantics) [16]. We define the relational representation of the PI-CS provenance for a query q which combines each tuple t in Q with all witness lists in $PI(q, t)$.

Definition 3 (Relational Representation). *The relational representation Q^{PI} for the PI-CS provenance of a query q is defined as:*

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

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

3.3 A Compositional Semantics

The declarative definition does not provide a direct way to compute provenance except for the brute force method of evaluating the conditions of the definition for each provenance candidate (subset of $\mathcal{W}(q)$). A more algorithmic approach is needed to simplify the development and correctness proofs for the algebraic rewrites. We derive compositional rules that define the provenance of an algebra operator based on the provenance of its inputs and prove that these rules are equivalent to the declarative definition of PI-CS.

Definition 4 (Compositional Semantics for PI-CS). *Figure 8 shows a compositional definition of PI-CS. Here $\perp$ (q) denotes a witness list for q with $\perp$ values only.*

Note that we omitted the rules for right and full outer-join and for nested subqueries [16]. The following theorem states the equivalence between the declarative definition and the compositional rules.

$$
\begin{aligned}
PI(R,t) &= \{< t >^n \mid t^n \in R\} \\
PI(\sigma_C(q_1),t) &= PI(q_1,t) \\
PI(\Pi_A(q_1),t) &= \{w^n \mid w^n \in PI(q_1,u) \wedge u.A = t\} \\
PI(\alpha_{G,agg}(q_1),t) &= \{w^n \mid w^n \in PI(q_1,u) \wedge u.G = t.G\} \cup \{<\perp> \mid Q_1 = \emptyset \wedge \mid G \mid = 0\} \\
PI(q_1 \bowtie_C q_2,t) &= \{(w_1 \blacktriangleright w_2)^{n \times m} \mid w_1^n \in PI(q_1, t.\mathbf{Q_1}) \wedge w_2^m \in PI(q_2, t.\mathbf{Q_2})\} \\
PI(q_1 \overline{\bowtie}_C q_2,t) &= \begin{cases} \{(w \blacktriangleright \perp (q_2))^n \mid w^n \in PI(q_1, t.\mathbf{Q_1})\} & \text{if } t \not\models C \\ PI(q_1 \bowtie_C q_2, t) & \text{else} \end{cases} \\
PI(q_1 \cup q_2,t) &= \{(w \blacktriangleright \perp (q_2))^n \mid w^n \in PI(q_1,t)\} \cup \{(\perp (q_1) \blacktriangleright w)^n \mid w^n \in PI(q_2,t)\} \\
PI(q_1 \cap q_2,t) &= \{(w_1 \blacktriangleright w_2)^{n \times m} \mid w^n \in PI(q_1,t) \wedge w_2^m \in PI(q_2,t)\} \\
PI(q_1 - q_2,t) &= \{(w \blacktriangleright \perp (q_2))^n \mid w^n \in PI(q_1,t)\}
\end{aligned}
$$

Fig. 8. Compositional Semantics of PI-CS

Theorem 1 (Equivalence with Declarative Semantics). *The declarative and compositional definitions of PI-CS are equivalent.*

Example 6. Consider the query from Figure 5 expressed in relational algebra as $q = \Pi_{name}(customer \bowtie_{ssn=owner} creditcard) \cup \Pi_{employee}(imports)$. Recall from Example 2 that the PI-CS provenance of tuple t_2 is $\{< u_2, c_2, \perp >, < u_2, c_3, \perp >\}$.

3.4 Algebraic Rewrites

Based on the compositional semantics we developed algebraic rewrite rules that generate the relational representation of PI-CS by propagating provenance tuples through the rewritten query. These rewrite rules are defined for single algebra operators and are applied recursively to rewrite a query. Each rule modifies both the structure of the algebra expression and an auxiliary data structure called the provenance attribute list. The provenance attribute list is the schema for the relational representation of a witness list (attributes storing provenance information). Using single operator rules allows us to support user created provenance information as long as it uses the same provenance representation as Perm and to limit provenance generation to parts of a query (see Example 4).

Definition 5 (Algebraic Rewrite Rules for PI-CS). *Let q be a query. The algebraic rewrite rules for PI-CS shown in Figure 7 transform q into a query q^+ that returns the relational representation of the PI-CS provenance for q. $\mathcal{P}(q^+)$ denotes the list of provenance attributes for query q^+, $P(R)$ is the list of provenance attribute names for relation R, and $=_\epsilon$ is an equality comparison operator that considers null values to be equal.*

Consider the rewrite rules for join (R6) and aggregation (R4) as an example of how these rules work. The rewrite rule for join rewrites the left and right input of the join and applies a projection to the result to achieve the correct

Structural Rewrite

$$q = R: \quad q^+ = \Pi_{\mathbf{R},\mathbf{R}\to P(\mathbf{R})}(R) \tag{R1}$$

$$q = \sigma_C(q_1): \quad q^+ = \sigma_C({q_1}^+) \tag{R2}$$

$$q = \Pi_A(q_1): \quad q^+ = \Pi_{A,\mathcal{P}(q^+)}({q_1}^+) \tag{R3}$$

$$q = \alpha_{G,agg}(q_1): \quad q^+ = \Pi_{G,agg,\mathcal{P}(q^+)}(\alpha_{G,agg}(q_1) \mathbin{⟕}_{G=_\epsilon X} \Pi_{G\to X,\mathcal{P}({q_1}^+)}({q_1}^+)) \tag{R4}$$

