ProvSQL : Gestion de provenance et de probabilités dans PostgreSQL

Pierre Senellart
DI ENS, ENS, CNRS, PSL University
& Inria & LTCI, Télécom ParisTech
Paris, France
pierre@senellart.com

Louis Jachiet
DI ENS, ENS, CNRS, PSL University
& Inria
Paris, France
louis.jachiet@ens.fr

Silviu Maniu
LRI, Université Paris-Sud,
Université Paris-Saclay
Orsay, France
silviu.maniu@lri.fr

Yann Ramusat
ENS, PSL University
Paris, France
yann.ramusat@ens.fr

ABSTRACT

Cette démonstration présente ProvSQL, un module open-source pour le système de gestion de base de données PostgreSQL qui lui ajoute un support pour le calcul de la provenance et des probabilités des résultats de requêtes. Une large gamme de formalismes de provenance sont pris en charge, en particuclier tous ceux capturés par les semi-anneaux de provenance, les semi-anneaux avec monus et la where-provenance. L'évaluation probabiliste de requête est rendue possible par l'usage d'outils de compilation des connaissances, en plus d'approches standard comme l'énumération des mondes possibles et l'échantillonage de Monte-Carlo. ProvSQL prend en charge un sous-ensemble important des requêtes SQL sans agrégation.

1 INTRODUCTION

When evaluating a query, it is often useful to capture meta-information about the result of a query, along with the result itself. The meta-information may indicate where the query result comes from, how it was computed, how many times each result was produced, what probability each result has, etc. Formal tools to capture such meta-information are *data provenance* [7] and *probabilistic databases* [29].

This demonstration introduces ProvSQL, an open-source and lightweight module for the PostgreSQL database management system that adds support for data provenance and probabilistic databases. Many different provenance formalisms have been introduced for relational data provenance. ProvSQL captures most of them: *provenance semirings* [19] that generalize previous formalisms such as *why-provenance* [7], lineages used in view maintenance [10], or the lineage used by the Trio uncertain management system [6]; *m-semirings* [17] that extend provenance semirings with support for negation; *where-provenance* [7], not captured by semiring-based formalisms. In addition, ProvSQL relies on provenance annotations to compute probabilities of query results, in the sense of probabilistic databases.

ProvSQL is application-independent and to our knowledge the first system supporting such a large range of provenance formalisms. In particular, it is the first system with support of m-semiring provenance and of specialization of provenance to arbitrary, user-defined, semirings; it is also the first system that provides an uniform framework to capture both semiring provenance and probabilistic query evaluation. Existing probabilistic relational database engines such as MayBMS [21], Trio [6], or Orion [9], as well as the Perm [18] system for relational provenance management, are all implemented by modifying the internals of a fixed, now obsolete, version of the PostgreSQL DBMS, which results in code that cannot be easily maintained or even compiled on modern operating systems. In contrast, ProvSQL is a lightweight extension to PostgreSQL, easily deployable on an existing PostgreSQL installation, not entangled with database engine code.

Perm [18] and GProM [4] are similar in scope with the non-probabilistic part of ProvSQL: they are systems to capture the provenance of queries in relational databases, with support for different forms of provenance, but not for probabilistic data. To solve the issues with tying Perm to a particular PostgreSQL version, GProM is instead built as a middleware between the user and the DBMS, rewriting queries to compute provenance annotations. The main differences between these systems and ProvSQL are as follows:

- they do not allow computation in user-defined semirings;
- they do not support the semantics of m-semirings;
- they do not distinguish between set (`DISTINCT`) and multiset query semantics in the same way as ProvSQL;
- the native provenance formalism, similar to why-provenance, stores provenance using additional attributes and rows of the result table – this approach is much less compact than the provenance circuit used by ProvSQL.

This demonstration also appeared in [28].

2 FOUNDATIONS

We now give a brief review of the main foundations of ProvSQL: (m-)semiring provenance, where-provenance, probabilistic databases. For a more in-depth review of provenance in relational databases, see [27].

