On the Impact of Provenance Semiring Theory on the Design of a Provenance-Aware Database System

Pierre Senellart
DI ENS, ENS, PSL University, CNRS, Paris, France
Inria, Paris, France
Institut Universitaire de France, Paris, France
CNRS@CREATE LTD, Singapore
IPAL, CNRS, Singapore

Abstract
We report on the impact that the theory of provenance semirings, developed by Val Tannen and his collaborators, has had on the design on a practical system for maintaining the provenance of query results over a relational database, namely ProvSQL.

2012 ACM Subject Classification Theory of computation → Data provenance; Information systems → Database management system engines

Keywords and phrases provenance, provenance semiring, ProvSQL

Digital Object Identifier 10.4230/OASIcs.Tannen.2024.9

Funding This work was funded in part by the French government under management of Agence Nationale de la Recherche as part of the “Investissements d’avenir” program, reference ANR-19-P3IA-0001 (PRAIRIE 3IA Institute). It is also part of the program DesCartes and is supported by the National Research Foundation, Prime Minister’s Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) program.

Acknowledgements ProvSQL is a collective effort; I acknowledge the contributions of Yann Ramusat, Silviu Maniu, Louis Jachiet, and Baptiste Lafosse to the development of the system.

1 Introduction

The important issue of keeping track of data throughout a complex process gave rise to the study of the provenance [4] of data, also sometimes called lineage [5]: extra information attached to query results which relates them to input data items. In a seminal work in 2007 [9], Todd J. Green, Grigoris Karvounarakis, and Val Tannen put forward provenance semirings as an algebraic framework to express a range of different forms of provenance over relational database systems, including the data lineage from [5], the why-provenance from [4], the Boolean provenance implicit in the early model of incomplete information of c-tables [12] (and made explicit in [10]), and many more. This work has had a considerable impact on the understanding of what data provenance is and how it can be computed, and was largely celebrated by the research community [11]. Val Tannen, in collaboration with a number of his colleagues, then further developed the theory of provenance semirings in other works, covering topics such as compact representation of provenance for recursive queries [7], provenance of non-monotone queries [2, 6], provenance of aggregate queries [3]. This line of work was also extended to other settings than the relational one and resulted in many different applications [18].

In 2016, inspired by this beautiful theoretical framework, motivated by applications of provenance to probabilistic databases [21, 10], and frustrated by the lack of maintained software that would implement provenance semirings, we embarked on the development of
On the Impact of Provenance Semiring Theory on the Design of ProvSQL

ProvSQL [19], a PostgreSQL extension that supports computation of semiring provenance and their extensions for SQL queries over relational databases. ProvSQL has first been demonstrated in 2018 [20] and has been updated and improved ever since.

In this paper, in honor of Val Tannen and the groundbreaking theory of provenance semirings, we want to reflect on the impact that theoretical research on provenance semirings has had on the design of ProvSQL: where ProvSQL directly implements the theory, where practical concerns require deviating from it, and when development is still lagging behind the theoretical framework.

We introduce semiring provenance and the way it is implemented in ProvSQL in Section 2, while in in Section 3 we discuss extensions that go beyond semiring provenance.

2 Semiring Provenance for Positive Relational Algebra Queries

We now discuss the semiring provenance framework from [9] for the positive relational algebra and how it is implemented in ProvSQL, in terms of data model, query evaluation, as well as representations of provenance expressions. We assume basic knowledge of the relational model and the relational algebra, see [1] for a primer.

2.1 Data Model

Theory

A semiring is an algebraic structure $(\mathbb{K}, \oplus, \otimes, 0, 1)$ where $(\mathbb{K}, \oplus, 0)$ and $(\mathbb{K}, \otimes, 1)$ are monoids (a set equipped with an associative binary operation and a neutral element), $\oplus$ is commutative, $\otimes$ distributes over $\oplus$, and 0 is an absorbing element for $\otimes$ (i.e., $\forall a \in \mathbb{K}, a \otimes 0 = 0 \otimes a = 0$). The semiring is commutative if $\otimes$ is commutative. The semiring is often referred to by $\mathbb{K}$ when the binary operations and neutral elements are clear. Given two semirings $\mathbb{K}$ and $\mathbb{K'}$, a semiring homomorphism from $\mathbb{K}$ to $\mathbb{K'}$ is a function from $\mathbb{K}$ to $\mathbb{K'}$ that maps neutral elements of $\mathbb{K}$ to the corresponding neutral elements of $\mathbb{K'}$ and that preserves the binary operations of the semirings.

