The *Perm* Provenance Management System in Action

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

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

ABSTRACT

In this demonstration we present the *Perm* provenance management system (PMS). *Perm* is capable of computing, storing and querying provenance information for the relational data model. Provenance is computed by using query rewriting techniques to annotate tuples with provenance information. Thus, provenance data and provenance computations are represented as relational data and queries and, hence, can be queried, stored and optimized using standard relational database techniques. This demo shows the complete *Perm* system and lets attendants examine in detail the process of query rewriting and provenance retrieval in *Perm*, the most complete data provenance system available today. For example, *Perm* supports lazy and eager provenance computation, external provenance and various *contribution* semantics for an almost complete subset of SQL..

1. INTRODUCTION

Data provenance is information about the origin of a *data item* and the *transformations* used to produce this data item. Provenance information is used in areas like curated databases, data warehouses and e-science to trace errors, estimate data quality and gain additional insights about data.

In the relational data model, data items are relations, tuples, and attribute values. Transformations are queries and functions defined over these data items. The provenance of a tuple t produced by a query q includes all tuples from the base relations accessed by the query, that *contributed* to the existence of t. Different definitions of *contribution* have been proposed in the literature (see [5]). Two prominent examples for *contribution* definitions are Why-provenance [2] and Where-provenance [1].

In [3] we introduced the novel *Perm* provenance management system. *Perm* uses query rewrite techniques to transform a query q into a query q^+ that computes the provenance of q. Our system has been implemented as an extension of a relational DBMS. By representing provenance data and provenance computation as relational data and queries, *Perm* benefits from the query, optimization and storage techniques developed for relational databases and supports provenance computation for complex SQL queries. The system supports different contribution semantics, lazy and eager computation of provenance, manually created provenance, and queries that combine provenance and 'normal' data. Hence, a user can pick the *contribution* definition that fits his needs and decide whether he will store the provenance of a query for later reuse or let the system compute it on the fly.

messages

mId	text	uId
1	lorem ipsum ...	3
4	hi there ...	2

users

uId	name
1	Bert
2	Gert
3	Gertrud

imports

mId	text	origin
2	hello ...	superForum
3	I don't ...	HiBoard

approved

uId	mId
2	2
1	4
2	4
3	4

q1:
```
SELECT mId, text FROM messages
  UNION SELECT mId, text FROM imports;
```

q2:
```
CREATE VIEW v1 AS q1;
```

q3:
```
SELECT count(*), text
  FROM v1 JOIN approved a ON (v1.mId = a.mId)
  GROUP BY v1.mId, text;
```

Figure 1: Example database and queries

1.1 Example database

Before we present the *Perm* system and the underlying concepts, we introduce a small example database that is used throughout this demo proposal to illustrate various aspects of our approach. The example database shown in Figure 1 represents the data of an online forum with *users*, *messages*, messages that were imported from other forums (*imports*), and a table that stores the information which user approved which message (*approved*). Some example queries are given in Figure 1. q_1 returns all messages entered by users of the forum or imported from other forums. q_2 stores q_1 as a view. q_3 outputs the text of each message together with the number of users that approved this message (messages without any approval are omitted from the result).

ACM SIGMOD '09 Providence, RI, USA

original result attributes		provenance attributes from *messages*			provenance attributes from *imports*		
mId	**text**	**p_mId**	**p_text**	**p_uId**	**p_mId**	**p_text**	**p_origin**
1	lorem ipsum ...	1	lorem ipsum ...	3	null	null	null
2	hello ...	null	null	null	2	hello ...	superForum
3	I don't ...	null	null	null	3	I don't ...	HiBoard
4	hi there ...	4	hi there ...	2	null	null	null

Figure 2: Query q_1 provenance

2. THE PERM SYSTEM

Perm is a provenance management system (PMS) that computes the provenance of relational queries on a tuple level granularity. The provenance of a query is calculated by using query rewrite techniques to annotate result tuples of a query q with provenance information.

