Databases

Views & Provenance

Pierre Senellart

PSL

13 May 2026

Outline

Views

- Views are queries given a name
- They are used like tables in other queries
- Semantics: the view is replaced by the result of evaluating the corresponding query

In SQL

CREATE VIEW v **AS SELECT** ...

CREATE MATERIALIZED VIEW v **AS SELECT** ...

(The latter is a PostgreSQL extension.)

Virtual vs. materialized views

- A view can be virtual (the default) or materialized
- No difference in semantics
- Operational difference, with an impact on query evaluation efficiency:

  virtual view: the query defining the view is re-evaluated every time the view is used in a query

  materialized view: the query defining the view is evaluated once, at view creation time, and the result is stored in an auxiliary table; this table is used directly every time the view is referenced

Why use views?

Logical independence: an application accesses views without needing to know the actual organization of the data (which may change transparently, by redefining views)

Access control: different access rights can be granted to base tables and to views, so that a given user or application only sees a restricted slice of the database

Data integration: views can be defined to bring together data from several sources with different relational schemas

Optimization: materialized views can be defined for frequent queries, to avoid re-evaluating them every time

Answering queries using views

- Problem: given a query on the base tables and a set of views, compute a rewriting equivalent to the query that uses only the views
- Sometimes there is no equivalent rewriting, but a maximally contained rewriting can be found
- Crucial in data integration, where views describe the data accessible through different sources in terms of a global schema
- Sometimes the cost of accessing each view must be taken into account
- Extensively studied; many algorithms exist for various query and view languages, see the survey [Halevy, 2001]

Views and updates

Views interact in non-trivial ways with updates (insert, modify, delete).

View maintenance: when an update happens on the base tables, the update must be reflected in the views

- No issue for virtual views
- More involved for materialized views, which must be maintained as updates occur

Views and updates

Views interact in non-trivial ways with updates (insert, modify, delete).

View maintenance: when an update happens on the base tables, the update must be reflected in the views

- No issue for virtual views
- More involved for materialized views, which must be maintained as updates occur

Update through views: under some circumstances we want to perform an update operation on a view and have it result in appropriate updates of the base tables

Views and updates

Views interact in non-trivial ways with updates (insert, modify, delete).

View maintenance: when an update happens on the base tables, the update must be reflected in the views

- No issue for virtual views
- More involved for materialized views, which must be maintained as updates occur

Update through views: under some circumstances we want to perform an update operation on a view and have it result in appropriate updates of the base tables

How can this be done?

Outline

Data model

- Relational data model: data decomposed into relations, with labelled attributes...

Data model

- Relational data model: data decomposed into relations, with labelled attributes...

name	position	city
John	Director	New York
Paul	Janitor	New York
Dave	Analyst	Paris
Ellen	Field agent	Berlin
Magdalen	Double agent	Paris
Nancy	HR director	Paris
Susan	Analyst	Berlin

Data model

- Relational data model: data decomposed into relations, with labelled attributes...
- ...and an extra provenance annotation for each tuple (think of it as a tuple identifier)

name	position	city	prov
John	Director	New York	t_1
Paul	Janitor	New York	t_2
Dave	Analyst	Paris	t_3
Ellen	Field agent	Berlin	t_4
Magdalen	Double agent	Paris	t_5
Nancy	HR director	Paris	t_6
Susan	Analyst	Berlin	t_7

Boolean provenance [Imielinski and Jr., 1984]

- $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ a finite set of Boolean events
- provenance annotation: Boolean function over $\mathcal{X}$, i.e., a function of the form $(\mathcal{X} \to \{\bot, \top\}) \to \{\bot, \top\}$
- Interpretation: possible-worlds semantics
  - each valuation $\nu : \mathcal{X} \to \{\bot, \top\}$ designates a possible world of the database
  - the provenance of a tuple evaluates under ν to $\bot$ or $\top$ depending on whether this tuple exists in this possible world
  - e.g., if every base tuple is annotated with the indicator function of a distinct Boolean event, the set of possible worlds is the set of all sub-instances