$$q = \delta(q_1): \quad q^+ = q_1^+ \tag{R5}$$

$$q = q_1 \bowtie_C q_2: \quad q^+ = \Pi_{\mathbf{Q_1},\mathbf{Q_2},\mathcal{P}(q^+)}({q_1}^+ \bowtie_C {q_2}^+) \tag{R6}$$

$$q = q_1 \mathbin{⟕}_C q_2: \quad q^+ = \Pi_{\mathbf{Q_1},\mathbf{Q_2},\mathcal{P}(q^+)}({q_1}^+ \mathbin{⟕}_C {q_2}^+) \tag{R7}$$

$$q = q_1 \cup q_2: \quad q^+ = ({q_1}^+ \times null(\mathcal{P}({q_2}^+))) \cup (\Pi_{\mathbf{Q_1},\mathcal{P}(q^+)}({q_2}^+ \times null(\mathcal{P}({q_1}^+)))) \tag{R8}$$

$$q = q_1 \cap q_2: \quad q^+ = \Pi_{\mathbf{Q_1},\mathcal{P}(q^+)}(\delta(q_1 \cap q_2) \bowtie_{\mathbf{Q_1}=_\epsilon X} \Pi_{\mathbf{Q_1}\to X,\mathcal{P}({q_1}^+)}({q_1}^+) \bowtie_{\mathbf{Q_1}=_\epsilon Y} \Pi_{\mathbf{Q_2}\to Y,\mathcal{P}({q_2}^+)}({q_2}^+)) \tag{R9}$$

$$q = q_1 - q_2: \quad q^+ = \Pi_{\mathbf{Q_1},\mathcal{P}(q^+)}(\delta(q_1 - q_2) \bowtie_{\mathbf{Q_1}=_\epsilon X} \Pi_{\mathbf{Q_1}\to X,\mathcal{P}({q_1}^+)}({q_1}^+) \times null(\mathcal{P}({q_2}^+))) \tag{R10}$$

Provenance Attribute List Rewrite

$$\mathcal{P}(q^+) = \begin{cases} \mathcal{P}({q_1}^+) & \text{if } q = \sigma_C(q_1) \mid \Pi_A(q_1) \mid \alpha_{G,agg}(q_1) \mid \delta(q_1) \\ P(R) & \text{if } q = R \\ \mathcal{P}({q_1}^+) \blacktriangleright \mathcal{P}({q_2}^+) & \text{else} \end{cases}$$

Fig. 9. *PI-CS* Algebraic Rewrite Rules

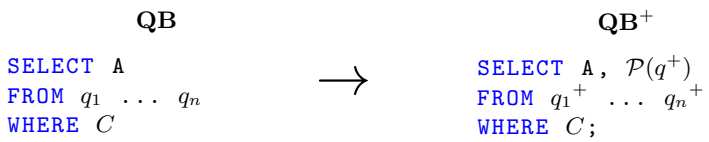

Fig. 10. SQL Query Block Rewrite

ordering between regular result attributes and provenance attributes. The list of provenance attributes for a rewritten join is the concatenation of the provenance attribute lists of its input. The rewrite rule for aggregation joins the original aggregation with the rewritten input on the group-by attributes. As can be seen in Figure 8, the provenance of an output tuple t from an aggregation contains the witness lists for all tuples from the input that have the same group-by attribute values as t, as precisely these tuples were used to compute t. The provenance attribute list for an aggregation is the provenance attribute list of its input. We refer the interested reader to Glavic [16] for detailed descriptions of the these rewrites. We presented a generic rewrite strategy [18] (called the *Gen* strategy) applicable for all types of nested subqueries by generating $\mathcal{W}(q)$ for the nested subquery using a cross product and filtering out tuples that do not belong to the provenance using additional nested subqueries and correlation. The following theorem states the correctness of the rewrite rules for PI-CS.

Theorem 2 (Rewrite Rules Correctness). *Given a query q, the query q^+ derived after Definition 5 generates the PI-CS provenance of q: $Q^+ = Q^{PI}$*

3.5 SQL Rewrites

In a final step, the algebraic rewrites are translated into SQL rewrites. First, we define a canonical translation between SQL queries and relational algebra expressions. We then classify types of SQL query blocks based on the algebra operators used in their translation. Finally, we develop an SQL rewrite rule for each of these block types. A block is translated into a relational algebra expression q, rewritten into expression q^+, and then q^+ is translated back into SQL. The SQL rewrite rule is then inferred from the original and rewritten SQL query.

Example 7 (SQL Rewrite Rules for PI-CS). Consider an SPJ (select-project-join) query block without aggregations as shown on the left of Figure 10. Such a query block is translated into an algebra expression q that is a list of joins followed by a selection and a projection. Applying the algebraic rewrites, then pulling and merging projections, we derive a rewritten expression q^+ which can be translated back into a single query block (shown as QB^+ in Figure 10).

3.6 Relationship with Provenance Polynomials

Recall from the introduction that the provenance polynomials introduced by Green et al. [22] generalize several other provenance semantics for positive relational algebra (USPJ queries). A natural question to ask is how PI-CS is related to this model. In contrast to Why-provenance [7], the PI-CS provenance of a tuple can not be derived from its provenance polynomial. The reason is that the structure of a witness list depends on the structure of the algebra expression q and this structure is not encoded in a provenance polynomial. However, the provenance polynomial of a tuple can be derived from its PI-CS provenance. Note that a polynomial can be written as a sum of products (called monomials). We transform the PI-CS provenance of a tuple t into a provenance polynomial by turning each witness list into a monomial and summing up the monomials for all witness lists of t.

