Row Level Security

by Bennie Swart

Postgres Conference US 2019

Overview

Recap: PostgreSQL Roles
Recap: PostgreSQL Privileges

PostgreSQL Row Security Policies

Database Users vs Application Users
Application Users: Typical Access Control
Application Users: Better Access Control using RLS

Access Control: Moving Everything to the Database

Performance impacts

Comments and Questions

Recap: PostgreSQL Roles

Recap: PostgreSQL Roles

Users – roles that can login
Groups – roles that cannot login } Everything is a role! (since 8.1)

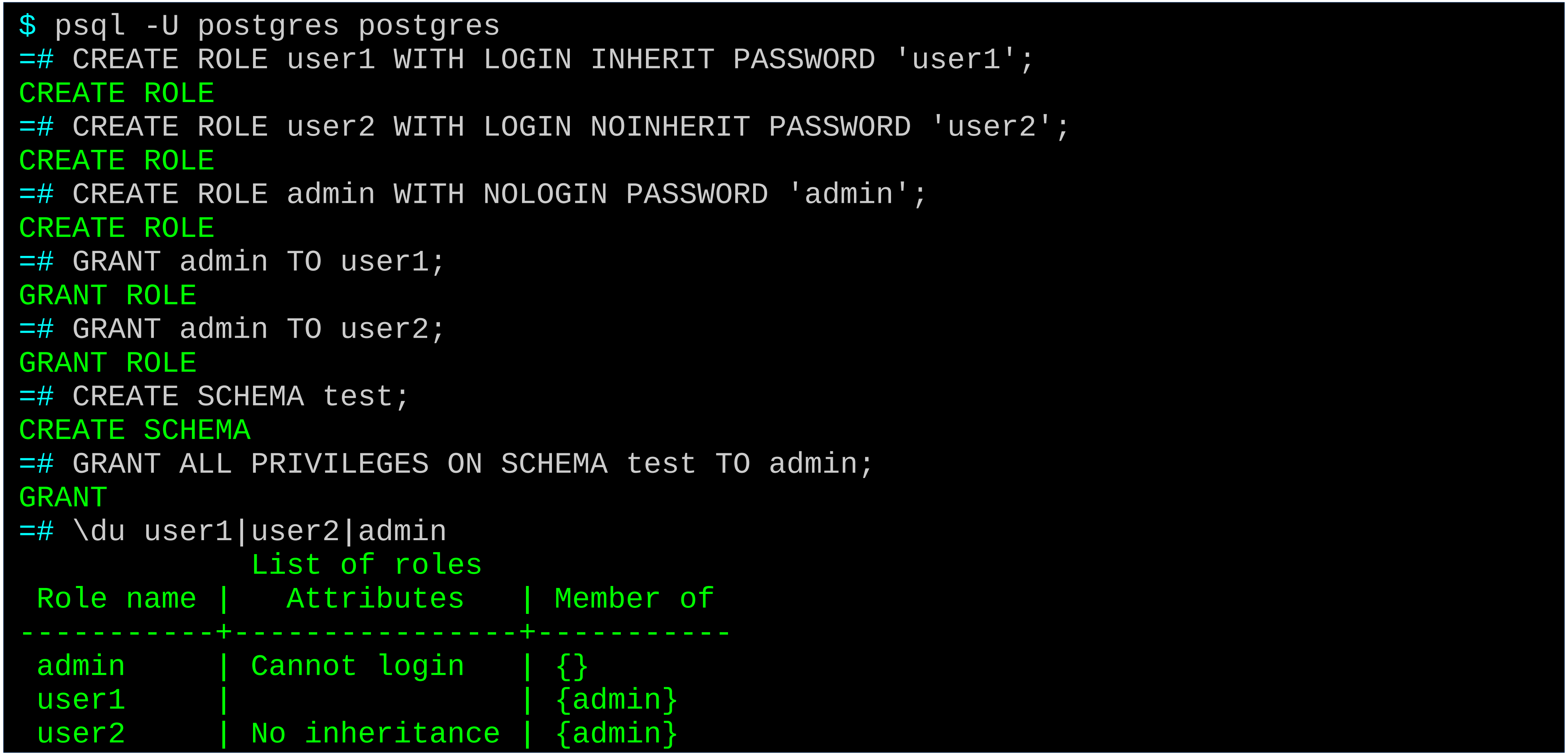
PostgreSQL roles are distinct from OS users

Roles can be members of other roles
- Allows to ‘become’ other roles, change privileges
- Or ‘inherit’ privileges from other roles automatically

One authenticated user per session/connection
session_user vs current_user

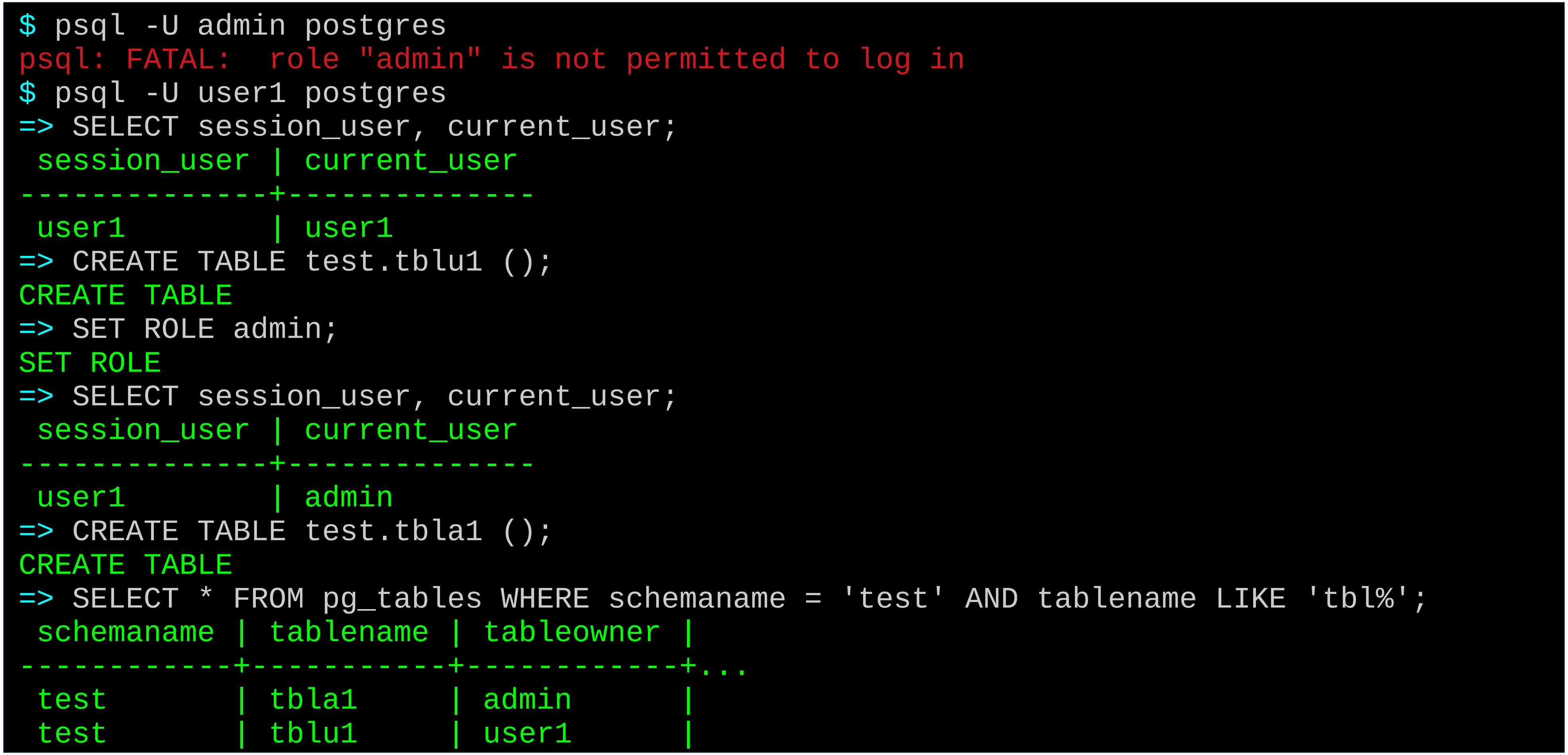
Implied PUBLIC role that all roles inherit from

