ProvSQL: Provenance and Probability Management in PostgreSQL

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

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

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

ABSTRACT

This demonstration showcases ProvSQL, an open-source module for the PostgreSQL database management system that adds support for computation of provenance and probabilities of query results. A large range of provenance formalisms are supported, including all those captured by provenance semirings, provenance semirings with monus, as well as where-provenance. Probabilistic query evaluation is made possible through the use of knowledge compilation tools, in addition to standard approaches such as enumeration of possible worlds and Monte-Carlo sampling. ProvSQL supports a large subset of non-aggregate SQL queries.

ACM Reference Format:
Pierre Senellart, Silviu Maniu, and Yann Ramusat. 2018. ProvSQL: Provenance and Probability Management in PostgreSQL. In *Proceedings of Submitted for Publication.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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. [28] Formal tools to capture such meta-information are *data provenance* [7] and *probabilistic databases* [29].

This demonstration introduces ProvSQL, an open-source 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 those: *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 those. 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 a large range of provenance formalisms. In contrast with Orchestra [18], it is freely available. In contrast with probabilistic relational database engines such as MayBMS [21], Trio [6], or Orion [9], it is a lightweight extension to PostgreSQL, easily deployable on an existing PostgreSQL installation, not entangled with database engine code. ProvSQL supports a large range of SQL queries. We believe it can be a tool of choice to develop applications needing meta-information on query results, as well as a useful system to teach about data provenance and probabilistic databases.

ProvSQL was first informally presented at the 2017 Dagstuhl workshop on *Recent Trends in Knowledge Compilation*. It has not been demonstrated in any formal setting so far. Some of the background material in Section 2 of is taken from [28].

We first present some of the foundations of provenance and probabilistic databases in Section 2, and the central notion of provenance term algebra circuit in Section 3. The ProvSQL system is then presented in Section 4, while the demonstration scenario is given in Section 5.

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

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 [28].

Provenance semirings. 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* [1] (selection, projection, renaming, cross product, union). A semantics for the provenance

Submitted for Publication, January 2018,
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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, 28].

Semirings with monus. 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 $\ominus$ [4, 17], that must verify some compatibility properties with $\oplus$ [4], 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.

Where-provenance. 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.

Probabilistic databases. 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, 25] that has been widely studied). In such a model, every subdatabase $D' \subseteq D$ (called a *possible world*) is assigned probability

$$\Pr(D') := \prod_{t \in D'} \Pr(t) \times \prod_{t \in D \setminus D'} (1 - \Pr(t)).$$

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] 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 [27], 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.

3 TERM ALGEBRA CIRCUIT

The main 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 [2, 15], which can be more compact [3, 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/.

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 of PostgreSQL from 9.4 to 10.1 (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 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.

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

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.

4.1 Query Rewriting Module

PostgreSQL provides *hooks* [26] 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);
- `SELECT DISTINCT` queries (i.e., queries with set semantics);
- `UNION`'s or `UNION ALL`'s of `SELECT` queries;
- `EXCEPT` of `SELECT` queries.

4.2 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` [27][4]).

Table 1: Table Personnel **for the personnel of an intelligence agency, used as a running example (from [28])**

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

We now describe our demonstration scenario, which will mostly use the example table in Table 1 (taken from [28]), the list of personnel from a fictitious intelligence agency, small and simple enough to be easily understandable in a demonstration setting. In addition, we will offer the user the possibility of running the queries on larger, more realistic datasets, to get a feel of the use and efficiency of ProvSQL in more practical applications.

Apart from visualization of provenance circuits, which relies on an external PDF viewer, the entire demonstration is done within PostgreSQL's `psql` command-line client.

We will illustrate the features of ProvSQL on Table 1 using the following example queries, also from [28], 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
```

[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

```
  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 of the table, 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.

Third, we will move on 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 a larger dataset will illustrate the difference of performances between these various methods.

Last, users interested in implementation aspects will be able to inspect the code; users who wish to re-use ProvSQL on their own will be shown the simple installation procedure.