Theorem 3 (Derive Provenance Polynomials from PI-CS). *Let $\mathbb{N}[X](q,t)$ denote the provenance polynomial for a tuple t in the result of a query q derived using the algebra with annotation propagation from Green et al. [22]. There exists a surjective function h from bags of witness lists to provenance polynomials so that for every positive relational algebra expression q and tuple $t \in Q$ the following holds:*

$$h(PI(q,t)) = \mathbb{N}[X](q,t)$$

There exists no function h' such that $h'(\mathbb{N}[X](q,t)) = PI(q,t)$ for every such q and t.

Proof. We construct such a function by deriving a monomial from a witness list w by multiplying all tuples from w (ignoring $\perp$ values) and summing up the monomials for all witness lists of a tuple. The equivalence of $h(PI(q,t))$ to $\mathbb{N}[X]$ can be proven by induction over the structure of an algebra expression.

$$h(PI(q,t)) = \sum_{w^m \in PI(q,t)} \left(\prod_{i \in \{1,n\} \wedge w[i] \neq \perp} w[i] \right)$$

The non-existence of h' is disproven by contradiction (see [19] for a similar proof).

Example 8. Reconsider the query q from Example 6. The result tuple t_2 was derived by joining the customer tuple u_2 with the credit card tuples c_2 and c_3. Thus, the PI-CS provenance of t_2 is $\{< u_2, c_2, \perp >, < u_2, c_3, \perp >\}$. The result of $h(PI(q,t))$ is $u_2 \times c_2 + u_2 \times c_3$, the provenance polynomial for t_2.

As mentioned before, the extension of provenance polynomials for aggregation stores provenance for individual aggregated values. The provenance attached to an attribute value by this model encapsulates both the influence of input tuples and the computation of the aggregation function result. Thus, it is not surprising that this type of provenance can not be derived from the PI-CS provenance. Similarly, some extensions of semiring provenance for set difference are

more informative than PI-CS with regard to this operation [15, 3, 15]. Whereas PI-CS only considers the left input of a set difference to contribute to the result, m-semirings [15] capture the positive influence of the right input in cases such as $q = R - (S - T)$. [7]

3.7 Optimizations

The rewrites implemented in Perm use several optimizations to speed up the execution of provenance queries. For queries without nested subqueries a standard DBMS optimizer will carry out most of the possible optimizations for us, e.g., by pushing down selections over provenance data into the provenance generation. For nested subqueries, the Gen strategy (see Section 3.4) leads to very complex nested subqueries that are hard to de-correlate and un-nest. Such unnesting is necessary to avoid cross-products in the outer query. Therefore, most of the optimizations for PI-CS target this type of query. Glavic and Alonso [18] presented two simple un-nesting strategies to optimize provenance computation for specific types of nested subqueries. The current version of Perm [16] extends this approach and applies a wide range of un-nesting and de-correlation techniques inspired by approaches for optimizing regular nested queries. For instance, we de-correlate correlated aggregation subqueries by using group-by and joins, and inject the outer-query block into a nested subquery to de-correlate universally quantified subqueries (`ALL`) with inequality predicates [28, 13, 32]. New de-correlation strategies can be applied in provenance computation that are not applicable to regular queries. For instance, under certain circumstances a correlated existentially quantified subquery (`EXISTS`) can be rewritten into a join without the need to eliminate duplicates as would be required for regular queries. Rewrite strategies are chosen heuristically, because at the level we apply the query rewrites we do not have access to cost estimates. We always prefer un-nesting and de-correlation techniques to other types of rewrites. This is a reasonable heuristic for provenance computation because avoiding the Gen strategy is almost always beneficial. Experimental results indicate that this heuristic can drastically improve performance [16].

3.8 Query Rewrite Example

We now demonstrate how Perm computes the provenance of the running example query (Figure 2) as specified in Example 4. Recall that the user decided to limit provenance generation to not trace into subquery `monthly`. The result of the SQL rewrites applied by Perm for this query is shown in Figure 11. The `SELECT` clause contains additional attributes to store the relational provenance representation. For simple relation accesses (`customer` and `creditcard` relations in the `FROM` clause of the outer query) these attributes are just renamed versions

[7] In this example, a tuple t from relation T can contribute to a result tuple, because it may cause a tuple s from S to not appear in the result of $(S - T)$ which in turn causes a tuple r from R to be in the result of q.