Recap: PostgreSQL Roles



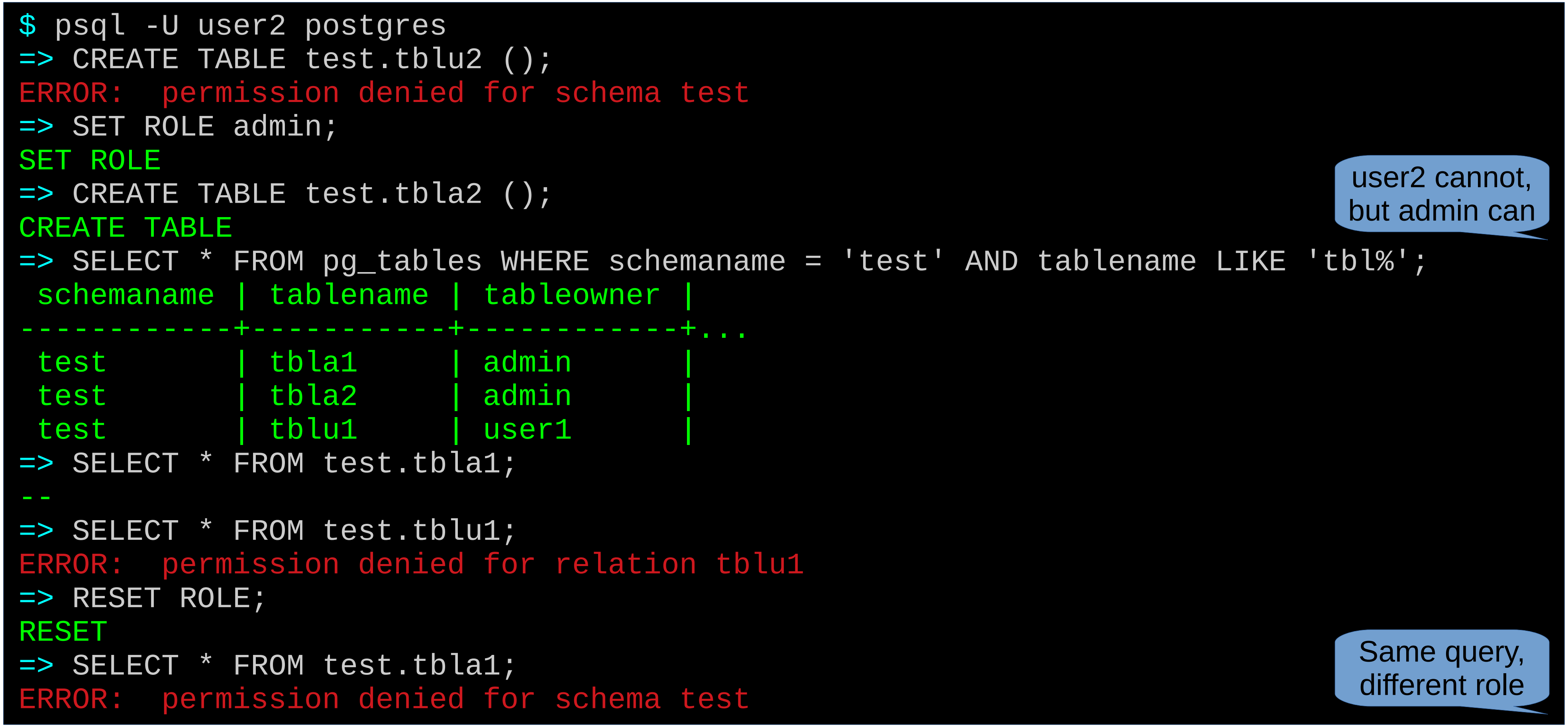
```
$ psql -U postgres postgres
=# CREATE ROLE user1 WITH LOGIN INHERIT PASSWORD 'user1';
CREATE ROLE
=# CREATE ROLE user2 WITH LOGIN NOINHERIT PASSWORD 'user2';
CREATE ROLE
=# CREATE ROLE admin WITH NOLOGIN PASSWORD 'admin';
CREATE ROLE
=# GRANT admin TO user1;
GRANT ROLE
=# GRANT admin TO user2;
GRANT ROLE
=# CREATE SCHEMA test;
CREATE SCHEMA
=# GRANT ALL PRIVILEGES ON SCHEMA test TO admin;
GRANT
=# \du user1|user2|admin
             List of roles
 Role name |   Attributes   | Member of
-----------+----------------+-----------
 admin     | Cannot login   | {}
 user1     |                | {admin}
 user2     | No inheritance | {admin}
```