2.1 Datamodel

Unlike other approaches *Perm* has a 'pure' relational representation of provenance data and provenance queries. The provenance of query is represented as a single relation that contains the original query results augmented with provenance information. Provenance information is attached to a query result by extending the original result tuples with the contributing tuples from the base relations accessed by the original query. Thus, all attributes from the relevant base relations are appended to the result schema of the original query. To distinguish between original attributes and provenance attributes, provenance attributes are identified by a prefix and the name of the relation they are derived from. To keep the examples compact we do not use this naming scheme for all examples, but instead identify provenance attributes with the prefix $p\_$. For example, the schema of the provenance of query q_1 in Figure 1 is:

$$(count, text, prov\_messages\_mId, prov\_messages\_text, \\ prov\_messages\_uId, prov\_imports\_mId, \\ prov\_imports\_text, prov\_imports\_origin)$$

A tuple t^+ of a provenance query result is built by attaching all contributing tuples to the original result tuple t. If there is more than one contributing tuple from one base relation, the orginal result tuple t has to be replicated. For instance, the provenance of query q_1 in the running example is depicted in Figure 1.1.

2.2 Provenance computation though query rewriting

Perm computes the provenance of a query q by applying a set of algebraic rewrite rules that transform q into a provenance query q^+. The provenance query q^+ generates the provenance representation introduced in the previous section. The rewrite rules are defined over an algebraic representation of a query and operate on a single algebraic operator. Each rule is defined over an input algebra expression and the list of provenance attributes of its input ($\mathcal{P}$). The result of a rewrite rule is a transformed algebra expression and provenance attribute list. As an example consider the rewrite rule for the projection operator:

$$(\Pi_A(T))^+ = \Pi_{A,\mathcal{P}(T^+)}(T^+) \text{ with } \mathcal{P}((\Pi_A(T))^+) = \mathcal{P}(T^+)$$

For an in depth explanation of the rewrite rules the interested reader is referred to [3] and [4]. In principle the rewrite rules are unaware of how the provenance attributes of their input were produced. This is a huge advantage, because it enable us to use the rewrite rules to propagate provenance information that was not produced by *Perm*. For example *Perm* can compute the provenance of queries that include data that was annotated with provenance information manually or by another provenance management system.

For some operators there is more than one rewrite rule that produces the provenance of the operator. For this type of operator the choice of rewrite rule influences the performance of the provenance computation. We use cost-based optimization to choose the best rewrite strategy for each situation.

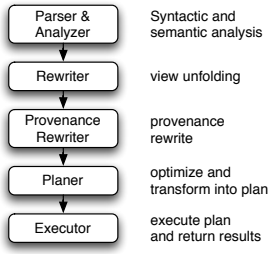

Figure 3: *Perm* architecture overview

2.3 Architecture

We have implemented *Perm* as an extension of the PostgreSQL DBMS (see Figure 3). Perm operates on the internal query tree representation of a query. The output of the Postgres query analyzer is passed to the *Perm* rewrite module. The rewrite module tranverses the query tree and applies the provenance query rewrite rules to transform the query (or part of the query) into a provenance query. The rewritten query tree produced by the *Perm* module is handed over to the Postgres query optimizer and, thus, *Perm* benefits from the query optimization techniques incorporated into PostgreSQL.

2.4 SQL-PLE: Perm's provenance SQL extension

Perm uses an SQL language extension called *SQL-PLE* to enable a user to issue provenance queries. The keyword PROVENANCE is employed to instruct *Perm* to compute the provenance of a query. An optional ON CONTRIBUTION modifier is used to specify the *contribution* definition for the provenance computation (at the current time *Perm* supports Why-provenance as keyword INFLUENCE and several types

of Where-provenance as keyword COPY). For example:

```
SELECT PROVENANCE INFLUENCE count(*), text
  FROM v1 JOIN approved a ON v1.mId = a.mId
  GROUP BY v1.mId;
```

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. Therefore a user cannot just receive provenance information, but also query provenance information, store it as a view, etc. For example, the following query can be used to output messages imported from the 'superForum' board that were approved by at least five users:

```
SELECT text, p_origin
  FROM
    (SELECT PROVENANCE count(*), text
    FROM v1 JOIN approved a ON v1.mId = a.mId
    GROUP BY v1.mId) AS prov
  WHERE count > 5 AND p_origin = 'superForum';
```

