UNIVERSITÉ
PARIS SCIENCES
& LETTRES

Optimizing Probabilistic Query Evaluation in ProvSQL

Pierre Senellart

SINFRA, 30 June 2026

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Probabilistic Query Evaluation

Data is uncertain

- Real-world data rarely comes with certainty:
  - sensor readings, image / text understood by an **ML model**
  - **information extraction** from the Web
  - **data integration**, deduplication, crowdsourcing

Data is uncertain

- Real-world data rarely comes with certainty:
  - sensor readings, image / text understood by an **ML model**
  - **information extraction** from the Web
  - **data integration**, deduplication, crowdsourcing
- Pragmatic model: attach to each fact a **probability of being correct**

Data is uncertain

- Real-world data rarely comes with certainty:
  - sensor readings, image / text understood by an **ML model**
  - **information extraction** from the Web
  - **data integration**, deduplication, crowdsourcing
- Pragmatic model: attach to each fact a **probability of being correct**
- We still want to **query** the data… and get, for each answer, the **probability that it holds**

Data is uncertain

- Real-world data rarely comes with certainty:
  - sensor readings, image / text understood by an **ML model**
  - **information extraction** from the Web
  - **data integration**, deduplication, crowdsourcing
- Pragmatic model: attach to each fact a **probability of being correct**
- We still want to **query** the data… and get, for each answer, the **probability that it holds**

This talk

How does **ProvSQL** [Sen et al., 2026], a provenance and probability extension of PostgreSQL, compute these probabilities **efficiently** – given that the problem is hard in general?

A (tiny) probabilistic database

Each input tuple carries an **independent** probability of being present – a **tuple-independent** database

AtRisk

person	Pr
Alice	0.6
Bob	0.5
Carol	0.5

TestedPos

person	Pr
Alice	0.3
Carol	0.8

Met

p	*q*	Pr
Alice	Carol	0.7
Bob	Carol	0.4

A (tiny) probabilistic database

Each input tuple carries an **independent** probability of being present – a **tuple-independent** database

AtRisk

person	Pr
Alice	0.6
Bob	0.5
Carol	0.5

TestedPos

person	Pr
Alice	0.3
Carol	0.8

Met

p	*q*	Pr
Alice	Carol	0.7
Bob	Carol	0.4

Each tuple = an independent Boolean random variable:

$$r_a, r_b, r_c,\ t_a, t_c,\ m_{ac}, m_{bc}$$

A (tiny) probabilistic database

Each input tuple carries an **independent** probability of being present – a **tuple-independent** database

AtRisk

person	Pr
Alice	0.6
Bob	0.5
Carol	0.5

TestedPos

person	Pr
Alice	0.3
Carol	0.8

Met

p	*q*	Pr
Alice	Carol	0.7
Bob	Carol	0.4

Each tuple = an independent Boolean random variable:

$$r_a, r_b, r_c,\ t_a, t_c,\ m_{ac}, m_{bc}$$

ProvSQL also handles **correlated** inputs – block-independent (BID) tables, view-derived correlations, a **continuous** tier – but we keep TID for the running example

From a query to a Boolean formula

Query Q_1: *is there someone at risk who tested positive?*

$$\exists x \ \ \mathrm{AtRisk}(x) \wedge \mathrm{TestedPos}(x)$$

From a query to a Boolean formula

Query Q_1: *is there someone at risk who tested positive?*

$$\exists x \;\; \mathrm{AtRisk}(x) \wedge \mathrm{TestedPos}(x)$$

Its **Boolean provenance** records which input tuples make it true:

$$\varphi_1 \;=\; \underbrace{(r_\mathrm{a} \wedge t_\mathrm{a})}_{\text{Alice}} \vee \underbrace{(r_\mathrm{c} \wedge t_\mathrm{c})}_{\text{Carol}}$$

From a query to a Boolean formula

Query Q_1: *is there someone at risk who tested positive?*

$$\exists x \;\; \mathrm{AtRisk}(x) \wedge \mathrm{TestedPos}(x)$$

Its **Boolean provenance** records which input tuples make it true:

$$\varphi_1 \;=\; \underbrace{(r_\mathrm{a} \wedge t_\mathrm{a})}_{\text{Alice}} \vee \underbrace{(r_\mathrm{c} \wedge t_\mathrm{c})}_{\text{Carol}}$$