Recap: PostgreSQL Roles



```
$ psql -U admin postgres
psql: FATAL:  role "admin" is not permitted to log in
$ psql -U user1 postgres
=> SELECT session_user, current_user;
 session_user | current_user
--------------+--------------
 user1        | user1
=> CREATE TABLE test.tblu1 ();
CREATE TABLE
=> SET ROLE admin;
SET ROLE
=> SELECT session_user, current_user;
 session_user | current_user
--------------+--------------
 user1        | admin
=> CREATE TABLE test.tbla1 ();
CREATE TABLE
=> SELECT * FROM pg_tables WHERE schemaname = 'test' AND tablename LIKE 'tbl%';
 schemaname | tablename | tableowner |
------------+-----------+------------+...
 test       | tbla1     | admin      |
 test       | tblu1     | user1      |
```

Recap: PostgreSQL Roles



```
$ psql -U user2 postgres
=> CREATE TABLE test.tblu2 ();
ERROR:  permission denied for schema test
=> SET ROLE admin;
SET ROLE
=> CREATE TABLE test.tbla2 ();
CREATE TABLE
=> SELECT * FROM pg_tables WHERE schemaname = 'test' AND tablename LIKE 'tbl%';
 schemaname | tablename | tableowner |
------------+-----------+------------+...
 test       | tbla1     | admin      |
 test       | tbla2     | admin      |
 test       | tblu1     | user1      |
=> SELECT * FROM test.tbla1;
--
=> SELECT * FROM test.tblu1;
ERROR:  permission denied for relation tblu1
=> RESET ROLE;
RESET
=> SELECT * FROM test.tbla1;
ERROR:  permission denied for schema test
```

user2 cannot, but admin can

Same query, different role

Recap: PostgreSQL Privileges

Recap: PostgreSQL Privileges

Privileges: required to perform commands (SELECT, INSERT, ...) on objects (TABLE, FUNCTION, ...)

When an object is created, it is assigned an owner. The owner is normally the role that executed the creation statement. For most kinds of objects, the initial state is that only the owner (or a superuser) can do anything with the object. To allow other roles to use it, privileges must be granted.

To grant/revoke privileges: GRANT/REVOKE

GRANTs are checked using OR (if any one satisfies the requirement, access is granted)

PostgreSQL grants default privileges on some types of objects to PUBLIC. No privileges are granted to PUBLIC by default on tables, table columns, sequences, [and more]. For other types of objects, the default privileges granted to PUBLIC are as follows: CONNECT and TEMPORARY (create temporary tables) privileges for databases; EXECUTE privilege for functions; and USAGE privilege for languages and data types (including domains)

Recap: PostgreSQL Privileges

Tables & Views: SELECT, INSERT, UPDATE, DELETE (and more)
Grantable on entire table, or per-column (S/I/U)

Functions: EXECUTE
SECURITY INVOKER vs SECURITY DEFINER

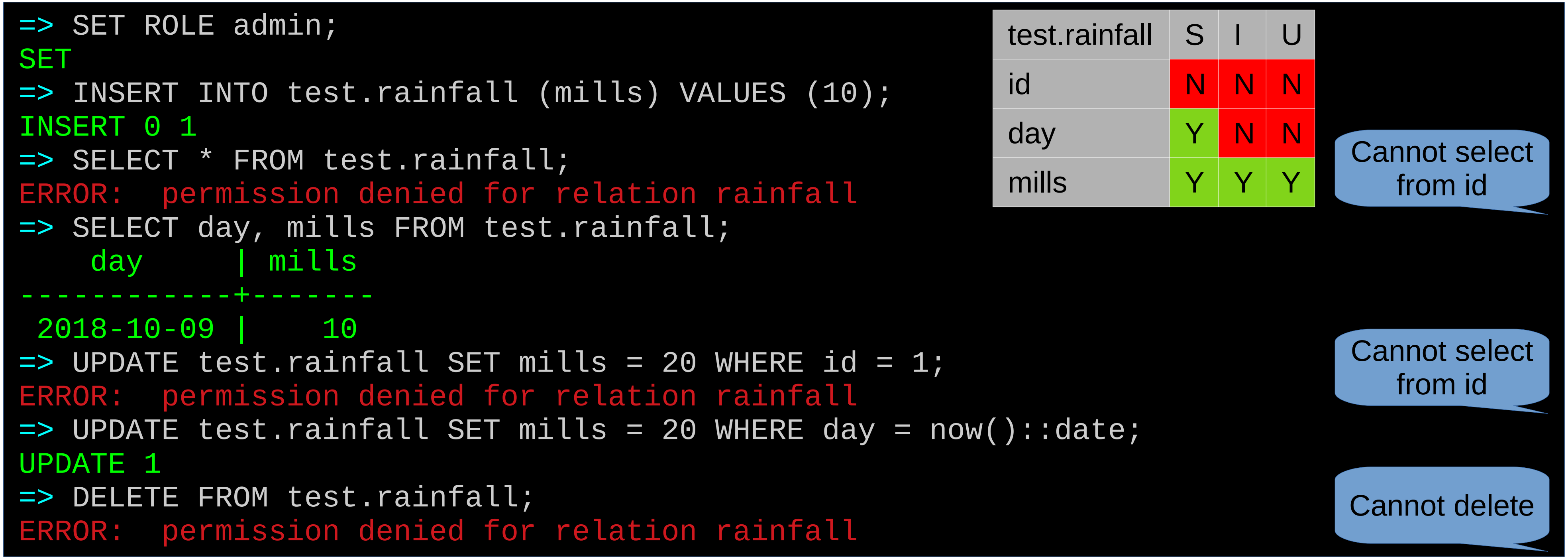
Recap: PostgreSQL Privileges

```
$ psql -U postgres postgres
=# SET ROLE user1;
SET
=> CREATE TABLE test.rainfall (
   id serial PRIMARY KEY, day date DEFAULT now(), mills int);
CREATE TABLE
=> GRANT SELECT (day, mills), INSERT (mills), UPDATE (mills)
   ON test.rainfall TO admin;
GRANT
=> SET ROLE admin;
SET
=> INSERT INTO test.rainfall (day, mills) VALUES (now(), 10);
ERROR:  permission denied for relation rainfall
=> INSERT INTO test.rainfall (mills) VALUES (10);
ERROR:  permission denied for sequence rainfall_id_seq
=> SET ROLE user1;
SET
=> GRANT USAGE ON SEQUENCE test.rainfall_id_seq TO admin;
GRANT
```

test.rainfall	S	I	U
id	N	N	N
day	Y	N	N
mills	Y	Y	Y

Cannot insert into day

Recap: PostgreSQL Privileges



```
=> SET ROLE admin;
SET
=> INSERT INTO test.rainfall (mills) VALUES (10);
INSERT 0 1
=> SELECT * FROM test.rainfall;
ERROR:  permission denied for relation rainfall
=> SELECT day, mills FROM test.rainfall;
    day     | mills
------------+-------
 2018-10-09 |    10
=> UPDATE test.rainfall SET mills = 20 WHERE id = 1;
ERROR:  permission denied for relation rainfall
=> UPDATE test.rainfall SET mills = 20 WHERE day = now()::date;
UPDATE 1
=> DELETE FROM test.rainfall;
ERROR:  permission denied for relation rainfall
```

