Formal Specification of the Policy Algebra

DAVID A. VENTIMIGLIA, Supabase, USA

This document formally specifies a *policy algebra* for PostgreSQL Row-Level Security (RLS). The algebra defines a decidable domain-specific language whose expressions compile deterministically to native PostgreSQL security artifacts. By restricting the language to atoms, clauses, and policies composed under well-defined lattice operations, the system enables static analysis — satisfiability, subsumption, redundancy, contradiction, and tenant-isolation proofs — that is impossible over arbitrary SQL. The specification spans the full governance lifecycle: definition, analysis, optimization, compilation, drift detection, and reconciliation.

Table 1 summarizes the notation used throughout this paper.

Running Example Schema

The specification uses the following multi-tenant SaaS schema throughout. Figure 1 shows the tables, primary keys, foreign keys, and the presence or absence of `tenant_id`.

1 Introduction & Motivation

1.1 The Problem: Arbitrary RLS Is Undecidable

PostgreSQL Row-Level Security allows attaching arbitrary SQL predicates to tables via `CREATE POLICY`. These predicates execute as part of every query, filtering rows according to security rules. The mechanism is powerful: any boolean SQL expression is a valid RLS predicate.

This power is also the fundamental problem. SQL is Turing-complete. An RLS predicate may invoke user-defined functions, reference arbitrary subqueries, or encode complex recursive logic. As a consequence:

THEOREM 1.1 (UNDECIDABILITY OF ARBITRARY RLS). *Given an arbitrary set of RLS policies expressed as SQL predicates, determining whether a given row is accessible to a given user is undecidable in general.*

SKETCH. Reduce from the halting problem. Encode a Turing machine's transition function as a PL/pgSQL function invoked within an RLS `USING` clause. The predicate returns `true` if and only if the machine halts. Determining row accessibility therefore requires solving the halting problem. □

This undecidability means that no tool can, in general:

- Prove that tenant isolation holds across all policies
- Detect contradictory policies that block all access
- Identify redundant policies that can be safely removed
- Verify that a policy change preserves the intended access semantics

Organizations managing hundreds of tables and dozens of interacting policies face an intractable verification burden if policies are authored as raw SQL.

1.2 The Compiler Insight

The solution is a shift in perspective: *do not analyze arbitrary SQL; instead, generate SQL from a language where analysis is decidable.*

This is precisely the strategy used by optimizing compilers. A compiler does not reason about arbitrary machine code. It operates on a structured intermediate representation (IR) where transformations are provably correct, then emits machine code as a final step.

Applied to RLS:

Author's Contact Information: David A. Ventimiglia, david.ventimiglia@supabase.io, Supabase, USA.

Table 1. Notation Conventions

Symbol	Meaning
$\wedge$	Logical conjunction (AND)
$\vee$	Logical disjunction (OR)
$\neg$	Logical negation (NOT)
$\bot$	Falsity / unsatisfiable / contradiction
$\top$	Truth / tautology
$\subseteq$	Subset or subsumption
$\supseteq$	Superset
$\sqsubseteq$	Lattice ordering (less restrictive than or equal)
$\sqcup$	Lattice join (least upper bound)
$\sqcap$	Lattice meet (greatest lower bound)
$\triangle$	Symmetric difference
$[\![\cdot]\!]$	Denotation (semantic interpretation)
$\vdash$	Entailment / proves
$\forall$	Universal quantifier
$\exists$	Existential quantifier
$\longrightarrow$	Implication or maps-to
$\emptyset$	Empty set
$\in$	Set membership
$\notin$	Not a member of
$\equiv$	Logical equivalence
$\sqsupseteq$	Reverse lattice ordering (more permissive than or equal)
$\mathcal{P}$	Power set
$\bigvee$	Indexed disjunction (big OR)
$\bigwedge$	Indexed conjunction (big AND)
$\lvert S\rvert$	Cardinality of set S

(1) Define a **domain-specific language** (DSL) with restricted expressiveness.
(2) Perform all **analysis and optimization** on the DSL's abstract syntax tree.
(3) **Compile** the DSL deterministically to PostgreSQL `CREATE POLICY` statements.

RLS becomes a *compilation target*, not an authoring surface. The DSL is designed so that the properties we care about — satisfiability, subsumption, isolation — are decidable by construction.

1.3 Relationship to Prior Work

The algebra draws on five established formalisms, each covering a distinct aspect of the system:

Bonatti et al.'s access-control algebra [2]. Bonatti, De Capitani di Vimercati, and Samarati formalized access-control policies as algebraic objects supporting union (grant), intersection (restriction), and difference (exception) operations. Our permissive/restrictive composition directly corresponds to their grant/restriction operators. The effective access predicate $(\bigvee \text{ permissive}) \wedge (\bigwedge \text{ restrictive})$ is an instance of their composition framework.

Lattice theory. Policies ordered by subsumption form a lattice. The join ($\sqcup$) corresponds to disjunction of permissive policies; the meet ($\sqcap$) to conjunction of restrictive policies. Redundancy detection reduces to identifying elements dominated by existing lattice members.

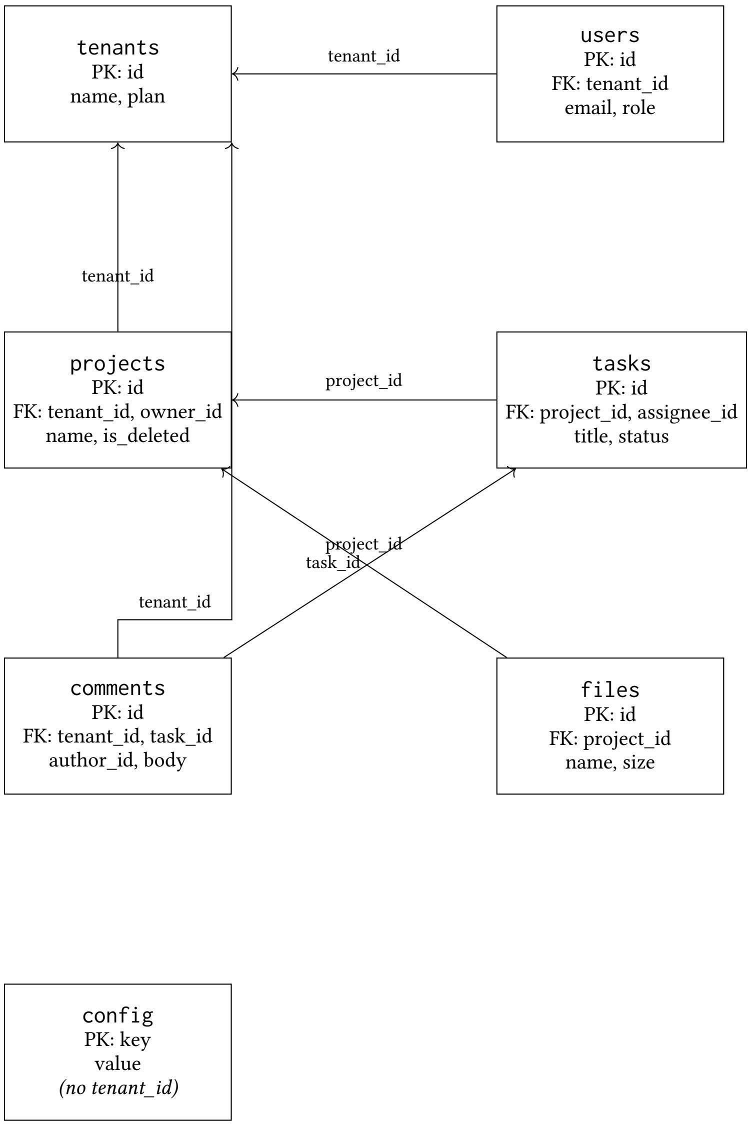

Fig. 1. Running example: multi-tenant SaaS schema. Tables `users`, `projects`, and `comments` carry `tenant_id` directly. Tables `tasks` and `files` inherit tenant context via FK to `projects`. Table `config` is global (no `tenant_id`).

SMT solving (Satisfiability Modulo Theories) [1, 4]. The atoms of our algebra — column comparisons with session variables and literals — fall within the quantifier-free fragments of linear integer arithmetic (QF-LIA) and equality with uninterpreted functions (QF-EUF). These are decidable theories supported by solvers such as Z3 and cvc5. We use SMT to check clause satisfiability, detect contradictions, and prove tenant isolation.

Formal Concept Analysis [5] (Ganter & Wille). Selectors — predicates over table metadata — define a Galois connection between the power set of tables and the power set of structural attributes. The closed sets (formal concepts) correspond to natural groupings of tables sharing common structure, providing a principled foundation for policy targeting.

Galois connections for compiler correctness [3]. The compilation function from DSL policies to SQL artifacts, paired with the denotational semantics of each, forms a Galois connection. This structure provides the framework for stating and proving that compilation preserves the intended access semantics.

1.4 Scope and Audience

This specification covers the **full lifecycle** of the policy algebra:

- **Definition**: atoms, clauses, policies, selectors, relationship traversal
- **Analysis**: satisfiability, subsumption, redundancy, contradiction, isolation
- **Optimization**: rewrite rules, normal forms, termination
- **Compilation**: deterministic translation to PostgreSQL artifacts
- **Monitoring**: drift detection and reconciliation

The intended audience is **senior engineers** implementing or evaluating the policy engine, as well as researchers interested in the formal foundations of database access control.

2 Atoms & Value Sources

Atoms are the irreducible predicates of the policy algebra. Every policy ultimately reduces to a boolean combination of atoms, each representing a single comparison.

2.1 Value Sources

A **value source** produces a scalar value for comparison. The algebra recognizes four kinds:

Definition 2.1 (Value source).

```
ValueSource ::= col(name)
              | session(key)
              | lit(v)
              | fn(name, args)
```

Where:

- `col(name)` — references a column of the table to which the policy is attached. The column must exist in the table's schema. Example: `col('tenant_id')`.
- `session(key)` — retrieves a runtime session variable via PostgreSQL's `current_setting(key)`. Example: `session('app.tenant_id')` compiles to `current_setting('app.tenant_id')`.
- `lit(v)` — a literal constant value: string, integer, boolean, or null. Example: `lit('admin')`, `lit(42)`, `lit(true)`.
- `fn(name, args)` — a call to a whitelisted, pure, deterministic function. The function must be registered in the policy engine's function allowlist. Example: `fn('auth.uid', [])` compiles to `auth.uid()`.

