Perm: Processing provenance and data on the same data model through query rewriting

Boris Glavic
Database Technology Research Group
University of Zurich
glavic@ifi.uzh.ch

Gustavo Alonso
Systems Group
Department of Computer Science ETH Zurich
alonso@inf.ethz.ch

***Abstract*— Data provenance is information that describes how a given data item was produced. The provenance includes source and intermediate data as well as the transformations involved in producing the concrete data item. In the context of a relational databases, the source and intermediate data items are relations, tuples and attribute values. The transformations are SQL queries and/or functions on the relational data items. Existing approaches capture provenance information by extending the underlying data model. This has the intrinsic disadvantage that the provenance must be stored and accessed using a different model than the actual data. In this paper, we present an alternative approach that uses query rewriting to annotate result tuples with provenance information. The rewritten query and its result use the same model and can, thus, be queried, stored and optimized using standard relational database techniques. In the paper we formalize the query rewriting procedures, prove their correctness, and evaluate a first implementation of the ideas using PostgreSQL. As the experiments indicate, our approach efficiently provides provenance information inducing only a small overhead on normal operations.**

I. Introduction

A large portion of data generated and stored by scientific databases, data warehouses, and workflow management systems is not entered manually by a user, but is derived from existing data using complex transformations. Understanding the semantics of such data and estimating its quality is not possible without extensive knowledge about the data's origin and the transformations that were used to create it. For these application domains and, in general, for every application domain where data is heavily transformed, data provenance is of essential importance.

Mere storage of provenance information is not very useful, if no query facilities for provenance are provided. Ideally it should be possible to use provenance and normal data in the same query. We use the term provenance management system (PMS) to refer to a system that is capable of computing and storing provenance and supports querying the provenance information [1]. A PMS should also be able to handle provenance imported from external sources (*external provenance*).

In this paper we describe a novel provenance management system called *Perm* (Provenance Extension of the Relational Model) that is capable of computing, storing and querying provenance for relational databases. *Perm* generates provenance by rewriting transformations (queries). For a given query q, *Perm* generates a single query q^+ that produces the same result as q but extended with additional attributes used to store provenance data. An important advantage of the approach used in Perm is that q^+ is also a regular relational algebra statement. Thus, we can use the full expressive power of SQL to, e.g, query the provenance of data items from the result of q, store q^+ as a materialized view, and apply standard query optimization techniques to the execution of q^+. *Perm* can be used both to compute provenance on the fly (i.e., at query time) and to store provenance persistently for future access. *Perm* also supports external provenance and incremental provenance computation reusing stored provenance information.

An important contribution of *Perm* is that, although not yet complete, it already covers a far wider range of relational algebra than existing systems. As shown in our experiments, *Perm* can run almost all of the queries of the TPC-H benchmark and efficiently compute their provenance. This is a significant improvement over existing systems, which typically can only run a few, if any at all, of the queries in the benchmark. Our experiments also demonstrate that *Perm* can efficiently compute the provenance of complex queries with only a minimal overhead on normal operations. As an example, on the few queries that are also supported by the *Trio* system, *Perm* outperforms *Trio* at least by a factor of 30.

The paper is organized as follows. In the next section we presented related work. Section III covers the formal foundations of the *Perm* approach. The prototype implementation of *Perm* is presented in section IV. An experimental evaluation of the prototype is discussed in section V. We conclude in section VI.

II. Related Work

Two types of information are generally considered to form the provenance of a data item. Information about the *source* data items that contributed to a result and the *transformations* used in the creation process of the data item. These types where first introduced in [2] as *provenance of data* and *provenance of a data product*.

Source provenance can be distinguished by the definition of *contribution* used to estimate if an input data item influenced a result data item. Buneman et al.[3] describe two definitions of contribution semantics: *Why-* and *Where-provenance*. Why-Provenance includes all data items that influence the creation of a data item. Where-Provenance includes only parts of the input literally copied to the output. In practice, these definitions are only two possibilities from a broader solution

space. For example, some systems consider the whole input of a transformation as the provenance of its output. To have a consistent terminology, we refer to these definitions as *input-, influence-* (for Why-provenance) and *copy-contribution* (for Where-provenance). There are basically two approaches to compute a mapping between sources and results of a transformation. One approach is to compute some kind of inverse of the transformation that maps a result (or a set of results) to the sources contributing to it. The other approach generates the mapping by propagating identifying annotations from the source to the result during the execution of the transformation.

Source provenance can be examined at different levels of granularity. A composite data item can be seen as a single data item or each of its components examined separately. For example, for the relational model, the provenance of data items could be at the granularity of relations, tuples or attribute values.

Some approaches compute provenance at the time new data is derived by a transformation, other approaches reconstruct provenance when is is requested by a user. In [2], Tan referred to these strategies as *eager* and *lazy*. A provenance management system that supports both strategies leaves this choice to the user.

In the area of data warehouses, Cui et al.[4] studied the problem of finding the portions of the input of a relational query that influenced a certain part of the its output (*influence-contribution*). This is achieved by generating inverse queries for an original query. The result of a provenance computation is a list of subsets of the base relations accessed by the original query that contributed to a specific subset of the result. A disadvantage of this approach is that the result of a provenance computation is a list of relations and the computation is not expressed in relational algebra. Hence, provenance queries cannot be used as subqueries of normal relational queries and only partially benefit from the query optimization of the underlying DBMS. Depending on the type of algebraic operators that are used in a query, the provenance computation may also require the instantiation of intermediate results.

Trio is a system extending the relational model with provenance and uncertainty [5]. *Trio* computes the provenance of a query during execution and stores the results in so-called lineage relations (*eager* provenance computation). To trace the provenance of a tuple produced by a sequence of transformations the system iteratively computes the contributing input tuples of each transformation using the lineage relations. To our knowledge, *Trio* currently supports only a small subset of features present in SQL (i.e., it does support neither aggregation nor subqueries, and supports only single set operations and a limited set of datatypes).

DBNotes [6] extends the relational model with attribute level annotations. Every attribute value can be annotated with a set of textual annotations. The *pSQL* language (a extension of a subset of SQL) allows to specify how annotations are propagated from the source to the results of a query. Two standard propagation schemes are defined from which one is independent under query rewrite. Base relations are extended with additional annotation attributes. Each annotation attribute stores the annotations of one of the original attributes. *pSQL* currently does not support aggregation, disjunctive queries and set operations.

Theoretical aspects of provenance have been studied by Buneman, Tan and Cheney in [7], [8]. Newer approaches use the nested relational algebra and calculus [9] instead of the simpler hierarchical language used in [10]. In [11], [12] the relationships between provenance and program dependency analysis are studied. While stating interesting theoretical results, their work has not yet been implemented so it is not possible to ascertain its impact on practical systems.

Provenance methods developed in the context of workflow systems and grid computing such as [13], [14] and [15] are more focused on transformation provenance and only analyze *input-contribution* source provenance. In general, these systems provide more extensive transformation provenance and some allow re-execution of transformations in case of changed source data.

The aforementioned systems represent provenance information in a data model that is either an extension or completely different from the data model for which the provenance is computed. This has the disadvantage that provenance data cannot be stored directly in the data repositories or queried using the query language used for the normal data. The *DBNotes* and *Trio* systems provide provenance computation and query facilities only for a small subset of SQL. We take a different approach by using solely relational algebra for provenance computation and representing provenance in the relational data model. At first sight this seems to be a strong restriction, but in fact allows us to achieve a broad range of functionality by reusing most of the query, storage, and optimization methods provided by relational databases. The *Perm* prototype supports provenance computation for the complete SQL language except correlated subqueries. For normal queries and queries on provenance data we support the complete SQL functionality implemented by PostgreSQL.