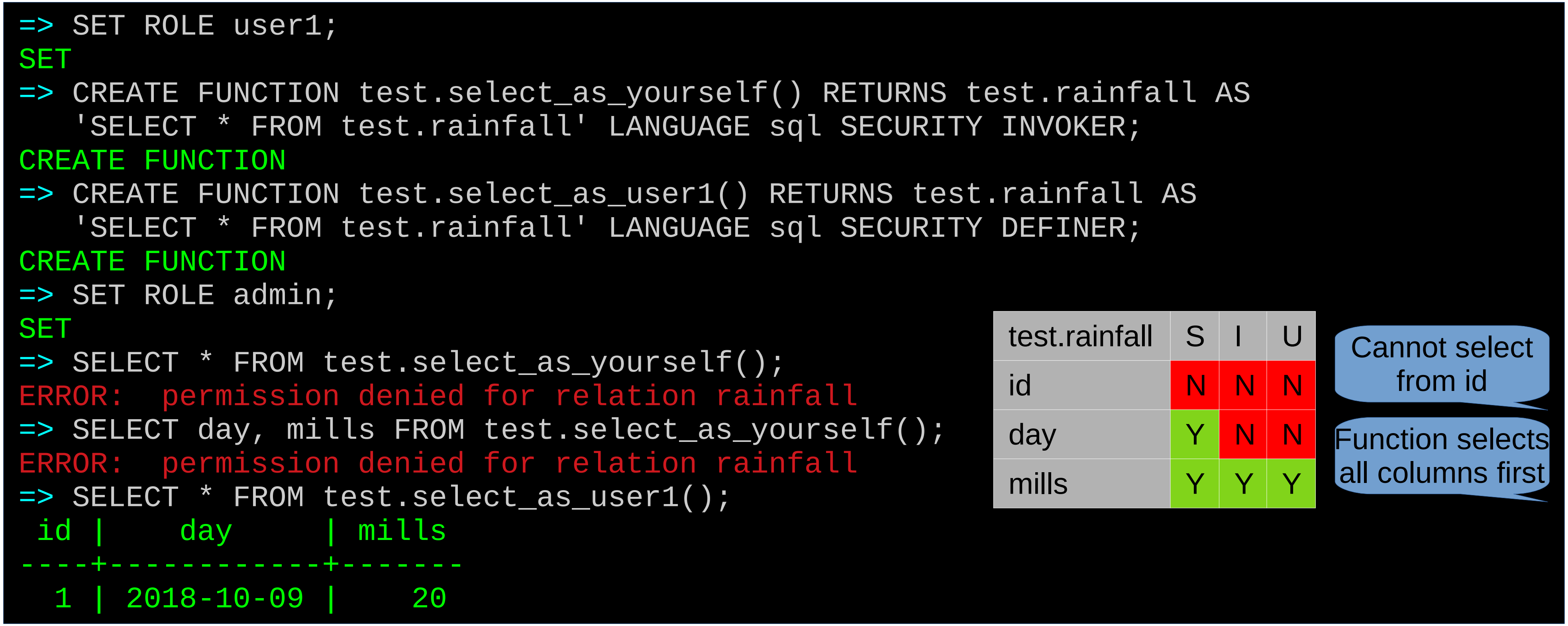
test.rainfall	S	I	U
id	N	N	N
day	Y	N	N
mills	Y	Y	Y

Cannot select from id

Cannot select from id

Cannot delete

Recap: PostgreSQL Privileges



```
=> SET ROLE user1;
SET
=> CREATE FUNCTION test.select_as_yourself() RETURNS test.rainfall AS
   'SELECT * FROM test.rainfall' LANGUAGE sql SECURITY INVOKER;
CREATE FUNCTION
=> CREATE FUNCTION test.select_as_user1() RETURNS test.rainfall AS
   'SELECT * FROM test.rainfall' LANGUAGE sql SECURITY DEFINER;
CREATE FUNCTION
=> SET ROLE admin;
SET
=> SELECT * FROM test.select_as_yourself();
ERROR:  permission denied for relation rainfall
=> SELECT day, mills FROM test.select_as_yourself();
ERROR:  permission denied for relation rainfall
=> SELECT * FROM test.select_as_user1();
 id |    day     | mills
----+------------+-------
  1 | 2018-10-09 |    20
```

test.rainfall	S	I	U
id	N	N	N
day	Y	N	N
mills	Y	Y	Y

Cannot select from id

Function selects all columns first

PostgreSQL Row Security Policies

PostgreSQL Row Security Policies

Row Level Security: control which **rows** can be read/modified
In contrast the SQL privilege system: control which **columns** can be read/modified

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

Policies created per table [per command] [per role]

USING: only operate on rows where **using_expression** is TRUE
when **using_expression** is FALSE, rows are 'hidden' (as if they don't exist)

WITH CHECK: used with INSERT and UPDATE to check new data
when **check_expression** is FALSE, error is thrown (invalid data)

PostgreSQL Row Security Policies

RLS must be explicitly enabled: ALTER TABLE … ENABLE ROW LEVEL SECURITY

Default deny: if RLS is enabled, one of the policies must pass for row to be visible

POLICYs for same command types for the same command are checked using OR

(if any one passes, row is visible)