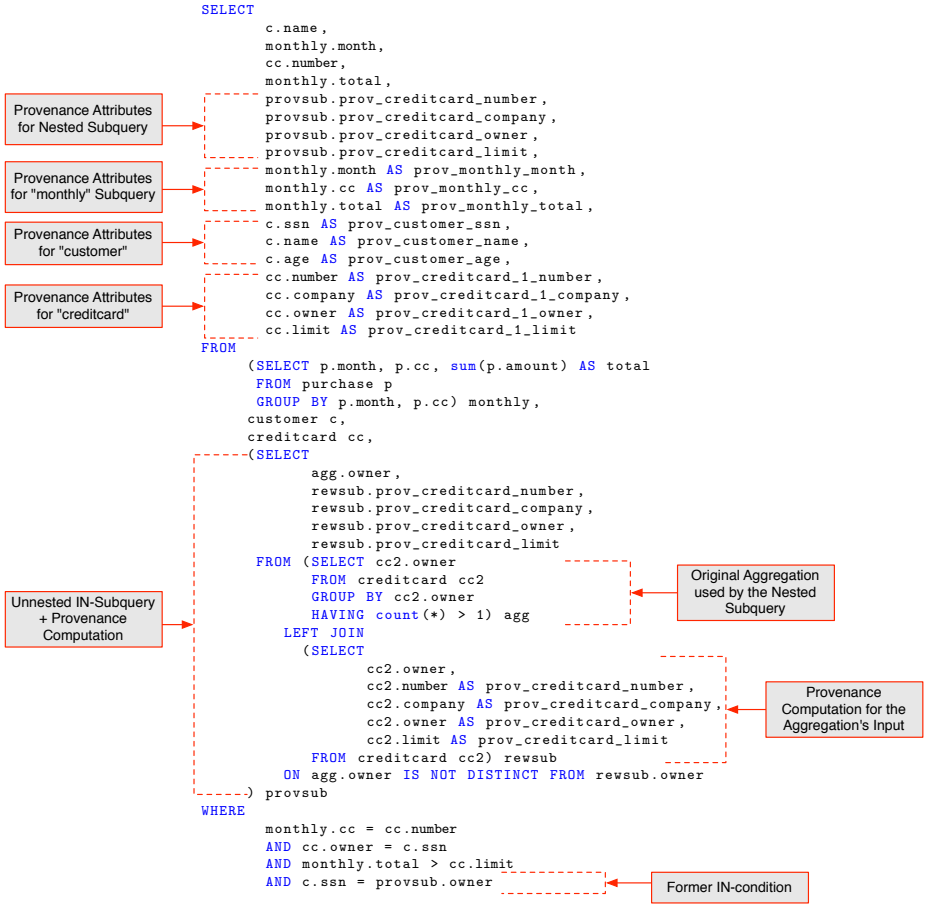

Fig. 11. Rewritten SQL query

of the attributes of these relations (e.g., `c.ssn AS prov_customer_ssn`). The same applies for the monthly subquery, because the user has instructed Perm to limit provenance generation to the results of this subquery using the `BASERELATION` keyword. Recall that the original query used an `IN`-subquery in the `WHERE` clause. This subquery was un-nested by turning it into a `FROM` clause subquery that implements both the selection condition containing the subquery and the provenance computation for this subquery. The `IN` condition has been translated into a simple selection condition (see Figure 11). The provenance computation for this subquery is realized by applying the rewrite rule for aggregation (joining the original aggregation with its rewritten input).

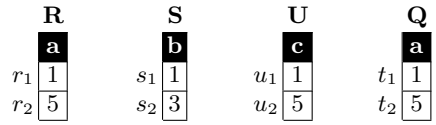

Fig. 12. C-CS Example

4 Copy Contribution Semantics (C-CS)

Copy contribution semantics (C-CS) is a restriction of PI-CS to input tuples that are copied (partially) to a result tuple. This is similar to Where-provenance [7, 5] except that we track copying at tuple granularity instead of attribute value granularity. Perm supports four variants of this provenance semantics based on the distinction of whether to consider equality conditions as an implicit form of copying values and the distinction between partially and completely copied tuples. We limit the discussion to the variant that takes partial and implicit copying into account. For the other variants and correctness theorems see Glavic [16]. Intuitively, it is apparent that the set of input tuples that have been copied to a tuple t is a subset of the tuples that contributed to t. Thus, it is reasonable to derive C-CS from PI-CS by filtering out tuples from the PI-CS witness lists that have not been copied to the output.

Example 9 (PI-CS vs. C-CS). Consider the query $q = \Pi_a(R \bowtie_{a<b} S) \cup U$ evaluated over the database instance shown in Figure 12. The PI-CS provenance of result tuple t_1 is $\{< r_1, s_2, \bot >, <\bot, \bot, u_1 >\}$. The a attribute value of t_1 has been copied from the a attribute of tuple r_1 and the c attribute of tuple u_1. Tuple s_2 was joined with tuple r_1 to produce t_1, but did not contribute any values to the result. Therefore, the C-CS provenance of t_1 is $\{< r_1, \bot, \bot >, <\bot, \bot, u_1 >\}$.

We use data structures called copy-maps to determine which tuples from a PI-CS witness list should be removed to form the corresponding witness list for C-CS. This data structures model from which attributes each result attribute of a query is copied. Formally, a copy-map is a function that maps an algebra expression q, one attribute a from one of its input relations, a result tuple t, and one witness list w in $PI(q,t)$ to the set of result attributes to which a is copied with respect to t and w. Copy-maps are defined recursively for all operators of the algebra in a similar fashion as the compositional semantics for PI-CS. Reconsider tuple t_1 from Example 9 as an example of why it is necessary to include a witness list as an input parameter for copy maps. The two witness lists in $PI(q,t_1)$ exhibit different copy behavior. According to the first witness list, the result attribute a is copied from the a attribute of tuple r_1. According to the second witness list, the result attribute a is copied from tuple u_1.