III. The Perm Approach

In this section we present the underlying algebra of the *Perm* approach, introduce a pure relational provenance representation, and demonstrate how provenance is computed using relational algebra exclusively. *Perm* uses the *influence-contribution*-semantics of Cui et al. [4], because it better captures user expectations than *copy-contibution*-semantics. For example, for a query computing the sum of an attribute for all tuples of an input relation the *influence-contribution* source consists of all input tuples, because each tuple *contributed* to the result of the sum. This seems to be more intuitive than none of the inputs which would be the result for *copy-contribution* semantics. We consider queries expressed in the extended version of relational algebra presented below. How the rewrite mechanism developed for this algebra can be applied to SQL statements is covered in section IV.

(1a) Set operators

$$\overset{S}{\Pi}_A(T) = \{a = (a_1, \ldots, a_n)^1 \mid \exists t \in T, t.A = a\}$$
$$T_1 \overset{S}{\cup} T_2 = \{t^1 \mid t \in T_1 \vee t \in T_2\}$$
$$T_1 \overset{S}{\cap} T_2 = \{t^1 \mid t \in T_1 \wedge t \in T_2\}$$
$$T_1 \overset{S}{-} T_2 = \{t^1 \mid t \in T_1 \wedge t \notin T_2\}$$

(1b) Bag operators

$$\overset{B}{\Pi}_A(T) = \{a = (a_1, \ldots, a_n)^{sum} \mid sum = \sum_{t^n \in T, t.A = a} (n)\}$$
$$T_1 \overset{B}{\cup} T_2 = \{t^{n+m} \mid t^n \in T_1 \wedge t^m \in T_2\}$$
$$T_1 \overset{B}{\cap} T_2 = \{t^{min(n,m)} \mid t^n \in T_1 \wedge t^m \in T_2\}$$
$$T_1 \overset{B}{-} T_2 = \{t^{n-m} \mid t^n \in T_1 \wedge t^m \in T_2\}$$

(1c) Set/Bag operators

$$\sigma_C(T) = \{t^n \mid t^n \in T \wedge t \models C\}$$
$$T_1 \times T_2 = \{(t_1, t_2)^{n \times m} \mid {t_1}^n \in T_1 \wedge {t_2}^m \in T_2\}$$
$$T_1 \bowtie_C T_2 = \{t = (t_1, t_2)^{n \times m} \mid {t_1}^n \in T_1 \wedge {t_2}^m \in T_2 \wedge t \models C\}$$
$$T_1 \overset{+}{\bowtie}_C T_2 = \{(t_1, t_2)^{n \times m} \mid {t_1}^n \in T_1 \wedge {t_2}^m \in T_2\} \cup \{(t_1, t_2^{null})^n \mid {t_1}^n \in T_1 \wedge (\nexists t_2 \in T_2 : (t_1, t_2) \models C)\}$$
$$\alpha_{G,aggr}(T) = \{(t.G, res_1, \ldots, res_n)^1 \mid t \in T \wedge \forall i \in \{1, n\} : res_i = aggr_i(\Pi_{B_i}(\sigma_{G = t.G}(T))\}$$

Fig. 1: The *Perm* algebra

A. Underlying Algebra

We have developed the *Perm* approach for the variant of relational algebra depicted in Figure 1. The algebra operates on bag-semantic relations. For a tuple t^n, n is the multiplicity of t. If no multiplicity is given, an arbitrary multiplicity is assumed. For convenience, a multiplicity of 0 or below is an alternative representation for a tuple that is not included in a relation. $t \in R$ is a shortcut for $t^n \in R$ with $n > 0$. We use R to refer to a base relation and T to refer to a base relation or the result of an algebra statement. Sometimes we abuse this notation and use T for result relations and algebra expressions interchangeably. $\mathbf{R}$ is the schema of a relation R.

Some of the algebra operators are provided as a set- and a bag-semantic version denoted by S or B (Figures 1a and 1b). Projection Π_A projects the attributes from T on A and is available as a set- and a bag-semantics version. A is a list of attributes, constants, attribute renamings and deterministic functions on one or more attributes or constants. Renaming attribute a in attribute b is presented as $a \rightarrow b$. The selection operator's condition C can contain boolean operations, comparison operators, attribute and constant values and functions on values, but is restricted to yield a boolean result. Crossproduct $\times$ and inner join $\bowtie_C$ are defined as in standard relational algebra. The left outer join operation $\overset{+}{\bowtie}_C$ is a union from the tuples of $\bowtie_C$ and all tuples from T_1 that did not have a join partner in T_2 (Figure 1c). t_2^{null}, used in the definition of the left outer join, is a single tuple relation with schema $\mathbf{T_2}$ and all attribute values set to $null$. Right outer join and full outer join are defined analogously and are therefore not included in the figure. The aggregation operator $\alpha_{agg,G}$ allows to specify a list of grouping attributes G on which the input tuples are grouped before the aggregation functions $agg_1, \ldots, agg_n$ from list agg are applied (Figure 1c). An aggregation function computes a single result value for a set of tuples with schema B. The result of the aggregation operator includes the results of the aggregation functions and the grouping attribute values. Set operations (union, intersection and difference) operate on union-compatible inputs T_1 and T_2 and are available as set- and bag-semantics versions. The result schema of a set operation is $\mathbf{T_1}$.

For convenience, renaming all attributes from a list of attributes A into B and applying a comparison operator $comp$ to all attributes from lists A and B is presented as $A \rightarrow B$ and $A\ comp\ B$. For a tuple t, $t.A$ is the projection of t on a list A of attributes, functions and renamings.

B. Perm Provenance Representation

Before we introduce the relational provenance representation used by *Perm*, we demonstrate the disadvantages of non-relational provenance representations by a motivating example (Figure 2). The example is a database of shops (with name and number of employees), items they are selling, and purchases (sales relation). Assume a user wants to query the total profits for each shop. This query can be expressed in relational algebra as:

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

If the user wants to investigate which source tuples contributed to the result tuple $(Merdies, 120)$ using *influence-contribution* semantics (Why-provenance) the result should include tuple $(Merdies, 3)$ from relation $shop$, tuples $(Merdies, 1), (Merdies, 2)$ and $(Merdies, 2)$ from relation $sales$ and tuples $(1, 100), (2, 10)$ from relation $items$. The approach from [4] would present this provenance data as the

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

result q_{ex}

name	sum(price)
Merdies	120
Joba	50

Fig. 2: Example database

following list of relations: [1]

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

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.

These shortcomings can be avoided if provenance is represented as a single result relation including complete provenance and normal data. We represent provenance information as a single relation by extending the original result tuples with contributing tuples from each base relation accessed by the original query. If there is more than one contributing tuple from a base relation, the original tuple has to be duplicated. For example, using this representation the provenance result for the example above would be:

$$\{(Merdies, 120, Merdies, 3, Merdies, 1, 1, 100),\\ (Merdies, 120, Merdies, 3, Merdies, 2, 2, 10),\\ (Merdies, 120, Merdies, 3, Merdies, 2, 2, 10)\}$$

It is obvious that this representation is only useful if descriptive names for the attributes are provided to preserve the information regarding which base relation and attribute are used to derive a value. For simplicity, we ignore this issue here and assume we have a appropriate naming scheme for provenance attributes. A concrete solution to this problem is given in the next section. In the following provenance attributes are represented as $\mathcal{P}(a)$ where a is an attribute from a base relation.