A *semiring* $(K, \mathbb{0}, \mathbb{1}, \oplus, \otimes)$ is a set K with distinguished elements $\mathbb{0}$ and $\mathbb{1}$, along with two binary operators: $\oplus$, an associative and commutative operator, with identity $\mathbb{0}$; $\otimes$, an associative operator, with identity $\mathbb{1}$. We further require $\otimes$ to distribute over $\oplus$, and $\mathbb{0}$ to be annihilating for $\otimes$. Examples of semirings include [19, 20, 23]:

- $(\mathbb{N}, 0, 1, +, \times)$: *counting* semiring;
- ({unclassified, restricted, confidential, secret, top secret}, top secret, unclassified, min, max): *security* semiring;
- $(\mathbb{N} \cup \{\infty\}, \infty, 0, \min, +)$: *tropical* semiring;
- ({positive Boolean funct. over X}, $\bot$, $\top$, $\vee$, $\wedge$): the semiring of *positive Boolean functions* over X.

Fix a semiring $(K, \mathbb{0}, \mathbb{1}, \oplus, \otimes)$ and assume that all tuples of a database come with provenance annotations from K. Consider a query Q from the *positive relational algebra* (selection, projection, renaming, cross product, union). A semantics for the provenance of a tuple $t \in Q(D)$ is defined inductively on the structure of Q, see [19] for details.

Using this inductive definition of semiring provenance, one can use different semirings to compute different meta-information on the output of a query:

counting semiring: the number of times a tuple can be derived;

security semiring: the minimum clearance level required to get a tuple as a result;

tropical semiring: minimum-weight way of deriving a tuple (as when computing shortest paths in a graph);

positive Boolean functions: Boolean provenance as in [22, 27].

Semiring provenance can only be defined for the positive fragment of the relational algebra, excluding non-monotone operations such as difference. However, some semirings can be straightforwardly equipped with a *monus* operator $\ominus$ [17], that must verify some compatibility properties with $\oplus$ [2], capturing non-monotone behavior. This is the case for the Boolean function semiring, which, equipped with the monus operator $a \ominus b = a \wedge \neg b$, forms a *semiring with monus*, or *m-semiring* for short. Once such an m-semiring is defined, provenance of the full relational algebra can be captured in that m-semiring.

One notable provenance formalism that was introduced early on [7] is where-provenance. The where-provenance is a bipartite graph that connects values in the output relation to values in the input relation to indicate where a specific value may come from in the input. [8] showed that where-provenance *cannot* be captured by semiring provenance: there is no semiring for which semiring provenance allows reconstructing the where-provenance of a query.

Assume that every tuple t of a database D come with an independent probability $\Pr(t)$ of being true (the simple model of *tuple-independent databases* [11, 16] that has been widely studied). In such a model, every subdatabase $D' \subseteq D$ (called a *possible world*) is assigned probability $\Pr(D') \coloneqq \prod_{t \in D'} \Pr(t) \times \prod_{t \in D \setminus D'} (1 - \Pr(t))$.

By definition, the probability of a tuple t to be in the result of a query Q over this database is the sum of the probabilities of all sub-databases D' such that $t \in Q(D')$. It was first observed in [20] (see also [29]) that, to compute this probability, one can first compute a provenance annotation for Q as a Boolean function (in the m-semiring of Boolean functions), and then compute the probability of this Boolean function. There are various approaches for this latter part such as direct enumeration of all possible worlds or Monte-Carlo sampling.

However, a more general approach is to resort to general *knowledge compilation* techniques [14]. Knowledge compilation is the problem of transforming Boolean functions of a certain form into another, more tractable, form. Over the years, a wide variety of techniques, results, heuristics, and tools have emerged from the knowledge compilation community. In particular, tools such as c2d [13], Dsharp [26], and d4 [24] compile arbitrary formulas in *conjunctive normal form* into *deterministic decomposable negation normal forms* (d-DNNF [12]), which are Boolean function representations on which probability computation can be done in linear-time. The use of knowledge compilation in a probabilistic database system is a novel contribution of ProvSQL.

3 TERM ALGEBRA CIRCUIT

One original idea of ProvSQL, which allows it to indifferently obtain semiring provenance, m-semiring provenance, where-provenance, and probabilities, is to compute the *provenance circuit* associated with a query in what we call the *provenance term algebra*, using the standard setting of term algebras [5]. The provenance term algebra is a generalization of the universal semiring of [19] and the universal m-semiring of [17]: we simply represent operations performed to obtain a query result as free terms over the following operators:

- $\otimes$ for cross product, as in semirings;
- $\oplus$ for union and duplicate elimination, as in semirings;
- $\ominus$ for set difference, as in m-semirings;
- Π for projection, used for where-provenance;
- $=$ for selection equality between columns, used for where-provenance.