The answer probability is the probability of this Boolean formula:

$$\Pr(\varphi_1) = 1 - (1 - 0.6 \cdot 0.3)(1 - 0.5 \cdot 0.8) = 0.508$$

From a query to a Boolean formula

Query Q_1: *is there someone at risk who tested positive?*

$$\exists x \;\; \mathrm{AtRisk}(x) \wedge \mathrm{TestedPos}(x)$$

Its **Boolean provenance** records which input tuples make it true:

$$\varphi_1 \;=\; \underbrace{(r_\mathrm{a} \wedge t_\mathrm{a})}_{\text{Alice}} \vee \underbrace{(r_\mathrm{c} \wedge t_\mathrm{c})}_{\text{Carol}}$$

The answer probability is the probability of this Boolean formula:

$$\Pr(\varphi_1) = 1 - (1 - 0.6 \cdot 0.3)(1 - 0.5 \cdot 0.8) = 0.508$$

Easy here: the two clauses share **no variable** $\Rightarrow$ a **read-once** formula $\Rightarrow$ multiply and combine, **linear time**

Probabilistic query evaluation, in general

PQE

Given a query and a tuple-independent database, compute the **probability** of each answer $=$ probability of its **Boolean provenance** φ, a Boolean function over independent variables

Probabilistic query evaluation, in general

PQE

Given a query and a tuple-independent database, compute the **probability** of each answer $=$ probability of its **Boolean provenance** φ, a Boolean function over independent variables

$$\Pr(\varphi) = \sum_{\nu \models \varphi} \prod_{x:\, \nu(x)=1} p_x \prod_{x:\, \nu(x)=0} (1 - p_x)$$

Probabilistic query evaluation, in general

PQE

Given a query and a tuple-independent database, compute the **probability** of each answer = probability of its **Boolean provenance** φ, a Boolean function over independent variables

$$\Pr(\varphi) = \sum_{\nu \models \varphi} \prod_{x:\nu(x)=1} p_x \prod_{x:\nu(x)=0} (1 - p_x)$$

- In ProvSQL the user just writes `probability_evaluate(provenance())`

Probabilistic query evaluation, in general

PQE

Given a query and a tuple-independent database, compute the **probability** of each answer = probability of its **Boolean provenance** φ, a Boolean function over independent variables

$$\Pr(\varphi) = \sum_{\nu \models \varphi} \prod_{x:\nu(x)=1} p_x \prod_{x:\nu(x)=0} (1 - p_x)$$

- In ProvSQL the user just writes `probability_evaluate(provenance())`
- Naively: sum over 2^n **possible worlds** – hopeless

Probabilistic query evaluation, in general

PQE

Given a query and a tuple-independent database, compute the **probability** of each answer = probability of its **Boolean provenance** φ, a Boolean function over independent variables

$$\Pr(\varphi) = \sum_{\nu \models \varphi} \prod_{x:\nu(x)=1} p_x \prod_{x:\nu(x)=0} (1 - p_x)$$

- In ProvSQL the user just writes `probability_evaluate(provenance())`
- Naively: sum over 2^n **possible worlds** – hopeless
- With all $p_x = \frac{1}{2}$: $\Pr(\varphi) = \frac{\#\{\text{satisfying assignments}\}}{2^n}$ – this is **model counting**

Why it is hard

Query Q_2: *is there someone at risk who met someone positive?*

$$\exists x, y \;\; \mathrm{AtRisk}(x) \wedge \mathrm{Met}(x, y) \wedge \mathrm{TestedPos}(y)$$

Why it is hard

Query Q_2: *is there someone at risk who met someone positive?*

$$\exists x, y \;\; \mathrm{AtRisk}(x) \wedge \mathrm{Met}(x, y) \wedge \mathrm{TestedPos}(y)$$

$$\varphi_2 = (r_a \wedge m_{ac} \wedge t_c) \;\vee\; (r_b \wedge m_{bc} \wedge t_c)$$

- Now t_c **is shared** across clauses – **not** read-once; the clauses are correlated