Let us now consider the provenance result format for an arbitrary algebra statement q. To produce a provenance result relation, the original result relation is extended with all attributes from all base relations accessed by q. Multiple references to a base relation are handled as separate relations. A result tuple of a provenance query is built by attaching contributing tuples to an original result tuple. Hence the original tuple has to be duplicated, if there is more than one contributing tuple from one of the base relations. The result schema of the rewritten query contains all attributes of the original query and in addition a set of provenance attributes for each base relation accessed by the original query. For each base relation we duplicate its attributes creating unique names for the duplicated attributes [2]. For example query q_{ex} accesses base relations $shop$, $sales$ and $items$. In consequence the provenance result schema for q_{ex} is:

$$(name, sum(price), \mathcal{P}(name), \mathcal{P}(numEmpl),\\ \mathcal{P}(sName), \mathcal{P}(itemId), \mathcal{P}(id), \mathcal{P}(price))$$

Alternatively we could produce two result relations. One with the original result and a second with provenance information. This representation omits the redundancy of duplicating result tuples, but has the same disadvantages as the representation from [4].

C. Provenance Query Rewrite Mechanism

Having presented the structure of a provenance query result and the algebra for which rewrites should be produced, we now present how a query is transformed by the *Perm* approach into a query that generates the desired provenance result schema and propagates provenance according to *influence-contribution*-semantics. *Perm* propagates provenance from the source to the result of a query. Recall that the approach from [4], based on inversion of queries, requires the instantiation of intermediate results, because there is no direct association between the original data and intermediate results. The *Perm* approach omits the instantiation of intermediate results. Each provenance propagation step depends exclusively on the result of its direct predecessor and is independent from previous stages. Thus we do not have to keep earlier results to compute the current step.

The *Perm* method transforms a query q into a provenance query q^+ by iteratively rewriting each relational operator of q. The rewritten form of an operator preserves the result of the original operator, but adds additional provenance attributes and propagates provenance data according to *influence-contribution*-semantics. The rewrite rules for each algebra operator are presented in Figure 3. To be able to rewrite a query incrementally, the rewrite rules need to support rewritten inputs, i.e., a rewrite has to distinguish between normal and provenance attributes. For each rewrite rule, $\mathcal{P}$ is the list of provenance attributes that are attached to the original result. For two $\mathcal{P}$-lists, $\mathcal{P}_1 = (p_1, \ldots, p_n)$ and $\mathcal{P}_2 = (q_1, \ldots, q_n)$, the list concatenation operation $\blacktriangleright$ is defined as $\mathcal{P}_1 \blacktriangleright \mathcal{P}_2 = (p_1, \ldots, p_n, q_1, \ldots, q_n)$.

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

[2] By unique we mean unique in the scope of q

R1	$R^+ = \Pi_{\mathbf{R},\mathcal{R}}(R)$ with $\mathcal{P}(R^+) = \mathcal{R}$ where $\mathcal{R}$ is a unique renaming of the attributes from $\mathbf{R}$
R2	$(\overset{S/B}{\Pi}_A(T))^+ = \overset{S/B}{\Pi}_{A,\mathcal{P}(T^+)}(T^+)$ with $\mathcal{P}((\overset{S/B}{\Pi}_A(T))^+) = \mathcal{P}(T^+)$
R3	$(\sigma_C(T))^+ = \sigma_C(T^+)$ with $\mathcal{P}((\sigma_C(T))^+) = \mathcal{P}(T^+)$
R4	$(T_1 \times T_2)^+ = {T_1}^+ \times {T_2}^+$ with $\mathcal{P}((T_1 \times T_2)^+) = \mathcal{P}({T_1}^+) \blacktriangleright \mathcal{P}({T_2}^+)$
R5	$(\alpha_{G,aggr}(T))^+ = \Pi_{G,aggr,\mathcal{P}(T^+)}(\alpha_{G,aggr}(T) \bowtie_{G=\hat{G}} \Pi_{G\to\hat{G},\mathcal{P}(T^+)}(T^+))$ with $\mathcal{P}((\alpha_{G,aggr}(T))^+) = \mathcal{P}(T^+)$
R6	$(T_1 \overset{S/B}{\cup} T_2)^+ = \Pi_{\mathbf{T_1},\mathcal{P}({T_1}^+),\mathcal{P}({T_2}^+)}(T_1 \overset{S/B}{\cup} T_2 \rtimes_{\mathbf{T_1}=\hat{T_1}} \Pi_{\mathbf{T_1}\to\hat{T_1},\mathcal{P}({T_1}^+)}({T_1}^+) \rtimes_{\mathbf{T_1}=\hat{T_2}} \Pi_{\mathbf{T_2}\to\hat{T_2},\mathcal{P}({T_2}^+)}({T_2}^+))$ with $\mathcal{P}((T_1 \overset{S/B}{\cup} T_2)^+) = \mathcal{P}({T_1}^+) \blacktriangleright \mathcal{P}({T_2}^+)$
R7	$(T_1 \overset{S/B}{\cap} T_2)^+ = \Pi_{\mathbf{T_1},\mathcal{P}({T_1}^+),\mathcal{P}({T_2}^+)}(T_1 \overset{S/B}{\cap} T_2 \bowtie_{\mathbf{T_1}=\hat{T_1}} \Pi_{\mathbf{T_1}\to\hat{T_1},\mathcal{P}({T_1}^+)}({T_1}^+) \bowtie_{\mathbf{T_1}=\hat{T_2}} \Pi_{\mathbf{T_2}\to\hat{T_2},\mathcal{P}({T_2}^+)}({T_2}^+))$ with $\mathcal{P}((T_1 \overset{S/B}{\cap} T_2)^+) = \mathcal{P}({T_1}^+) \blacktriangleright \mathcal{P}({T_2}^+)$
R8	$(T_1 \overset{S}{-} T_2)^+ = \Pi_{\mathbf{T_1},\mathcal{P}({T_1}^+),\mathcal{P}({T_2}^+)}(T_1 \overset{S}{-} T_2 \bowtie_{\mathbf{T_1}=\hat{T_1}} \Pi_{\mathbf{T_1}\to\hat{T_1},\mathcal{P}({T_1}^+)}({T_1}^+)) \rtimes {T_2}^+)$ with $\mathcal{P}((T_1 \overset{S}{-} T_2)^+) = \mathcal{P}({T_1}^+) \blacktriangleright \mathcal{P}({T_2}^+)$
R9	$(T_1 \overset{B}{-} T_2)^+ = \Pi_{\mathbf{T_1},\mathcal{P}({T_1}^+),\mathcal{P}({T_2}^+)}(T_1 \overset{B}{-} T_2 \bowtie_{\mathbf{T_1}=\hat{T_1}} \Pi_{\mathbf{T_1}\to\hat{T_1},\mathcal{P}({T_1}^+)}({T_1}^+) \rtimes_{\mathbf{T_1}\neq\mathbf{T_2}} {T_2}^+)$ with $\mathcal{P}((T_1 \overset{B}{-} T_2)^+) = \mathcal{P}({T_1}^+) \blacktriangleright \mathcal{P}({T_2}^+)$

Fig. 3: *Perm* provenance rewrite rules

We now discuss each rewrite rule in detail. The rewrite rule **R1** for base relations duplicates the attributes from a base relation R and renames them according to a provenance attribute naming scheme.

The rule **R2** for $\Pi_A(T)$ rewrites a projection by adding the list of provenance attributes from T^+ to the projection list. For example, if T is the base relation $items$, $\mathcal{P}(T^+)$ is $(\mathcal{P}(id), \mathcal{P}(price))$. So, $(\Pi_A(item))^+$ preserves the complete tuples that were used to compute the result of $\Pi_A(item)$.

For a selection operator (rule **R3**), the unmodified selection is applied to its rewritten input, because a selection only filters out tuples but does not change or add tuples.