Other query operators (such as selections comparing a column to a constant) are not represented, as they do not impact provenance in any of the provenance formalisms. Provenance terms contain enough information to reconstruct any provenance formalism, or to compute probabilities of query results. Instead of representing these free terms as formulas – the usual approach [19, 29] – we represent them as (arithmetic) circuits, as in [15]. Usually, the circuit representation can be more compact than formulas [1, 31]. ProvSQL thus maintains a provenance term algebra circuit and performs all operations on it, as explained next.

4 THE PROVSQL SYSTEM

We now briefly describe the ProvSQL system. ProvSQL is an open-source software implemented in SQL, PL/pgSQL (the procedural programming language of PostgreSQL), C, and C++, freely available at https://github.com/PierreSenellart/provsql/ and as a Docker container at `inriavalda/provsqldemo`.

ProvSQL is implemented as a *module* of the PostgreSQL database management system[1], which means it can be deployed in a straightforward manner on top of an existing installation of PostgreSQL. ProvSQL has been tested with versions PostgreSQL 9.5, 9.6, and 10 (inclusive), under Linux and MacOS X.

ProvSQL functions by adding a separate column, `provsql`, to all *ProvSQL-aware* tables of the database, which contains *provenance tokens*. Provenance tokens are 128-bit *universally unique identifiers* (UUIDs) that are generated using the `uuid-ossp` PostgreSQL module. These provenance tokens are identifiers of gates in a *provenance circuit* that is constructed and maintained by ProvSQL. Provenance

[1] https://www.postgresql.org/

tokens on base ProvSQL-aware tables are fresh UUIDs and correspond to input gates of the circuits. ProvSQL generates new provenance tokens for results of queries on ProvSQL-aware tables, which identify inner gates of the provenance circuit. UUIDs are assigned in a reproducible manner, so that results to two identical queries are assigned identical provenance tokens.

ProvSQL consists of two distinct parts, presented next in detail: a *query rewriting module* that automatically computes the provenance of query results as gates of a provenance circuit; *user-defined functions* (UDFs) that introduce provenance annotations to existing tables and allow computation of various forms of provenance and probabilities from the provenance circuit.

Query Rewriting Module. PostgreSQL provides *hooks* [25] at different stages of the query execution engine that modules can use to change the behavior of the software. ProvSQL uses one such hook, `planner_hook`, to perform query rewriting after the query has been parsed and before it is sent to the query planner.

The query rewriting module only operates for queries that reference one or more ProvSQL-aware tables. Such a query is rewritten in two steps: first, all direct references to the `provsql` column are ignored – this column is not meant to be directly manipulated by the user; second, a new `provsql` column is generated for the query result: the values contained in this column are provenance tokens identifying gates of the provenance circuit that encode all operations performed to produce this result, in the provenance term algebra.

SQL is a very rich language, and the structure of the query parsed by PostgreSQL reflects this richness. This means that the rewriting needs to take into account every possible feature of the SQL language. The following types of SQL queries are currently supported by ProvSQL:

- simple **`SELECT ... FROM ... WHERE`** queries, i.e., conjunctive queries with multiset semantics;
- **`JOIN`** queries (regular joins only; outer, semijoins, and antijoins are not currently supported);
- **`SELECT`** queries with nested **`SELECT`** subqueries in the **`FROM`** clause;
- **`GROUP BY`** queries (without aggregation) – we note that aggregation support would require moving from semirings to the semimodules of [3], which has not yet been implemented to the best of our knowledge;
- **`SELECT DISTINCT`** queries (i.e., queries with set semantics);
- **`UNION`**'s or **`UNION ALL`**'s of **`SELECT`** queries;
- **`EXCEPT`** of **`SELECT`** queries.

User-Defined Functions. User-defined functions provide a SQL interface to the ProvSQL system, defined within a separate `provsql` schema [30]. In particular, the following user-defined functions are available:

`add_provenance(table):` turns a regular PostgreSQL table into a ProvSQL-aware table, with a `provsql` attribute containing fresh provenance tokens.

`provenance():` returns the provenance token encoding the provenance of the current query.

`create_provenance_mapping(mapping,table,column):` constructs a *provenance mapping* as a new `mapping` table, mapping the provenance tokens of table `table` to the values within the column `column` of that table; provenance mappings are used by further UDFs to assign an elementary value to input provenance tokens.

`view_circuit(token, mapping):` provides a PDF visualization of the subcircuit rooted at `token` of the provenance circuit, using `mapping` to label input gates.