Definition 6 (C-CS and Copy-Map). *The C-CS provenance $C(q,t)$ of a tuple t from the result of a query q is a multiset of witness lists defined as*

$$
\begin{aligned}
\mathcal{CM}(R, a, w, t) =& \{a\} \\
\mathcal{CM}(\sigma_C(q_1), a, w, t) =& \mathcal{CM}(q_1, a, w, t) \cup \\
& \{x \mid \exists y : (x = y) \in C \wedge t \models (x = y) \wedge y \in \mathcal{CM}(q_1, a, w, t)\} \\
\mathcal{CM}(\Pi_A(q_1), a, w, t) =& \{x \mid (x \in \mathcal{CM}(q_1, a, w, y) \wedge x \in A \wedge y.A = t)\} \\
& \cup \{x \mid (b \to x) \in A \wedge b \in \mathcal{CM}(q_1, a, w, y) \wedge y.A = t)\} \\
& \cup \{x \mid \ if\ (C)\ then\ (x)\ else\ (e) \in A \wedge x \in \mathcal{CM}(q_1, a, w, y) \\
& \wedge y.A = t \wedge y \models C)\} \\
& \cup \{x \mid \ if\ (C)\ then\ (e)\ else\ (x) \in A \wedge x \in \mathcal{CM}(q_1, a, w, y) \\
& \wedge y.A = t \wedge y \not\models C)\} \\
\mathcal{CM}(q_1 \bowtie_C q_2, a, w, t) =& \mathcal{CM}(q_1, a, w[q_1], t.\mathbf{Q_1}) \cup \mathcal{CM}(q_2, a, w[q_2], t.\mathbf{Q_2}) \\
& \cup \{x \mid \exists y : (x = y) \in C \wedge w \models (x = y) \\
& \wedge (y \in \mathcal{CM}(q_1, a, w[q_1], t.\mathbf{Q_1}) \vee y \in \mathcal{CM}(q_2, a, w[q_2], t.\mathbf{Q_2}))\}
\end{aligned}
$$

Fig. 13. Copy-Map Definition

follows (the copy-map $\mathcal{CM}(q, a, w, t)$ is defined in Figure 13).

$$
\begin{aligned}
C(q, t) &= \{\hat{w}^n \mid w^n \in PI(q, t)\} \\
\hat{w}[i] &= \begin{cases} w[i] & \text{if } \exists a \in \mathbf{Q_i} : \mathcal{CM}(q, a, w, t) \neq \emptyset \\ \bot & \text{else} \end{cases}
\end{aligned}
$$

We use $w[q_1]$ to denote the part of a witness list corresponding to subquery q_1.

As an example, consider the copy-map definition for projection. For a tuple t and one of its witness lists w, the value of an attribute a has been copied to a result attribute x if one of the following holds: (1) x is in the copy map of a for a tuple from the input of the projection that has been projected on t (first line); (2) the same applies for an attribute b that has been renamed to x (second line); or (3) the projection contains an if-then-else expression (`CASE` in SQL) with x being the result expression for either the "then" respective "else" branch and the condition is fulfilled respective not fulfilled (third and fourth line).

4.1 Algebraic Rewrites

We use the fact that C-CS is defined as filtering out parts from PI-CS witness lists to develop rewrite rules for this provenance type. We first apply modified versions of the PI-CS rewrite rules to generate a rewritten query q^{C+}. These rules use additional projections expressions, called *copy expressions*, to iteratively build a relational encoding of the copy-map for the query. Afterwards, a final projection is added to the rewritten query to conditionally replace provenance attribute values with null values based on the copy expression information. The relational encoding of a copy map is a list of set valued attributes. Each of these attributes is used to store the result of a copy-map for one input attribute

a ($\mathcal{CM}(q, a, t, w)$). Conditional projection expressions (if-then-else in algebra or `CASE` in SQL) are used whenever the inclusion of an output attribute into the copy-map is conditional. The final projection determines for each input relation if at least one attribute from this relation has been copied to the output (for the current tuple and witness list) using a disjunction of comparisons between copy expressions and the empty-set. If this expression evaluates to false, the provenance attributes for this input relation are replaced with null values.

Definition 7 (C-CS Rewrite Rules). *Let $\mathcal{B}(q)$ denote the list of all attributes from the relations accessed by query q. A query q is rewritten into a provenance generating query q^C according to C-CS as shown below. Query q^C uses projection expressions $\mathcal{P}^*(q^{C+})$ to filter out tuples from witness lists over a rewritten version q^{C+} of q.*

$$
\begin{aligned}
q^C &= \Pi_{\mathbf{Q}, \mathcal{P}^*(q^{C+})}(q^{C+}) \\
\mathcal{P}^*(q^{C+}) &= \blacktriangleright_{a \in \mathcal{B}(q)} \; if\ (\mathcal{C}^+(a))\ then\ (P(a))\ else\ (\varepsilon) \to P(a) \\
\mathcal{C}^+(a) &= (\mathcal{C}(b_1) \neq \emptyset \vee \ldots \vee \mathcal{C}(b_x) \neq \emptyset)\ for\ a \in \mathbf{Q_j} = (b_1, \ldots, b_x)
\end{aligned}
$$

Each of the adapted PI-CS rewrites adds the copy expressions to the rewritten query. We present the rule for projection as an example:

$$
q = \Pi_A(q_1): \qquad q^{C+} = \Pi_{A, \mathcal{P}(q^{C+}), CM(q)}({q_1}^{C+})
$$

The copy expressions $CM(q)$ for a query q are defined in Figure 14.

4.2 Optimizations

Generating the C-CS provenance of a query q requires the generation of copy-expressions in addition to generating the PI-CS provenance of q. However, for a wide range of algebra expressions the query that generates *C-CS* can be simplified based on the following observations. *Instance Independent Copy Expressions*: Often, we can deduce that some conditional clauses used in copy expressions evaluate to a constant result independent of the data. For example, this holds for projections without conditional expressions. We identify and evaluate constant copy expressions at query compile time to avoid unnecessary computations at run-time. *Omit Rewrite*: If the provenance attributes for an input relation are guaranteed to be ε, it is not necessary to compute any provenance for this relation. Thus, we can avoid rewriting a sub-expression if it exclusively accesses input relations with this property.

