

Row-Level Security sucks. Can we make it usable?

PGDay UK 2025

Jimmy Angelakos

About me

- Systems & Database Architect, based in Edinburgh
- Open Source user & contributor (25+ years)
- PostgreSQL exclusively (17+ years), Contributor
- Member, PostgreSQL Europe Diversity Committee
- Author, PostgreSQL Mistakes and How to Avoid Them
- Co-author, PostgreSQL 16 Administration Cookbook
- **pg_statviz** PostgreSQL extension



Contents

- What is RLS?
- When to use it
- How it works
- How to use it
- What's wrong with it
- What to do about it
- More things to try

Motivation, etc.

- Customer wanted application users to not see each other's data
- Duh? But:
- Customer was used to application being badly coded
- REST URLs like `/user/1234/data`

What is Row-Level Security (RLS)?

- Fine-grained control over which rows are visible to which users
- Provides additional security beyond table or column level privileges
- It's a type of Access Control List (ACL)
- Saves you application-side security filtering

When would you use RLS?

- Confidential data
 - Restrict access to sensitive records
- Role / department separation
 - e.g. only HR sees HR-related content
- Multi-tenant systems
 - Separate data for each customer/tenant in the same DB
- Finer-grained visibility control (row vs table)

How does RLS work?

(i)

- From user perspective, rows they're not allowed to see "don't exist"
- Key concepts:
 - Policy
 - Conditions for reading/modifying rows
 - Security barrier
 - Query optimizer doesn't inline/restructure query to bypass RLS

How does RLS work?

(ii)

- It's exactly an ACL
- Internally, you are effectively adding WHERE conditions to the query
- Permissive / Restrictive policies
 - Permissive: `policy_A OR policy_B` (default)
 - Restrictive: `policy_C AND policy_D`

How does RLS work?

(iii)

- `pg_catalog.pg_policy`
 - `polrelid`: The table to which the policy applies
 - `polcmd`: The command for which the policy is:
SELECT, INSERT, UPDATE, DELETE, all
 - `polpermissive`: Policy permissive (`true`) or restrictive
 - `polroles`: Array of roles that the policy applies to
 - `polqual`: **USING** clause
 - `polwithcheck`: **WITH CHECK** clause

How do I use RLS?

(i)

```
ALTER TABLE customers  
ENABLE ROW LEVEL SECURITY;
```

- Remember: deny by default

How do I use RLS?

(ii)

```
CREATE POLICY custpolicy  
ON customer  
FOR ALL  
TO public  
USING customer_user = CURRENT_USER;
```

How do I use RLS?

(iii)

```
SELECT * FROM customer;
```



```
SELECT * FROM customer  
(WHERE customer_user = CURRENT_USER);
```

Okay, but what about your clickbait title?

- It does suck
 - And RLS sucks too
 - Why?

What's wrong with how RLS works? (i)

- It assumes that your application works a certain way
- People generally don't have data separated **by database user** that accesses it
- You don't want Postgres to manage your application users
 - Roles system has global scope
 - Can't store user attributes/preferences

What's wrong with how RLS works? (ii)

- Your application connects to DB using a single user
 - Makes auditing difficult
 - Changing this would require a significant rewrite
- Aligning application users and DB roles is tedious
 - Spaghetti of **GRANTS**
 - You have to keep them in sync too

What can we do?

Row-Level Security sucks. Can we make it usable?

A possible solution

(i)

- **SET** variables and use those in the **POLICY**

```
CREATE POLICY transpolicy
```

```
ON transaction FOR ALL TO public
```

```
USING
```

```
    (tenant = current_setting('app.tenant'));
```

```
SET app.tenant = 'Megacorp';
```

A possible solution

(ii)

- Feeling paranoid?

```
CREATE POLICY transpolicy  
ON transaction FOR ALL TO public  
USING (tenant =  
        current_setting('app.tenant')::uuid);  
SET app.tenant =  
    '465f2480-bbca-4eb0-8dd5-c6310b724e37';
```

A possible solution

(iii)

- Depending on whether you use connection pooling:

```
SET LOCAL app.tenant =  
    '465f2480-bbca-4eb0-8dd5-c6310b724e37';
```

Want to take this a step further?



ACL + RBAC

- Explicit Access Control List
and Role-Based Access Control
- Add an ACL column to the table:

ALTER TABLE transaction

ADD acl uuid[] NOT NULL DEFAULT '{}'::uuid[]

- **ARRAY** of **uuid** (if we use UUIDs for role identifiers)

(i)

ACL + RBAC

(ii)

- **SET** the **roles** that are granted access in the ACL

```
SET app.tenant_roles =  
    '{dda71d2d-67d8-4f00-b877-41ab442e62ea,  
     039746dc-48a1-4e2a-b765-968f689ac84f}';
```

ACL + RBAC

(iii)

- What does the RLS policy look like?

```
CREATE POLICY transrolepolicy  
ON transaction FOR ALL TO public  
USING (acl &&  
current_setting('app.tenant_roles')::uuid[]  
= true);
```

```
ALTER TABLE transaction  
ENABLE ROW LEVEL SECURITY;
```

ACL + RBAC

(iv)

- The policy checks if any of the tenant roles are inside the ACL
- RBAC
 - Roles can have attributes that define their privileges
 - Like Postgres roles, can be thought of as “groups” (of one or more tenants)
 - Can be granted to other roles, and then you have an aggregate of the privileges
 - Yes, we parallel the PostgreSQL roles system 😂😭

Want to dive even deeper?