Definition 2.2 (Value source type). *Each value source has an associated type drawn from* {`text`, `integer`, `bigint`, `uuid`, `boolean`, `timestamp`, `jsonb`}. *Type compatibility is enforced at policy definition time, not at compilation.*

2.2 Atoms

Definition 2.3 (Atom). *An atom is a triple* (*left*, *op*, *right*) *where:*

- *left and right are value sources*
- *op is a comparison operator from the set* {$=$, $\neq$, $<$, $>$, $\leq$, $\geq$, `IN`, `NOT IN`, `IS NULL`, `IS NOT NULL`, `LIKE`, `NOT L`

For unary operators (`IS NULL`*,* `IS NOT NULL`*), right is omitted (or equivalently, right =* `lit(null)`*).*

BNF fragment:

```
<atom>        ::= <value_source> <binary_op> <value_source>
                | <value_source> <unary_op>

<binary_op>   ::= "=" | "!=" | "<" | ">" | "<=" | ">="
                | "IN" | "NOT IN" | "LIKE" | "NOT LIKE"

<unary_op>    ::= "IS NULL" | "IS NOT NULL"

<value_source> ::= "col(" <identifier> ")"
                | "session(" <string_literal> ")"
                | "lit(" <literal_value> ")"
                | "fn(" <identifier> "," "[" <arg_list> "]" ")"
```

Examples:

Atom	Informal meaning
`(col('tenant_id'), =, session('app.tenant_id'))`	Row's tenant matches session tenant
`(col('role'), =, lit('admin'))`	User role is admin
`(col('is_deleted'), =, lit(false))`	Row is not soft-deleted
`(col('status'), IN, lit(['active','pending']))`	Status is active or pending
`(col('deleted_at'), IS NULL, _)`	No deletion timestamp

2.3 Atom Normal Form

To enable comparison and deduplication, atoms are normalized to a canonical form.

DEFINITION 2.4 (ATOM NORMAL FORM). *An atom is in normal form when:*

(1) ***Column-left ordering****: if exactly one operand is* `col(...)`*, it appears on the left. If both are columns, they are ordered lexicographically by column name.*
(2) ***Operator canonicalization****: $>$ is rewritten to $<$ (with operands swapped); $>=$ to $<=$; $!=$ to* `NOT =`.
(3) ***Literal simplification****:* `lit(true)` *in a boolean equality is absorbed (e.g.,* `col('active') = lit(true)` *normalizes to* `col('active') IS NOT NULL` *only for boolean columns where null means false; otherwise left as-is).*

Algorithm: `normalize_atom`$(a) \to a'$ applies rules 1–3 in sequence.

2.4 Atom Equivalence and Subsumption

DEFINITION 2.5 (ATOM EQUIVALENCE). *Two atoms a_1 and a_2 are equivalent, written $a_1 \equiv a_2$, if and only if their normal forms are syntactically identical.*

DEFINITION 2.6 (ATOM SUBSUMPTION). *Atom a_1 subsumes atom a_2, written $a_1 \sqsubseteq a_2$, if every row satisfying a_2 also satisfies a_1. Equivalently, $a_2 \vdash a_1$ (a_2 entails a_1).*

Examples of subsumption:

- `col('x') IS NOT NULL` $\sqsubseteq$ `col('x') = lit(5)` — equality implies non-null
- `col('x') IN lit([1,2,3])` $\sqsubseteq$ `col('x') IN lit([1,2])` — subset of IN-list

2.5 Decidability of Atom Satisfiability

Property 2.1 (Decidability). *The satisfiability of any finite conjunction of atoms is decidable.*

Sketch. Each atom translates to a formula in the quantifier-free theory of linear integer arithmetic with equality and uninterpreted functions (QF-LIA ∪ QF-EUF). Column references become free variables; session variables become distinct free variables; literals become constants. The conjunction of translated atoms is a QF-LIA/EUF formula, which is decidable by the Nelson-Oppen combination procedure as implemented in SMT solvers. □

3 Clauses

A clause is the fundamental unit of row-level access control: a conjunction of atoms that must all be satisfied for a row to match.

3.1 Definition

Definition 3.1 (Clause). *A clause c is a finite set of atoms, interpreted as their conjunction:*

$$c = \{a_1, a_2, \ldots, a_n\} \quad \textit{meaning} \quad a_1 \wedge a_2 \wedge \cdots \wedge a_n$$

The empty clause $\{\}$ is the trivial clause, equivalent to $\top$ (always true).

BNF fragment:

```
<clause>   ::= <atom>
             | <atom> "AND" <clause>
```

3.2 Clause Normal Form

Definition 3.2 (Clause normal form). *A clause is in normal form when:*

1. *Every constituent atom is in atom normal form (Def. 2.4).*
2. *Atoms are sorted lexicographically by their normal-form string representation.*
3. *Duplicate atoms (by equivalence, Def. 2.5) are removed.*
4. *If any pair of atoms is contradictory, the entire clause is replaced by $\bot$.*

A pair of atoms is contradictory when their conjunction is unsatisfiable. Common cases detected syntactically:

- $\texttt{col}(x) = \texttt{lit}(v_1) \wedge \texttt{col}(x) = \texttt{lit}(v_2)$ where $v_1 \neq v_2$
- $\texttt{col}(x)$ `IS NULL` $\wedge\ \texttt{col}(x) = \texttt{lit}(v)$ for any non-null v
- $\texttt{col}(x) = \texttt{lit}(v) \wedge \texttt{col}(x) \neq \texttt{lit}(v)$

Algorithm: `normalize_clause`$(c) \longrightarrow c'$:

```
function normalize_clause(c):
    c' <- {normalize_atom(a) | a in c}
    c' <- deduplicate(c')
    if has_syntactic_contradiction(c'):
        return bot
    return sort(c')
```

3.3 Clause Properties

Property 3.1 (Clause satisfiability). *A normalized clause c is satisfiable if and only if $c \neq \bot$. For non-syntactic contradictions, SMT solving (Property 2.1) provides a complete decision procedure.*

Property 3.2 (Clause subsumption). *Clause c_1 subsumes clause c_2, written $c_1 \sqsubseteq c_2$, if and only if every atom in c_1 is subsumed by some atom in c_2 or implied by the conjunction of atoms in c_2.*

A sufficient syntactic check: $c_1 \subseteq c_2$ (every atom in c_1 appears in c_2) implies $c_2 \sqsubseteq c_1$. Note the direction: more atoms means more constraints, hence fewer matching rows, hence the clause with more atoms is subsumed by (is less permissive than) the clause with fewer atoms.

More precisely: if $c_1 \subseteq c_2$ then $[\![c_2]\!] \subseteq [\![c_1]\!]$ (the denotation of c_2 is a subset of the denotation of c_1), so $c_1 \sqsubseteq c_2$ in the "more permissive" ordering.

Property 3.3 (Idempotence). *For any clause c: $c \land c = c$.*

Proof. Clause conjunction merges atom sets. Deduplication yields the original set. □

Example:

```
c1 = {col('tenant_id') = session('app.tenant_id')}
c2 = {col('tenant_id') = session('app.tenant_id'),
      col('role') = lit('editor')}

c1 subsumes c2: c1 (*@$\subseteq$@*) c2, so (*@$\denote{c_2} \subseteq \denote{c_1}
     $@*)
    c1 is more permissive (fewer constraints, more rows match)
```

4 Policies

A policy is a named, typed collection of clauses that applies to specific SQL commands on tables selected by a selector predicate.

4.1 Definition

Definition 4.1 (Policy). *A policy is a 5-tuple:*

$$p = (name,\ type,\ commands,\ selector,\ clauses)$$

Where:

- *name* $\in$ String — *a unique identifier for the policy*
- *type* $\in \{\texttt{permissive}, \texttt{restrictive}\}$
- *commands* $\subseteq \{\texttt{SELECT}, \texttt{INSERT}, \texttt{UPDATE}, \texttt{DELETE}\}$, *non-empty*
- *selector — a selector predicate (Section 6) determining which tables this policy applies to*
- *clauses* $= \{c_1, c_2, \ldots, c_n\}$ — *a non-empty finite set of clauses*

BNF fragment:

```
<policy>         ::= "POLICY" <identifier>
                     <policy_type>
                     <command_list>
                     <selector_clause>
                     <clause_block>

<policy_type>    ::= "PERMISSIVE" | "RESTRICTIVE"
<command_list>   ::= "FOR" <command> ("," <command>)*
<command>        ::= "SELECT" | "INSERT" | "UPDATE" | "DELETE"
<selector_clause> ::= "SELECTOR" <selector>
<clause_block>   ::= "CLAUSE" <clause> ("OR" "CLAUSE" <clause>)*
```

4.2 Policy Denotation

Definition 4.2 (Policy denotation). *The denotation of a policy p is the disjunction of its clauses:*

$$[\![p]\!] = [\![c_1]\!] \vee [\![c_2]\!] \vee \cdots \vee [\![c_n]\!]$$

A row satisfies a policy if it satisfies any *of the policy's clauses.*

4.3 USING vs WITH CHECK

For write commands (`INSERT`, `UPDATE`, `DELETE`), PostgreSQL [7] distinguishes:

- **USING**: filters which existing rows are visible (relevant for `UPDATE`, `DELETE`, and `SELECT` within an `UPDATE`/`DELETE`)
- **WITH CHECK**: validates new or modified rows (relevant for `INSERT` and the new values in `UPDATE`)

In this algebra, each policy carries a single set of clauses that serves as both `USING` and `WITH CHECK` by default. A policy may optionally specify separate `with_check_clauses` when the write-validation predicate differs from the read-visibility predicate.

Definition 4.3 (Policy with distinct check). *An extended policy is a 6-tuple* (*name*, *type*, *commands*, *selecto*
where check_clauses defaults to using_clauses if unspecified.

4.4 Example: Tenant Isolation Policy

```
POLICY tenant_isolation
  PERMISSIVE
  FOR SELECT, INSERT, UPDATE, DELETE
  SELECTOR has_column('tenant_id')
  CLAUSE col('tenant_id') = session('app.tenant_id')
```

This defines a single permissive policy with one clause containing one atom. The selector `has_column('tenant_id')` causes it to apply to `users`, `projects`, and `comments` in our running example, but *not* to `tasks`, `files` (no direct `tenant_id`), or `config` (global table).

5 Composition — The Policy Lattice

This section defines how multiple policies on a single table combine to produce an effective access predicate. The composition rules follow PostgreSQL's native semantics and correspond to Bonatti et al.'s [2] access-control algebra.