Example 10 (C-CS Optimizations). Consider the query $q = \Pi_a(R \bowtie_{a<b} S)$ evaluated over the instance from Figure 12. The single attribute of each result tuple of q is copied from the a attribute of a tuple from relation R. Therefore, we can apply the original PI-CS rewrites to relation R and avoid rewriting S at all.

$$
\begin{aligned}
CM(R) &= \underset{a \in \mathbf{R}}{\blacktriangleright} \{a\} \rightarrow \mathcal{C}(a) \\
CM(\sigma_C(q_1)) &= \underset{a \in \mathcal{B}(q_1)}{\blacktriangleright} (\mathcal{C}^*(q_1, a) \cup \mathcal{C}(a)) \rightarrow \mathcal{C}(a) \\
CM(q_1 \bowtie_C q_2) &= \underset{a \in \mathcal{B}(q_1)}{\blacktriangleright} (\mathcal{C}^*(q_1, a) \cup \mathcal{C}(a)) \rightarrow \mathcal{C}(a) \underset{a \in \mathcal{B}(q_2)}{\blacktriangleright} (\mathcal{C}^*(q_2, a) \cup \mathcal{C}(a)) \rightarrow \mathcal{C}(a) \\
\mathcal{C}^*(q, a) &= \bigcup_{x \in \mathbf{Q} \wedge ((x=y) \in C \vee (y=x) \in C)} if\ ((x = y) \wedge x \in \mathcal{C}(a))\ then\ (\{y\})\ else\ (\emptyset) \\
CM(\alpha_{G,agg}(q_1)) &= \underset{a \in \mathcal{B}(q_1)}{\blacktriangleright} (\mathcal{C}(a) \cap G) \rightarrow \mathcal{C}(a) \\
CM(\Pi_A(q_1)) &= \underset{a \in \mathcal{B}(q_1)}{\blacktriangleright} \left(\bigcup_{x \in A} \mathcal{C}^*(a, x) \rightarrow \mathcal{C}(a) \right) \\
\mathcal{C}^*(a, x) &= \begin{cases}
\{x\} \cap \mathcal{C}(a) & \text{for } x \in \mathbf{Q_1} \\
if\ (C)\ then\ (\{y\} \cap \mathcal{C}(a))\ else\ (\emptyset) & \text{for } x = \ if\ (C)\ then\ (y)\ else\ (e) \\
if\ (C)\ then\ (\emptyset)\ else\ (\{y\} \cap \mathcal{C}(a)) & \text{for } x = \ if\ (C)\ then\ (e)\ else\ (y) \\
if\ (y \in \mathcal{C}(a))\ then\ (\{z\})\ else\ (\emptyset) & \text{for } x = (y \rightarrow z) \\
\emptyset & \text{else}
\end{cases}
\end{aligned}
$$

Fig. 14. C-CS Copy Expressions

4.3 SQL Rewrites

The translation of the algebraic C-CS rewrite rules into SQL rewrites is analogous to the translation for PI-CS except for modeling copy expressions and filtering provenance attributes in the outermost projection. An efficient way to model the set-valued attributes used in the copy expressions are bit-arrays (natively supported by PostgreSQL). The copy-expressions for all attributes of one input relation are represented as a single bit-array using n bits (where n is the number of query result attributes) to represent the result set for each attribute. UDFs are used to speed up common operations on the bit-array type.[8]

Example 11 (Example SQL Rewrite). The query shown in Figure 15 removes outlier values (values outside some predefined bound) from a relation R by replacing them with a per-id default value from a relation S. This kind of query is similar to queries used in data cleaning or fusion. The user can request the C-CS provenance of this query to understand from where the values in the result are copied (Figure 15 shows an excerpt of the rewritten query). Consider the expression that determines the value for the provenance attribute `prov_S_c`. A bit-array of length four is constructed to store which of the two result attributes are copied from which of the two attributes of relation S. The construction consists of an outer bitwise-or and inner conditional construction of bit-arrays. For example, if the condition $R.a < 20$ holds, then attribute $S.c$ is copied to the first

[8] For a DBMS without support for a bit-array datatype or UDFs, we could simulate a bit-array as a list of boolean attributes.

```
SELECT PROVENANCE ON CONTRIBUTION (COPY PARTIAL TRANSITIVE)
         CASE
             WHEN r.a < 20 THEN r.a
             ELSE s.c
         END AS cleana,
         CASE
             WHEN r.b < 30 THEN r.b
             ELSE s.c
         END AS cleanb
FROM r NATURAL JOIN s;
```

R

	id	a	b
r_1	1	1	40
r_2	2	51	60

S

	id	c
s_1	1	10
s_2	2	20

Q

	cleana	cleanb
t_1	1	10
t_2	20	20

```
SELECT
       CASE
           WHEN r.a < 20 THEN r.a
           ELSE s.c
       END AS cleana,
       CASE
           WHEN r.b < 30 THEN r.b
           ELSE s.c
       END AS cleanb,
       ...
       CASE
           WHEN NOT biteq(bitor(
             CASE
                 WHEN NOT (r.a < 20) THEN B'0010' ELSE B'0000'
             END,
             CASE
                 WHEN NOT (r.b < 30) THEN B'0001' ELSE B'0000'
             END), B'0000')
           THEN s.c
           ELSE NULL
        END AS prov_s_c
FROM
     r NATURAL JOIN s
```

Fig. 15. Example C-CS SQL Rewrite

result attribute (0010). Similar, if $R.b < 30$ holds, then $S.c$ is copied to the second result tuple (0001). The outer-most `CASE` construct checks whether at least one attribute from relation S has been copied to one of the result attributes, i.e., if the constructed bit-array is not equal to a sequence of zeros (0000).