Examples of possible worlds

name	position	city	prov
John	Director	New York	t_1
Paul	Janitor	New York	t_2
Dave	Analyst	Paris	t_3
Ellen	Field agent	Berlin	t_4
Magdalen	Double agent	Paris	t_5
Nancy	HR director	Paris	t_6
Susan	Analyst	Berlin	t_7

ν:

t_1	t_2	t_3	t_4	t_5	t_6	t_7
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Examples of possible worlds

name	position	city	prov
John	Director	New York	t_1
Dave	Analyst	Paris	t_3
Magdalen	Double agent	Paris	t_5
Susan	Analyst	Berlin	t_7

ν:

t_1	t_2	t_3	t_4	t_5	t_6	t_7
$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$

Boolean provenance of query results

- $\nu(D)$: the sub-instance of D where every tuple whose provenance evaluates to $\bot$ under ν is removed
- The Boolean provenance $\mathrm{prov}_{q,D}(t)$ of a tuple $t \in q(D)$ is the function

$$\nu \mapsto \begin{cases} \top & \text{if } t \in q(\nu(D)) \\ \bot & \text{otherwise} \end{cases}$$

Example (Which cities appear in the Personnel table?)

name	position	city	prov
John	Director	New York	t_1
Paul	Janitor	New York	t_2
Dave	Analyst	Paris	t_3
Ellen	Field agent	Berlin	t_4
Magdalen	Double agent	Paris	t_5
Nancy	HR director	Paris	t_6
Susan	Analyst	Berlin	t_7

city	prov
New York	$t_1 \vee t_2$
Paris	$t_3 \vee t_5 \vee t_6$
Berlin	$t_4 \vee t_7$

Commutative semiring $(K, \mathbb{0}, \mathbb{1}, \oplus, \otimes)$

- Set K with two distinguished elements $\mathbb{0}$, $\mathbb{1}$
- $\oplus$ is associative, commutative, with identity $\mathbb{0}_K$:
  - $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
  - $a \oplus b = b \oplus a$
  - $a \oplus \mathbb{0} = \mathbb{0} \oplus a = a$
- $\otimes$ is associative, commutative, with identity $\mathbb{1}_K$:
  - $a \otimes (b \otimes c) = (a \otimes b) \otimes c$
  - $a \otimes b = b \otimes a$
  - $a \otimes \mathbb{1} = \mathbb{1} \otimes a = a$
- $\otimes$ distributes over $\oplus$:

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$$

- $\mathbb{0}$ annihilates for $\otimes$:

$$a \otimes \mathbb{0} = \mathbb{0} \otimes a = \mathbb{0}$$

Example semirings

- $(\mathbb{N}, 0, 1, +, \times)$: counting semiring
- $(\{\bot, \top\}, \bot, \top, \vee, \wedge)$: Boolean semiring
- $(\{\mathit{public} < \mathit{confidential} < \mathit{secret} < \mathit{top\ secret} < \mathit{unavailable}\}, \mathit{unavailable}, \mathit{public}, \min, \max)$: security semiring
- $(\mathbb{N} \cup \{\infty\}, \infty, 0, \min, +)$: tropical semiring
- $(\{\text{Boolean functions over } \mathcal{X}\}, \bot, \top, \vee, \wedge)$: semiring of Boolean functions over $\mathcal{X}$
- $(\mathbb{N}[\mathcal{X}], 0, 1, +, \times)$: semiring of integer polynomials over variables in $\mathcal{X}$ (also called How or universal semiring; see below)
- $(\mathcal{P}(\mathcal{P}(\mathcal{X})), \emptyset, \{\emptyset\}, \cup, \Cup)$: Why semiring on $\mathcal{X}$ ($A \Cup B := \{a \cup b \mid a \in A, b \in B\}$)

Semiring provenance [Green et al., 2007]

- Fix a semiring $(K, \mathbb{0}, \mathbb{1}, \oplus, \otimes)$
- Assume provenance annotations live in K
- Consider a query q in the positive relational algebra (selection, projection, renaming, cartesian product, union; joins are simulated via renaming, cartesian product, selection, projection)
- Define the provenance of a tuple $t \in q(D)$ recursively on the structure of q

Selection, renaming

Provenance annotations of selected tuples are unchanged.