The crossproduct operator concatenates input tuples from T_1 and T_2. Adding provenance attributes to the input relations T_1 and T_2 does not change the original part of the result. Hence rule **R4** rewrites a crossproduct as the crossproduct of the rewritten inputs T_1 and T_2. Notice that we do not provide rewrite rules for other join types than cross product, because rewrite rules for inner join, natural join and outer joins can be defined using algebraic equivalents for these join types and the given set of rewriting rules. The rewrite rules that are produced by these equivalence rules are of the form (where $\diamond$ is one of the defined join types):

$$(T_1 \diamond T_2)^+ = {T_1}^+ \diamond {T_2}^+$$

The rewrite rule **R5** 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. So the aggregation is applied to the original input T. The results of the aggregation are joined with the rewritten version of T using a equality condition on the grouping attributes. This is feasible because, according to the definition of aggregation, all tuples with the same grouping attribute values $\vec{g}$ contribute to a single result tuple with grouping attribute values $\vec{g}$.

For set operations the result of the operation would change if we add attributes or tuples to its input relations. In addition if $\mathbf{T_1}^+ \neq \mathbf{T_2}^+$ holds the set operations can not be applied to ${T_1}^+$ and ${T_2}^+$. Thus for set operations, the original query must be conserved. Provenance is attached by joining the original query with its rewritten inputs. Rewrite rules **R6** and **R7** for union and intersection use joins on $\mathbf{T_1}$ and $\mathbf{T_2}$ because, according to the definition of these operations, a tuple t from T_1 or T_2 contributes to a result tuple t' if their attribute values are equal. For union, the left outer join operation is used because some tuples might be only present in T_1 or T_2

For set-difference, the provenance of tuple t includes all tuples from T_2 that are different from t. The rewrite rules **R8** and **R9** use the left join operation on $\mathbf{T_1} \neq \mathbf{T_2}$ to attach these tuples to the result. For the set-semantics version of set-difference, the condition can be omited because tuples fulfilling this condition can not occur in the result of this operation.

D. Example Query Rewrite

As an example, query q_{ex} from the example in the previous section can be rewritten as shown in Figure 4. The top level operation of q_{ex} is an aggregation operator. Applying rewrite rule $R5$ we get ${q_{ex}}^+$ as in 4 (**step 1**). Rule $R5$ states that the $\mathcal{P}$-list for ${q_{ex}}^+$ equals $\mathcal{P}(T^+)$. At this point, T^+ has not been computed so $\mathcal{P}({q_{ex}}^+)$ is left undefined for now. The remaining subquery T is a selection, which is left untouched by the rewrite. The cross-product $shop \times sales \times item$ is handled by rewrite rule $R4$ (see 4 (**step 2**)). The $\mathcal{P}$-list of a rewritten cross-product is the concatenation of the $\mathcal{P}$-lists of the subqueries

original query	$q_{ex} = \alpha_{name,sum(price)}(\sigma_{name=sName \wedge itemid=id}(shop \times sales \times items))$
step 1	${q_{ex}}^{+} = \Pi_{name,sum(price),\mathcal{P}(T^{+})}(\alpha_{name,sum(price)}(T) \bowtie_{name=na\hat{m}e} \Pi_{name \to na\hat{m}e,\mathcal{P}(T^{+})}(T^{+}))$ $T = \sigma_{name=sName \wedge itemid=id}(shop \times sales \times items)$ $\mathcal{P}({q_{ex}}^{+}) = \mathcal{P}(T^{+})$
step 2	$T^{+} = \sigma_{name=sName \wedge itemid=id}(shop^{+} \times sales^{+} \times items^{+})$ $\mathcal{P}(T^{+}) = \mathcal{P}(shop^{+}) \blacktriangleright \mathcal{P}(sales^{+}) \blacktriangleright \mathcal{P}(items^{+})$
step 3	$shop^{+} = \Pi_{name,numEmpl,name \to pName,numEmpl \to pNumEmpl}(shop)$ $\mathcal{P}(shop^{+}) = (pName, pNumEmpl)$
step 4	$sales^{+} = \Pi_{sName,itemId,sName \to pSName,itemId \to pItemid}(sales)$ $\mathcal{P}(sales^{+}) = (pSName, pItemId)$
step 5	$items^{+} = \Pi_{id,price,id \to pId,price \to pPrice}(items)$ $\mathcal{P}(items^{+}) = (pId, pPrice)$

$result\ relation\ {q_{ex}}^{+}$

name	sum(price)	pName	pNumEmpl	pSName	pItemId	pId	pPrice
Merdies	120	Merdies	3	Merdies	1	1	100
Merdies	120	Merdies	3	Merdies	2	2	10
Merdies	120	Merdies	3	Merdies	2	2	10
Joba	50	Joba	14	Joba	3	3	25
Joba	50	Joba	14	Joba	3	3	25

Fig. 4: Example application of rewrite rules

used in the cross-product. In this case, the provenance attribute lists of rewritten base relations $shop$, $sales$ and $items$. In Figure 4 (**steps 3-5**) rewrite rule $R1$ is used to derive the rewritten base relations $shop^{+}$, $sales^{+}$ and $items^{+}$.

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 rewrite top-down and computed the $\mathcal{P}$-lists in a second bottom-up tree-traversal. 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 $\mathcal{P}$-lists of sub-expressions of a query q needed to compute q's $\mathcal{P}$-list are immediately available, but as we will see in section IV, the bottom-up approach has other disadvantages.

Using the set of rewrite rules, we are able to transform a query q into a *single* relational algebra query q^{+} propagating provenance. A major advantage of our provenance representation format is that provenance is presented as complete base relation tuples and is directly associated with the original 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, i.e., we can use provenance and original attributes in conditions and projections.

E. Correctness Proof

It can be shown that the *influence-contribution*-semantics defined by the *Perm* rewrite rules is equivalent to the semantics introduced in [4]. The only difference is that we might loose the information of the multiplicity of original result tuples for some operations. Due to space constraints we present only a proof sketch. The equivalence is proven by first showing that the original attribute part of a rewritten query's result tuples is the same as the original result tuples except for multiplicity:

$$\overset{S}{\Pi}_{\mathbf{T}}(T^{+}) = \overset{S}{\Pi}_{\mathbf{T}}(T)$$

This can be proven by induction over the set of algebra operators. Having shown that we do not create spurious result tuples or omit original result tuples, we prove in a second step that the provenance attached to the original result tuples represent exactly the provenance produced by Cui's approach [4]. This step uses a mapping between our provenance representation and the representation presented in [4] and induction over the set of algebra operators.

IV. Implementation

In this section we present the prototype implementation of the *Perm* approach. Before being able to implement a *Perm*-prototype, we had to define a SQL language extension for provenance and define a mechanism for provenance attribute handling.

A. SQL Provenance Language Extension

1) Provenance Attribute Handling: To enable a user to query provenance data, a mechanism for addressing provenance attributes in a query is needed. Either a new language construct has to be introduced or provenance and normal attributes are distinguished by name. We decided to reserve a distinct set of names for provenance attributes. A provenance attribute name consists of the fixed prefix $prov\_$, the name of the base relation, the attribute is derived from, and the original attribute name. Each part of a provenance attribute name is separated by an underline character. If a relation is referenced more than once in a query, an identifying number is attached to the relation name. For example, the names of the provenance attributes from the example query q of the last section are *prov_shop_name, prov_shop_numEmpl, prov_sales_sName, prov_sales_itemId, prov_items_id* and *prov_items_price*. To keep the following examples concise, the praefix p is used as a shortcut for provenance attribute names (e.g. $pName$ instead of $prov\_shop\_name$).