Perm supports incremental provenance computation by allowing the manual specification of provenance attributes of a relation or subquery, and providing language constructs to stop the rewrite process at a certain point. E.g., consider a query over a view where the user is interested in the tuples from the view that contributed to the query result (in contrast to the base relation tuples that contributed to the query result). The keyword BASERELATION is appended to a subquery or view to instructed *Perm* to handle it like a base relation. To manually specify the provenance attributes of a view, base relation or subquery, the keyword PROVENANCE followed by a list of attribute names has to be appended to a FROM-clause item. For example consider the following query defined over view v_1 in our running example:

```
SELECT PROVENANCE text
  FROM
    v1 BASERELATION
  WHERE count > 3;
```

In this example view v_1 will be handled like base relation. Therefore, the rewrite rules are not applied to the view definition of v_1, but the attributes of the view query result are renamed and attached to the query result.

3. DEMONSTRATION

In the demonstration we will illustrate the functionality of *Perm* by executing a few example queries. The *Perm-browser* client application used in the demonstration enables a user to send queries to the system (see Figure 4 marker 1), view query results (see Figure 4 marker 5), activate or deactivate rewrite strategies, and choose between different contribution semantics. In addition to the query results, the browser presents the rewritten query as an SQL statement (see Figure 4 marker 2) together with algebra trees for the original (see Figure 4 marker 3) and rewritten query (see Figure 4 marker 4).

The demonstration will be divided into the following parts:

- **Query execution**: At first we will run queries on the example database introduced in this paper and analyze the produced results.

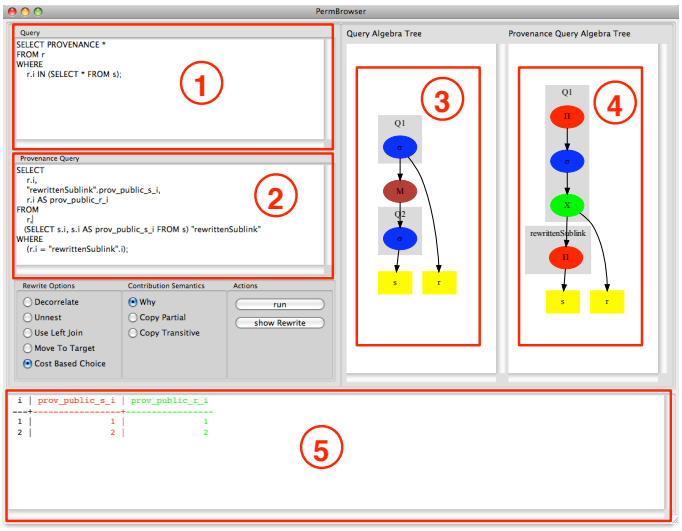

Figure 4: *Perm* browser

- **Rewrite analysis**: In this part of the demonstration we will illustrate the rewrite process for some example queries using the algebra trees and rewritten SQL-statements generated by the *Perm-browser*.

- **Implementation details**: Depending on demands by the participants we will reveal implementation details.

- **Complex queries**: At the end of the demonstrationt we will let participants run queries with the *Perm-browser* and discuss the results and applied rewrite rules.

4. CONCLUSION

In this demonstration proposal we presented the *Perm* PMS that provides efficient provenance computation and query facilities for the relational data model. Our system is able to handle both manually created provenance and provenance produced by other PMS. A user can choose between different *contribution semantics* for provenance computation, store provenance for later investigation or incremental provenance computation, and query provenance with the full expressive power of SQL.

5. REFERENCES

[1] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT '01*, pages 316–330, 2001.

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

[3] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE '09 (to appear)*, 2009.

[4] B. Glavic and G. Alonso. Provenance for nested subqueries. In *EDBT '09 (to appear)*, 2009.

[5] B. Glavic and K. R. Dittrich. Data provenance: A categorization of existing approaches. In *BTW '07*, pages 227–241, 2007.