Example $(\rho_{\text{name}\to\text{n}}(\sigma_{\text{city}=\text{“New York”}}(R)))$

name	position	city	prov
John	Director	New York	t_1
Paul	Janitor	New York	t_2
Dave	Analyst	Paris	t_3
Ellen	Field agent	Berlin	t_4
Magdalen	Double agent	Paris	t_5
Nancy	HR director	Paris	t_6
Susan	Analyst	Berlin	t_7

n	position	city	prov
John	Director	New York	t_1
Paul	Janitor	New York	t_2

Projection

Provenance annotations of merged identical tuples are $\oplus$-ed.

Example ($\pi_{\text{city}}(R)$)

name	position	city	prov
John	Director	New York	t_1
Paul	Janitor	New York	t_2
Dave	Analyst	Paris	t_3
Ellen	Field agent	Berlin	t_4
Magdalen	Double agent	Paris	t_5
Nancy	HR director	Paris	t_6
Susan	Analyst	Berlin	t_7

city	prov
New York	$t_1 \oplus t_2$
Paris	$t_3 \oplus t_5 \oplus t_6$
Berlin	$t_4 \oplus t_7$

Union

Provenance annotations of merged identical tuples are $\oplus$-ed.

Example

$\pi_{\text{city}}(\sigma_{\textit{ends-with}(\text{position},\text{“agent”})}(R)) \cup \pi_{\text{city}}(\sigma_{\text{position}=\text{“Analyst”}}(R))$

name	position	city	prov
John	Director	New York	t_1
Paul	Janitor	New York	t_2
Dave	Analyst	Paris	t_3
Ellen	Field agent	Berlin	t_4
Magdalen	Double agent	Paris	t_5
Nancy	HR director	Paris	t_6
Susan	Analyst	Berlin	t_7

city	prov
Paris	$t_3 \oplus t_5$
Berlin	$t_4 \oplus t_7$

Cartesian product and join

Provenance annotations of combined tuples are $\otimes$-ed.

Example

$\pi_{\text{city}}(\sigma_{\textit{ends-with}(\text{position},\text{“agent”})}(R)) \bowtie \pi_{\text{city}}(\sigma_{\text{position}=\text{“Analyst”}}(R))$

name	position	city	prov
John	Director	New York	t_1
Paul	Janitor	New York	t_2
Dave	Analyst	Paris	t_3
Ellen	Field agent	Berlin	t_4
Magdalen	Double agent	Paris	t_5
Nancy	HR director	Paris	t_6
Susan	Analyst	Berlin	t_7

city	prov
Paris	$t_3 \otimes t_5$
Berlin	$t_4 \otimes t_7$

What can we do with this?

Counting semiring:	number of times a tuple can be derived; multiset semantics
Boolean semiring:	whether a tuple exists when a sub-database is selected
Security semiring:	minimum clearance level required to obtain a tuple as a result
Tropical semiring:	minimum-cost way of obtaining a tuple (think shortest path in a graph)
Boolean functions:	Boolean provenance, as defined above
Integer polynomials:	universal provenance; see below
Why semiring:	why-provenance [Buneman et al., 2001], sets of tuple-combinations needed for a tuple to exist

Why-provenance example

$$\pi_{\text{city}}(\sigma_{\text{name}<\text{name2}}(\pi_{\text{name,city}}(R) \bowtie \rho_{\text{name}\to\text{name2}}(\pi_{\text{name,city}}(R))))$$

name	position	city	prov
John	Director	New York	t_1
Paul	Janitor	New York	t_2
Dave	Analyst	Paris	t_3
Ellen	Field agent	Berlin	t_4
Magdalen	Double agent	Paris	t_5
Nancy	HR director	Paris	t_6
Susan	Analyst	Berlin	t_7

city	prov
New York	$\{\{t_1, t_2\}\}$
Paris	$\{\{t_3, t_5\}, \{t_3, t_6\}, \{t_5, t_6\}\}$
Berlin	$\{\{t_4, t_7\}\}$