5.1 Table Policy Set

Definition 5.1 (Table policy set). *For a given table T and command* CMD*, the* table policy set *is the set of all policies whose selector matches T and whose command set includes* CMD*:*

$$\text{Policies}(T, \text{CMD}) = \{p \mid \text{match}(p.selector, T) \wedge \text{CMD} \in p.commands\}$$

This set partitions into permissive and restrictive subsets:

$$P(T, \text{CMD}) = \{p \in \text{Policies}(T, \text{CMD}) \mid p.type = \texttt{permissive}\}$$
$$R(T, \text{CMD}) = \{p \in \text{Policies}(T, \text{CMD}) \mid p.type = \texttt{restrictive}\}$$

5.2 Effective Access Predicate

Definition 5.2 (Effective access predicate). *The effective access predicate for table T under command* CMD *is:*

$$\text{effective}(T, \text{CMD}) = \left(\bigvee_{p \in P} [\![p]\!] \right) \wedge \left(\bigwedge_{r \in R} [\![r]\!] \right)$$

Expanding policy denotations:

$$\text{effective}(T, \text{CMD}) = \left(\bigvee_{p \in P} \bigvee_{c \in p.clauses} [\![c]\!] \right) \wedge \left(\bigwedge_{r \in R} \bigvee_{c \in r.clauses} [\![c]\!] \right)$$

5.3 Default Deny

Definition 5.3 (Default deny). *If* $P(T, \text{CMD}) = \emptyset$ *(no permissive policies apply), then* $\text{effective}(T, \text{CMD}) = \bot$. *No rows are accessible.*

This follows from the convention that an empty disjunction is $\bot$. Restrictive policies alone cannot grant access — they can only further restrict access already granted by permissive policies.

5.4 Connection to Bonatti's Algebra

Bonatti et al. [2] define an access-control algebra with three operators:

Bonatti operator	This algebra	Effect
+ (grant/union)	Permissive policy disjunction	Expands accessible rows
& (restriction/intersection)	Restrictive policy conjunction	Narrows accessible rows
− (exception/difference)	Not directly supported	Would allow row-level exceptions

The effective predicate formula maps directly:

$$\text{effective} = \left(+_{p \in P} [\![p]\!] \right) \ \& \ \left(\&_{r \in R} [\![r]\!] \right)$$

The absence of the exception operator (−) is deliberate: exceptions complicate analysis and are not needed for the patterns targeted by this algebra (tenant isolation, role-based access, soft-delete filtering).

5.5 Monotonicity Properties

Property 5.1 (Monotonicity of permissive extension). *Adding a permissive policy to P can only increase (or maintain) the set of accessible rows.*

Proof. Let $P' = P \cup \{p_{\text{new}}\}$. Then:

$$\bigvee_{p \in P'} [\![p]\!] = \left(\bigvee_{p \in P} [\![p]\!] \right) \vee [\![p_{\text{new}}]\!] \supseteq \bigvee_{p \in P} [\![p]\!]$$

Since $A \vee B \supseteq A$ for any predicates A, B (in terms of satisfying rows), the effective predicate's permissive component can only grow. □

Property 5.2 (Anti-monotonicity of restrictive extension). *Adding a restrictive policy to R can only decrease (or maintain) the set of accessible rows.*

Proof. Let $R' = R \cup \{r_{\text{new}}\}$. Then:

$$\bigwedge_{r \in R'} [\![r]\!] = \left(\bigwedge_{r \in R} [\![r]\!]\right) \wedge [\![r_{\text{new}}]\!] \subseteq \bigwedge_{r \in R} [\![r]\!]$$

Since $A \wedge B \subseteq A$ for any predicates A, B. □

5.6 Policy Subsumption and Redundancy

Definition 5.4 (Policy subsumption). *Permissive policy p_1 subsumes permissive policy p_2, written $p_1 \sqsupseteq p_2$, if:*

$$[\![p_2]\!] \subseteq [\![p_1]\!]$$

That is, every row accessible under p_2 is also accessible under p_1.

Definition 5.5 (Policy redundancy). *A policy p in policy set S is* redundant *if removing it does not change the effective access predicate:*

$$\text{effective}_S(T, \text{CMD}) = \text{effective}_{S \setminus \{p\}}(T, \text{CMD})$$

Lemma 5.1 (Subsumed permissive policy is redundant). *If permissive policy p_2 is subsumed by another permissive policy $p_1 \in P$, then p_2 is redundant in P.*

Sketch. Since $[\![p_2]\!] \subseteq [\![p_1]\!]$:

$$\bigvee_{p \in P} [\![p]\!] = [\![p_1]\!] \vee [\![p_2]\!] \vee \bigvee_{p \in P \setminus \{p_1, p_2\}} [\![p]\!] = [\![p_1]\!] \vee \bigvee_{p \in P \setminus \{p_1, p_2\}} [\![p]\!]$$

by absorption ($A \vee B = A$ when $B \subseteq A$). Removing p_2 leaves the disjunction unchanged. □

A sufficient syntactic condition for policy subsumption: $p_1 \sqsupseteq p_2$ if for every clause $c_2 \in p_2.\mathit{clauses}$, there exists a clause $c_1 \in p_1.\mathit{clauses}$ such that $c_1 \sqsubseteq c_2$ (i.e., c_1 has a subset of c_2's atoms, so c_1 is at least as permissive).

5.7 Worked Example

Consider two policies on the `projects` table for `SELECT`:

```
POLICY tenant_isolation      POLICY soft_delete
  PERMISSIVE                   RESTRICTIVE
  FOR SELECT                   FOR SELECT
  SELECTOR has_column('tenant_id') SELECTOR has_column('is_deleted')
  CLAUSE                       CLAUSE
    col('tenant_id') =           col('is_deleted') = lit(false)
      session('app.tenant_id')
```

Both selectors match `projects` (which has both `tenant_id` and `is_deleted`).
Partition: $P = \{\text{tenant\_isolation}\}$, $R = \{\text{soft\_delete}\}$.
Effective predicate:

$$\begin{aligned}\text{effective}(\texttt{projects}, \texttt{SELECT}) &= [\![\text{tenant\_isolation}]\!] \wedge [\![\text{soft\_delete}]\!] \\ &= (\texttt{col('tenant\_id')} = \texttt{session('app.tenant\_id')}) \\ &\quad \wedge (\texttt{col('is\_deleted')} = \texttt{lit(false)})\end{aligned}$$

A `SELECT` on `projects` returns only rows where the tenant matches *and* the row is not soft-deleted.

6 Selectors & Table Matching

Selectors decouple policies from specific table names. Instead of enumerating tables, a policy declares structural criteria. Tables matching those criteria receive the policy automatically — including tables added in the future.

6.1 Selector Predicates

Definition 6.1 (Selector). *A selector is a predicate over table metadata, constructed from the following grammar:*

```
<selector>      ::= <base_selector>
                   | <selector> "AND" <selector>
                   | <selector> "OR" <selector>
                   | "NOT" <selector>
                   | "(" <selector> ")"
                   | "ALL"

<base_selector> ::= "has_column(" <identifier> ("," <type>)? ")"
                   | "in_schema(" <identifier> ")"
                   | "named(" <pattern> ")"
                   | "tagged(" <tag> ")"
```

Where:

- `has_column(name, type?)` — matches tables that have a column with the given name, optionally restricted to a specific type.
- `in_schema(s)` — matches tables in the specified PostgreSQL schema.
- `named(pat)` — matches tables whose name matches the given pattern (SQL `LIKE` syntax).
- `tagged(t)` — matches tables that carry the specified metadata tag.
- `ALL` — matches every table in the governed set.

6.2 Table Metadata Context

Definition 6.2 (Table metadata context). *The metadata context M is the set of structural facts about all tables in the governed database, extracted from* `pg_catalog`*:*

$$M = \{(table\_name, schema\_name, columns, tags) \mid table \in governed\_tables\}$$

Where columns is a set of $(column\_name, column\_type)$ *pairs, and tags is a set of string labels.*

6.3 Matching Function

Definition 6.3 (Selector matching). *The function* match *evaluates a selector against the metadata context to produce a set of matching tables:*

$$\begin{aligned}
\text{match} &: \text{Selector} \times M \to \mathcal{P}(\text{Table}) \\
\text{match}(\texttt{has\_column}(n,t), M) &= \{T \in M \mid (n,t') \in T.\mathit{columns} \wedge (t = \_ \vee t = t')\} \\
\text{match}(\texttt{in\_schema}(s), M) &= \{T \in M \mid T.\mathit{schema} = s\} \\
\text{match}(\texttt{named}(\mathit{pat}), M) &= \{T \in M \mid T.\mathit{name}\ \texttt{LIKE}\ \mathit{pat}\} \\
\text{match}(\texttt{tagged}(t), M) &= \{T \in M \mid t \in T.\mathit{tags}\} \\
\text{match}(s_1\ \texttt{AND}\ s_2, M) &= \text{match}(s_1, M) \cap \text{match}(s_2, M) \\
\text{match}(s_1\ \texttt{OR}\ s_2, M) &= \text{match}(s_1, M) \cup \text{match}(s_2, M) \\
\text{match}(\texttt{NOT}\ s, M) &= M \setminus \text{match}(s, M) \\
\text{match}(\texttt{ALL}, M) &= M
\end{aligned}$$

Example: In our running schema:

$$\begin{aligned}
\text{match}(\texttt{has\_column('tenant\_id')}, M) &= \{\texttt{users}, \texttt{projects}, \texttt{comments}\} \\
\text{match}(\texttt{has\_column('is\_deleted')}, M) &= \{\texttt{projects}\} \\
\text{match}(\texttt{ALL}, M) &= \{\texttt{users}, \texttt{projects}, \texttt{tasks}, \texttt{comments}, \texttt{files}, \texttt{config}\}
\end{aligned}$$

6.4 Connection to Formal Concept Analysis

The selector mechanism admits a natural interpretation in Formal Concept Analysis (FCA) [5]:

- **Objects** = the set of governed tables
- **Attributes** = structural properties (has column X, in schema Y, etc.)
- **Incidence relation** = table T has attribute A iff the corresponding base selector is satisfied

A **formal concept** is a pair (*extent*, *intent*) where:

- *extent* is a maximal set of tables sharing all attributes in *intent*
- *intent* is a maximal set of attributes shared by all tables in *extent*

The closure operators forming the Galois connection between $\mathcal{P}$(Tables) and $\mathcal{P}$(Attributes) are:

$$\alpha(T_{\text{set}}) = \{a \in \text{Attributes} \mid \forall T \in T_{\text{set}} : T \text{ has } a\}$$
$$\beta(A_{\text{set}}) = \{T \in \text{Tables} \mid \forall a \in A_{\text{set}} : T \text{ has } a\}$$

A selector s defines an attribute set, and $\text{match}(s, M)$ computes β applied to that set. This means selectors are computing extents of (possibly non-closed) attribute sets. The formal concepts represent the natural "policy groups" — maximal clusters of tables sharing structural properties.