POLICYs for different command types for the same command are checked using AND

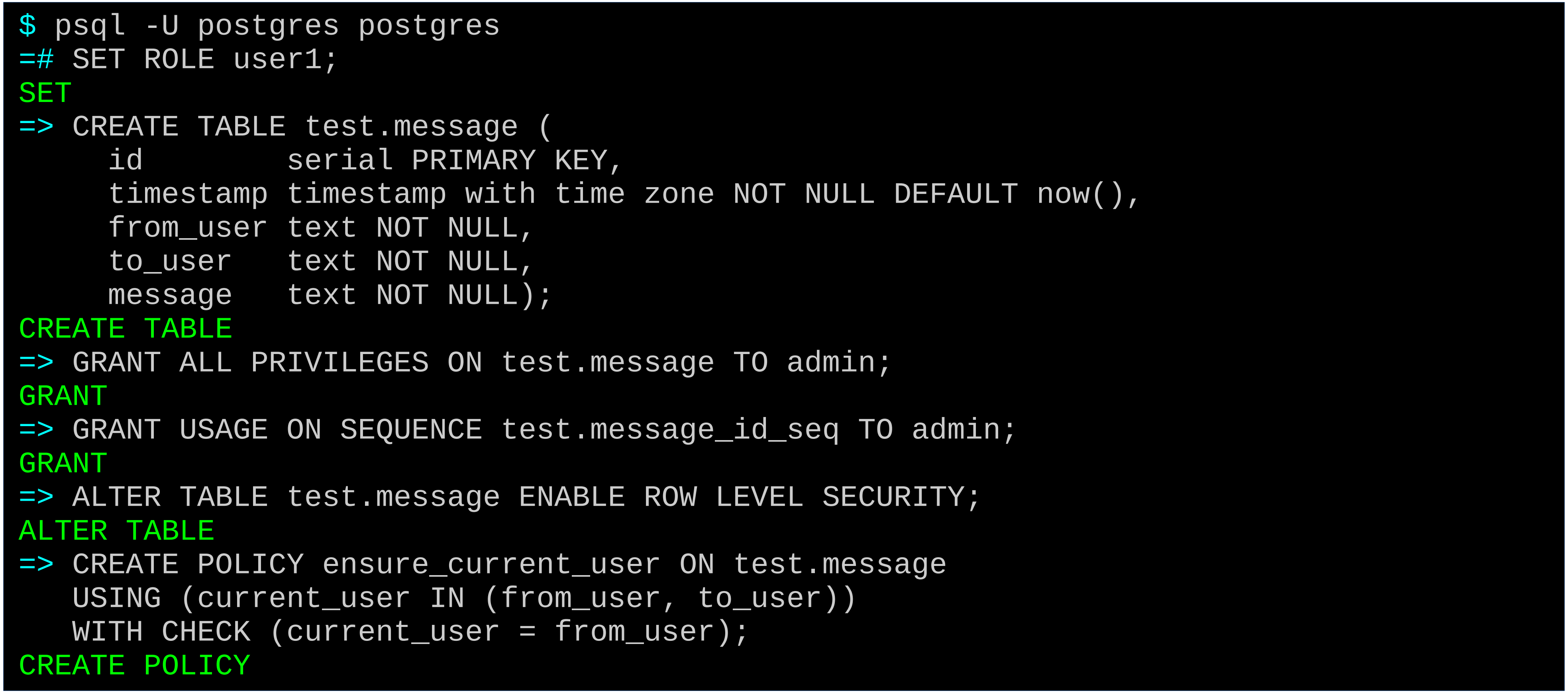
(e.g. UPDATE … WHERE requires UPDATE *and* SELECT permission)

Only table owners can create policies

Superusers and users with BYPASSRLS *always* bypass RLS

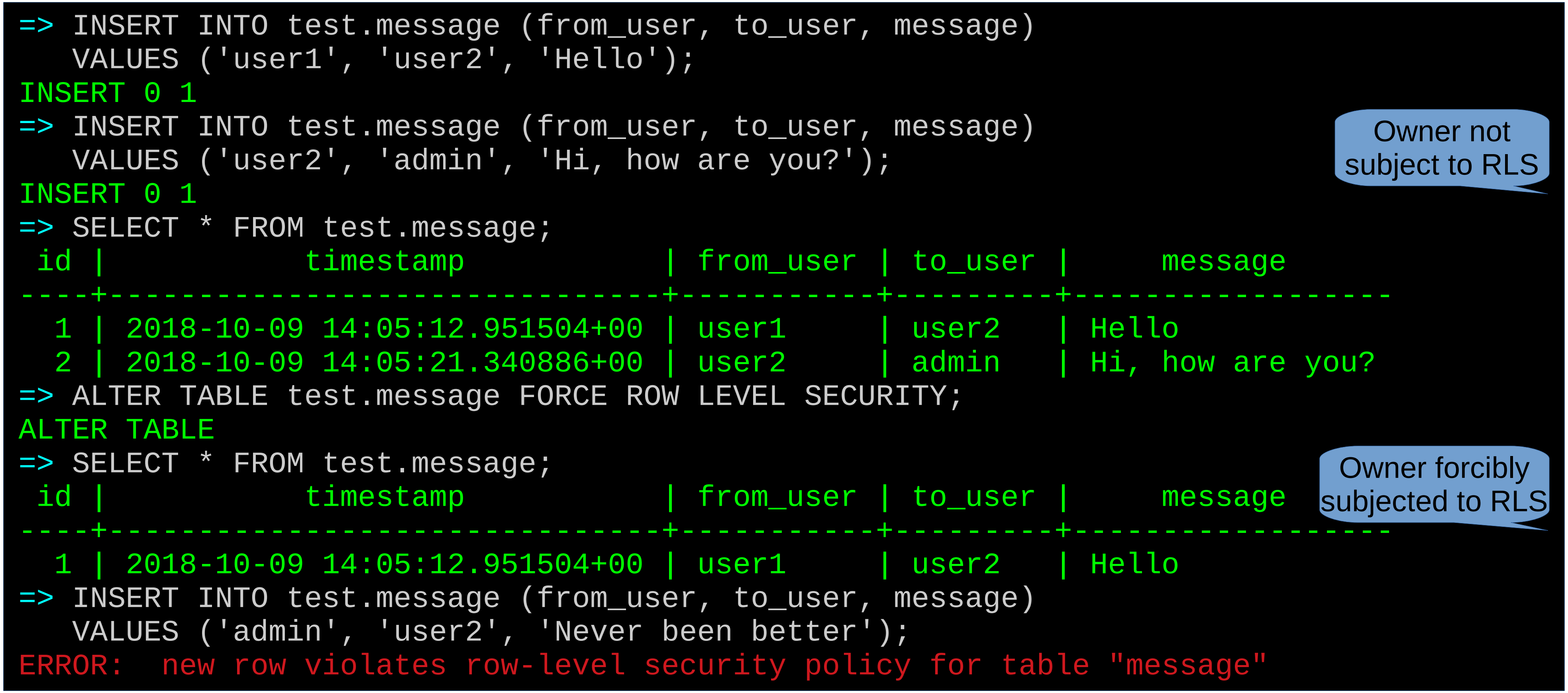
Table owners bypass RLS by default: ALTER TABLE … FORCE ROW LEVEL SECURITY