▶ Example 1. The following are classical examples of semirings, with applications to provenance:
- $(\mathbb{B} = \{\bot, \top\}, \lor, \land, \bot, \top)$ is the semiring of Booleans;
- $(\mathbb{N}, +, \times, 0, 1)$ is the counting semiring;
- $(\mathbb{S} = \{\text{unclassified} < \text{restricted} < \text{confidential} < \text{secret} < \text{top_secret} < \text{unavailable}\}, \min, \max, \text{unavailable}, \text{unclassified})$ is the security semiring of security clearance levels;
- For any finite set $X$ of variables, $(\mathbb{N}[X], +, \times, 0, 1)$ is the integer polynomial semiring that is sometimes also called how-semiring.

See [17] for many more examples and their application to provenance.

Given a commutative semiring $(\mathbb{K}, \oplus, \otimes, 0, 1)$, [9] introduces a $\mathbb{K}$-relation (or relation annotated by $\mathbb{K}$) over a finite set of attributes $A$ as a function $R$ that maps tuples over $A$ to an element of $\mathbb{K}$ such that $\{t \mid R(t) \neq 0\}$ is finite. Homomorphisms are extended to $\mathbb{K}$-relations: for a homomorphism $h : \mathbb{K} \to \mathbb{K'}$ and a $\mathbb{K}$-relation $R$, $h \circ R$ is a $\mathbb{K'}$-relation. Finally, $\mathbb{K}$-databases are databases formed of (labeled) $\mathbb{K}$-relations, and semiring homomorphisms extend to $\mathbb{K}$-databases in the natural way.

▶ Example 2. $\mathbb{B}$-relations over a set of attributes $A$ are simply relations in the usual sense: finite set of tuples over $A$. 


<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>position</th>
<th>city</th>
<th>classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>Director</td>
<td>New York</td>
<td>unclassified</td>
</tr>
<tr>
<td>2</td>
<td>Paul</td>
<td>Janitor</td>
<td>New York</td>
<td>restricted</td>
</tr>
<tr>
<td>3</td>
<td>Dave</td>
<td>Analyst</td>
<td>Paris</td>
<td>confidential</td>
</tr>
<tr>
<td>4</td>
<td>Ellen</td>
<td>Field agent</td>
<td>Berlin</td>
<td>secret</td>
</tr>
<tr>
<td>5</td>
<td>Magdalen</td>
<td>Double agent</td>
<td>Paris</td>
<td>top_secret</td>
</tr>
<tr>
<td>6</td>
<td>Nancy</td>
<td>HR</td>
<td>Paris</td>
<td>restricted</td>
</tr>
<tr>
<td>7</td>
<td>Susan</td>
<td>Analyst</td>
<td>Berlin</td>
<td>secret</td>
</tr>
</tbody>
</table>

Consider the example personnel relation in Table 1. If $t_1, \ldots, t_7$ are elements of a semiring $K$ distinct from $0$, then this depicts a $K$-relation where every tuple of the relation is associated with a non-0 element of $K$. For example, assume that for every $1 \leq i \leq 7$, $t_i$ is set to the value of the classification attribute of the corresponding tuple; then this is a $S$-relation.

**Example 3.** For example, take $X = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$ and the valuation $v$ that maps each of these tuple ids to the corresponding security level from the classification attribute in Table 1. Then the unique homomorphism $h : \mathbb{N}[X] \to S$ preserving this valuation is the one that maps $t_1 t_2 + t_3 t_5$ to

$$\min(\max(v(t_1), v(t_2)), \max(v(t_4), v(t_4), v(t_5))) = \min(\max(v(t_1), v(t_2)), \max(v(t_4), v(t_5)))$$

$$= \min(\text{restricted}, \text{top_secret})$$

$$= \text{restricted}.$$  

**Implementation in ProvSQL**

ProvSQL relies on the universality of the integer polynomial semiring by representing every relation as an $\mathbb{N}[U]$-relation where $U$ is a set of unique identifiers of base tuples. One major difference with the theoretical framework is that relations in SQL are not sets of tuples but multisets: the definition of a $K$-relation thus needs to be modified to allow multiple annotations for the same tuple, resulting in multiple occurrences of this tuple.

To create an $\mathbb{N}[U]$-relation, ProvSQL provides an `add_provenance` function, that takes as input a regular PostgreSQL relation and modifies it to add to this relation an additional `provsql` attribute initialized with universally unique identifiers (UUIDs), generated at random.