6.5 Selector Monotonicity

Property 6.1 (Selector monotonicity). *For a fixed selector s, if a new table T_{new} is added to the governed database and* $\text{match}(s, M)$ *included tables $T_1, \dots, T_k$, then* $\text{match}(s, M \cup \{T_{\text{new}}\}) \supseteq \{T_1, \dots, T_k\}$.

Proof. Selector evaluation depends only on each table's own metadata. Adding a new table cannot change the metadata of existing tables, so existing matches are preserved. The new table either matches (expanding the set) or doesn't (leaving it unchanged). □

This property ensures that policy coverage is stable under schema evolution: existing protections are never silently dropped when new tables are added.

7 Relationship Traversal

Some tables do not carry a direct `tenant_id` column but inherit tenant context through foreign-key relationships. The `tasks` table in our running example has no `tenant_id` but references `projects`, which does. Relationship traversal allows policies to express this indirect access pattern.

7.1 Declared Relationships

Definition 7.1 (Relationship). *A declared relationship is a 4-tuple:*

$$\texttt{rel}(\mathit{source\_table}, \mathit{source\_col}, \mathit{target\_table}, \mathit{target\_col})$$

Where source_table.source_col is a foreign key referencing target_table.target_col.

Example: `rel(tasks, project_id, projects, id)` declares that `tasks.project_id` references `projects.id`.

Relationships are declared explicitly in the policy configuration, not inferred from database constraints. This ensures that only intentional access paths are used for policy traversal.

7.2 Traversal Atoms

Definition 7.2 (Traversal atom). *A traversal atom extends the atom grammar with an existential subquery:*

```
<traversal_atom> ::= "exists(" <relationship> "," <clause> ")"
```

Semantics: $\texttt{exists}(\texttt{rel}(S, sc, T, tc), \mathit{clause})$ *is satisfied for a row* r *of table* S *if there exists a row* r' *in table* T *such that* $r.sc = r'.tc$ *and* $\mathit{clause}(r')$ *holds.*

Example:

```
exists(
  rel(tasks, project_id, projects, id),
  {col('tenant_id') = session('app.tenant_id')}
)
```

This atom on the `tasks` table checks: "there exists a project row whose `id` matches this task's `project_id` and whose `tenant_id` matches the session tenant." This provides tenant isolation for `tasks` through the relationship to `projects`.

Extended BNF:

```
<atom> ::= <value_source> <binary_op> <value_source>
         | <value_source> <unary_op>
         | <traversal_atom>

<traversal_atom> ::= "exists(" <relationship> "," <clause> ")"

<relationship> ::= "rel(" <identifier> "," <identifier> ","
                         <identifier> "," <identifier> ")"
```

7.3 Traversal Depth

Definition 7.3 (Traversal depth). *The* depth *of an atom is defined recursively:*

$$\begin{aligned} \text{depth}(\mathit{value\_source}\ \mathit{op}\ \mathit{value\_source}) &= 0 \\ \text{depth}(\mathit{value\_source}\ \mathit{unary\_op}) &= 0 \\ \text{depth}(\texttt{exists}(\mathit{rel}, \mathit{clause})) &= 1 + \max(\{\text{depth}(a) \mid a \in \mathit{clause}\}) \end{aligned}$$

The depth of a clause is $\max(\{\text{depth}(a) \mid a \in \textit{clause}\})$.

Definition 7.4 (Maximum traversal depth). *The policy engine enforces a global maximum traversal depth D (default $D = 2$). Any atom with* $\text{depth}(a) > D$ *is rejected at definition time.*

7.4 Properties

Property 7.1 (Bounded compilation). *A traversal atom of depth d compiles to at most d nested* `EXISTS` *subqueries. With maximum depth D, the compiled SQL has at most D levels of nesting.*

Proof. By structural induction on the traversal atom. Base case: a non-traversal atom compiles to a flat SQL expression (depth 0). Inductive step: `exists(`*rel*, *clause*`)` compiles to `EXISTS (SELECT 1 FROM T WHERE join_cond AND compile(clause))`, adding one nesting level to whatever `compile(`*clause*`)` produces. □

Property 7.2 (No recursive traversal). *The algebra does not support recursive relationship traversal. Hierarchical access patterns (e.g., org trees) require pre-computed closure tables rather than recursive policy expressions.*

This restriction is essential for decidability. Recursive traversal would require fixpoint computation, pushing the algebra beyond the decidable fragment.

7.5 Example: Tenant Isolation via Traversal

For `tasks` (no `tenant_id`) and `files` (no `tenant_id`):

```
POLICY tenant_isolation_via_project
  PERMISSIVE
  FOR SELECT, INSERT, UPDATE, DELETE
  SELECTOR named('tasks') OR named('files')
  CLAUSE
    exists(
      rel(_, project_id, projects, id),
      {col('tenant_id') = session('app.tenant_id')}
    )
```

When applied to `tasks`, the compiled SQL becomes:

```
CREATE POLICY tenant_isolation_via_project ON tasks
  USING (EXISTS (
    SELECT 1 FROM projects
    WHERE projects.id = tasks.project_id
      AND projects.tenant_id = current_setting('app.tenant_id')
  ));
```

8 Analysis

Because the algebra is decidable, policies can be analyzed *at design time*, before any SQL is generated or executed. This section defines the key analysis operations: satisfiability, subsumption, redundancy, contradiction, and tenant isolation proofs.

8.1 Satisfiability

Satisfiability asks: "Can this clause/policy ever match any row?" An unsatisfiable clause is a contradiction — a bug in the policy definition.

8.1.1 SMT Encoding. Each atom is encoded as an SMT formula in the combined theory QF-LIA ∪ QF-EUF:

```
function encode_atom(a) -> SMT formula:
   match a:
      (col(x), =, col(y))     -> x_var = y_var
      (col(x), =, session(k)) -> x_var = k_var
      (col(x), =, lit(v))     -> x_var = v_const
      (col(x), !=, lit(v))    -> x_var != v_const
      (col(x), <, lit(v))     -> x_var < v_const
      (col(x), IN, lit([v1..])) -> x_var = v1 | ... | x_var = vn
      (col(x), IS NULL, _)    -> x_null = true
      (col(x), IS NOT NULL, _) -> x_null = false
      exists(rel, clause)      -> encode_traversal(rel, clause)

function encode_clause(c) -> SMT formula:
   return conjunction of encode_atom(a) for a in c

function encode_traversal(rel(S, sc, T, tc), clause):
   target_vars <- fresh_vars(T)
   join_cond  <- sc_var = target_vars[tc]
   return exists target_vars: join_cond and
         encode_clause(clause)[T -> target_vars]
```

The satisfiability check: submit the formula to an SMT solver. If the solver returns UNSAT, the clause is a contradiction.

8.1.2 Pseudocode.

```
function check_satisfiability(clause c) -> {SAT, UNSAT, UNKNOWN}:
   c' <- normalize_clause(c)
   if c' = bot:
      return UNSAT      -- Syntactic contradiction detected
   phi <- encode_clause(c')
   result <- smt_solve(phi, timeout=5s)
   return result
```

8.1.3 Example. Consider the clause: {`col('role') = lit('admin')`, `col('role') = lit('viewer')`}.

After normalization, syntactic contradiction detection finds two equality atoms on the same column with different literal values. The clause reduces to ⊥ without needing the SMT solver.

For a subtler case: {`col('age') > lit(65)`, `col('age') < lit(18)`}. Syntactic checks may not catch this. The SMT encoding produces:

$$age\_var > 65 \land age\_var < 18$$

The solver returns UNSAT: no integer satisfies both constraints.

8.2 Subsumption

Subsumption determines whether one policy's access grant is entirely contained within another's.

Definition 8.1 (Policy subsumption via clauses). *Permissive policy p_1 subsumes permissive policy p_2, written $p_1 \sqsupseteq p_2$, if:*

$$\forall c_2 \in p_2.clauses,\ \exists c_1 \in p_1.clauses : c_1 \sqsubseteq c_2$$

That is, every clause of p_2 is subsumed by some clause of p_1.

Algorithm:

```
function check_subsumption(p1, p2) -> bool:
    for each c2 in p2.clauses:
        found <- false
        for each c1 in p1.clauses:
            if clause_subsumes(c1, c2):
                found <- true
                break
        if not found:
            return false
    return true

function clause_subsumes(c1, c2) -> bool:
    -- Syntactic check: c1's atoms are a subset of c2's
    if atoms(c1) is subset of atoms(c2):
        return true
    -- Semantic check: ask SMT if c2 entails c1
    phi <- encode_clause(c2) and not encode_clause(c1)
    return smt_solve(phi) = UNSAT
```

8.3 Redundancy

Definition 8.2 (Redundancy). *Policy p is redundant in policy set S if:*

$$\text{effective}_S(T, \text{CMD}) \equiv \text{effective}_{S\setminus\{p\}}(T, \text{CMD})$$

Algorithm:

```
function check_redundancy(p, S, T, CMD) -> bool:
    if p.type = permissive:
        P_others <- P(T, CMD) \ {p}
        for each c in p.clauses:
            subsumed <- false
            for each p' in P_others:
                for each c' in p'.clauses:
                    if clause_subsumes(c', c):
                        subsumed <- true; break
                if subsumed: break
            if not subsumed:
                return false
        return true
    else: -- restrictive
        phi_perm <- encode_permissive_disjunction(P(T, CMD))
        phi_p    <- encode_policy(p)
        phi      <- phi_perm and not phi_p
        return smt_solve(phi) = UNSAT
```

8.4 Contradiction

Definition 8.3 (Contradiction). *The effective access predicate for table T under command* CMD *is* contradictory *if it is unsatisfiable:*

$$\text{effective}(T, \text{CMD}) = \bot$$

This means no rows are ever accessible — likely a policy authoring error.

Algorithm:

```
function check_contradiction(T, CMD, S) -> bool:
    phi <- encode(effective(T, CMD))
    return smt_solve(phi) = UNSAT
```

Example: If the only permissive policy on `projects` requires `col('role') = lit('admin')` and the only restrictive policy requires `col('role') = lit('viewer')`, the effective predicate is:

$$\texttt{col('role')} = \texttt{lit('admin')} \wedge \texttt{col('role')} = \texttt{lit('viewer')}$$

This is unsatisfiable. The analysis flags a contradiction.