5 Transformation Provenance

Transformation provenance models what parts of a transformation contribute to a result tuple [19]. We represent the transformation provenance of a query q using annotated algebra trees for q. For a result tuple t and a witness list w in PI(q,t), the transformation provenance includes an annotated algebra tree for q with 1 and 0 annotations on the operators. A 1 indicates this operator on w influences t, a 0 indicates it does not.

Definition 8 (Annotated Algebra Tree). *An annotated algebra tree for a query q is a pair $(Tree_q, \theta)$ where $Tree_q = (V, E)$ is a tree that contains a node for each algebra operator used in q and $\theta : V \to \{0, 1\}$ is a function that associates each operator in the tree with an annotation from $\{0, 1\}$. We define a preorder on the nodes to give each node an identifier (and to order the children of binary operators). Let $I(op)$ denote the identifier assigned to node op.*

We define transformation provenance based on PI-CS provenance. Intuitively, each witness list in the PI-CS provenance of a tuple t represents one evaluation of an algebra expression q. For one witness list, each part of the algebra expression has either contributed to the result of evaluating q on w or not. We represent the transformation provenance as a set of annotated algebra trees of q with one member per witness list w. PI-CS provenance is used to decide whether an operator op in q is annotated with 0 or 1. If evaluating the subtree sub_{op} under op on w results in the empty set ($sub_{op}(w) = \emptyset$), then op has contributed nothing to the result t and should not be included in the transformation provenance.

Definition 9 (Transformation provenance). *The transformation provenance $\mathcal{T}(q, t)$ of a tuple t in the result of a query q is a set of annotated trees defined as:*

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

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

5.1 Algebraic Rewrites

Transformation provenance is defined by evaluating subexpressions of a query over the PI-CS provenance. However, we have shown that it is possible to generate the transformation provenance of a query without instantiating its PI-CS provenance. The rewrite rules for transformation provenance rewrite a query q into a query q^T adding an additional attribute $\mathcal{T}$ to its schema that is used

$$
\begin{array}{lll}
q = R: & q^T = \Pi_{\mathbf{R}, \mathcal{T}(q^T) \to \mathcal{T}}(R) & \mathcal{T}(q^T) = \{R\} \\
q = \sigma_C(q_1): & q^T = \Pi_{\mathbf{Q_1}, \mathcal{T}(q^T) \to \mathcal{T}}(\sigma_C({q_1}^T)) & \mathcal{T}(q^T) = \{\sigma_c(q_1)\} \cup \mathbf{Q_1}.\mathcal{T} \\
q = \Pi_A(q_1): & q^T = \Pi_{A, \mathcal{T}(q^T) \to \mathcal{T}}({q_1}^T) & \mathcal{T}(q^T) = \{\Pi_A(q_1)\} \cup \mathbf{Q_1}.\mathcal{T} \\
q = q_1 \cup q_2: & q^T = \Pi_{\mathbf{Q_1}, \mathcal{T}(q^T) \to \mathcal{T}}({q_1}^T \cup {q_2}^T) & \mathcal{T}(q^T) = \{q_1 \cup q_2\} \cup \mathbf{Q_1}.\mathcal{T}
\end{array}
$$

Fig. 16. Transformation Provenance Rewrite Rules

to store transformation provenance information. Recall that the *transformation* provenance of a result tuple t is a set of annotated algebra trees (one tree per witness list w). The elements of this set represents the same algebra tree with different annotation functions θ_w. Therefore, we can factor out the tree and store only the annotation functions. Each value of attribute $\mathcal{T}$ stores θ_w for one witness-list w of t (represented as the set of operators that carry a 1-annotation).

Each *transformation* provenance rewrite rule computes a new set of annotations from the annotation sets of the rewritten inputs of the operator. Fig. 16 presents the rewrite rules for some algebra operators (see Glavic [16] for the remaining operators). The rewrite rule for a base relation access adds the singleton annotation set for the operator $\{I(R)\}$ as the value for attribute $\mathcal{T}$ to all result tuples. A selection is rewritten by applying the unmodified selection and then adding the identifier of the selection to the annotation set. The rewrite rules for projection and union work analogously.

5.2 SQL Rewrites and Optimizations

We represent an annotation set as a bit-array in the SQL rewrites, because its space requirements are low, and the union operation used frequently in the rewrite rules is efficient (bit-wise disjunction). Similar to C-CS, we can precompute the transformation provenance for a sub-expression if it is independent of the input and avoid rewriting this sub-expression. To provide a useful *transformation* provenance representation to the user the bit-vector representation is transformed into either SQL text with markup or XML (chosen by using the keyword *TRANSSQL* or *TRANSXML* to trigger provenance computation) by applying a UDF f_{SQL} or f_{XML} in the outermost projection of the rewritten query.[9] The SQL representation encloses parts of the original query text with `<NOT>` and `</NOT>` to indicate which parts do not belong to the *transformation* provenance. The XML representation is a hierarchical representation of the query that models each clause as an XML element.

[9] UDFs are used to increase performance. In principle, the `CASE` construct and string concatenation are sufficient for producing these representations.