How would you protect from application?

- After all, the application can connect to the DB and change roles and policies
- You hide direct access to this system from the application
- Why?
 - You don't trust your / third party application
- Let's assume Django app

Database-side

(i)

```
CREATE ROLE django;  
CREATE TABLE transaction (  
  id uuid PRIMARY KEY DEFAULT  
    gen_random_uuid(),  
  amount numeric,  
  created_at timestampz DEFAULT CURRENT_TIME,  
  acl uuid[] NOT NULL DEFAULT '{}'::uuid[]  
);
```

Database-side

(ii)

- To speed up ACL enforcement, we need an index that supports **ARRAY** operations on it:

```
CREATE INDEX ON transaction  
USING GIN (acl array_ops);
```

Database-side

(iii)

```
CREATE TABLE tenant_role (  
    role_id uuid PRIMARY KEY,  
    role_name text NOT NULL UNIQUE,  
    role_description text  
);  
CREATE TABLE tenant_role_member (  
    tenant_id uuid REFERENCES tenant(tenant_id),  
    role_id uuid REFERENCES tenant_role(role_id);  
);  
CREATE INDEX ON tenant_role_member(tenant_id);
```

Database-side

(iv)

- Remove the ability of DB user django to see inner workings:
REVOKE ALL ON tenant_role FROM django;
REVOKE ALL ON tenant_role_member FROM django;
REVOKE SELECT ON transaction FROM django;
GRANT SELECT (id, amount, created_at)
ON transaction TO django;

Database-side

(v)

- Create the policy:

```
CREATE POLICY trans_rls ON transaction  
USING (acl &&  
current_setting('app.tenant_roles')::uuid[ ]  
= true);
```

```
ALTER TABLE transaction  
ENABLE ROW LEVEL SECURITY;
```

Expose RBAC to Django through functions

- `create_tenant_role(_role_name text,
_role_description text) RETURNS uuid`
enables creation of tenant roles
- `get_tenant_roles(_tenant_id uuid) RETURNS uuid[]`
returns the roles that have been assigned to a tenant
- `set_tenant_roles(_tenant_id uuid, _roles uuid[])
RETURNS boolean`
sets all roles for a tenant

One more thing...

- For each table, we need an “add role to row acl” function and a “remove role from row acl function”
- These can be called e.g. by overriding Django’s `.save()`
- Important when using **SECURITY DEFINER**:
SET `search_path = public, pg_temp;`
(at the bottom of each function definition)

"Add role to row ACL" function

```
CREATE OR REPLACE FUNCTION order_acl_add(_id uuid, _role_id uuid)
RETURNS boolean AS $$
DECLARE _current_roles uuid[]; r uuid;
BEGIN
UPDATE order SET acl = (
    CASE WHEN _role_id = ANY(acl) THEN acl
    ELSE array_append(acl, _role_id) END)
WHERE id = _id;
IF FOUND THEN RETURN true;
ELSE RETURN false;
END IF;
END $$
LANGUAGE plpgsql
SECURITY DEFINER
SET search_path = public, pg_temp; -- IMPORTANT with SECURITY DEFINER
```

Finally, a few potential RLS catches (i)

- Policies can add overhead to queries
 - Especially complex conditions
 - Keep policies simple and explicit
- Superuser can bypass all RLS checks
- Table owner can bypass RLS check if **FORCE ROW LEVEL SECURITY** is not specified

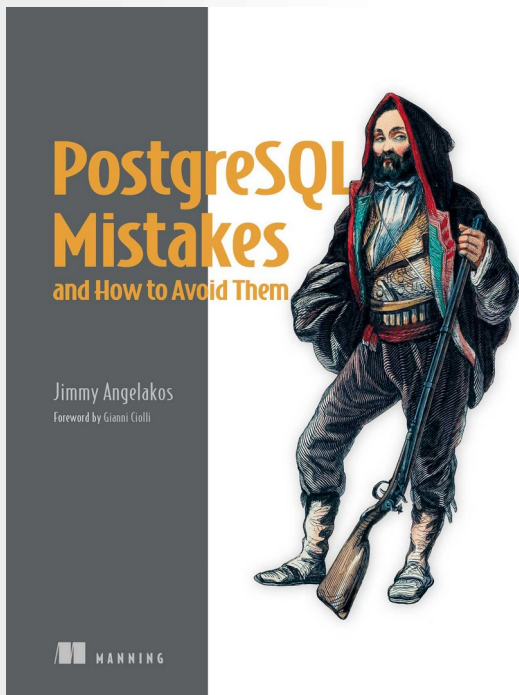
Finally, a few potential RLS catches (ii)

- Set a restrictive **DELETE** policy
 - So that people can't delete rows they can read but not update
- Make sure you reset variables between sessions
 - PgBouncer statement mode won't work with **SET/SET LOCAL**
- Ensure **WITH (SECURITY BARRIER)** is in place for views
 - To stop malicious function overrides with cost 0.00000000000001 etc.

Find me on socials

- YouTube: <https://youtube.com/JimmyAngelakos>
- Mastodon: <https://fosstodon.org/@vyruss>
- BlueSky: <https://bsky.app/profile/vyruss.org>
- LinkedIn: <https://linkedin.com/in/vyruss>

45% off everything!
Code: **pgd25ja**



Questions?