8.5 Tenant Isolation Proof

The most important analysis: proving that tenant data is properly isolated. The question is: *can any session ever access a row belonging to a different tenant?*

8.5.1 Formal Statement.

Definition 8.4 (Tenant isolation). *Table T satisfies tenant isolation if there is no row r and two distinct sessions $s_1 \neq s_2$ (differing in* `app.tenant_id`*) such that both sessions can access r:*

$$\neg\exists\, r, s_1, s_2 : \ s_1.\mathit{tenant\_id} \neq s_2.\mathit{tenant\_id} \wedge \text{effective}(T, \text{CMD})[s \mapsto s_1](r) \wedge \text{effective}(T, \text{CMD})[s \mapsto s_2](r)$$

*If this formula is **unsatisfiable**, tenant isolation holds.*

8.5.2 SMT Encoding.

```
function prove_tenant_isolation(T, CMD, S)
    -> {PROVEN, FAILED, UNKNOWN}:
    s1_vars <- fresh_session_vars("s1")
    s2_vars <- fresh_session_vars("s2")
    row_vars <- fresh_row_vars(T)

    phi_diff <- s1_vars['app.tenant_id']
                != s2_vars['app.tenant_id']
    phi_eff1 <- encode(effective(T, CMD))
                [session -> s1_vars, row -> row_vars]
    phi_eff2 <- encode(effective(T, CMD))
                [session -> s2_vars, row -> row_vars]

    phi <- phi_diff and phi_eff1 and phi_eff2

    result <- smt_solve(phi)
    if result = UNSAT:
        return PROVEN
    else if result = SAT:
        return FAILED
    else:
        return UNKNOWN
```

8.5.3 Sufficient Condition.

Theorem 8.1 (Sufficient condition for tenant isolation). *If every permissive clause for table T contains the atom* `col('tenant_id') = session('app.tenant_id')` *(directly or via a depth-1 traversal to a table with such a clause), then tenant isolation holds for T.*

Sketch. Suppose two sessions s_1 and s_2 with different tenant IDs both access row r. Each must satisfy at least one permissive clause (by default deny). Every permissive clause requires the row's `tenant_id` (direct or via traversal) to equal the session's `app.tenant_id`. So:

$$r.tenant\_id = s_1.\texttt{app.tenant\_id}$$
$$r.tenant\_id = s_2.\texttt{app.tenant\_id}$$

Therefore $s_1.\texttt{app.tenant\_id} = s_2.\texttt{app.tenant\_id}$, contradicting the assumption that they differ.

□

9 Optimization & Rewrite Rules

The policy engine applies rewrite rules to simplify policies before compilation. Each rule preserves the denotation (semantic equivalence) while reducing syntactic complexity.

9.1 Rewrite Rules

Rule 1: Idempotence.

$$a \wedge a = a$$

Duplicate atoms within a clause are removed.

Example: $\{\texttt{col('x')} = \texttt{lit(1)}, \texttt{col('x')} = \texttt{lit(1)}\} \rightarrow \{\texttt{col('x')} = \texttt{lit(1)}\}$.

Rule 2: Absorption.

$$c_1 \vee (c_1 \wedge c_2) = c_1$$

In a disjunction of clauses within a policy, if clause c_1 subsumes clause $c_1 \cup c_2$ (because $c_1 \subseteq c_1 \cup c_2$), the more restrictive clause is absorbed.

Rule 3: Contradiction Elimination.

$$\texttt{col}(x) = \texttt{lit}(v_1) \wedge \texttt{col}(x) = \texttt{lit}(v_2) \rightarrow \bot \quad \text{when } v_1 \neq v_2$$

A clause containing contradictory atoms is replaced by $\bot$ and removed from the policy's clause set.

Rule 4: Tautology Detection.

$$\texttt{col}(x) = \texttt{col}(x) \rightarrow \top$$

A tautological atom is removed from a clause (since $a \wedge \top = a$). If all atoms in a clause are tautological, the clause becomes $\top$. A policy containing a $\top$ clause is equivalent to $\top$ (since $\top \vee c = \top$).

Rule 5: Subsumption Elimination in Disjunctions.

$$\text{If } c_1 \sqsubseteq c_2 \ (c_1 \text{ subsumes } c_2), \text{ then } c_1 \vee c_2 = c_1$$

Within a policy's clause set, if one clause subsumes another, the subsumed (more restrictive) clause is removed.

Rule 6: Atom Merging.

$$\texttt{col}(x) = \texttt{lit}(v) \wedge \texttt{col}(x) \texttt{ IN } \texttt{lit}([v, w_1, w_2, \ldots]) \rightarrow \texttt{col}(x) = \texttt{lit}(v)$$

When an equality atom and an IN-list atom reference the same column, and the equality value appears in the IN-list, the IN-list is redundant.

More generally: $\texttt{col}(x) \texttt{ IN } \texttt{lit}(S_1) \wedge \texttt{col}(x) \texttt{ IN } \texttt{lit}(S_2) \rightarrow \texttt{col}(x) \texttt{ IN } \texttt{lit}(S_1 \cap S_2)$.

9.2 Policy Normal Form

Definition 9.1 (Policy normal form). *A policy is in normal form when:*

1. *Every clause is in clause normal form (Def. 3.2).*
2. *All unsatisfiable clauses (⊥) have been removed from the clause set.*
3. *No clause in the set is subsumed by another clause in the same set.*
4. *No further rewrite rules (1–6) apply.*

If removing unsatisfiable clauses leaves the clause set empty, the policy itself is unsatisfiable and is flagged as an error.

9.3 Normalization Algorithm

```
function normalize_policy(p) -> p':
    -- Phase 1: normalize individual clauses
    clauses <- {normalize_clause(c) | c in p.clauses}

    -- Phase 2: remove unsatisfiable clauses
    clauses <- {c in clauses | c != bot}

    -- Phase 3: apply rewrite rules until fixpoint
    changed <- true
    while changed:
        changed <- false

        -- Absorption / subsumption elimination (Rules 2, 5)
        for each pair (c1, c2) in clauses x clauses, c1 != c2:
            if atoms(c1) is subset of atoms(c2):
                clauses <- clauses \ {c2}
                changed <- true
                break

        -- Atom merging within each clause (Rule 6)
        for each c in clauses:
            c' <- merge_atoms(c)
            if c' != c:
                clauses <- (clauses \ {c}) union {c'}
                changed <- true

    if clauses = empty:
        flag_error("Policy is entirely unsatisfiable")

    return p with clauses <- clauses
```

9.4 Termination

Property 9.1 (Termination). *The normalization algorithm terminates.*

Proof. Define a complexity measure on a policy as the pair $(|clauses|, \sum_{c \in clauses} |\text{atoms}(c)|)$ under lexicographic ordering. Each rewrite rule strictly reduces this measure:

- Contradiction elimination (Rule 3): removes a clause, reducing $|clauses|$.
- Absorption/subsumption elimination (Rules 2, 5): removes a clause.
- Atom merging (Rule 6): reduces $|\text{atoms}(c)|$ for some clause.

- Idempotence (Rule 1): reduces $|\text{atoms}(c)|$ for some clause.
- Tautology detection (Rule 4): reduces $|\text{atoms}(c)|$ for some clause.

Since the measure is a natural number pair in a well-order, the algorithm must terminate. □

9.5 Correctness

Property 9.2 (Correctness). *Each rewrite rule preserves the denotation of the policy:* $[\![p]\!] = [\![\text{normalize}(p)]\!]$.

Sketch. Each rule is a standard logical equivalence:

- Idempotence: $a \wedge a \equiv a$
- Absorption: $A \vee (A \wedge B) \equiv A$
- Contradiction elimination: removing $\bot$ from a disjunction does not change it
- Tautology detection: $a \wedge \top \equiv a$
- Subsumption elimination: $A \vee B \equiv A$ when $B \subseteq A$
- Atom merging: $(x = v) \wedge (x \in S)$ where $v \in S \equiv x = v$

Each preserves the set of satisfying rows. □

9.6 Worked Example

Starting from a policy with 4 components:

```
POLICY example_policy PERMISSIVE FOR SELECT SELECTOR ALL
  CLAUSE c1: {col('tenant_id') = session('tid')}
  CLAUSE c2: {col('tenant_id') = session('tid'),
              col('active') = lit(true)}
  CLAUSE c3: {col('role') = lit('admin'),
              col('role') = lit('viewer')}
  CLAUSE c4: {col('is_deleted') = lit(false)}
```

Step 1: Normalize clauses.

- c_1: already normal.
- c_2: already normal.
- c_3: contradiction detected (`role = 'admin'` $\wedge$ `role = 'viewer'`) $\to \bot$.
- c_4: already normal.

Step 2: Remove unsatisfiable clauses. $c_3 = \bot \to$ removed. Remaining: $\{c_1, c_2, c_4\}$.

Step 3: Subsumption elimination. $\text{atoms}(c_1) \subseteq \text{atoms}(c_2) \to c_1$ subsumes $c_2 \to$ remove c_2. Remaining: $\{c_1, c_4\}$. No further subsumption.

Result: 4 atoms $\to$ 2 clauses with 1 atom each. The simplest correct enforcement.

10 Compilation

Compilation is the deterministic translation of normalized policies to native PostgreSQL security artifacts. This section defines the compilation function, proves its correctness, and specifies the naming conventions for generated artifacts.

10.1 PostgreSQL Artifact Set

Definition 10.1 (Artifact set). *The compilation output for a governed table T is a set of SQL statements drawn from:*

- `ALTER TABLE T ENABLE ROW LEVEL SECURITY`
- `ALTER TABLE T FORCE ROW LEVEL SECURITY`
- `GRANT <privileges> ON T TO <role>`

- `CREATE POLICY <name> ON T [AS {PERMISSIVE|RESTRICTIVE}] [FOR <cmd>] USING (<expr>) [WITH CHECK (<expr>)]`

10.2 Compilation Function

Compilation is defined as structural recursion over the policy algebra's types.