A 12-minute video preview 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. We also acknowledge the many colleagues we discussed the ProvSQL system with and who gave helpful feedback, in particular Antoine Amarilli, Adnan Darwiche, Dan Olteanu, Charles Paperman.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.
[2] Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. 2015. Provenance Circuits for Trees and Treelike Instances. In *ICALP*.
[3] Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. 2016. Tractable Lineages on Treelike Instances: Limits and Extensions. In *PODS*.
[4] K. Amer. 1984. *Algebra Universalis* 18, 1 (1984).
[5] Franz Baader and Tobias Nipkow. 1998. *Term rewriting and all that.* Cambridge University Press.
[6] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. 2006. ULDBs: Databases with uncertainty and lineage. In *VLDB*.
[7] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *ICDT*.
[8] James Cheney, Laura Chiticariu, and Wang Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2009).
[9] Reynold Cheng, Sarvjeet Singh, and Sunil Prabhakar. 2005. U-DBMS: A Database System for Managing Constantly-Evolving Data. In *VLDB*.
[10] Yingwei Cui and Jennifer Widom. 2000. Practical lineage tracing in data warehouses. In *ICDE*.
[11] Nilesh Dalvi and Dan Suciu. 2007. Efficient query evaluation on probabilistic databases. *The VLDB Journal* 16, 4 (2007).
[12] Adnan Darwiche. 2001. On the tractable counting of theory models and its application to truth maintenance and belief revision. *J. Applied Non-Classical Logics* 11, 1-2 (2001).
[13] Adnan Darwiche. 2004. New advances in compiling CNF to decomposable negation normal form. In *ECAI*.
[14] Adnan Darwiche and Pierre Marquis. 2002. A knowledge compilation map. *J. Artificial Intelligence Research* 17, 1 (2002).
[15] Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. 2014. Circuits for Datalog Provenance.. In *ICDT*.
[16] Norbert Fuhr and Thomas Rölleke. 1997. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Transactions on Information Systems* 15, 1 (1997).
[17] Floris Geerts and Antonella Poggi. 2010. On database query languages for K-relations. *J. Applied Logic* 8, 2 (2010).
[18] Grigoris Green, Todd J. and Karvounarakis, Zach Ives, and Val Tannen. 2010. Provenance in ORCHESTRA. *IEEE Data Eng. Bull* 33, 3 (2010).
[19] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *PODS*.
[20] Todd J. Green and Val Tannen. 2006. Models for Incomplete and Probabilistic Information. *IEEE Data Eng. Bull.* 29, 1 (2006).
[21] Jiewen Huang, Lyublena Antova, Christoph Koch, and Dan Olteanu. 2009. MayBMS: a probabilistic database management system. In *SIGMOD*.
[22] Tomasz Imielinski and Witold Lipski Jr. 1984. Incomplete Information in Relational Databases. *J. ACM* 31, 4 (1984).
[23] Grigoris Karvounarakis and Todd J Green. 2012. Semiring-annotated data: queries and provenance? *ACM SIGMOD Record* 41, 3 (2012).
[24] Jean-Marie Lagniez and Pierre Marquis. 2017. An improved decision-DNNF compiler. In *IJCAI*.
[25] Laks VS Lakshmanan, Nicola Leone, Robert Ross, and Venkatramanan Siva Subrahmanian. 1997. Probview: A flexible probabilistic database system. *ACM Transactions on Database Systems* 22, 3 (1997).
[26] Guillaume Lelarge. 2012. Hooks in PostgreSQL. (2012). https://github.com/gleu/Hooks-in-PostgreSQL Talk at FOSDEM 2012 and pgCon 2012.
[27] Christian J Muise, Sheila A McIlraith, J Christopher Beck, and Eric I Hsu. 2012. Dsharp: Fast d-DNNF Compilation with sharpSAT.. In *Canadian Conference on AI*.
[28] Pierre Senellart. 2017. Provenance and Probabilities in Relational Databases: From Theory to Practice. *SIGMOD Record* 46, 4 (2017).
[29] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. 2011. *Probabilistic Databases.* Morgan & Claypool.
[30] The PostgreSQL Global Development Group. 2017. *PostgreSQL 10.1 Documentation.* Chapter 5.8 (Schemas). https://www.postgresql.org/docs/10/static/ddl-schemas.html
[31] Ingo Wegener. 1987. *The complexity of Boolean functions.* Wiley.