When one wants to interpret such an annotated relation as a $K$-relation for a semiring $K$, it suffices to provide the valuation $v : U \to K$ (called in ProvSQL a *provenance mapping*) as a PostgreSQL relation, as well as a description of the homomorphism from $\mathbb{N}[U]$ to $K$, which amounts to explaining how to interpret in the semiring $K$ the 0, 1 elements of $\mathbb{N}[U]$, as well as the binary operations $+$ and $\times$ of $\mathbb{N}[U]$. ProvSQL provides a `provenance_evaluate` function for this purpose; operations of the semirings are typically coded as PL/pgSQL functions, PostgreSQL’s user-defined function language.
2.2 Positive Relational Algebra Query Evaluation

Theory

The same paper [9] shows how the different operators of the positive relational algebra (selection, projection, union, projection, cross product or join, and renaming) can be defined on \( K \)-relations: selection and renaming have no effect on the annotations; tuples merged as a result of a projection or a union are combined with the \( \oplus \) operation of the semiring; tuples jointly participating in producing a new tuple in a product or join are combined with the \( \otimes \) operation of the semiring. Given a query \( q \) of the positive relational algebra and a \( K \)-relation \( R \), \( q(R) \) is the \( K \)-relation obtained by inductively applying these operations.

▶ Example 4. Consider the Boolean query

\[
\Pi_{id}(\sigma_{id < id2}(\text{personnel} \bowtie \text{city}) \Pi_{id2,\text{city}}(\rho_{id \rightarrow id2}(\text{personnel})))
\]

over the running example schema which returns whether there exists a city with two distinct individuals. The result of evaluating this query over the \( \mathbb{N}[X] \)-relation from Table 1 is the annotated relation with a single nullary tuple annotated with the polynomial \( t_1t_2 + t_3t_5 + t_3t_6 + t_5t_6 + t_4t_7 \). If instead this is a \( S \)-relation with the annotation from the classification attribute, the resulting annotation is \( \min(\text{restricted}, \text{secret}, \text{top_secret}, \text{top_secret}, \text{secret}) = \text{restricted} \).

The reason why this is the right definition is given by two results from [9]. First, some standard identities of the relational algebra are preserved [9, Proposition 3.4]. Second, \textit{query evaluation commutes with semiring homomorphism} [9, Proposition 3.5]: if \( h \) is a homomorphism from \( K \) to \( K' \), \( q \) a positive relational algebra query and \( R \) a \( K \)-relation, then \( h \circ q = q \circ h \).

Implementation in ProvSQL

Again, the theory needs to be adapted to reflect the fact that SQL uses a multiset semantics and not a set semantics. This has an impact for projections (which does not imply duplicate eliminations in SQL) and for \texttt{UNION ALL} unions: they do not change the provenance annotations. On the other hand, \texttt{DISTINCT} and \texttt{GROUP BY} operators in SQL result in duplicate elimination and thus in the application of the \( \oplus \) operator of the semiring.

All operations are done in the \( \mathbb{N}[U] \) semiring. Because of the commutativity of semiring homomorphisms and query evaluation, it is possible to evaluate the result of a query in a different semiring by first evaluating it in the \( \mathbb{N}[U] \) semiring and then apply the semiring homomorphism.

In ProvSQL, the \( \oplus \) and \( \otimes \) operations of the semiring are respectively implemented by a \texttt{provenance_plus} and \texttt{provenance_times} user-defined function. At planning time, the query sent to PostgreSQL is rewritten so that the resulting relation includes a \texttt{provsql} column whose content is computed using these two functions, following the operations specified in the query.

▶ Example 5. Consider the following SQL query, which uses the usual \texttt{SELECT DISTINCT} 1 trick to mimic the behavior of the Boolean query from Example 4:
SELECT DISTINCT 1 FROM (  
    SELECT p1.city  
    FROM personnel p1  
    JOIN personnel p2 ON p1.city=p2.city  
    WHERE p1.id<p2.id  
    GROUP BY p1.city  
) inner_query;

In ProvSQL, this query gets rewritten to the following one so as to produce in a new provsql attribute the correct provenance annotation: provenance_times gets called to reflect the join, while provenance_plus gets called to reflect both the GROUP BY and DISTINCT operators (the latter being converted to a GROUP BY).