PostgreSQL Row Security Policies



```
$ psql -U postgres postgres
=# SET ROLE user1;
SET
=> CREATE TABLE test.message (
     id        serial PRIMARY KEY,
     timestamp timestamp with time zone NOT NULL DEFAULT now(),
     from_user text NOT NULL,
     to_user   text NOT NULL,
     message   text NOT NULL);
CREATE TABLE
=> GRANT ALL PRIVILEGES ON test.message TO admin;
GRANT
=> GRANT USAGE ON SEQUENCE test.message_id_seq TO admin;
GRANT
=> ALTER TABLE test.message ENABLE ROW LEVEL SECURITY;
ALTER TABLE
=> CREATE POLICY ensure_current_user ON test.message
   USING (current_user IN (from_user, to_user))
   WITH CHECK (current_user = from_user);
CREATE POLICY
```

PostgreSQL Row Security Policies

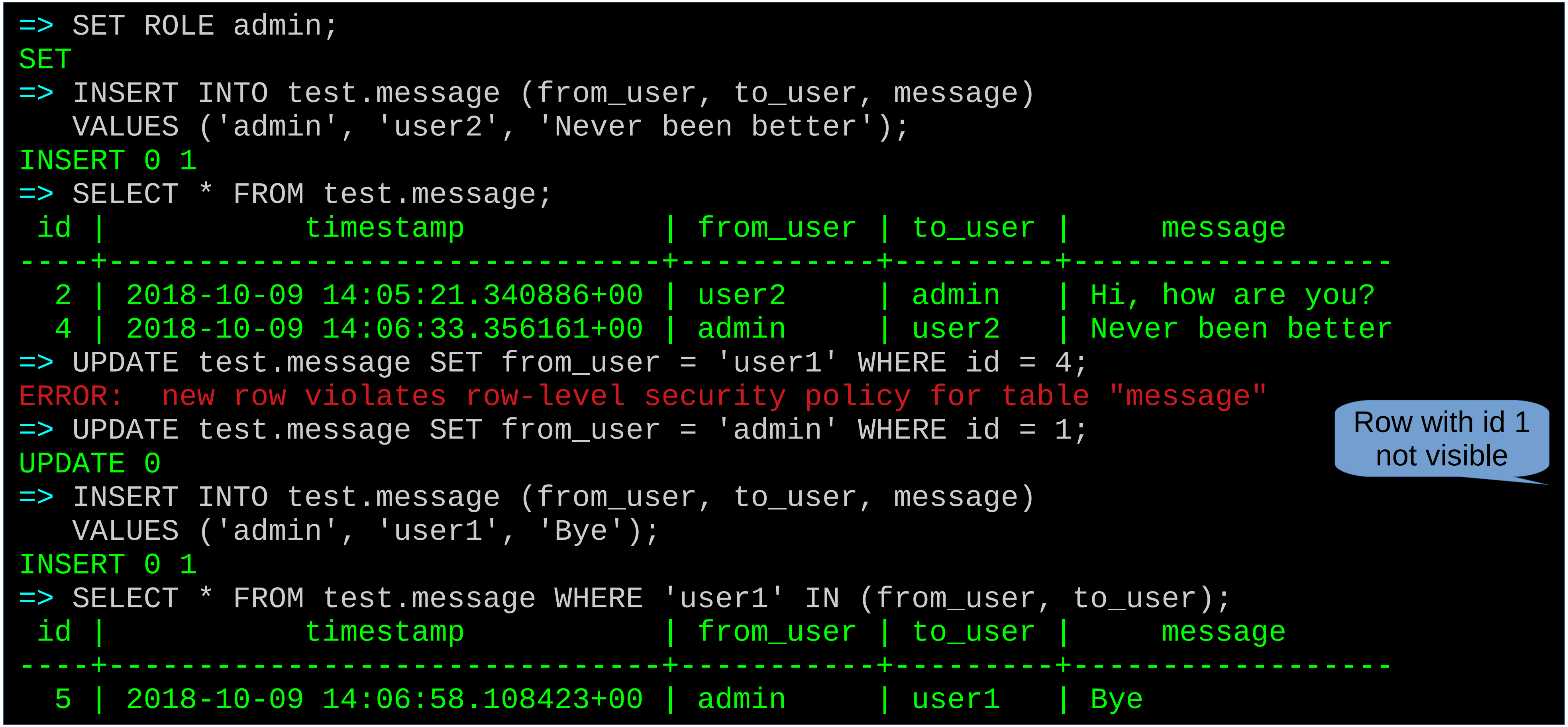


```
=> INSERT INTO test.message (from_user, to_user, message)
   VALUES ('user1', 'user2', 'Hello');
INSERT 0 1
=> INSERT INTO test.message (from_user, to_user, message)
   VALUES ('user2', 'admin', 'Hi, how are you?');
INSERT 0 1
=> SELECT * FROM test.message;
 id |           timestamp           | from_user | to_user |      message
----+-------------------------------+-----------+---------+------------------
  1 | 2018-10-09 14:05:12.951504+00 | user1     | user2   | Hello
  2 | 2018-10-09 14:05:21.340886+00 | user2     | admin   | Hi, how are you?
=> ALTER TABLE test.message FORCE ROW LEVEL SECURITY;
ALTER TABLE
=> SELECT * FROM test.message;
 id |           timestamp           | from_user | to_user |      message
----+-------------------------------+-----------+---------+------------------
  1 | 2018-10-09 14:05:12.951504+00 | user1     | user2   | Hello
=> INSERT INTO test.message (from_user, to_user, message)
   VALUES ('admin', 'user2', 'Never been better');
ERROR:  new row violates row-level security policy for table "message"
```