Notes [Green et al., 2007]

- Computing provenance carries only a polynomial-time overhead
- Two equivalent queries can produce different provenance annotations on the same database, in some semirings (e.g., in the Why semiring)
- Semiring homomorphisms commute with provenance computation: if h is a homomorphism from K to K', computing provenance in K then applying h yields the same result as computing provenance directly in K'
- The semiring of integer polynomials is universal: there is a unique homomorphism into any other commutative semiring respecting a given valuation of the variables
- Hence all computations can be carried out in the universal semiring, and homomorphisms applied afterwards

Beyond semirings

- For non-monotone queries, one needs semirings with monus [Geerts and Poggi, 2010], even if the theory is not as clean as for plain semirings [Amsterdamer et al., 2011b]
- It is also possible to give a semantics to the provenance of aggregate queries, but one has to introduce an algebraic structure at the value level (provenance semimodules [Amsterdamer et al., 2011a])
- It also works for recursive queries such as Datalog [Green and Tannen, 2006, Deutch et al., 2014], but only for some semantics [Ramusat, 2022]

Outline

Application: Probabilistic databases

[Green and Tannen, 2006, Suciu et al., 2011]

- Independent DB: every tuple t is annotated with an independent probability $\Pr(t)$ of existing
- Probability of a possible world $D' \subseteq D$:

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

- Probability of a tuple in the result of q on D:

$$\Pr(t \in q(D)) = \sum_{\substack{D' \subseteq D \\ t \in q(D')}} \Pr(D')$$

- Setting $\Pr(x_i) := \Pr(t_i)$, where x_i is the provenance annotation of tuple t_i: $\Pr(t \in q(D)) = \Pr(\mathrm{prov}_{q,D}(t))$
- Computing the probability of a query in a probabilistic database reduces to computing Boolean provenance and then computing the probability of a Boolean function
- Works for richer probabilistic models too

Probability-computation example

name	position	city	prov	prob
John	Director	New York	t_1	0.5
Paul	Janitor	New York	t_2	0.7
Dave	Analyst	Paris	t_3	0.3
Ellen	Field agent	Berlin	t_4	0.2
Magdalen	Double agent	Paris	t_5	1.0
Nancy	HR director	Paris	t_6	0.8
Susan	Analyst	Berlin	t_7	0.2

city	prov
New York	$t_1 \vee t_2$
Paris	$t_3 \vee t_5 \vee t_6$
Berlin	$t_4 \vee t_7$

Probability-computation example

name	position	city	prov	prob
John	Director	New York	t_1	0.5
Paul	Janitor	New York	t_2	0.7
Dave	Analyst	Paris	t_3	0.3
Ellen	Field agent	Berlin	t_4	0.2
Magdalen	Double agent	Paris	t_5	1.0
Nancy	HR director	Paris	t_6	0.8
Susan	Analyst	Berlin	t_7	0.2

city	prov	prob
New York	$t_1 \vee t_2$	$1 - (1 - 0.5) \times (1 - 0.7) = 0.85$
Paris	$t_3 \vee t_5 \vee t_6$	1.00
Berlin	$t_4 \vee t_7$	$1 - (1 - 0.2) \times (1 - 0.2) = 0.36$

Outline

Insertions

For positive relational algebra, the impact of an insertion can be computed incrementally:

$$\sigma_\varphi(R \cup \Delta R) = \sigma_\varphi(R) \cup \sigma_\varphi(\Delta R)$$
$$\pi_X(R \cup \Delta R) = \pi_X(R) \cup \pi_X(\Delta R)$$
$$\rho_{A\to B}(R \cup \Delta R) = \rho_{A\to B}(R) \cup \rho_{A\to B}(\Delta R)$$
$$(R \cup \Delta R) \cup (S \cup \Delta S) = (R \cup S) \cup \Delta R \cup \Delta S$$
$$(R \cup \Delta R) \times (S \cup \Delta S) = (R \times S) \cup (R \times \Delta S) \cup (\Delta R \times S) \cup (\Delta R \times \Delta S)$$
$$(R \cup \Delta R) \bowtie (S \cup \Delta S) = (R \bowtie S) \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie S) \cup (\Delta R \bowtie \Delta S)$$

