PostgreSQL Row Level Security (RLS) Infosheet

Row Level Security (RLS) was introduced in PostgreSQL v9.5 (2015), finally giving the database a much more flexible and granular security model suitable for supporting any number of users. With RLS, row access is determined by policies containing SQL expression, these policies run against each database row and define if it can be seen and/or written.

Permissive policies - only one policy must pass

Permissive RLS policies are effectively combined with boolean `OR`, meaning only one needs to pass for a row to be operated on. If there is no policy covering a table/operation then no rows can be operated on. (Restrictive policies do the same, but all must pass for a row to be operated on.)

Enabling RLS on a table

Once RLS is enabled on a table, only superusers and the table owner may operate on rows within that table until a policy grants access.

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

One role, millions of users

With RLS, one additional database role (e.g. our unprivileged "`graphql_role`") can represent any number of users. We still need to grant this role the ability to interact with the table:

```
GRANT
  SELECT,
  INSERT (name, public),
  UPDATE (name, public),
  DELETE
ON albums
TO graphql_role;
```

(Note this role still cannot view or write rows until policies are in place.)

You can use **"transaction variables"** (local settings, cleared when the transaction exits) to indicate the current user. It's advisable to use a helper function ("`viewer_id()`") to avoid repetition:

```
CREATE FUNCTION viewer_id() RETURNS int AS $$
  SELECT nullif(
    current_setting('my_app.user_id', TRUE),
    ''
  )::int;
$$ LANGUAGE sql STABLE;
```

If there's a risk of someone gaining access to your DB, it's advisable to use a session identifier rather than the user ID to identify the user, this prevents the attacker from impersonating another user.

Creating a policy: syntax

The less commonly used parts are in green text.

```
CREATE POLICY name ON table_name
  [ AS { PERMISSIVE | RESTRICTIVE } ]
  [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE }
  [ TO { role_name | PUBLIC | CURRENT_USER
|SESSION_USER } [, ...] ]
  [ USING ( using_expression ) ]
  [ WITH CHECK ( check_expression ) ]
```

AS determines how policies combine; all `RESTRICTIVE` policies and at least one `PERMISSIVE` policy must pass for a row to be accessed; `PERMISSIVE` is assumed by default.

FOR specifies which operations the policy applies to.

TO specifies the database roles this policy applies to; by default it applies to `PUBLIC` (all roles).

USING acts as a hidden "`WHERE`" clause determining which rows can be "seen" by the operation; it applies to `SELECT`, `UPDATE` and `DELETE`.

WITH CHECK is similar to `USING`, but it is performed on the new/updated row before it is written to the database; it applies to `INSERT` and `UPDATE`. If `WITH CHECK` is omitted, the `USING` clause will be used in its place.

PostgreSQL Row Level Security (RLS) Infosheet Continued...

`USING` and `WITH CHECK` expressions

These SQL expressions run against each row to determine access, they may contain subqueries and function calls.

```
-- Everyone can see public albums:
CREATE policy select_public ON albums
FOR SELECT USING ( public IS TRUE );

-- You can always see your own albums:
CREATE policy select_own ON albums
FOR SELECT USING ( owner_id = viewer_id() );

-- You can see a photo if you can see
-- its album
CREATE POLICY select_where_album_visible
  ON photos FOR SELECT USING(
    EXISTS(
      SELECT 1
      FROM albums
      WHERE albums.id = photos.album_id
    )
  );
```

Beware stack exhaustion!

Subqueries in RLS policies respect the RLS policies of the tables they reference. Cyclic dependencies risk infinite recursion. To solve, use a function marked as `SECURITY DEFINER` to bypass RLS. (This technique can also be used to improve RLS policy **performance**.)

```
CREATE FUNCTION viewer_member_album_ids()
RETURNS SETOF int AS $$
  SELECT album_id
  FROM album_members
  WHERE user_id = viewer_id();
$$ LANGUAGE sql STABLE
SECURITY DEFINER; -- Bypass RLS

CREATE POLICY select_members ON albums
  FOR SELECT
  USING (
    id IN ( SELECT viewer_member_album_ids() )
);
```

Trying it out

You can see the effects of RLS in a simple transaction. Become the unprivileged role ("`graphql_role`"), set your transaction variables, and then perform your operations.

```
BEGIN;
SET LOCAL role TO graphql_role;
SET LOCAL my_app.user_id TO '3';
SELECT viewer_id();
SELECT * FROM photos;
COMMIT;
```

Notes...

PostGraphile