10.2.1 *Compile Atom.*

```
function compile_atom(a) -> SQL expression:
   match a:
      (col(x), =, session(k)) -> "x = current_setting('k')"
      (col(x), =, lit(v))     -> "x = v"
      (col(x), !=, lit(v))    -> "x <> v"
      (col(x), <, lit(v))     -> "x < v"
      (col(x), >, lit(v))     -> "x > v"
      (col(x), <=, lit(v))    -> "x <= v"
      (col(x), >=, lit(v))    -> "x >= v"
      (col(x), IN, lit(vs))   -> "x IN (v1, v2, ...)"
      (col(x), NOT IN, lit(vs)) -> "x NOT IN (v1, v2, ...)"
      (col(x), IS NULL, _)    -> "x IS NULL"
      (col(x), IS NOT NULL, _) -> "x IS NOT NULL"
      (col(x), LIKE, lit(v))  -> "x LIKE 'v'"
      (col(x), =, col(y))     -> "x = y"
      (col(x), =, fn(f, args)) -> "x = f(args)"
      exists(rel, clause)      -> compile_traversal(rel, clause)
```

10.2.2 *Compile Traversal.*

```
function compile_traversal(rel(S, sc, T, tc), clause)
   -> SQL expression:
   inner <- compile_clause(clause)
   return "EXISTS (SELECT 1 FROM T
          WHERE T.tc = S.sc AND inner)"
```

Where S in `S.sc` refers to the outer table being policy-protected.

10.2.3 *Compile Clause.*

```
function compile_clause(c) -> SQL expression:
   parts <- [compile_atom(a) | a in c, sorted]
   return join(parts, " AND ")
```

An empty clause ($\top$) compiles to `true`.

10.2.4 *Compile Policy.*

```
function compile_policy(p, T) -> SQL statement:
   type_clause <- "AS " + upper(p.type)
   cmd_clause  <- "FOR " + join(p.commands, ", ")
   using_expr  <- join([compile_clause(c) |
                     c in p.using_clauses], " OR ")
   check_expr  <- join([compile_clause(c) |
                     c in p.check_clauses], " OR ")

   sql <- "CREATE POLICY " + p.name + "_" + T.name
       + " ON " + T.qualified_name
       + " " + type_clause
       + " " + cmd_clause
       + " USING (" + using_expr + ")"

   if check_expr != using_expr
       and p.commands intersect {INSERT, UPDATE} != empty:
        sql <- sql + " WITH CHECK (" + check_expr + ")"

   return sql
```

10.2.5 Compile Policy Set for Table.

```
function compile_table(T, CMD, S) -> [SQL statement]:
    statements <- []
    statements.append("ALTER TABLE " + T.qualified_name
                   + " ENABLE ROW LEVEL SECURITY")
    statements.append("ALTER TABLE " + T.qualified_name
                   + " FORCE ROW LEVEL SECURITY")
    for each p in Policies(T, CMD):
        statements.append(compile_policy(p, T))
    return statements
```

10.3 Compilation Correctness

Theorem 10.1 (Compilation correctness). *For any table T, command* CMD*, and policy set S, the set of rows accessible under the compiled SQL policies equals the set of rows satisfying* effective(T, CMD)*:*

$$\{r \mid r \textit{ accessible under compiled SQL}\} = \{r \mid \mathrm{effective}(T, \mathrm{CMD})(r) = \textit{true}\}$$

Sketch (by structural induction). **Base case** (atoms). Each atom compiles to a SQL expression that evaluates to `true` on exactly the rows satisfying the atom's semantics:

- `col`(x) = `session`(k) compiles to `x = current_setting('k')`. PostgreSQL evaluates `current_setting('k')` at query time, returning the session value. The comparison produces the same boolean result as the atom's denotation.
- `col`(x) = `lit`(v) compiles to `x = v`. Direct correspondence.
- `exists`(`rel`(S, sc, T, tc), *clause*) compiles to `EXISTS (SELECT 1 FROM T WHERE T.tc = S.sc AND ...)`. The EXISTS subquery returns true iff there exists a matching row in T satisfying the join condition and the compiled clause — matching the traversal atom's semantics.

Inductive step (clauses). A clause $\{a_1, \ldots, a_n\}$ compiles to `compile`(a_1) `AND ... AND` `compile`(a_n). By the base case, each compiled atom has the correct denotation. SQL AND has standard conjunction semantics.

Inductive step (policies). A policy's clause set $\{c_1, \ldots, c_k\}$ compiles to `compile`(c_1) `OR ... OR` `compile`(c_k) in the USING expression. SQL OR has standard disjunction semantics.

Inductive step (composition). PostgreSQL composes permissive policies by OR and restrictive policies by AND, then takes their conjunction. This exactly mirrors Definition 5.2. □

10.4 Connection to Galois Connections

The compilation function and the denotational semantics form an adjunction [3]. Define:

- L = the lattice of DSL policy expressions, ordered by subsumption
- R = the lattice of SQL predicate expressions, ordered by logical implication
- $\alpha : L \to R$ = the compilation function (`compile`)
- $\gamma : R \to L$ = the abstraction function (parsing compiled SQL back to DSL, where possible)

The pair (α, γ) forms a Galois connection when:

$$\forall\, l \in L,\ r \in R: \quad \alpha(l) \sqsubseteq_R r \iff l \sqsubseteq_L \gamma(r)$$

In practice, γ is partial (not all SQL can be parsed back). The important direction is α: compilation preserves the ordering.

Property 10.1 (Monotonicity of compilation). *If policy p_1 subsumes policy p_2 in the DSL ($p_1 \sqsupseteq p_2$), then the compiled SQL of p_1 is at least as permissive as the compiled SQL of p_2.*

10.5 Determinism

Property 10.2 (Determinism). *Two policies with identical normal forms produce identical SQL output.*

Proof. The compilation function is purely structural with no randomness or ambient state dependency. Normal form is unique (by confluence of rewrite rules). Therefore the output is determined entirely by the normal form. □

10.6 Naming Convention

Generated artifacts follow a deterministic naming scheme:

$$\langle \textit{policy\_name} \rangle\_\langle \textit{table\_name} \rangle$$

Examples:

- Policy `tenant_isolation` on table `projects` → `CREATE POLICY tenant_isolation_projects ON projects ...`
- Policy `soft_delete` on table `projects` → `CREATE POLICY soft_delete_projects ON projects ...`

10.7 Full Compilation Example

Given the running example policies from Sections 4 and 5 applied to `projects`:

Input (normalized policies):

```
POLICY tenant_isolation PERMISSIVE FOR SELECT
  CLAUSE {col('tenant_id') = session('app.tenant_id')}

POLICY soft_delete RESTRICTIVE FOR SELECT
  CLAUSE {col('is_deleted') = lit(false)}
```

Compiled output:

```
-- Enable RLS
ALTER TABLE public.projects ENABLE ROW LEVEL SECURITY;
ALTER TABLE public.projects FORCE ROW LEVEL SECURITY;

-- Permissive: tenant isolation
CREATE POLICY tenant_isolation_projects
  ON public.projects
  AS PERMISSIVE
  FOR SELECT
  USING (tenant_id = current_setting('app.tenant_id'));

-- Restrictive: soft delete filter
CREATE POLICY soft_delete_projects
```

```
    ON public.projects
    AS RESTRICTIVE
    FOR SELECT
    USING (is_deleted = false);
```

Effective SQL predicate (what PostgreSQL enforces):

```
-- (OR of permissive) AND (AND of restrictive)
(tenant_id = current_setting('app.tenant_id'))
AND
(is_deleted = false)
```

11 Drift Detection & Reconciliation

After compilation and application, the database state must be continuously monitored to ensure it matches the intended policy state. *Drift* is any discrepancy between the observed database state and the expected state derived from the policy algebra.

11.1 Observed and Expected State

Definition 11.1 (Observed state). *The observed state O is the set of security-relevant facts extracted from the live database via introspection:*

$$\begin{aligned} O = \{&\mathit{rls\_enabled} : \text{Table} \longrightarrow \text{bool},\\ &\mathit{rls\_forced} : \text{Table} \longrightarrow \text{bool},\\ &\mathit{policies} : \text{Table} \longrightarrow \text{Set}(\text{PolicyFact}),\\ &\mathit{grants} : \text{Table} \longrightarrow \text{Set}(\text{GrantFact})\ \} \end{aligned}$$

Where PolicyFact *captures the name, type (permissive/restrictive), command, roles, USING expression, and WITH CHECK expression of each live policy.*

Definition 11.2 (Expected state). *The expected state E is the output of the compilation function (Section 10) applied to the current policy set:*

$$E = \text{compile}(\text{PolicySet}, \text{governed\_tables})$$

11.2 Drift

Definition 11.3 (Drift). *Drift is the symmetric difference between observed and expected states:*

$$\text{Drift} = O \bigtriangleup E = (O \setminus E) \cup (E \setminus O)$$

11.3 Drift Classification

Drift is classified into the following types:

11.4 Drift Detection Algorithm

```
function detect_drift(S, governed_tables) -> Set(DriftItem):
    drift <- empty set
    E <- compile(S, governed_tables)
    O <- introspect(governed_tables)

    for each T in governed_tables:
        if not O.rls_enabled(T):
            drift.add(DriftItem(T, "rls_disabled"))
```

Table 2. Drift classification

Drift type	Description	Severity
Missing policy	Expected policy not found in database	Critical
Extra policy	Unmanaged policy found on governed table	Warning
Modified policy	Policy exists but USING/CHECK expression differs	Critical
Missing GRANT	Expected GRANT not present	Critical
Extra GRANT	Unmanaged GRANT on governed table	Warning
RLS disabled	`relrowsecurity = false` on governed table	Critical
RLS not forced	`relforcerowsecurity = false` on governed table	High

```
        if not O.rls_forced(T):
            drift.add(DriftItem(T, "rls_not_forced"))

        expected_policies <- E.policies(T)
        observed_policies <- O.policies(T)

        for each ep in expected_policies:
            op <- find_by_name(observed_policies, ep.name)
            if op = null:
                drift.add(DriftItem(T, "missing_policy", ep))
            else if op.using_expr != ep.using_expr
                    or op.check_expr != ep.check_expr
                    or op.type != ep.type:
                drift.add(DriftItem(T, "modified_policy",
                          ep, op))

        for each op in observed_policies:
            if not find_by_name(expected_policies, op.name):
                drift.add(DriftItem(T, "extra_policy", op))

    return drift
```

The `introspect` function queries PostgreSQL system catalogs [8]:

```
-- Policy introspection
SELECT schemaname, tablename, policyname, permissive,
       roles, cmd, qual, with_check
FROM pg_policies
WHERE schemaname = 'public';

-- RLS status
SELECT relname, relrowsecurity, relforcerowsecurity
FROM pg_class
WHERE relnamespace = 'public'::regnamespace;
```

11.5 Reconciliation Strategies

When drift is detected, three strategies are available:

Auto-remediate: Automatically re-apply the expected state. Suitable for `missing_policy`, `modified_policy`, `rls_disabled`, and `rls_not_forced` drift types.

Alert: Notify operators without taking action. Suitable for `extra_policy` and `extra_grant` drift types, which may represent intentional manual overrides that require human review.