Why it is hard

Query Q_2**:** *is there someone at risk who met someone positive?*

$$\exists x, y \;\; \mathsf{AtRisk}(x) \wedge \mathsf{Met}(x, y) \wedge \mathsf{TestedPos}(y)$$

$$\varphi_2 = (r_\mathsf{a} \wedge m_\mathsf{ac} \wedge t_\mathsf{c}) \;\vee\; (r_\mathsf{b} \wedge m_\mathsf{bc} \wedge t_\mathsf{c})$$

- Now t_c **is shared** across clauses – **not** read-once; the clauses are correlated
- Computing $\Pr(\varphi_2)$ at scale is **#P-hard** [Dalvi and Suciu, 2007] – as hard as counting solutions of a SAT formula [Valiant, 1979]

Why it is hard

Query Q_2**:** *is there someone at risk who met someone positive?*

$$\exists x, y \;\; \mathrm{AtRisk}(x) \wedge \mathrm{Met}(x, y) \wedge \mathrm{TestedPos}(y)$$

$$\varphi_2 = (r_a \wedge m_{ac} \wedge t_c) \;\vee\; (r_b \wedge m_{bc} \wedge t_c)$$

- Now t_c **is shared** across clauses – **not** read-once; the clauses are correlated
- Computing $\Pr(\varphi_2)$ at scale is **#P-hard** [Dalvi and Suciu, 2007] – as hard as counting solutions of a SAT formula [Valiant, 1979]

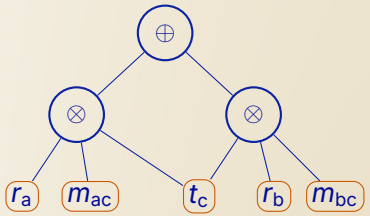
Dichotomy [Dalvi and Suciu, 2012]

Every (union of) conjunctive query is **either** computable in **polynomial time** (**safe**) **or** #P-hard – and we can tell which from the query

Optimizing PQE in ProvSQL

ProvSQL: provenance circuits inside PostgreSQL

- A **lightweight extension** of PostgreSQL [Sen et al., 2026]
- Every query builds, by **query rewriting**, a **provenance circuit**: a DAG of $\oplus$/$\otimes$ gates over the input tuples
- One circuit, **many uses**: different **provenance** notions (why, how…), **Shapley values**, and – our focus – **probabilities**
- PQE $=$ **build** this circuit $+$ **evaluate** it as a Boolean function



Boolean provenance φ_2 (t_c shared)

Two ways to go fast

Rather than one general-purpose algorithm, ProvSQL adapts to each query and circuit, combining:

1. Exploiting structure

Recognize that the query / circuit falls in a **tractable island**, and route it to a **dedicated linear-time** method – turning a #P-hard problem into an easy one

2. Picking the cheapest method

When no obvious structure: a **cost-based optimizer** chooses, per circuit, the cheapest algorithm meeting the **guarantee** you asked for

Two ways to go fast

Rather than one general-purpose algorithm, ProvSQL adapts to each query and circuit, combining:

1. Exploiting structure

Recognize that the query / circuit falls in a **tractable island**, and route it to a **dedicated linear-time** method – turning a #P-hard problem into an easy one

2. Picking the cheapest method

When no obvious structure: a **cost-based optimizer** chooses, per circuit, the cheapest algorithm meeting the **guarantee** you asked for

Both are **transparent**: same exact answer, automatically

Exploiting structure: tractable islands

Each is a **sufficient condition** ProvSQL recognizes and routes to a dedicated, certified mechanism

Condition on the query / data	Cost	ProvSQL route
hierarchical CQ, no self-joins	$\Theta(\lvert D\rvert)$	safe-query rewrite $\rightarrow$ read-once [Dalvi and Suciu, 2007]
inversion-free UCQ (self-joins ok)	$O(\lvert D\rvert)$	structured d-DNNF [Jha and Suciu, 2011]
safe UCQ via Möbius cancellation	$O(\lvert D\rvert^e)$	signed Möbius sweep [Dalvi and Suciu, 2012]
provenance of treewidth $\leqslant k$	$2^{O(k)}\lvert D\rvert$	tree decomposition [Amarilli et al., 2020]
reachability on treelike graphs	$2^{O(k^2)}\lvert D\rvert$	network-reliability compiler [Amarilli et al., 2015]
bounded joint data+annotation width	$2^{O(k^e)}\lvert D\rvert$	joint-width compiler [Amarilli, 2016]