2) Provenance Extension: A simple way to extend SQL with provenance rewrites is to add an optional keyword PROVENANCE to the SQL select-clause. A query or subquery is rewritten, if the PROVENANCE keyword is present in the select-clause. For example, the query q_1 from the example of the last section is expressed in SQL with the provenance language extension (SQL-PLE) as following:

```
SELECT pId
  FROM
    (SELECT PROVENANCE name, sum(price) AS sum
    FROM shop, sales, items
    WHERE name=sName AND itemId=id
    GROUP BY name) AS prov
  WHERE sum > 100;
```

3) Incremental Provenance Computation: To support external provenance and incremental provenance computation a user can define that a subset of a from-clause item's attributes are provenance attributes by appending PROVENANCE $(attrlist)$ to the text of the from-clause item. The *Perm* module is instructed by the PROVENANCE clause to stop rewriting when processing the subquery and to accept it as an already rewritten subquery. This is especially useful to compute provenance incrementally starting at a stored result of a provenance query.

For example, assume a view $totalItemPrice$ is used to store provenance data. If $totalItemPrice$ is used in a provenance query the provenance stored in the pId and $pPrice$ attributes of this view should be used in the provenance computation:

```
CREATE VIEW totalItemPrice AS
  SELECT PROVENANCE sum(price) AS total
    FROM items;

SELECT PROVENANCE total * 10
  FROM totalItemPrice PROVENANCE (pId, pPrice);
```

4) Limited Provenance Scope: A from-clause item is marked to be handled as a base relation by specifying the key word BASERELATION. The BASERELATION keyword should be used if a user does not want to trace provenance down to the base relations, but is interested in the influence a view or subquery had on the query results. In the following example the rewriter would use rewrite rule $R1$ for subquery sub instead of applying rule $R5$.

```
SELECT PROVENANCE total * 10 FROM
 (SELECT sum(price) AS total
  FROM items) BASERELATION AS sub;
```

B. Extensions to PostgreSQL

We have implemented a first prototype of *Perm* by extending the PostgreSQL DBMS [16]. In the system, the *Perm* module is located below the postgres rewriter (see Figure 5). Our module does not need to care about semantic checks and view unfolding, because this is done by the analyzer and rewriter modules. The output of the provenance rewriter is passed to the optimizer, thus we benefit from the full query optimization of PostgreSQL. Small changes had to be made to the parser and analyzer module for recognition of the provenance language extensions and handling of references to provenance attributes in select-, where-, group-by- and having-clauses.

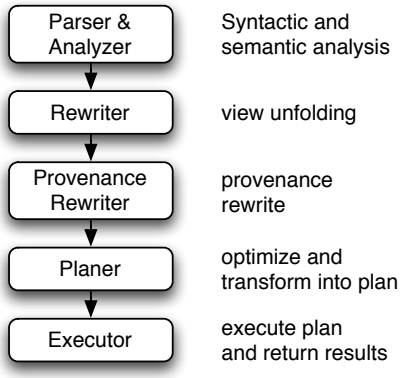

Fig. 5: *Perm* architecture overview

To integrate the rewrite mechanism presented in the last section in PostgreSQL, the rewrite rules have to be adapted for SQL or the internal algebraic operator tree representation of the database system. In most database systems, including PostgreSQL, the result of the SQL-parser is a so-called query tree. Each query node in the query tree represents one or more relational algebra operators. The main components of a query node are the target list, the range table and the set operation tree. The target list is a list of expressions on attributes and constants that define the schema of the result relation of the query. The range table contains references to base relations and query nodes of subqueries. The set operation tree, used only in set operation queries, consists of set operations and references to range table entries that are accessed by these set operations. For PostgreSQL, each node in the query tree

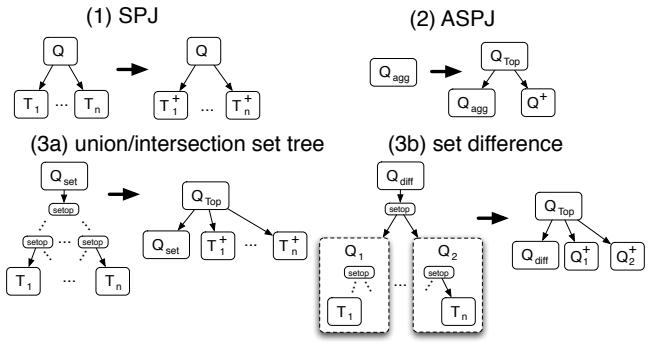

Fig. 6: Query tree rewrite

is either an aggregate-select-project-join (ASPJ) segment or a set-operation segment. If we distinguish between ASPJ nodes with and without aggregation there are three different cases:

1) select-project-join (SPJ) query
2) aggregate-select-project-join (ASPJ) query
3) set operations

We distinguish between these cases because for each case a rewritten subtree can be computed according to the *Perm* rewrite rules.

1) SPJ: In relational algebra case 1 is equivalent to the query $q = \Pi_A(\sigma_C(T_1 \diamond \ldots \diamond T_n))$ where $\diamond$ is one of the join operations ($\times, \bowtie, \rightarrow\!\!\bowtie, \bowtie\!\!\leftarrow, \rightarrow\!\!\bowtie\!\!\leftarrow$). By using the rewrite rules and induction over n, it can be shown that q^+ is equivalent to $\Pi_{A,\mathcal{P}(T_1)\cup\ldots\cup\mathcal{P}(T_n))}(\sigma_C({T_1}^+ \diamond \ldots \diamond {T_n}^+))$. Thus a SPJ query node is rewritten by rewriting its rangetable entries and adding all provenance attributes from the subqueries to its target list (see Figure 6.(1)).

2) ASPJ: The algebra representation of an ASPJ query node Q_{agg} is $\Pi_A(\sigma_{having}(\alpha_{G,aggr}(\sigma_C(T_1 \diamond \ldots \diamond T_n))))$. Remember that the result of the rewrite rule for aggregation includes the original aggregation operation (Q_{agg}) and the rewritten subquery (Q^+) of the original operation without aggregation. For an ASPJ query node we duplicate the original query node Q_{agg}, strip off the aggregation operation, selection on $having$ and projection on A and rewrite the duplicated node. Afterwards, a new query node Q_{top} is created that joins Q_{agg} and Q^+ according to the rewrite rule for aggregation (see Figure 6.(2)). The selection on $having$ and projection on A are defined on attributes from $G, aggr$. Which means they can be omitted in Q^+ without affecting the result of the join between Q_{agg} and Q^+.

3) Set-operations: Set operation query nodes are equivalent to an algebra statement of the following form: $q = T_1\ setop\ \ldots\ setop\ T_n$ with possible association of set operations using brackets. The position of the brackets is defined by the structure of the set operation tree. If none of the set operations is a set-difference, q^+ is of the form $q^{\diamond}{}_{T_1=\hat{T_1}}\Pi_{\mathbf{T_1}\rightarrow\hat{T_1},\mathcal{P}(T_1)}({T_1}^+)\ldots{}^{\diamond}{}_{T_1=\hat{T_n}}\Pi_{\mathbf{T_n}\rightarrow\hat{T_n},\mathcal{P}(T_n)}({T_n}^+)$ (same argument as for case 1). In this case, the operator $\diamond$ represents a left join for $setop = \cup$ and an inner join for $setop = \cap$. A set operation query node is rewritten by duplicating the query nodes of all Ts, rewriting the duplicates and creating a new top node Q_{top} that joins the original query node Q_{set} with the rewritten duplicates (see Figure 6.(3a)).