Quarantine: For unmanaged tables, log the finding and optionally restrict access until reviewed.

```
function reconcile(drift_items, strategy) -> Set(SQL):
    actions <- empty set
    for each item in drift_items:
        match (strategy, item.type):
            (auto, "missing_policy") -> actions.add(
                                          item.expected_sql)
            (auto, "modified_policy") -> actions.add(
                                          drop(item.observed))
                                       actions.add(
                                          item.expected_sql)
            (auto, "rls_disabled")  -> actions.add(
                                          enable_rls(item.table))
            (auto, "rls_not_forced") -> actions.add(
                                          force_rls(item.table))
            (alert, "extra_policy") -> notify(item)
            (alert, "extra_grant")  -> notify(item)
            (quarantine, _)         -> quarantine(item.table)
    return actions
```

12 The Governance Loop

The governance loop is the top-level operational cycle that ties together all components of the policy algebra: definition, analysis, compilation, application, monitoring, and reconciliation.

12.1 Six Phases

Figure 2 illustrates the six-phase governance loop.

(1) **Define**: Data stewards author policies using the DSL (atoms, clauses, selectors, traversals). Policies are version-controlled.
(2) **Analyze**: The analysis engine (Section 8) validates all policies: satisfiability, contradiction detection, redundancy identification, and tenant isolation proofs. If errors are found, the policy set is rejected and authors are notified.
(3) **Compile**: Validated policies are compiled (Section 10) to PostgreSQL artifacts. The output is deterministic and reproducible.
(4) **Apply**: Compiled SQL statements are executed against the target database in a transaction.
(5) **Monitor**: The drift detection engine (Section 11) periodically introspects the database and compares observed state to expected state.
(6) **Reconcile**: When drift is detected, the reconciliation engine applies the appropriate strategy (auto-remediate, alert, or quarantine) and feeds findings back into the Define phase.

12.2 Governance State Machine

Definition 12.1 (Governance state). *A governance state is a pair:*

$$G = (S, D)$$

Where S is the current policy set and D is the current database state (the observed state from Section 11).

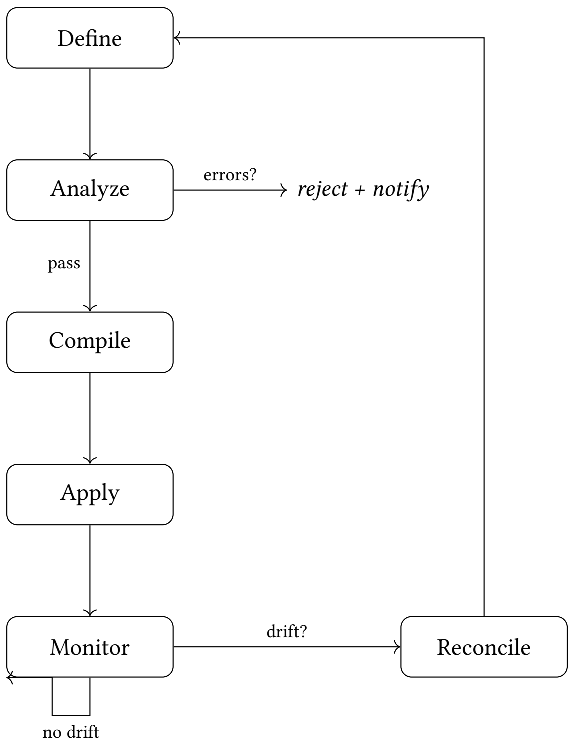

Fig. 2. The six-phase governance loop: Define ⟶ Analyze ⟶ Compile ⟶ Apply ⟶ Monitor ⟶ Reconcile.

Definition 12.2 (Governance transitions). *The governance loop defines transitions:*

$$\begin{aligned}
\text{define} &: (S, D) \longrightarrow (S', D) \\
\text{analyze} &: (S, D) \longrightarrow (S, D) \mid \mathit{error} \\
\text{compile} &: (S, D) \longrightarrow (S, D, E) \\
\text{apply} &: (S, D, E) \longrightarrow (S, D') \\
\text{monitor} &: (S, D) \longrightarrow (S, D, \Delta) \\
\text{reconcile} &: (S, D, \Delta) \longrightarrow (S, D')
\end{aligned}$$

12.3 Convergence

Property 12.1 (Convergence). *Absent external changes to the database, one complete cycle of the governance loop brings drift to zero:*

$$\text{drift}(\text{apply}(\text{compile}(\text{analyze}(S))), S) = \emptyset$$

Sketch. The `compile` function produces expected state E from policy set S. The `apply` function executes E against the database, making observed state O equal to E. The `monitor` function computes $O \triangle E$, which is $E \triangle E = \emptyset$. □

The "absent external changes" caveat is essential: another actor (DBA, migration script, another tool) may modify the database between `apply` and `monitor`, introducing drift that requires another cycle.

12.4 Idempotence

Property 12.2 (Idempotence). *Applying the same compiled artifacts twice produces the same database state:*

$$\text{apply}(E, \text{apply}(E, D)) = \text{apply}(E, D)$$

Sketch. Each compiled artifact is a `CREATE POLICY ... IF NOT EXISTS` or an idempotent `ALTER TABLE ... ENABLE ROW LEVEL SECURITY`. Re-executing these statements on a database already in the target state is a no-op. For `CREATE POLICY` without `IF NOT EXISTS`, the engine uses `DROP POLICY IF EXISTS` followed by `CREATE POLICY`, which is idempotent by construction. □

12.5 Key Invariants

The governance loop maintains the following invariants at steady state (zero drift):

(1) **RLS enabled**: Every governed table has `relrowsecurity` = `true`.
(2) **RLS forced**: Every governed table has `relforcerowsecurity` = `true`.
(3) **Policy match**: For every governed table T, the set of policies on T matches the compiled output of the policy set.
(4) **Grant match**: For every governed table T, the grants on T match the compiled output.
(5) **Tenant isolation**: For every governed table T that is subject to a tenant isolation policy, the isolation proof (Section 8) holds.

13 Complete BNF Grammar

This section assembles all grammar fragments from Sections 2–7 into a standalone grammar for the policy algebra DSL.

```
(* ============================================= *)
(* Policy Algebra DSL -- Complete BNF Grammar *)
(* ============================================= *)

(* --- Top Level --- *)

<policy_set>    ::= <policy>*

<policy>        ::= "POLICY" <identifier>
                    <policy_type>
                    <command_list>
                    <selector_clause>
                    <clause_block>

<policy_type>   ::= "PERMISSIVE" | "RESTRICTIVE"
<command_list>  ::= "FOR" <command> ("," <command>)*
<command>       ::= "SELECT" | "INSERT"
                  | "UPDATE" | "DELETE"
<selector_clause> ::= "SELECTOR" <selector>

(* --- Selectors --- *)

<selector>      ::= <base_selector>
                   | <selector> "AND" <selector>
                   | <selector> "OR" <selector>
                   | "NOT" <selector>
                   | "(" <selector> ")"
```

```
                    | "ALL"

<base_selector> ::= "has_column(" <identifier>
                     ("," <type>)? ")"
                    | "in_schema(" <identifier> ")"
                    | "named(" <pattern> ")"
                    | "tagged(" <tag> ")"

(* --- Clauses --- *)

<clause_block>  ::= "CLAUSE" <clause>
                    ("OR" "CLAUSE" <clause>)*
<clause>        ::= <atom> ("AND" <atom>)*

(* --- Atoms --- *)

<atom>          ::= <value_source> <binary_op>
                    <value_source>
                   | <value_source> <unary_op>
                   | <traversal_atom>

<traversal_atom> ::= "exists(" <relationship> ","
                    <clause> ")"

<relationship>  ::= "rel(" <identifier> ","
                    <identifier> "," <identifier> ","
                    <identifier> ")"

(* --- Value Sources --- *)

<value_source>  ::= "col(" <identifier> ")"
                   | "session(" <string_literal> ")"
                   | "lit(" <literal_value> ")"
                   | "fn(" <identifier> ","
                     "[" <arg_list>? "]" ")"

<arg_list>      ::= <value_source>
                    ("," <value_source>)*

(* --- Operators --- *)

<binary_op>     ::= "=" | "!=" | "<" | ">" | "<="
                    | ">=" | "IN" | "NOT IN"
                    | "LIKE" | "NOT LIKE"

<unary_op>      ::= "IS NULL" | "IS NOT NULL"

(* --- Literals and Identifiers --- *)

<literal_value> ::= <string_literal>
                    | <integer_literal>
                    | <boolean_literal>
```

Table 3. Summary of properties and lemmas

#	Name	Statement
T1.1	Undecidability of arbitrary RLS	Row accessibility under arbitrary SQL RLS predicates is undecidab
P2.1	Decidability of atom satisfiability	Satisfiability of any finite conjunction of atoms is decidable
P3.1	Clause satisfiability	A normalized clause is satisfiable iff $c \neq \bot$
P3.2	Clause subsumption	$c_1 \subseteq c_2$ implies $[\![c_2]\!] \subseteq [\![c_1]\!]$
P3.3	Idempotence	$c \wedge c = c$ for any clause c
P5.1	Monotonicity of permissive extension	Adding a permissive policy can only increase accessible rows
P5.2	Anti-monotonicity of restrictive extension	Adding a restrictive policy can only decrease accessible rows
L5.1	Subsumed permissive redundancy	A permissive policy subsumed by another is redundant
P6.1	Selector monotonicity	Adding tables preserves existing selector matches
P7.1	Bounded compilation	Traversal depth $d \to$ at most d nested EXISTS
P7.2	No recursive traversal	Hierarchies require closure tables
T8.1	Tenant isolation sufficient condition	If every permissive clause has the tenant atom, isolation holds
P9.1	Termination of normalization	Normalization terminates (strict reduction under lex ordering)
P9.2	Correctness of normalization	Each rewrite rule preserves denotation
T10.1	Compilation correctness	Accessible rows under SQL = rows satisfying effective(T, CMD)
P10.1	Monotonicity of compilation	Subsumption in DSL preserved in compiled SQL
P10.2	Determinism of compilation	Same normal form $\to$ identical SQL output
P12.1	Convergence	One governance cycle brings drift to zero
P12.2	Idempotence of application	Applying same artifacts twice = same state

```
                    | <null_literal>
                    | <list_literal>

<list_literal>  ::= "[" <literal_value>
                     ("," <literal_value>)* "]"

<string_literal> ::= "'" <character>* "'"
<integer_literal> ::= ["-"] <digit>+
<boolean_literal> ::= "true" | "false"
<null_literal>  ::= "null"

<identifier>    ::= <letter>
                     (<letter> | <digit> | "_")*

<pattern>       ::= <string_literal>
<tag>           ::= <string_literal>

<type>          ::= "text" | "integer" | "bigint"
                    | "uuid" | "boolean" | "timestamp"
                    | "jsonb"
```

14 Summary of Properties & Lemmas

Table 3 lists all theorems, properties, and lemmas established in this specification.