Exploiting structure: tractable islands

Each is a **sufficient condition** ProvSQL recognizes and routes to a dedicated, certified mechanism

Condition on the query / data	Cost	ProvSQL route
hierarchical CQ, no self-joins	$\Theta(\|D\|)$	safe-query rewrite $\rightarrow$ read-once [Dalvi and Suciu, 2007]
inversion-free UCQ (self-joins ok)	$O(\|D\|)$	structured d-DNNF [Jha and Suciu, 2011]
safe UCQ via Möbius cancellation	$O(\|D\|^e)$	signed Möbius sweep [Dalvi and Suciu, 2012]
provenance of treewidth $\leqslant k$	$2^{O(k)}\|D\|$	tree decomposition [Amarilli et al., 2020]
reachability on treelike graphs	$2^{O(k^2)}\|D\|$	network-reliability compiler [Amarilli et al., 2015]
bounded joint data+annotation width	$2^{O(k^e)}\|D\|$	joint-width compiler [Amarilli, 2016]

The compilers emit **deterministic, decomposable circuits** (d-Ds) – with a **certificate** that makes the downstream probability a single **linear** pass. No external tool

The cost optimizer: ask for a *guarantee*, not a method

You request **what you need**; ProvSQL decides **how**:

exact	the true probability (default)
relative (ε, δ)	within a factor $1 \pm \varepsilon$, right for **rare events**
additive (ε, δ)	within $\pm\varepsilon$

The cost optimizer: ask for a *guarantee*, not a method

You request **what you need**; ProvSQL decides **how**:

exact	the true probability (default)
relative (ε, δ)	within a factor $1 \pm \varepsilon$, right for **rare events**
additive (ε, δ)	within $\pm\varepsilon$

A **cost-based chooser** then runs the **cheapest** method that meets the request, per query:

- tolerances **nest**: an approximate request still returns the **exact** value when an exact method is cheaper;

The cost optimizer: ask for a *guarantee*, not a method

You request **what you need**; ProvSQL decides **how**:

exact	the true probability (default)
relative (ε, δ)	within a factor $1 \pm \varepsilon$, right for **rare events**
additive (ε, δ)	within $\pm\varepsilon$

A **cost-based chooser** then runs the **cheapest** method that meets the request, per query:

- tolerances **nest**: an approximate request still returns the **exact** value when an exact method is cheaper;
- hard-to-predict methods run **under a budget** and **escalate** instead of stalling on a pathological circuit

One circuit, many methods – the chooser picks

A portfolio of 13 methods: the optimizer auto-selects 11 per **setting** and guarantee, the setting itself fixes 2 more at build time *(measured picks)*.

setting	exact	rel. ε=0.1	rel. ε=0.3	additive
read-once	**independent**	independent	independent	independent
safe self-join–free	**sq-rewrite**	sq-rewrite	sq-rewrite	sq-rewrite
safe self-join	**inv.-free**	inv.-free	inv.-free	inv.-free
safe UCQ (Möbius)	**möbius**	möbius	möbius	möbius
bounded joint width	**bounded-jw**	bounded-jw	bounded-jw	bounded-jw
low-treewidth circuit	**tree-decomp.**	**d-tree**	d-tree	**monte-carlo**
small DNF	**sieve**	sieve	**karp-luby**	sieve
big clique	**compilation:d4**	d-tree	**stopping-rule**	monte-carlo
few tuples	**possible-worlds**	possible-worlds	possible-worlds	possible-worlds

One circuit, many methods – the chooser picks

A portfolio of 13 methods: the optimizer auto-selects 11 per **setting** and guarantee, the setting itself fixes 2 more at build time *(measured picks)*.

