Row Level Security

Stephen Frost
stephen.frost@crunchydatasolutions.com

Crunchy Data Solutions, Inc.

September 19, 2014

Stephen Frost

- Chief Technology Officer @ Crunchy
- Committer
- Major Contributor
- Implemented the roles system in 8.3
- Column-level privileges in 8.4
- Contributions to PL/pgSQL, PostGIS

What is RLS?

Provide an individual view of the data in the system for each user of the system, while maintaining relational consistency and normalization with a single set of tables.

- Filtering records returned from all queries
- Limit records allowed to be added to a table
- Support filtering and limiting based on user
- Require policies to always be applied
- Default Deny approach

Security Barrier Views

- Build a view over tables to filter records
- Must use security_barrier option
- Permissions on view independent of table
- Access to tables through view uses permisisions of view owner

```
CREATE TABLE mytable (
    mydata  text,
    myuser  name
);

CREATE VIEW myview WITH (security_barrier) AS
    SELECT * FROM mytable
            WHERE myuser = current_user();
```

WITH CHECK Option

- "Simple" views are automatically updatable
- View could be used to add records not visible to user
- WITH CHECK prevents adding records which can't be seen
- CASCADED vs LOCAL

```
CREATE VIEW myview
  WITH (security_barrier, check_option) AS
    SELECT * FROM mytable
            WHERE myuser = current_user();
```

Or

```
CREATE VIEW myview WITH (security_barrier) AS
    SELECT * FROM mytable
            WHERE myuser = current_user
    WITH CHECK OPTION;
```

Multi-schema Approach

Create a schema and set of tables for each user, use GRANT system for managing access control.

- Difficult to maintain consistency
- Maintenance nightmare for updating table definitions
- Dynamic SQL may be required
- Views or inheiritance for bulk operations
- Partitioning must match authorization requirement

ALTER TABLE ... (ENABLE/DISABLE) ROW LEVEL SECURITY

ENABLE:

- Enables RLS for the table
- Requires all access to be through a policy
- Default-Deny policy
- Excludes superuser and table owner by default

DISABLE:

- Disables RLS for the table
- Policies are no longer used at all
- Does not remove policies on the table

SET ROW_SECURITY ... = (ON/OFF/FORCE)

- ON - Normal mode, policies applied as appropriate
- OFF - Never applies RLS, will throw an ERROR if necessary
- FORCE - RLS applied even for superusers and table owners

CREATE POLICY

- Policies are table-specific
- Applied for specific commands, or all commands
- Can be for specific roles or all roles
- "default-deny" policy
- USING and WITH CHECK clauses
- WITH CHECK defaults to USING if not specified

```
Command:     CREATE POLICY
Description: define a new row-security policy for a table
Syntax:
CREATE POLICY name ON table_name
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { role_name | PUBLIC } [, ...] ]
    [ USING ( expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

CREATE POLICY

Example policy creation:

```
CREATE POLICY p1 ON mytable
        USING (myuser = current_user);
```

CREATE POLICY

Example policy creation:

```
CREATE POLICY p1 ON mytable
        USING (myuser = current_user);
```

Same:

```
CREATE POLICY p1 ON mytable
        USING (myuser = current_user)
   WITH CHECK (myuser = current_user);
```

CREATE POLICY

Example policy creation:

```
CREATE POLICY p1 ON mytable
        USING (myuser = current_user);
```

Same:

```
CREATE POLICY p1 ON mytable FOR ALL
        USING (myuser = current_user)
   WITH CHECK (myuser = current_user);
```

CREATE POLICY

Example policy creation:

```
CREATE POLICY p1 ON mytable
        USING (myuser = current_user);
```

Same:

```
CREATE POLICY p1 ON mytable FOR ALL TO PUBLIC
       USING (myuser = current_user)
  WITH CHECK (myuser = current_user);
```

CREATE POLICY FOR SELECT

- Applies only to SELECT queries
- Only allows USING clause

```
CREATE POLICY p1 ON mytable
    FOR SELECT USING (myuser = current_user);
```

CREATE POLICY FOR INSERT

- Applies only to INSERT queries
- Only allows WITH CHECK clause

```
CREATE POLICY p1 ON mytable
    FOR INSERT WITH CHECK (myuser = current_user);
```

CREATE POLICY FOR UPDATE

- Applies only to UPDATE queries
- Alllows both USING and WITH CHECK clauses
- If WITH CHECK is omitted, USING clause will be used

Allows updating records to be hidden from current user:

```
CREATE POLICY p1 ON mytable FOR UPDATE
           USING (myuser = current_user)
      WITH CHECK (true);
```

CREATE POLICY FOR DELETE

- Applies only to DELETE queries
- Alllows only USING clause