`semiring(token, mapping):` a different UDF is defined for different (m-)semirings, which returns the result of the evaluation of the subcircuit rooted at `token` of the provenance circuit in the corresponding m-semiring, using `mapping` to map input provenance tokens to (m-)semiring elements.

`where_provenance(token):` returns a textual representation of the where-provenance for the provenance token `token`.

`probability_evaluate(token, mapping, method, a):` computes the probability of the provenance token `token`, using `mapping` to map input gates to probabilities; `method` and `a` specify the method used to evaluate the probability (enumeration of possible world, Monte-Carlo sampling, knowledge compilation to a d-DNNF) and additional arguments specifying the number of iterations or the knowledge compiler used (`c2d` [13][2], `d4` [24][3], or `dsharp` [26][4]).

Table 1: Table Personnel **for the personnel of an intelligence agency, used as a running example (from [27])**

id	name	position	city	classification	
1	John	Director	New York	unclassified	t_1
2	Paul	Janitor	New York	restricted	t_2
3	Dave	Analyst	Paris	confidential	t_3
4	Ellen	Field agent	Berlin	secret	t_4
5	Magdalen	Double agent	Paris	top_secret	t_5
6	Nancy	HR	Paris	restricted	t_6
7	Susan	Analyst	Berlin	secret	t_7

5 DEMONSTRATION SCENARIO

Our demonstration scenario will use three different databases:

- the example Table 1 (taken from [27]), the list of personnel from a fictitious intelligence agency, small and simple enough to be easily understandable in a demonstration setting;
- a synthetic large probabilistic graph, used to demonstrate the difference of performances between various methods;
- a real-world database of a large public transport network[5] (involving several tables, with more than 10 million tuples) used to showcase realistic examples.

Most of the demonstration is done within PostgreSQL's `psql` command-line client; a special Web-based GUI is provided for visualizing where-provenance, and circuit visualization uses an external PDF viewer.

[2] http://reasoning.cs.ucla.edu/c2d/download.php
[3] http://www.cril.univ-artois.fr/KC/d4.html
[4] https://bitbucket.org/haz/dsharp
[5] https://opendata.stif.info/explore/dataset/offre-horaires-tc-gtfs-idf/table/

We will illustrate the features of ProvSQL on Table 1 using the following example queries, also from [27], as well as variations thereof:

- A monotone query Q_1 that asks for cities with at least two persons in the agency:
```
SELECT DISTINCT P1.city
FROM Personnel P1 JOIN Personnel P2
  ON P1.city = P2.city
WHERE P1.id < P2.id
```
- A non-monotone query Q_2 that asks for cities with exactly one person in the agency:
```
  SELECT DISTINCT city FROM Personnel
EXCEPT
  SELECT DISTINCT P1.city
  FROM Personnel P1 JOIN Personnel P2
    ON P1.city = P2.city
  WHERE P1.city = P2.city AND P1.id < P2.id
```

The user will also be free to write her own queries, helping her get familiar with ProvSQL's features and limitations.

First, we will show how to use the `add_provenance` UDF to add provenance support to an existing table, assigning fresh provenance tokens to all tuples, and the `create_provenance_mapping` UDF to create provenance mappings that will be used in further queries. Basic SQL queries will be run to show that every query results is annotated with provenance annotations.

Second, we will show the result of executing Q_1 or Q_2 on the table, as a table with new provenance tokens. Using the `view_circuit` UDF, we will display the provenance circuit corresponding to these provenance tokens, and will also show how this circuit is stored in the database. We will then illustrate how this circuit can be used to compute provenance annotations in any provenance semiring (or, for the case of the non-monotone query Q_2, in any m-semiring): specifically, we will show how to compute, for instance, the security level needed to access each tuple in the result of Q_1, or how to get a Boolean formula expressing how the result of Q_2 depends on the presence or absence of individual tuples in the input, using UDFs defined for each provenance (m-)semiring. If the user so decides, we can also demonstrate implementing a new semiring (such as the tropical semiring) by writing the corresponding UDF. Finally, we will ask ProvSQL to display the where-provenance of query results using the `where_provenance` UDF, through a Web-based interface: when the user hovers over a value in a query result, the system highlights values in the input that produced this data value.