If the set operation tree contains set-difference operations, the structure of q^+ depends on the order and association of the set operations. This means we have to split a set-difference set operation query node into a query node for the set-difference operator Q_{diff} and two new nodes for its left and right subtree (Q_1 and Q_2). If the set-difference operation is not the top level set operation, we add Q_{diff} to the range table of its parent query and replace the complete subtree under Q_{diff} by a reference to the new range table entry for Q_{diff}. Q_{diff} is rewritten using the method $3a$ outlined above (see Figure 6.(3b)).

```
variables: pStack
input: the root node of a query tree
output: the root node of the rewritten query tree

traverseQueryTree (QueryNode q) {
  if (IsMarkedForRewrite(q)) {
    q+ := rewriteQueryNode(q);
  else
    foreach subquery q̂ in rangetable q
      traverseQueryTree(q̂)
}

rewriteQueryNode (Querynode q) {
  P−list := ∅ // Provenance attrs of q
  case is SPJ node
    foreach q̂ in range table
      rewriteQueryNode (q̂)
      P−list := pop P−lists of all q̂
  case is ASPJ node
    top := createQuerynode
    d := duplicate (q)
    add q and d to rangetable of top
    create join conditions for q and d
    strip of aggregation from d
    rewriteQueryNode (d)
    P−list := pop P−list of d
  case is set operation node
    create QueryNodes q1 and q2
        for children of top level set operation
    range table q := ∅
    add q1 and q2 to range table of q
    top := createQueryNode
    d1, d2 := duplicate (q1, q2)
    add q, d1, d2 to range table of top
    rewriteQueryNode(d1)
    rewriteQueryNode(d2)
    P−list := pop P−lists of d1 and d2
  case is base relation
    push R on pStack;
    return
  push P−list on pStack
  return q
}
```

Fig. 7: Rewrite algorithm

C. Rewrite Algorithm

With the results from the three cases of query node handling we are able to explain the rewrite algorithm used in the provenance rewrite module. The input to the algorithm is a query tree produced by the PostgreSQL rewriter module. Remember that the query tree has been semantically checked

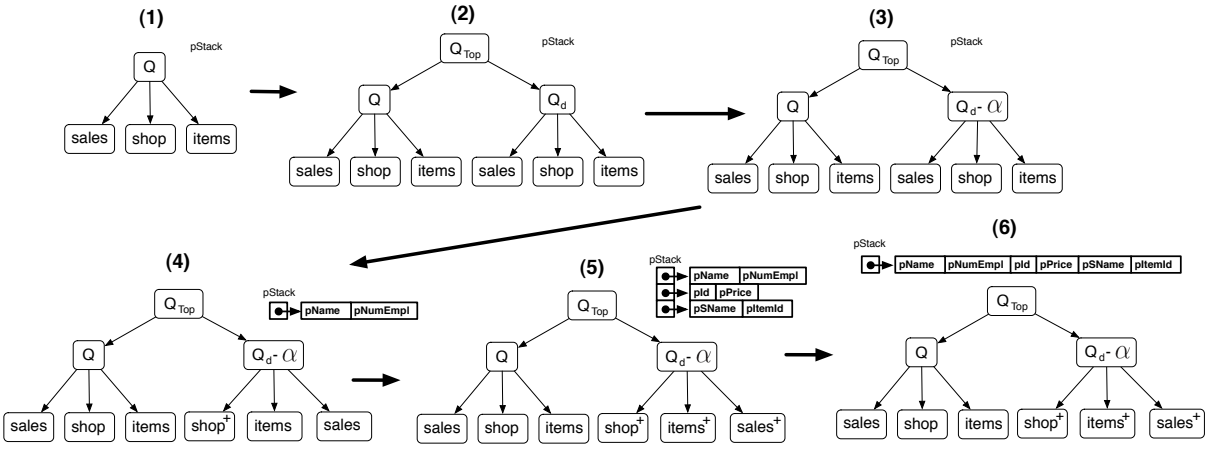

Fig. 8: Query tree rewrite

by the analyzer and views have been unfolded by the rewriter. A pseudo-code algorithm for the rewrite method is given in Figure 7. The main procedure (*traverseQueryTree*) of the provenance rewriter module traverses the query tree until a query node q marked for rewrite is found. Node q is passed to the *rewriteQueryNode* procedure that computes q^+. Afterwards q is replaced by q^+ in the query tree of the original query. If the traversal is finished the (possibly modified) query tree is passed to the planer module.

Procedure *rewriteQueryNode* computes q^+ top down by first rewriting the node q, leaving the subqueries in its range table untouched. In a second step, all direct child query nodes are rewritten by recursive calls to *rewriteQueryNode*. Afterwards the $\mathcal{P}$-list of q is computed using the $\mathcal{P}$-lists of its child nodes. Provenance attribute lists are stored on a stack data structure (*pStack*). Before returning from *rewriteQueryNode* q's $\mathcal{P}$-list is pushed on the stack. The first step distinguishes between the query node types presented above by inspecting the set operation tree and the boolean variable $hasAggs$ of q.[3] Note that we do not compute the rewrite bottom-up, because we would have to keep references to unmodified copies of subtrees of the original query in memory. For example, for an aggregation over a projection, the bottom-up computation would rewrite the projection first. To rewrite the aggregation the original projection is needed. This means we have to keep a copy of the original projection. For complex query trees access to certain subtrees of the original query would be cumbersome. Hence we compute provenance top-down to avoid this additional complexity.

D. Rewrite Example

Reconsider the example from section III. Algebraic statement q_{ex} can be expressed in SQL as follows:

```
SELECT PROVENANCE name, sum(price)
  FROM shop, sales, items
  WHERE name=sName AND itemId=id
  GROUP BY name;
```

[3]For set operation queries we included only the set-difference case to keep the algorithm short.

The query tree representation of q_{ex} is depicted in Figure 8.1. The *traverseQueryTree* procedure recognizes that the top query node Q is marked for rewrite and calls *rewriteQueryNode*. Q is a ASPJ query node (case 2), so *rewriteQueryNode* will create a new query node Q_{top}, create a duplicate Q_D of Q and add Q and Q_D to the range table of Q_{top} (Figure 8.2). In the next step, aggregation target entries are removed from Q_D, $hasAggs$ is set to false, and group-by- and having-clauses are removed (8.3). Afterwards *rewriteQueryNode* is applied to the base relations $shop$, $items$ and $sales$. Figure 8.4 captures the state after the rewrite of $shop$. Note that the $\mathcal{P}$-list from $shop^+$ is pushed on $pStack$. After the rewrite of all base relations there are three attribute lists on $pStack$ (see Figure 8.5). In the last step these lists are poped from $pStack$ and are combined to form the provenance attribute list for Q_{top}, which is pushed on $pStack$ before *rewriteQueryNode* returns (8.6).

E. Subqueries in Expressions

Until now we did not explain how subqueries in conditions are handled by the provenance rewrite module. To distinguish between from-clause subqueries and expression subqueries, we refer to the latter as *sublinks*. A sublink is called *correlated*, if it contains attribute references of the query it is used in. The algebra we introduced in section III has no support for subqueries in expressions (IN, EXISTS, ...), but the current implementation of *Perm* is able to rewrite sublinks, if they are uncorrelated. We plan to add support for correlated sublinks in the future.

Sublinks are rewritten by adding the rewritten sublink query to the range table of the query they are used in and join them with the other range table entries on the expression C the sublink is used in. In the join condition the sublink is replaced by a normal comparison operator. According to the contribution definition of Cui et. al, the provenance of a tuple includes all tuples accessed by the sublink query, if the expression C is true independent of the truth-value of the sublink condition C_{sub}. In this case we have to add the crossproduct of all base relations accessed by the sublink to the provenance result. For example consider the following query

Query	Overhead		
	absolute	relative 10MB	relative 100MB
1	0.007670s	1.0 %	0.10 %
3	0.001096s	2.3 %	0.20 %
5	0.000018s	0.2 %	0.02 %
6	0.000116s	0.3 %	0.03 %
7	0.024633s	38.0 %	6.60 %
8	0.013342s	25.0 %	1.50 %
9	0.001982s	1.5 %	0.10 %
10	0.000012s	0.2 %	0.01 %
11	0.000261s	3.3 %	0.03 %
12	0.000310s	0.5 %	0.05 %
13	0.000064s	0.2 %	0.01 %
14	0.000513s	1.4 %	0.10 %
15	0.000513s	1.3 %	0.10 %
16	0.000074s	0.4 %	0.02 %
19	0.000808s	1.3 %	0.10 %

Fig. 9: TPC-H: Compilation Time Overhead for Normal Queries

with a disjunctive where condition:

```
SELECT PROVENANCE name
  FROM shop
  WHERE numEmpl < 10
    OR name IN (SELECT sName FROM sales);
```

$$C = (numEmpl < 10 \vee C_{sub})$$
$$C_{sub} = name\ IN\ (SELECT\ sName\ FROM\ sales)$$

For the tuple $t = (Merdies, 3)$ from the *shop* relation the where condition C is true whether or not the sublink condition C_{sub} is fulfilled. This means the provenance of tuple t includes all tuples from the $sales$ relation.