A Full Lifecycle Worked Example

This appendix traces the complete governance lifecycle for our running example schema, exercising every definition in the specification.

A.1 Define

We define three policies:

```
POLICY tenant_isolation
  PERMISSIVE
  FOR SELECT, INSERT, UPDATE, DELETE
  SELECTOR has_column('tenant_id')
  CLAUSE col('tenant_id') = session('app.tenant_id')

POLICY tenant_isolation_via_project
  PERMISSIVE
  FOR SELECT, INSERT, UPDATE, DELETE
  SELECTOR named('tasks') OR named('files')
  CLAUSE
    exists(
      rel(_, project_id, projects, id),
      {col('tenant_id') = session('app.tenant_id')}
    )

POLICY soft_delete
  RESTRICTIVE
  FOR SELECT
  SELECTOR has_column('is_deleted')
  CLAUSE col('is_deleted') = lit(false)
```

A.2 Selector Evaluation

Evaluate selectors against the running example metadata:

Selector	Matching tables
`has_column('tenant_id')`	users, projects, comments
`named('tasks') OR named('files')`	tasks, files
`has_column('is_deleted')`	projects

Policy-to-table mapping:

Table	Policies applied
users	tenant_isolation
projects	tenant_isolation, soft_delete
tasks	tenant_isolation_via_project
comments	tenant_isolation
files	tenant_isolation_via_project
config	*(none — default deny)*

A.3 Normalize

All three policies are already in normal form:

- Each clause has one atom, in atom normal form.
- No unsatisfiable clauses.
- No subsumption between clauses within the same policy.

A.4 Analyze

A.4.1 Satisfiability.

- `tenant_isolation` clause: `col('tenant_id') = session('app.tenant_id')` — satisfiable (session variable can equal any `tenant_id` value).
- `tenant_isolation_via_project` clause: `exists(rel(...), ...)` — satisfiable (there can exist a matching project row).
- `soft_delete` clause: `col('is_deleted') = lit(false)` — satisfiable.

All clauses pass satisfiability.

A.4.2 Contradiction Check. For `projects` (SELECT):

effective(`projects`, SELECT) = (`col('tenant_id') = session('app.tenant_id')`)∧(`col('is_deleted')`

SMT encoding: $tid\_var = session\_tid \wedge is\_deleted\_var = false$. Satisfiable. No contradiction.

A.4.3 Tenant Isolation Proof.

- For `users`: The sole permissive clause contains `col('tenant_id') = session('app.tenant_id')`. By Theorem 8.1, isolation holds.
- For `projects`: Same as `users`.
- For `tasks`: The permissive clause uses traversal to check `tenant_id` on `projects`. By the extended form of Theorem 8.1 (depth-1 traversal), isolation holds.
- For `comments`: Same as `users`.
- For `files`: Same reasoning as `tasks`.
- For `config`: No permissive policies → default deny → no access at all → isolation trivially holds.

Result: Tenant isolation proven for all tables.

A.5 Compile

Generated SQL for each governed table:

```
-- ====================================
-- users
-- ====================================
ALTER TABLE public.users
  ENABLE ROW LEVEL SECURITY;
ALTER TABLE public.users
  FORCE ROW LEVEL SECURITY;

CREATE POLICY tenant_isolation_users
  ON public.users
  AS PERMISSIVE
  FOR SELECT, INSERT, UPDATE, DELETE
  USING (tenant_id =
    current_setting('app.tenant_id'));
```

```
-- ====================================
-- projects
-- ====================================
ALTER TABLE public.projects
  ENABLE ROW LEVEL SECURITY;
ALTER TABLE public.projects
  FORCE ROW LEVEL SECURITY;

CREATE POLICY tenant_isolation_projects
  ON public.projects
  AS PERMISSIVE
  FOR SELECT, INSERT, UPDATE, DELETE
  USING (tenant_id =
    current_setting('app.tenant_id'));

CREATE POLICY soft_delete_projects
  ON public.projects
  AS RESTRICTIVE
  FOR SELECT
  USING (is_deleted = false);

-- ====================================
-- tasks
-- ====================================
ALTER TABLE public.tasks
  ENABLE ROW LEVEL SECURITY;
ALTER TABLE public.tasks
  FORCE ROW LEVEL SECURITY;

CREATE POLICY tenant_isolation_via_project_tasks
  ON public.tasks
  AS PERMISSIVE
  FOR SELECT, INSERT, UPDATE, DELETE
  USING (EXISTS (
    SELECT 1 FROM public.projects
    WHERE public.projects.id
          = public.tasks.project_id
      AND public.projects.tenant_id
          = current_setting('app.tenant_id')
  ));

-- ====================================
-- comments
-- ====================================
ALTER TABLE public.comments
  ENABLE ROW LEVEL SECURITY;
ALTER TABLE public.comments
  FORCE ROW LEVEL SECURITY;

CREATE POLICY tenant_isolation_comments
  ON public.comments
  AS PERMISSIVE
  FOR SELECT, INSERT, UPDATE, DELETE
  USING (tenant_id =
    current_setting('app.tenant_id'));

-- ====================================
-- files
-- ====================================
ALTER TABLE public.files
  ENABLE ROW LEVEL SECURITY;
ALTER TABLE public.files
  FORCE ROW LEVEL SECURITY;

CREATE POLICY tenant_isolation_via_project_files
  ON public.files
  AS PERMISSIVE
  FOR SELECT, INSERT, UPDATE, DELETE
  USING (EXISTS (
    SELECT 1 FROM public.projects
    WHERE public.projects.id
          = public.files.project_id
      AND public.projects.tenant_id
          = current_setting('app.tenant_id')
  ));
```

A.6 Apply

Execute the compiled SQL in a transaction against the target PostgreSQL database. All statements succeed.

A.7 Simulate Drift

A DBA manually runs:

```
ALTER TABLE public.projects
  DISABLE ROW LEVEL SECURITY;

CREATE POLICY manual_override ON public.users
  AS PERMISSIVE FOR SELECT
  USING (true);
```

This introduces two drift items:

(1) RLS disabled on `projects`
(2) Extra (unmanaged) policy on `users`

A.8 Detect

The drift detection algorithm (Section 11) runs:

```
detect_drift(S, {users, projects, tasks,
                 comments, files}) ->
{
    DriftItem(projects, "rls_disabled"),
    DriftItem(users, "extra_policy",
              "manual_override")
```

```
  }
```

A.9 Reconcile

- `projects` / `rls_disabled` → **auto-remediate**: re-enable RLS.
- `users` / `extra_policy` → **alert**: notify operators about unmanaged policy `manual_override`.

Remediation SQL:

```
ALTER TABLE public.projects
  ENABLE ROW LEVEL SECURITY;
```

After remediation, the next monitoring cycle detects zero drift (assuming the `extra_policy` alert has been acknowledged or the manual policy has been reviewed and either adopted into the policy set or dropped).

B Glossary

Atom An irreducible boolean comparison: (*left_source*, *operator*, *right_source*). The smallest unit of the policy algebra.

Clause A conjunction (AND) of atoms. Represents a single access condition that must be fully satisfied.

Compilation The deterministic translation of a policy set to PostgreSQL SQL artifacts.

Default deny The principle that if no permissive policy grants access, no rows are accessible.

Denotation The semantic interpretation $[\![\cdot]\!]$ of a policy expression: the set of rows it matches.

Drift Any discrepancy between the observed database state and the expected state derived from the policy algebra.

Effective access predicate The combined predicate ($\bigvee$ permissive) $\wedge$ ($\bigwedge$ restrictive) that determines row accessibility.

FCA Formal Concept Analysis. A mathematical framework for deriving concept hierarchies from object-attribute relations.

Galois connection A pair of monotone functions between ordered sets satisfying an adjunction property. Used to relate DSL and SQL semantics.

Governance loop The six-phase cycle: Define → Analyze → Compile → Apply → Monitor → Reconcile.

Normalization The process of applying rewrite rules to reduce a policy to its canonical form.

Permissive policy A policy whose clauses are OR'd together with other permissive policies. Grants access.

Policy A named, typed collection of clauses with a selector and command set.

Policy set The complete collection of policies governing a database.

Reconciliation The process of resolving drift between observed and expected database state.

Relationship A declared foreign-key link between tables, used for traversal atoms.

Restrictive policy A policy whose clauses are AND'd with the permissive disjunction. Narrows access.

RLS Row-Level Security. PostgreSQL's mechanism for attaching row-filtering predicates to tables.

Selector A predicate over table metadata that determines which tables a policy applies to.

SMT Satisfiability Modulo Theories. A decision procedure for logical formulas over combined theories.

Subsumption Relation where one policy/clause is at least as permissive as another.

Traversal atom An atom that uses `exists`(*relationship*, *clause*) to follow a foreign-key relationship.

Value source A typed scalar producer: column reference, session variable, literal, or function call.

References

[1] Clark Barrett and Cesare Tinelli. 2024. SMT-LIB: The Satisfiability Modulo Theories Library. https://smtlib.cs.uiowa.edu/.

[2] Piero A. Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. 2002. An Algebra for Composing Access Control Policies. *ACM Transactions on Information and System Security* 5, 1 (2002), 1–35.

[3] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '77)*. ACM, 238–252.

[4] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of TACAS 2008 (LNCS, Vol. 4963)*. Springer, 337–340.

[5] Bernhard Ganter and Rudolf Wille. 1999. *Formal Concept Analysis: Mathematical Foundations*. Springer.

[6] Ruoming Pang, Ramon Lanber, Ricardo Lopes, Vinicius Mussatti, Tarun Nandi, Tejo Narayan, Jateen Padhye, Flavio Pereira, Sergey Petrov, Benjamin Rae, Christopher Simpson, and Yonggang Zhao. 2019. Zanzibar: Google's Consistent, Global Authorization System. In *Proceedings of the 2019 USENIX Annual Technical Conference*. USENIX Association.

[7] PostgreSQL Global Development Group. 2024. Row Security Policies. https://www.postgresql.org/docs/current/ddl-rowsecurity.html. PostgreSQL Documentation.

[8] PostgreSQL Global Development Group. 2024. System Catalogs — pg_policies. https://www.postgresql.org/docs/current/view-pg-policies.html. PostgreSQL Documentation.