Owner not subject to RLS

Owner forcibly subjected to RLS

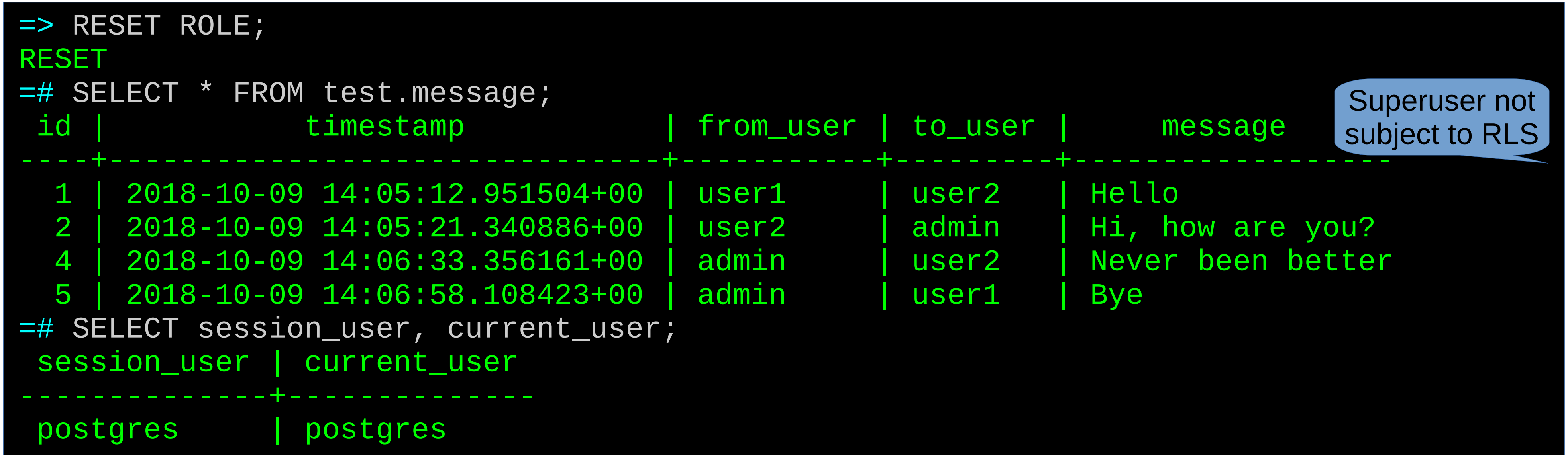
PostgreSQL Row Security Policies



```
=> SET ROLE admin;
SET
=> INSERT INTO test.message (from_user, to_user, message)
   VALUES ('admin', 'user2', 'Never been better');
INSERT 0 1
=> SELECT * FROM test.message;
 id |           timestamp           | from_user | to_user |     message
----+-------------------------------+-----------+---------+-------------------
  2 | 2018-10-09 14:05:21.340886+00 | user2     | admin   | Hi, how are you?
  4 | 2018-10-09 14:06:33.356161+00 | admin     | user2   | Never been better
=> UPDATE test.message SET from_user = 'user1' WHERE id = 4;
ERROR:  new row violates row-level security policy for table "message"
=> UPDATE test.message SET from_user = 'admin' WHERE id = 1;
UPDATE 0
=> INSERT INTO test.message (from_user, to_user, message)
   VALUES ('admin', 'user1', 'Bye');
INSERT 0 1
=> SELECT * FROM test.message WHERE 'user1' IN (from_user, to_user);
 id |           timestamp           | from_user | to_user |     message
----+-------------------------------+-----------+---------+-------------------
  5 | 2018-10-09 14:06:58.108423+00 | admin     | user1   | Bye
```

Row with id 1
not visible

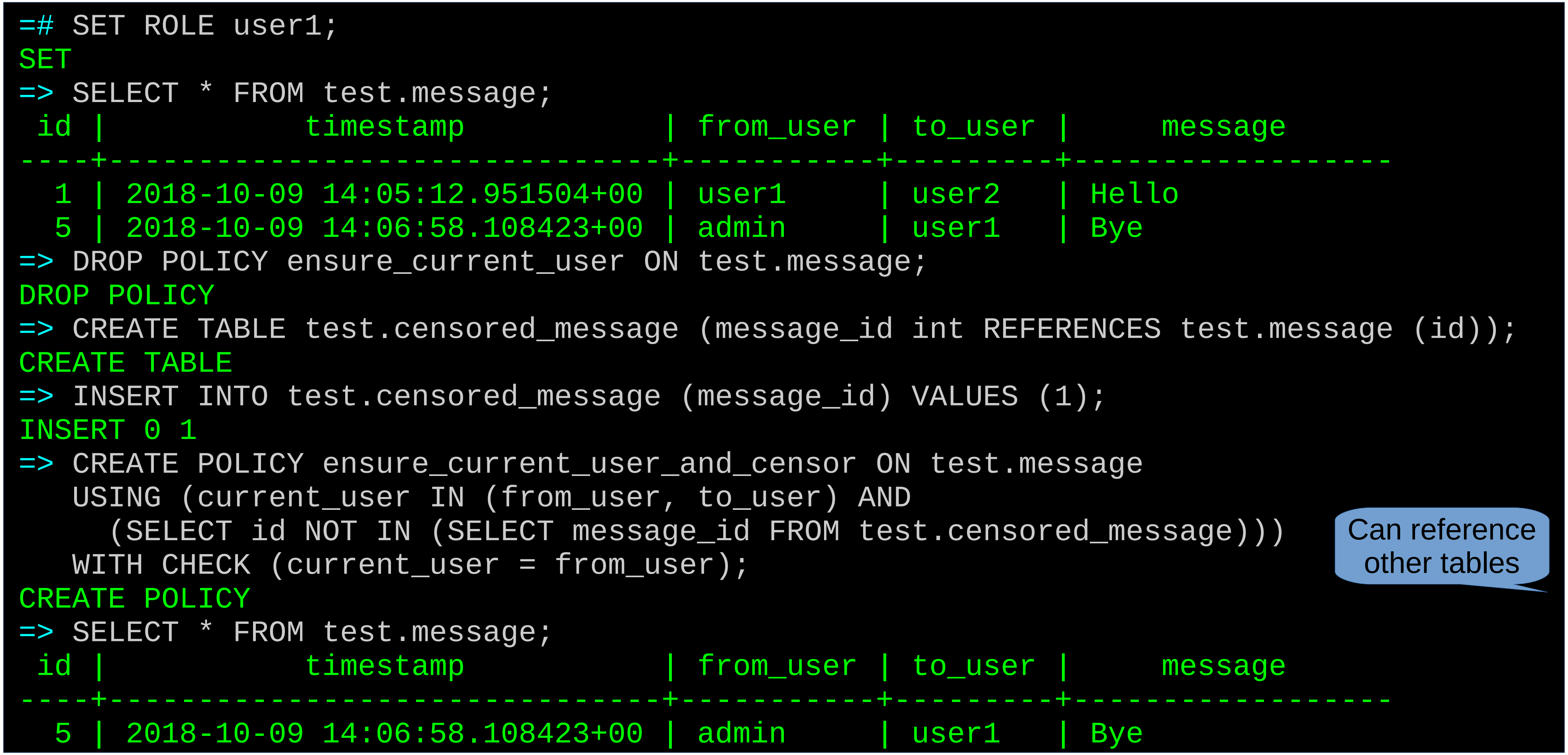
PostgreSQL Row Security Policies



```
=> RESET ROLE;
RESET
=# SELECT * FROM test.message;
 id |           timestamp           | from_user | to_user |     message
----+-------------------------------+-----------+---------+------------------
  1 | 2018-10-09 14:05:12.951504+00 | user1     | user2   | Hello
  2 | 2018-10-09 14:05:21.340886+00 | user2     | admin   | Hi, how are you?
  4 | 2018-10-09 14:06:33.356161+00 | admin     | user2   | Never been better
  5 | 2018-10-09 14:06:58.108423+00 | admin     | user1   | Bye
=# SELECT session_user, current_user;
 session_user | current_user
--------------+--------------
 postgres     | postgres
```