V. Experiments

Before explaining the experimental evaluation of the *Perm* approach, we summarize the functionality of our system. With the *Perm* system, a user can compute the provenance of a SQL-query, create a view for the provenance computation of this query, and store provenance by using *SELECT INTO*. Stored and external provenance is used in a provenance computation, if the user specifies a list of provenance attributes in the *PROVENANCE*-clause. Provenance queries can be used as subqueries, hence we provide query-functunality with full expressive power of SQL.

We used the TPC-H benchmark [17] to evaluate the cost of provenance computation with the *Perm* prototype. This decision support benchmark consists of 22 complex queries with aggregation and subqueries in expressions (sublinks). The *Perm* prototype currently supports all SQL-features implemented by PostgreSQL except correlated sublinks, thus we can not compute the provenance of queries 2,4,17,18,20,21 and 22 from the benchmark. Each benchmark query was run in three different configurations. One configuration is a normal PostgreSQL instance. One configuration used the *Perm* prototype without computing provenance information, to evaluate the overhead our system introduces for normal operations. In the last configuration the queries are executed with provenance computation. All experiments are performed for test database sizes of 10MB, 100MB and 1GB on a 2GHz Intel-dual-core machine with 1GB of main memory. The TPC-H benchmark is provided with a query generator that randomly sets the parameters of a query and adapts the queries for a certain database size. We used this generator to generate a set of 100 versions for each benchmark query and database size. The same sets were used in all configurations. We stopped queries after 12 hours of execution if they had not terminated by then. These queries are represented by black table cells in the tables.

Q	10MB		100MB		1GB	
	norm.	prov.	norm.	prov.	norm.	prov.
1	0.764s	2.258s	7.610s	28.794s	77.579s	235.334s
3	0.042s	0.070s	0.594s	1.095s	39.165s	227.809s
5	0.010s	0.018s	0.083s	0.098s	0.889s	0.818s
6	0.036s	0.074s	0.373s	0.770s	18.152s	35.818s
7	0.063s	0.105s	0.374s	1.096s	31.779s	278.191s
8	0.051s	0.080s	0.884s	1.949s	56.043s	91.027s
9	0.122s	16.896s	1.950s	4985.910s	76.577s	
10	0.008s	0.012s	0.081s	0.093s	0.801s	0.903s
11	0.008s	0.102s	0.838s	11.693s	3.638s	1038.826s
12	0.057s	0.058s	0.569s	0.585s	17.780s	17.948s
13	0.036s	0.459s	0.476s	5.235s	10.463s	397.869s
14	0.036s	0.074s	0.370s	0.777s	17.216s	36.609s
15	0.038s	0.044s	0.371s	0.421s	17.996s	18.480s
16	0.020s	4.917s	0.280s		7.318s	
19	0.061s	0.06s	0.626s	1.323s	20.266s	40.647s

Fig. 10: TPC-H: Execution Time Comparison

Q	10MB		100MB		1GB	
	norm.	prov.	norm.	prov.	norm.	prov.
1	4	60'175	4	600'572	4	6'001'215
3	111	287	1'149	3'008	11'478	30'222
5	0	0	0	0	0	0
6	1	0	1	0	1	0
7	4	56.22	4	575	4	5'844
8	2	23.04	2	240	2	2'429
9	173	3'313	175	30'755	175	
10	0	0	0	0	0	0
11	14	3'376	104	410'011	1'469	38'535'360
12	0	0	0	0	0	0
13	32	15'336	37	153'364	42	1'533'833
14	1	0	1	0	1	0
15	0	0	0	0	0	0
16	296	119'224	2'820		18'193	
19	1	1	1	12	1	118

Fig. 11: TPC-H: Number of Result Tuples

A. TPC-H Benchmark Results

1) Overhead for Normal Queries: The average absolute and relative overhead introduced by *Perm* for normal query execution is depicted in Figure 9. The overhead depends only on the algebraic structure of the query and thus is independent of the database size. This experiment clearly shows that the overhead of *Perm* for normal operations is negligible. The maximal overhead was 25 milliseconds. We omited the results for database size 1GB, because the overhead is insignificant small in comparison with the overall execution time.

numSetOp	10MB		100MB		1GB	
	norm.	prov.	norm.	prov.	norm.	prov.
1	0.005s	0.008s	0.032s	0.061s	0.298s	0.588s
2	0.004s	0.021s	0.041s	0.415s	0.447s	1.077s
3	0.005s	0.079s	0.055s	1.037s	0.579s	1.686s
4	0.006s	0.166s	0.064s	1.243s	0.737s	2.593s
5	0.008s	0.193s	0.081s	2.284s	0.893s	4.256s

Fig. 12: Set-operations: Execution Time Comparison

Query	10MB		100MB		1GB	
	norm.	prov.	norm.	prov.	norm.	prov.
1	0.006s	0.015s	0.055s	0.102s	0.573s	1.037s
2	0.008s	0.015s	0.084s	0.147s	0.826s	1.439s
3	0.011s	0.019s	0.093s	0.158s	0.939s	1.601s
4	0.012s	0.019s	0.118s	0.200s	1.037s	1.757s
5	0.015s	0.022s	0.119s	0.194s	1.347s	1.887s
6	0.022s	0.031s	0.131s	0.207s	1.201s	1.954s

Fig. 13: SPJ Operations: Execution Time Comparison

agg	10MB		100MB		1GB	
	norm.	prov.	norm.	prov.	norm.	prov.
1	0.0002s	0.0011s	0.0014s	0.0067s	0.0117s	0.0634s
2	0.0005s	0.0019s	0.0041s	0.0215s	0.0181s	0.1721s
3	0.0011s	0.0151s	0.0024s	0.0139s	0.0668s	0.3777s
4	0.0012s	0.0067s	0.0094s	0.0498s	0.0197s	0.1896s
5	0.0015s	0.0098s	0.0118s	0.0683s	0.1160s	0.7483s
6	0.0019s	0.0138s	0.0144s	0.0899s	0.1425s	0.9554s
7	0.0022s	0.0188s	0.0171s	0.1153s	0.1696s	1.1937s
8	0.0025s	0.0218s	0.0197s	0.1407s	0.1954s	1.5011s
9	0.0028s	0.0267s	0.0225s	0.1712s	0.2212s	1.7876s
10	0.0031s	0.0326s	0.0251s	0.2043s	0.2480s	2.0680s

Fig. 14: Aggregation Operations: Execution Time Comparison

2) Costs of Provenance Computation: A comparison between the run times of normal and provenance queries is shown in Figure 10. To have an estimate for the complexity of a provenance computation in comparison with normal queries, the average number of result tuples of provenance and normal queries is presented in Figure 11. [4]

Except for queries 9,11 and 16 the overhead of provenance queries lies in a range of factor 3 to 30. In most cases a higher factor indicates a huge increase in the number of result tuples. For example, for query 1 and database size 10MB the number of result tuples increases approximately by a factor of 15000, but the execution time only increases by a factor of 4. The queries with high factors are usually queries with sublinks, which result in inefficient algebra expressions and in general generate huge amounts of result tuples. Query 9 is an aggregation over a join on 8 tables grouping on a functional expression. Thus, the rewritten query is a join on this functional expression with a table consisting of a join over 8 tables. Of course this type of query is expensive to compute. Query 11 is an aggregation with a sublink in the having clause condition. The sublink is an aggregation that accesses three base relations. In the rewritten form, this query includes a theta-join on the rewritten sublink, which itself is a join of its aggregation operation with the accessed base tables. The rewritten form of query 11 produces approximately 38 million tuples in an execution time that increases only by a factor of 300. Query 16 is an aggregation containing a negated sublink. According to the contribution definition, the provenance of a single tuple of the result of query 16 includes every tuple from the subquery that did not fulfill the sublink condition.