Deletions

- Just use Boolean provenance!
- Drop every tuple whose provenance evaluates to $\bot$

Deletions

- Just use Boolean provenance!
- Drop every tuple whose provenance evaluates to $\bot$

name	position	city	prov
John	Director	New York	t_1
Paul	Janitor	New York	t_2
Dave	Analyst	Paris	t_3
Ellen	Field agent	Berlin	t_4
Magdalen	Double agent	Paris	t_5
Nancy	HR director	Paris	t_6
Susan	Analyst	Berlin	t_7

city	prov
New York	$t_1 \wedge t_2$
Paris	$t_3 \wedge t_5 \vee t_3 \wedge t_6 \vee t_5 \wedge t_6$
Berlin	$t_4 \wedge t_7$

If t_1 disappears

Deletions

- Just use Boolean provenance!
- Drop every tuple whose provenance evaluates to $\bot$

name	position	city	prov
John	Director	New York	t_1
Paul	Janitor	New York	t_2
Dave	Analyst	Paris	t_3
Ellen	Field agent	Berlin	t_4
Magdalen	Double agent	Paris	t_5
Nancy	HR director	Paris	t_6
Susan	Analyst	Berlin	t_7

city	prov
New York	$t_1 \wedge t_2$
Paris	$t_3 \wedge t_5 \vee t_3 \wedge t_6 \vee t_5 \wedge t_6$
Berlin	$t_4 \wedge t_7$

If t_1 disappears, New York disappears from the view.

What about PostgreSQL?

- Automatic view maintenance is not implemented!
- A materialized view can be refreshed manually with:
  `REFRESH MATERIALIZED VIEW v`
- For automatic maintenance, turn the materialized view into a regular table, and add triggers on the base tables to handle update operations on a case-by-case basis

```
CREATE TRIGGER tab_trigger
  AFTER INSERT ON tab
  FOR EACH ROW EXECUTE PROCEDURE tab_insert()
```

Outline

Insertions

Hard in general:

- In a join with the join value projected out, a join value must be "invented" so that tuples can be added with that value to several base tables
- In a plain projection, base-table fields are filled with **NULL**
- With aggregation: essentially impossible, except by defining an ad hoc strategy for each problem

Deletions [Buneman et al., 2002]

- A use case for Why-provenance!
- To delete a tuple t from a view, pick a minimal subset of tuples (in size, or in terms of side-effects on other view tuples) whose annotation appears in every annotation set of the Why-provenance of t
- NP-complete in general

Deletions [Buneman et al., 2002]

- A use case for Why-provenance!
- To delete a tuple t from a view, pick a minimal subset of tuples (in size, or in terms of side-effects on other view tuples) whose annotation appears in every annotation set of the Why-provenance of t
- NP-complete in general

name	position	city	prov
John	Director	New York	t_1
Paul	Janitor	New York	t_2
Dave	Analyst	Paris	t_3
Ellen	Field agent	Berlin	t_4
Magdalen	Double agent	Paris	t_5
Nancy	HR director	Paris	t_6
Susan	Analyst	Berlin	t_7

city	prov
New York	$\{\{t_1, t_2\}\}$
Paris	$\{\{t_3, t_5\}, \{t_3, t_6\}, \{t_5, t_6\}\}$
Berlin	$\{\{t_4, t_7\}\}$

To delete Paris

Deletions [Buneman et al., 2002]

- A use case for Why-provenance!
- To delete a tuple t from a view, pick a minimal subset of tuples (in size, or in terms of side-effects on other view tuples) whose annotation appears in every annotation set of the Why-provenance of t
- NP-complete in general

name	position	city	prov
John	Director	New York	t_1
Paul	Janitor	New York	t_2
Dave	Analyst	Paris	t_3
Ellen	Field agent	Berlin	t_4
Magdalen	Double agent	Paris	t_5
Nancy	HR director	Paris	t_6
Susan	Analyst	Berlin	t_7

city	prov
New York	$\{\{t_1, t_2\}\}$
Paris	$\{\{t_3, t_5\}, \{t_3, t_6\}, \{t_5, t_6\}\}$
Berlin	$\{\{t_4, t_7\}\}$

To delete Paris, drop two tuples among t_3, t_5, t_6.