```
CREATE POLICY p1 ON mytable
    FOR SELECT USING (myuser = current_user);
```

ALTER POLICY

- Allows changing policy definitions
- Set of roles the policies applies to can also be changed
- COMMAND for policy is not able to be changed- drop and recreate instead
- Policies can also be renamed

```
Command:     ALTER POLICY
Description: change the definition of a row-security polic
Syntax:
ALTER POLICY name ON table_name
    [ RENAME TO new_name ]
    [ TO { role_name | PUBLIC } [, ...] ]
    [ USING ( expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

ALTER POLICY

```
ALTER POLICY p1 ON mytable USING (myuser = 'r1');

ALTER POLICY p1 ON mytable WITH CHECK (myuser = 'r1');

ALTER POLICY p1 ON mytable TO r1, r2;

ALTER POLICY p1 ON mytable TO r1, r2
        USING (myuser = current_user)
        WITH CHECK (myuser = current_user);
```

DROP POLICY

- Policies can be dropped
- Remember- default deny policy if no policies found

```
Command:     DROP POLICY
Description: remove a row-security policy from a table
Syntax:
DROP POLICY [ IF EXISTS ] name ON table_name
```

```
DROP POLICY p1 ON mytable;
```

POLICY vs. GRANT

- GRANT system
  - Table-level SELECT, INSERT, UPDATE, DELETE
  - Column-level SELECT, INSERT, UPDATE
  - All-or-Nothing for rows
  - Views traditionally used to limit data
  - Everything-or-Error
- POLICY system
  - Table-level row selection
  - Conditional added to queries
  - ERROR for rows added which violate POLICY

Simple USING Policy

```
postgres=# \d t1
     Table "public.t1"
 Column | Type | Modifiers
--------+------+-----------
 myuser | name |
 mydata | text |
Policies:
    POLICY "p1" EXPRESSION (myuser = "current_user"())

postgres=# table t1;
 myuser | mydata
--------+---------
 r1     | r1 data
 r2     | r2 data
 r3     | r3 data
(3 rows)
```

Simple USING Policy

- Results filtered based on user

```
postgres=# set role r1;
SET
postgres=> table t1;
 myuser | mydata
--------+---------
 r1     | r1 data
(1 row)
```

- If no WITH CHECK clause specified, USING clause is used

```
postgres=> insert into t1 values ('r2','my data');
ERROR:  new row violates WITH CHECK OPTION for "t1"
DETAIL:  Failing row contains (r2, my data).
```

Data Modification Policies

- WITH CHECK used for adding records
- Allows creating records not visible to current user also

```
postgres=# alter policy p1 on t1
postgres-# using (myuser = current_user) with check (true)
ALTER POLICY
postgres=# set role r1;
SET
postgres=> insert into t1 values ('r2','my data');
INSERT 0 1
postgres=> table t1;
 myuser | mydata
--------+---------
 r1     | r1 data
(1 row)
```

External Policy Systems

- With expressions, any function can be used
- Could call out to extensions for, eg: SELinux

```
postgres=# alter policy p1 on t1
postgres-# using (check_selinux(mydata));
ALTER POLICY
```

Leak-Proof Functions

- Only leak-proof functions can be pushed down
- Allows them to be executed before other qualifications
- May be able to more efficiently use indexes
- Only superuser can mark functions leak-proof
- Leak-proof functions must actually be leak proof!

EXPLAIN Example 1

```
postgres=> explain table t1;
                         QUERY PLAN
-----------------------------------------------------
 Seq Scan on t1  (cost=0.00..19.60 rows=3 width=96)
   Filter: (myuser = "current_user"())
 Planning time: 0.341 ms
(3 rows)
```

EXPLAIN Example 2

```
postgres=> explain table t1;
                              QUERY PLAN
----------------------------------------------------------------------
 Index Scan using myuser_idx on t1  (cost=0.13..8.15 rows=1 width=96)
   Index Cond: (myuser = "current_user"())
 Planning time: 1.035 ms
(3 rows)
```

EXPLAIN Example 3

```
postgres=> explain select * from t1 where mydata ~ '^r2';
                                  QUERY PLAN
-----------------------------------------------------------------------------------
 Subquery Scan on t1  (cost=0.13..8.16 rows=1 width=96)
   Filter: (t1.mydata ~ '^r2'::text)
   ->  Index Scan using myuser_idx on t1 t1_1  (cost=0.13..8.15 rows=1 width=96
         Index Cond: (myuser = "current_user"())
 Planning time: 0.538 ms
(5 rows)
```

Thank You

- Questions?