SELECT 1, provenance_plus(ARRAY_AGG(provsql)) AS provsql FROM (  
    SELECT p1.city, provenance_plus(ARRAY_AGG(provenance_times(p1.provsql,p2.provsql))) AS provsql  
    FROM personnel p1 JOIN personnel p2 ON p1.city=p2.city  
    WHERE p1.id<p2.id  
    GROUP BY p1.city  
) inner_query GROUP BY 1;

Another challenge of the practical implementation is that PostgreSQL’s internal data structures do not fully match the abstract view of the relational algebra; instead, every operator that exists in the SQL language gets reflected in a special way, which requires handling many subcases (and which means SQL support in ProvSQL is still not complete to this date).

2.3 Provenance Representations

Theory

Though [9] does not explicitely give complexity results, it is clear that provenance tracking can be done in polynomial-time. The exact complexity, however, depends on how costly the ⊕ and ⊗ operations of the semiring are. If they can be reasonably counted to be in $O(1)$ for certain application semirings (e.g., the security semiring or even the counting semiring if integers involved are bounded), one needs to be more careful about the complexity of operations in more complex semirings such as the integer polynomial semiring. Indeed, if one were to require expanding every polynomial to a sum of monomial, it is easy to construct examples where this results in exponentially-sized expressions.

The question of compact representation of provenance led Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen to propose in [7] arithmetic circuits to represent provenance annotations in a way that allows sharing and does not require copying entire subexpressions or expanding them. This was done in the context of recursive queries (see 3.1) but is also useful for non-recursive ones.

Implementation in ProvSQL

Since all provenance in ProvSQL is $\mathbb{N}[U]$ provenance, compact representation is paramount for efficiency of query evaluation as well as reduced use of storage. ProvSQL thus stores provenance as an arithmetic circuit, whose internal gates are the semiring operators and leaves are base UUIDs. This way, the provsql column of provenance-aware relations can
simply be pointers to the corresponding gates in the circuit (in practice, we also use UUIDs as identifiers of internal gates, and these UUIDs are stored in the provsql columns). The provenance_plus and provenance_times functions add new gates to the circuit. This raises the question of where to store the circuit. We have successively experimented with three different storage mechanisms:

1. Initially, the provenance circuit was stored as a table within the same database, managed by the database engine. Unfortunately, this is extremely inefficient, as this means that every query results in many different updates (each time a gate is created in the circuit) on the provenance circuit table; this was also a nightmare in terms of concurrency control as every query turned into a batch of updates.

2. We then moved to storing the circuit in main memory, using the shared memory buffers of PostgreSQL. This was much more efficient and made it easier to address concurrency issues, but this was not a viable solution either, as the amount of shared memory buffers is limited, and this solution does not provide any way to ensure persistence of storage of the circuit.

3. In our latest implementation, the circuit is stored on disk, in memory-mapped files that are accessed through a single process. This solution resolves the issues of persistence and concurrency control, while memory mapping helps with keeping access to the circuit efficient in practice.

3 Beyond Positive Relational Algebra Queries and Semirings

We now briefly discuss theoretical ways that have been proposed to go beyond the positive relational algebra and the semiring frameworks, and their influence in the design of ProvSQL.

3.1 Recursive Queries

Theory

The original paper on the semiring framework [9] also dealt with recursive queries in the form of Datalog programs. Green, Karvounarakis, and Tannen showed that their provenance could be captured by semirings, as long as those satisfied some technical conditions (in particular, being $\omega$-continuous). In this setting, most of the results for the positive relational algebra can be recovered: commutativity of Datalog queries and semiring homomorphisms, as well as the existence of a universal semiring, i.e., the semiring of formal power series. Algorithms for computing the provenance of recursive queries were then refined in [7] with the introduction of provenance circuits. Recent work by other authors [14] study in more detail conditions for convergence of Datalog queries involving provenance.

Implementation in ProvSQL

Unfortunately, support for recursive queries cannot be added to ProvSQL in a straightforward way. This is due to the fact that the computation of the provenance annotation in the special provsql columns requires aggregation to combine annotations of different tuples, and that SQL forbids aggregation within \texttt{WITH RECURSIVE} recursive queries. There does not seem to be any easy way around this without reimplementing a query evaluation engine, which is out of the scope of the ProvSQL project. Note, however, that together with Yann Ramusat and Silviu Maniu, we have proposed and experimented various algorithms for evaluation of the provenance of recursive queries [15, 16], inspired by [9, 7], but in a simpler setting outside of a database engine.
3.2 Non-Monotone Queries

Theory

A natural direction beyond the positive relational algebra is to add negation, by adding the difference operator of the full relational algebra. One way to add them to the framework of provenance semirings is to extend semirings with a \( \text{monus} \odot \text{operator} \) (which results in what is called \( m\)-semirings), as proposed by Floris Geerts and Antonella Poggi [8]; for example, in the Boolean semiring it is as expected defined as \( a \odot b = a \land \neg b \) and in the counting semiring as \( a \odot b = \max(0, a - b) \). However, Yael Amsterdamer, Daniel Deutch, Tova Milo, and Val Tannen have showed in [2] that this definition results in some counter-intuitive results (some common axioms, such as distributivity over \( \otimes \) over \( \odot \), fail); in addition, \( \mathbb{N}[X] \) is not a universal semiring with monus [8].