What about PostgreSQL?

- The SQL standard only allows updates through very simple views (selection, projection, renaming):
  - no joins
  - no union, intersection, difference
  - no aggregation
  - no recursive query
  - no set semantics
- For everything else, define an INSTEAD OF trigger on the view

```
CREATE TRIGGER vue_trigger
  INSTEAD OF INSERT OR UPDATE OR DELETE ON vue
  FOR EACH ROW EXECUTE PROCEDURE vue_update()
```

Outline

Triggers in PostgreSQL

- Functions defined inside the DBMS, fired when certain events occur:
  - **BEFORE** an update on a table or view
  - **AFTER** an update on a table or view
  - **INSTEAD OF** an update on a view
- Can be fired once per affected tuple (**FOR EACH ROW**) or once per statement (**FOR EACH STATEMENT**)
- Can be defined in several languages; the most common is PL/pgSQL
- See `https://www.postgresql.org/docs/current/sql-createtrigger.html`

PL/pgSQL

- PostgreSQL's imperative programming language
- Used to define triggers:

```
CREATE FUNCTION tab_insert() RETURNS TRIGGER
  LANGUAGE plpgsql
AS $$
  BEGIN
    ...
    RETURN NEW;
  END;
$$;
```

- Inside a row-level trigger function, **OLD** is the previous tuple value (for modification or deletion), **NEW** is the new value (for insertion or modification)

Outline

Why a provenance-aware DBMS?

- Provenance automatically maintained as the user interacts with the database
- Provenance computation benefits from query optimization inside the DBMS
- Probability computation on top of provenance
- Any form of provenance can be computed: Boolean, semiring, why, where, aggregate provenance, on demand

ProvSQL: Provenance within PostgreSQL

[Senellart et al., 2018, Sen et al., 2026]

- PostgreSQL extension, works for all PostgreSQL versions $\geqslant 10$, tested on Linux, macOS (x86 / ARM), WSL
- Provenance annotations stored as Universally Unique Identifiers (UUIDs), in an extra attribute of every provenance-aware relation
- UUIDs of base tuples randomly generated; UUIDs of query results generated deterministically (so identical sub-queries share gates)
- A provenance circuit relating UUIDs of elementary provenance annotations and arithmetic gates is stored in the DBMS shared memory (and on disk, per-database since v1.3)
- All computations carried out in the universal semiring (or in the free semiring with monus, or, for where-provenance, in a free term algebra)

How it works

- Query rewriting (after parsing, before planning) to compute output provenance from input provenance:
  - duplicate elimination (DISTINCT, set union) $\Rightarrow$ $\oplus$-aggregation of provenance values
  - cartesian products and joins $\Rightarrow$ $\otimes$-combination of provenance values
  - difference rewritten as a join with $\ominus$
- Extra circuit gates on projection and join for where-provenance
- Extra gates for aggregation (agg, `semimod`)
- Probability computation from the provenance circuits, via several methods (linear-time evaluation on read-once circuits, knowledge compilation to d-DNNF for low-treewidth circuits [Amarilli et al., 2020], Monte-Carlo sampling, external compilers `d4` / `c2d` / `dsharp`)
- (Expected) Shapley and Banzhaf value computation [Karmakar et al., 2024]

Engineering challenges

- Low-level access to PostgreSQL data structures
- No simple query rewriting mechanism in PostgreSQL; ProvSQL installs a planner hook
- SQL is much less clean than relational algebra, with multiset semantics by default and many ways to express the same thing
- Inherent limitations: e.g., no aggregation within recursive queries
- Provenance computation should not slow down the rest of the system; per-database mmap circuit store, lazy input gates, gate cache
- Update provenance now partially supported [Bourhis et al., 2020, Widiaatmaja et al., 2025], but transactional support is still missing