Third, we will move to probabilistic query evaluation, modifying the input table to add probability values onto individual tuples. Using the `probability_evaluate` UDF, we will show how the probability of individual query results can be computed by different approaches: naive enumeration of possible worlds, Monte-Carlo sampling (with a parameter specifying the number of samples used), and knowledge compilation using either `c2d`, `d4`, or `dsharp`. Probabilistic query evaluation on the synthetic probabilistic graph will illustrate the difference of performances between these methods.

Last, we consider realistic examples. On the transport network dataset, queries can be run, for instance, to compute all stations reachable with at most one transfer from some station. After the computation is performed, the provenance of the result can be used to obtain different kinds of meta-information, without rerunning the query: Is the station reachable for a wheelchair user (using evaluation in the Boolean semiring and accessibility attributes on lines and stations)? What is the probability to encounter a failure on a route given failure probability on each line?

A 12-minute video preview of part of the demonstration is available at https://youtu.be/iqzSNfGHbEE?vq=hd1080.

Acknowledgment. We are grateful to Peter Buneman for discussions on integration of where-provenance into ProvSQL, and to Boris Glavic for insight on Perm and GProM. We also acknowledge Antoine Amarilli, Adnan Darwiche, Dan Olteanu, Charles Paperman, for feedback on the ProvSQL system.

REFERENCES

[1] A. Amarilli, P. Bourhis, and P. Senellart. Tractable lineages on treelike instances: Limits and extensions. In *PODS*, pages 355–370, 2016.
[2] K. Amer. Equationally complete classes of commutative monoids with monus. *Algebra Universalis*, 18(1):129–131, 1984.
[3] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *PODS*, pages 153–164, 2011.
[4] B. S. Arab, S. Feng, B. Glavic, S. Lee, X. Niu, and Q. Zeng. GProM - A swiss army knife for your provenance needs. *IEEE Data Eng. Bull.*, 41(1):51–62, 2018.
[5] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.
[6] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, pages 953–964, 2006.
[7] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.
[8] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.
[9] R. Cheng, S. Singh, and S. Prabhakar. U-DBMS: A database system for managing constantly-evolving data. In *VLDB*, pages 1271–1274, 2005.
[10] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *ICDE*, pages 367–378, 2000.
[11] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, 2007.
[12] A. Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *J. Applied Non-Classical Logics*, 11(1-2):11–34, 2001.
[13] A. Darwiche. New advances in compiling CNF to decomposable negation normal form. In *ECAI*, pages 318–322, 2004.
[14] A. Darwiche and P. Marquis. A knowledge compilation map. *J. Artificial Intelligence Research*, 17(1):229–264, 2002.
[15] D. Deutch, T. Milo, S. Roy, and V. Tannen. Circuits for Datalog provenance. In *ICDT*, pages 201–212, 2014.
[16] N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM TOIS*, 15(1):32–66, 1997.
[17] F. Geerts and A. Poggi. On database query languages for K-relations. *J. Applied Logic*, 8(2):173–185, 2010.
[18] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, pages 174–185, 2009.
[19] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.
[20] T. J. Green and V. Tannen. Models for incomplete and probabilistic information. *IEEE Data Eng. Bull.*, 29(1):17–24, 2006.
[21] J. Huang, L. Antova, C. Koch, and D. Olteanu. MayBMS: a probabilistic database management system. In *SIGMOD*, pages 1071–1074, 2009.
[22] T. Imielinski and W. L. Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.
[23] G. Karvounarakis and T. J. Green. Semiring-annotated data: queries and provenance? *ACM SIGMOD Record*, 41(3):5–14, 2012.
[24] J.-M. Lagniez and P. Marquis. An improved decision-DNNF compiler. In *IJCAI*, pages 667–673, 2017.
[25] G. Lelarge. Hooks in PostgreSQL, 2012. Talk at FOSDEM 2012 and pgCon 2012.
[26] C. J. Muise, S. A. McIlraith, J. C. Beck, and E. I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In *Canadian Conference on AI*, pages 356–361, 2012.
[27] P. Senellart. Provenance and probabilities in relational databases: From theory to practice. *SIGMOD Record*, 46(4):5–15, 2017.
[28] P. Senellart, L. Jachiet, S. Maniu, and Y. Ramusat. ProvSQL: Provenance and probability management in PostgreSQL. *PVLDB*, 11(12), 2018. Demonstration.
[29] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases.* Morgan & Claypool, 2011.

[30] The PostgreSQL Global Development Group. *PostgreSQL 10.1 Documentation*, chapter 5.8 (Schemas). 2017.
[31] I. Wegener. *The complexity of Boolean functions.* Wiley, 1987.