As an alternative to semirings with monus, Katrin M. Dannert, Erich Grädel, Matthias Naaf, and Val Tannen have proposed [6] a very general logical framework for computing the provenance of recursive queries (in the form of fixpoint logics) with negation. This is based on a trick of associating with every positive provenance token a corresponding negative one and considering the semiring of integer polynomials (or formal series in the recursive case) with variables both positive and negative tokens.

A final alternative for provenance with difference is given by the work on provenance aggregate [3] that we discuss in the next section.

Implementation in ProvSQL

ProvSQL follows the m-semiring approach, despite its limitations identified in [2]. Since \( \mathbb{N}[X] \) is not universal any longer, we need to work with the actual universal m-semiring, which is simply the free m-semiring [8], i.e., the m-semiring of free terms constructed using \( \ominus, \otimes, \odot \), quotiented by the equivalence relations imposed by the m-semiring structure. In practice, this means adding a \text{provenance_monus} function, used when the \text{EXCEPT} SQL keyword is used, that adds a \( \ominus \) gate in the provenance circuit.

3.3 Aggregate Queries

Theory

Another very commonly used query feature that goes beyond the relational algebra is aggregates. Yael Amsterdamer, Daniel Deutch, and Val Tannen have proposed a solution [3] in the form of \textit{provenance semimodules} for the case of aggregate functions that are associative and commutative, such as \textit{min}, \textit{max}, \textit{sum}, or \textit{count}. The scalar aggregate values form a monoid, which is combined with the provenance semiring to form a semimodule. Note that the resulting semimodule values are now annotating attribute values instead of annotating tuples. In addition to these semimodule attribute values, [3] introduces an additional \( \delta \) operator to the provenance semiring that is used to determine the tuple annotation of tuples that include an aggregate computation (see [3]). Finally, when selection can be done on the result on an aggregation, additional comparison operators are introduced to build provenance annotations (these operators can then be used to define a semantics for difference).
Example 6. Consider the query that counts the number of distinct cities in the running example schema. When evaluating this query over the $\mathbb{N}[X]$-relation from Table 1 we obtain a unary tuple with semimodule value

$$(t_1 \oplus t_2) \star 1 + (t_3 \oplus t_5 \oplus t_6) \star 1 + (t_4 \oplus t_7) \star 1$$

where $\star$ is a tensor product allowing combining elements of the aggregation monoid with elements of the provenance semiring and $\oplus$ is the aggregation monoid operation (here, addition). The provenance annotation of this tuple is 1 (the 1-element of the semiring) as it is always present. In the security semiring, this is:

$$\text{unclassified} \star 1 + \text{confidential} \star 1 + \text{secret} \star 1$$

with provenance annotation “unclassified”.

Implementation in ProvSQL

ProvSQL carefully follows the theoretical framework of [3] to support provenance computation of aggregate queries. At the moment, aggregates are only supported when they are the final operation performed, though we have plans of adding support of nested aggregates in the future. In order to keep a compact representation of the aggregate values, these are also added to the provenance circuit, using extra gate types for the tensor product and monoid aggregate operators.

4 Conclusion

In this paper, we have presented the tremendous impact that the work of Val Tannen and his collaborators on provenance semirings has had on the design of a practical system for computing the provenance of query results. It is remarkable that so many of these theoretical works lead themselves to practical implementations that can be made efficient.

Though ProvSQL is already usable as is (and, in addition to provenance, provides features for computations of probabilities [17] and Shapley(-like) values [13]), there remains a number of features to implement and optimizations to perform. The most direct given the previous discussion is the support for nested aggregates; recursive queries are unfortunately unlikely to be supported in a near future. Performing all computations in the universal semiring (or the universal m-semiring) has the advantage of being a generic approach, but means that many optimizations that are possible in a given semiring cannot be applied – some engineering is required to allow a user to request ProvSQL to capture only specific forms of provenance and ensure all possible optimizations for this particular algebraic structure are performed.

References


On the Impact of Provenance Semiring Theory on the Design of ProvSQL