Try it out

- Source code, documentation, tutorial, case studies: `https://provsql.org/`
- GitHub: `https://github.com/PierreSenellart/provsql`
- Five worked case studies bundled as regression tests (intelligence agency, open science, public transit, government ministers over time, wildlife photo archive)
- Companion ProvSQL Studio web UI for provenance inspection and on-the-fly semiring evaluation (`pip install provsql-studio`)
- Docker image bundles the extension and Studio: `docker run -p 8000:8000 -p 5432:5432 inriavalda/provsql:1.4.0`

Outline

References

- Foundational paper on provenance semirings [Green et al., 2007]
- Survey on answering queries using views [Halevy, 2001]
- Chapter 22 of [Abiteboul et al., 1995]
- Accessible overview of provenance and probabilistic databases [Senellart, 2017]; tutorial survey [Senellart, 2019]
- Reflection on how semiring theory shaped ProvSQL's design [Senellart, 2024]
- PostgreSQL documentation on triggers and PL/pgSQL: `https://www.postgresql.org/docs/`
- ProvSQL, a PostgreSQL extension for provenance and probabilities: `https://provsql.org/` [Senellart et al., 2018, Sen et al., 2026]

Bibliography I

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN 0-201-53771-0. URL `http://www-cse.ucsd.edu/users/vianu/book.html`.

Antoine Amarilli, Florent Capelli, Mikaël Monet, and Pierre Senellart. Connecting knowledge compilation classes and width parameters. *Theory Comput. Syst.*, 64(5):861–914, 2020.

Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for aggregate queries. In *PODS*, pages 153–164, 2011a.

Yael Amsterdamer, Daniel Deutch, and Val Tannen. On the limitations of provenance for queries with difference. In *TaPP*, 2011b.

Pierre Bourhis, Daniel Deutch, and Yuval Moskovitch. Equivalence-invariant algebraic provenance for hyperplane update queries. In *SIGMOD*, pages 415–429, 2020.

Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

Bibliography II

Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. On propagation of deletions and annotations through views. In *PODS*, pages 150–158, 2002.

Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. Circuits for Datalog provenance. In *ICDT*, pages 201–212, 2014.

Floris Geerts and Antonella Poggi. On database query languages for K-relations. *J. Appl. Log.*, 8(2):173–185, 2010.

Todd J. Green and Val Tannen. Models for incomplete and probabilistic information. In *EDBT Workshops*, pages 278–296, 2006.

Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4): 270–294, 2001.

Tomasz Imielinski and Witold Lipski Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

Bibliography III

Pratik Karmakar, Mikaël Monet, Pierre Senellart, and Stéphane Bressan. Expected shapley-like scores of boolean functions: Complexity and applications to probabilistic databases. *Proc. ACM Manag. Data*, 2(2):92, 2024.

Yann Ramusat. *The Semiring-Based Provenance Framework for Graph Databases*. PhD thesis, Ecole normale supérieure – ENS PARIS, PSL University, 2022.

Aryak Sen, Silviu Maniu, and Pierre Senellart. ProvSQL: a general system for keeping track of the provenance and probability of data. In *Proc. ICDE*, Montréal, Canada, 2026. Also arXiv abs/2504.12058.

Pierre Senellart. Provenance and probabilities in relational databases. *SIGMOD Rec.*, 46(4):5–15, 2017.

Pierre Senellart. Provenance in databases: Principles and applications. In *Reasoning Web*, LNCS, pages 104–109, 2019.

Pierre Senellart. On the impact of provenance semiring theory on the design of a provenance-aware database system. In *Tannen's Festschrift*, OASIcs, pages 9:1–9:10, 2024.

Bibliography IV

Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. ProvSQL: provenance and probability management in PostgreSQL. *Proc. VLDB Endow.*, 11(12):2034–2037, 2018. Demonstration.

Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Morgan & Claypool, 2011.

Albert Ariel Widiaatmaja, Belkis Djeffal, Ashish Dandekar, and Pierre Senellart. Demonstration of ProvSQL update provenance through temporal databases. In *Provenance Week*, pages 71–76, 2025. Demonstration.