6 Conclusions

We presented an overview of the Perm approach for integrating efficient on-demand provenance support in relational databases and discussed its contributions with respect to the requirements for a provenance system outlined in Section 1. Perm stands out for using a pure relational representation of provenance information which is generated and queried by executing standard SQL queries, thus, taking full advantage of the DBMS optimizer. We demonstrated the flexibility of the approach by implementing several provenance types including Where-provenance and provenance polynomials. The Perm approach enables a wide range of optimizations such as using algebraic equivalences to develop more efficient rewrites (used to optimize nested subqueries for PI-CS), static analysis of queries to avoid unnecessary generation of provenance information (used for C-CS and transformation provenance), and DBMS specific optimizations using specialized data types (mainly used for C-CS, transformation provenance, and the provenance polynomial implementation). Perm provides a platform for exploring advanced topics such as provenance-aware physical operators, cost-based optimization for provenance generation, provenance compression and summarization, and provenance of updates. In addition to these topics, we plan to extend the approach to support provenance for complete transactions.

References

1. U. Acar, P. Buneman, J. Cheney, J. van den Bussche, N. Kwasnikowska, and S. Vansummeren. A graph model of data and workflow provenance. In *TaPP*, 2010.
2. P. Agrawal, O. Benjelloun, A. Das Sarma, C. Hayworth, S. U. Nabar, T. Sugihara, and J. Widom. Trio: A System for Data, Uncertainty, and Lineage. In *VLDB*, pages 1151–1154, 2006.
3. Y. Amsterdamer, D. Deutch, and V. Tannen. On the Limitations of Provenance for Queries with Difference. In *TaPP*, 2011.
4. Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for Aggregate Queries. *PODS*, pages 153–164, 2011.
5. D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An Annotation Management System for Relational Databases. *VLDB Journal*, 14(4):373–396, 2005.
6. R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Computing Surveys*, 37(1):1–28, 2005.
7. P. Buneman, S. Khanna, and W.-C. Tan. Why and Where: A Characterization of Data Provenance. In *ICDT*, pages 316–330, 2001.
8. J. Cheney. Program Slicing and Data Provenance. *IEEE Data Engineering Bulletin*, 30(4):22–28, 2007.
9. J. Cheney. Causality and the Semantics of Provenance. In *DCM*, pages 63–74, 2010.
10. J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.
11. L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. DBNotes: a Post-it System for Relational Databases based on Provenance. In *SIGMOD*, pages 942–944, 2005.

12. Y. Cui, J. Widom, and J. L. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *TODS*, 25(2):179–227, 2000.
13. U. Dayal. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *VLDB*, pages 197–208, 1987.
14. J. N. Foster, T. J. Green, and V. Tannen. Annotated XML: Queries and Provenance. In *PODS*, pages 271–280, 2008.
15. F. Geerts and A. Poggi. On database query languages for K-relations. *Journal of Applied Logic*, 8(2):173–185, 2010.
16. B. Glavic. *Perm: Efficient Provenance Support for Relational Databases*. PhD thesis, University of Zurich, 2010.
17. B. Glavic and G. Alonso. Perm: Processing Provenance and Data on the same Data Model through Query Rewriting. In *ICDE*, pages 174–185, 2009.
18. B. Glavic and G. Alonso. Provenance for Nested Subqueries. In *EDBT*, pages 982–993, 2009.
19. B. Glavic, G. Alonso, R. J. Miller, and L. M. Haas. TRAMP: Understanding the Behavior of Schema Mappings through Provenance. In *VLDB*, pages 1314–1325, 2010.
20. T. J. Green, Z. G. Ives, and V. Tannen. Reconcilable Differences. In *ICDT*, pages 212–224, 2009.
21. T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update Exchange with Mappings and Provenance. In *VLDB*, pages 675–686, 2007.
22. T. J. Green, G. Karvounarakis, and V. Tannen. Provenance Semirings. In *PODS*, pages 31–40, 2007.
23. T.J. Green. Containment of conjunctive queries on annotated relations. *Theory of Computing Systems*, 49(2):429–459, 2011.
24. G. Karvounarakis and T.J. Green. Semiring-Annotated Data: Queries and Provenance. *SIGMOD Record*, 41(3):5–14, 2012.
25. G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *SIGMOD*, pages 951–962, 2010.
26. A. Kementsietsidis and M. Wang. On the Efficiency of Provenance Queries. In *ICDE*, pages 1223–1226, 2009.
27. A. Kementsietsidis and M. Wang. Provenance Query Evaluation: What's so Special about it? In *CIKM*, pages 681–690, 2009.
28. W. Kim. On Optimizing an SQL-like Nested Query. *TODS*, 7(3):443–469, 1982.
29. E. V. Kostylev and P. Buneman. Combining dependent annotations for relational algebra. In *ICDT*, pages 196–207, 2012.
30. A. Meliou, W. Gatterbauer, K.F. Moore, and D. Suciu. The Complexity of Causality and Responsibility for Query Answers and non-Answers. *PVLDB*, 4(1):34–45, 2010.
31. J. Park, D. Nguyen, and R. Sandhu. A provenance-based access control model. In *PST*, pages 137–144. IEEE, 2012.
32. P. Seshadri, H. Pirahesh, and T.Y.C. Leung. Complex Query Decorrelation. In *ICDE*, pages 450–458, 1996.
33. W.-C. Tan. Containment of Relational Queries with Annotation Propagation. In *DBPL*, pages 37–53, 2003.
34. J. Widom. Trio: A System for Managing Data, Uncertainty, and Lineage. *Managing and Mining Uncertain Data*, pages 113–148, 2008.
35. J. Widom, M. Theobald, and A. Das Sarma. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. In *ICDE*, pages 1023–1032, 2008.