B. Results for Artificial Queries

In addition to the TPC-H benchmark, we generated artificial queries to test the performance of specific types of queries.

1) Set-operation Queries: We generated queries consisting only of set operations with simple selections on the TPC-H base table *part* restricted to a random range of primary key attribute values as inputs for the set operations. A set operation query is a random set operation tree structure with $numSetOp$ leaf nodes (Selections on *part*). For one value of $numSetOp$, we measured the average execution time of 100 of these set operation queries. In the worst case, the provenance computation of a set-difference operation can degrade to a crossproduct of the subqueries used in the right operand (T_2) of the set difference. The number of result tuples grows exponentially in the number of set difference operations. Therefore, we used only union and intersections in the experiment to evaluate the effect of the computational complexity of a provenance query instead of the effect of exponential result growth. The results from Figure 12 show that union and intersection operations are handled well by the *Perm* system. Note that the current version of *Perm* uses the simpler version of set operation rewriting (see Figure 6 (3.b)). We expect a significant speedup using the other set rewrite variant (3.a), because it omits the creation of unnecessary intermediate results.

2) SPJ Queries: The second set of artificial queries tests the performance of SPJ-queries. Each query is built using $numSub$ leaf subqueries. For each query a random query tree is created. A run of this experiment is built using 100 queries. The results indicate that provenance computation of SPJ queries is a very efficient operation. As shown in Figure 13, the provenance computation results in a maximal average overhead of a factor of 10. This is expected behavior, because the rewrite algorithm only adds new attributes to the target list of a SPJ query and does not change its structure.

3) ASPJ Queries: In the next set of experiments, the performance of nested aggregation operations is tested. An aggregation test query consists of agg aggregation operations. Each aggregation operates on the result of its child aggregation. The leaf operation accesses the TPC-H table *part*. Every aggregation groups the input on a range of primary key attribute values. The ranges are chosen so that each operation performs approximately the same number of aggregation function computations. This is achieved by grouping on the primary key attribute divided by $numGrp = \sqrt[agg]{|\,part\,|}$. Figure 14 demonstrates that the execution time grows linear in

[4]For some queries the number of provenance results is 0, but the number of original results is 1. These queries are aggregations over an empty relation, which lead to a single result tuple with all attribute values set to *null*.

System	10MB	100MB	1GB
Trio	113s	922s	9309s
Perm	3s	25s	249s

Fig. 15: Execution Time Comparison with Trio

database size and number of aggregation operations, because each aggregation operation introduces a new join for the provenance query.[5]

C. Comparison with the Trio Approach

As a final experiment, we compared the execution of queries between *Perm* and the *Trio* system. For these experiments we had to use simple SPJ queries and one level set operations, because other query types are not supported by *Trio*. We generated 1000 simple selections on a range of primary key attribute values of relation $supplier$. Figure 15 presents the overall execution times in seconds for the complete set of queries. *Perm* outperforms *Trio* by a factor of at least 30. Note that *Trio* does not support lazy provenance computation, so the provenance was computed beforehand. The measured execution time includes only the time to query the stored provenance. For *Perm* the provenance was computed lazily. Note that a certain amount of the observed overhead may be due to the uncertainty management features provided by *Trio*. We did not compare the performance of our system to the approach presented in [12] because this approach uses different contribution semantics.

VI. Conclusion

In this paper we have described the *Perm* system, a provenance management system supporting an almost complete subset of the SQL language. *Perm* uses query rewrite mechanisms to compute the provenance of an SQL query. In addition to providing lazy and eager computation of provenance, SQL query facilities, and support for external provenance, the experiments described in the paper indicate that *Perm* can outperform existing approaches by a factor of 30. Moreover, the experiments show that normal database operations are not effected by the provenance extensions. Queries with multiple set-difference operations and subqueries in negated or disjunctive expressions generate enormous amounts of provenance data and thus are still an open problem. This is partly due to the contribution definition, so a more restrictive definition should be used for provenance computation in future versions of the system.

We plan to investigate if the *Perm* approach can be extended for transformation provenance, data manipulation queries and correlated sublinks. Further areas of interest include different contribution semantics, different data item granularity and more efficient set-operation, aggregation and sublink computation using specialized plan operators.

[5]For some values of agg the expression $numGrp$ yields an integer result. In these cases the groupby expression is of less computational complexity, leading to faster query execution.

VII. Acknowledgements

We would like to dedicate this work to Klaus R. Dittrich, who passed away on 20 Nov 2007. Klaus helped to establish the *Perm* project and advised Boris in the preliminary research that led to the work reported in this paper.

References

[1] B. Glavic and K. R. Dittrich, "Data provenance: A categorization of existing approaches," in *Proc. BTW'07*, pp. 227–241.
[2] W. Tan, "Research problems in data provenance," *IEEE Data Engineering Bulletin*, vol. 27, no. 4, pp. 42–52, 2004.
[3] P. Buneman, S. Khanna, and W. C. Tan, "Why and where: A characterization of data provenance," in *Proc. ICDT '01*, pp. 316–330.
[4] Y. Cui and J. Widom, "Lineage tracing in a data warehousing system," in *Proc. ICDE '00*, p. 683.
[5] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom, "An introduction to ULDBs and the Trio system," *IEEE Data Engineering Bulletin*, vol. 29, no. 1, pp. 5–16, 2006.
[6] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, "DBNotes: a post-it system for relational databases based on provenance," in *Proc. SIGMOD '05*, pp. 942–944.
[7] P. Buneman, S. Khanna, and W.-C. Tan, "Computing provenance and annotations for views," in *Workshop on Data Derivation and Provenance*. Chicago IL., October 2002.
[8] P. Buneman, A. Chapman, and J. Cheney, "Provenance management in curated databases," in *SIGMOD'06*, pp. 539–550.
[9] P. Buneman, J. Cheney, and S. Vansummeren, "On the expressiveness of implicit provenance in query and update languages," *Proc. ICDT'07*, p. 209.
[10] P. Buneman and J. Cheney, "A copy-and-paste model for provenance in curated databases," The University of Edinburgh, Tech. Rep., 2005.
[11] J. Cheney, A. Ahmed, and U. Acar, "Provenance as dependency analysis," *Proc. DBPL'07*.
[12] J. Cheney, "Program slicing and data provenance," *IEEE Data Bulletin Engineering*, vol. 30, no. 4, pp. 22–28, 2007.
[13] D. Lui and M. Franklin, "GridDB: A data-centric overlay for scientific grids," *Proc. VLDB'04*, pp. 600–611.
[14] P. Groth, S. Miles, and L. Moreau, "PReServ: Provenance recording for services," in *Proc. AHM'05*.
[15] T. Heinis and G. Alonso, "Efficient lineage tracking for scientific workflows," in *SIGMOD'08*, pp. 1007–1018.
[16] B. Momjian, *PostgreSQL: Introduction and Concepts*. Boston, MA: Addison-Wesley, 2001.
[17] Transaction Processing Performance Council. (2008) TPC-H benchmark specification. [Online]. Available: http://www.tpc.org/tpch/