setting	exact	rel. ε=0.1	rel. ε=0.3	additive
read-once	**independent**	independent	independent	independent
safe self-join–free	**sq-rewrite**	sq-rewrite	sq-rewrite	sq-rewrite
safe self-join	**inv.-free**	inv.-free	inv.-free	inv.-free
safe UCQ (Möbius)	**möbius**	möbius	möbius	möbius
bounded joint width	**bounded-jw**	bounded-jw	bounded-jw	bounded-jw
low-treewidth circuit	**tree-decomp.**	**d-tree**	d-tree	**monte-carlo**
small DNF	**sieve**	sieve	**karp-luby**	sieve
big clique	**compilation:d4**	d-tree	**stopping-rule**	monte-carlo
few tuples	**possible-worlds**	possible-worlds	possible-worlds	possible-worlds

- exact **linear** methods for safe / low-width provenance;
- **FPRAS** samplers (Karp–Luby [Karp et al., 1989], stopping-rule) only where needed – never on a cell it can do exactly

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Wrap - Up

Takeaways

- PQE – query answers **with probabilities** – is #**P-hard** in general, but riddled with **tractable islands**

Takeaways

- PQE – query answers **with probabilities** – is #**P-hard** in general, but riddled with **tractable islands**
- **ProvSQL** makes it practical inside PostgreSQL by:
  - recognizing structure and **routing** to dedicated linear-time compilers (safe queries, circuit treewidth, joint data+annotation width, reachability);
  - a **cost-based optimizer**: *ask for a guarantee, not a method* – exact when cheap, approximate only when needed, never hanging

Takeaways

- PQE – query answers **with probabilities** – is #**P-hard** in general, but riddled with **tractable islands**
- **ProvSQL** makes it practical inside PostgreSQL by:
  - recognizing structure and **routing** to dedicated linear-time compilers (safe queries, circuit treewidth, joint data+annotation width, reachability);
  - a **cost-based optimizer**: *ask for a guarantee, not a method* – exact when cheap, approximate only when needed, never hanging
- Exact (or sometimes approximate) PQE is **feasible at scale** on the structure real data tends to have

Takeaways

- PQE – query answers **with probabilities** – is #**P-hard** in general, but riddled with **tractable islands**
- **ProvSQL** makes it practical inside PostgreSQL by:
  - recognizing structure and **routing** to dedicated linear-time compilers (safe queries, circuit treewidth, joint data+annotation width, reachability);
  - a **cost-based optimizer**: *ask for a guarantee, not a method* – exact when cheap, approximate only when needed, never hanging
- Exact (or sometimes approximate) PQE is **feasible at scale** on the structure real data tends to have

Open source – try it: `https://provsql.org/`

In your browser, no install: `https://provsql.org/playground/`

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Thank you!

References I

Antoine Amarilli. *Leveraging the Structure of Uncertain Data*. PhD thesis, Télécom ParisTech, 2016. tel-01345836.

Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. Provenance circuits for trees and treelike instances. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, Lecture Notes in Computer Science, pages 56–68. Springer, 2015. doi: 10.1007/978-3-662-47666-6_5.

Antoine Amarilli, Florent Capelli, Mikaël Monet, and Pierre Senellart. Connecting knowledge compilation classes and width parameters. *Theory Comput. Syst.*, 64(5):861–914, 2020. doi: 10.1007/S00224-019-09930-2.

Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4):523–544, 2007. doi: 10.1007/s00778-006-0004-3.

Nilesh N. Dalvi and Dan Suciu. The dichotomy of probabilistic inference for unions of conjunctive queries. *J. ACM*, 59(6):30:1–30:87, 2012. doi: 10.1145/2395116.2395119.

References II

Abhay Kumar Jha and Dan Suciu. Knowledge compilation meets database theory: Compiling queries to decision diagrams. In *Database Theory - ICDT 2011, 14th International Conference, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 162–173. ACM, 2011. doi: 10.1145/1938551.1938574.

Richard M. Karp, Michael Luby, and Neal Madras. Monte-carlo approximation algorithms for enumeration problems. *J. Algorithms*, 10(3):429–448, 1989. doi: 10.1016/0196-6774(89)90038-2.

Aryak Sen, Silviu Maniu, and Pierre Senellart. ProvSQL: A general system for keeping track of the provenance and probability of data. In *Proc. IEEE 42nd International Conference on Data Engineering (ICDE), Montréal, Canada, May 2026*, 2026. Also arXiv abs/2504.12058.

Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8 (3):410–421, 1979. doi: 10.1137/0208032.