Superuser not subject to RLS

PostgreSQL Row Security Policies



```
=# SET ROLE user1;
SET
=> SELECT * FROM test.message;
 id |           timestamp           | from_user | to_user |     message
----+-------------------------------+-----------+---------+------------------
  1 | 2018-10-09 14:05:12.951504+00 | user1     | user2   | Hello
  5 | 2018-10-09 14:06:58.108423+00 | admin     | user1   | Bye
=> DROP POLICY ensure_current_user ON test.message;
DROP POLICY
=> CREATE TABLE test.censored_message (message_id int REFERENCES test.message (id));
CREATE TABLE
=> INSERT INTO test.censored_message (message_id) VALUES (1);
INSERT 0 1
=> CREATE POLICY ensure_current_user_and_censor ON test.message
   USING (current_user IN (from_user, to_user) AND
     (SELECT id NOT IN (SELECT message_id FROM test.censored_message)))
   WITH CHECK (current_user = from_user);
CREATE POLICY
=> SELECT * FROM test.message;
 id |           timestamp           | from_user | to_user |     message
----+-------------------------------+-----------+---------+------------------
  5 | 2018-10-09 14:06:58.108423+00 | admin     | user1   | Bye
```

Can reference other tables

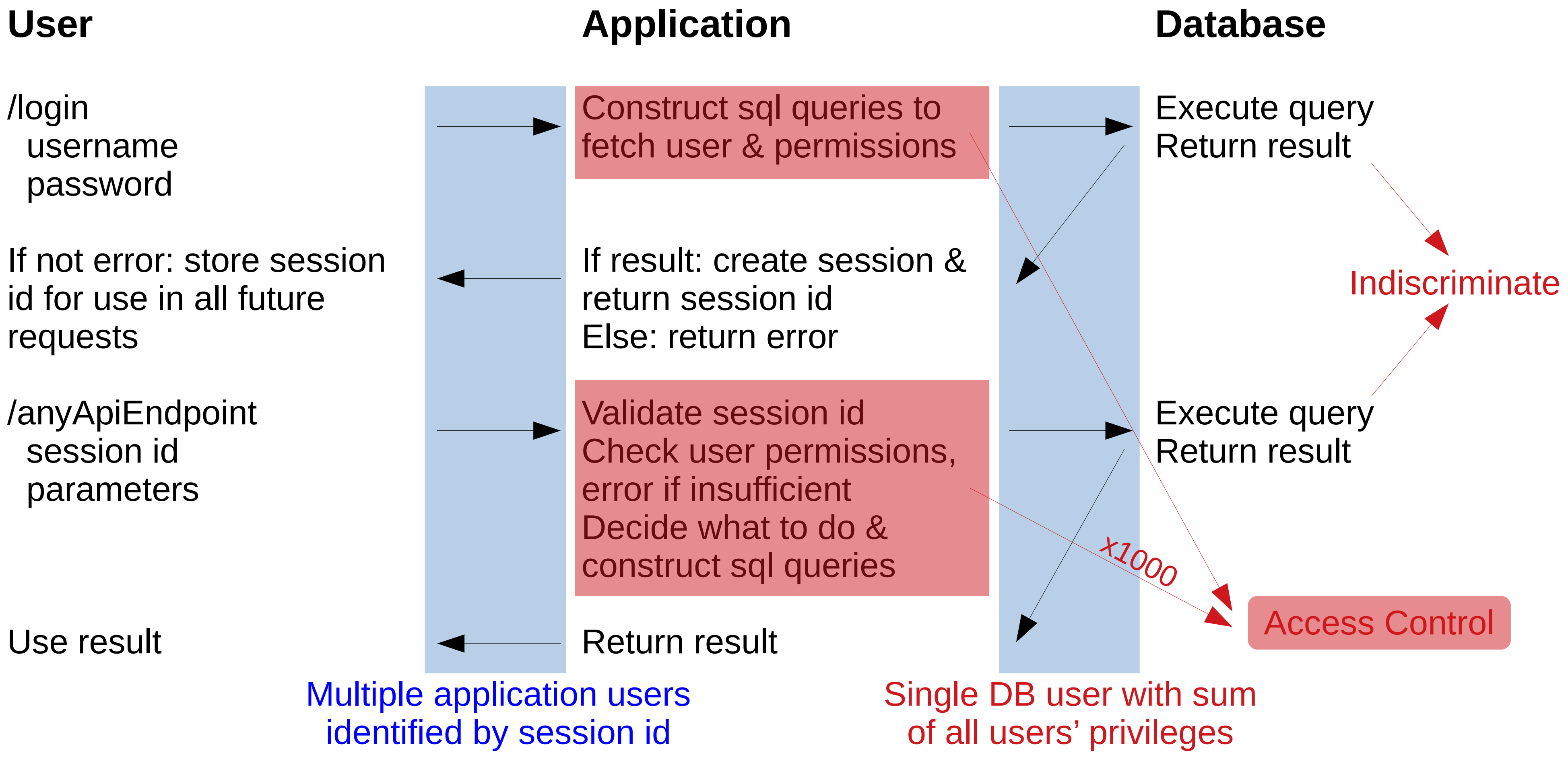
Database Users vs Application Users

Database Users vs Application Users

	Database Users **connect with user**	**Database Users** **connect with other, SET ROLE user**	**Application Users**
User list	System table pg_catalog.pg_authid		Any custom table e.g. myschema.myuser
DB connections	Many: one DB user per connection	One: multiplex DB users using SET ROLE	One: multiplex application users over one DB user
Secure from vulnerable application code	Yes	No	No
User scope	Global across database cluster		Local, typically to database or schema
Easily manage dynamic permissions	Yes (role membership)		Yes (custom scheme)

Application Users: Typical Access Control

Application Users: Typical Access Control



User

Application

Database

/login
username
password

Construct sql queries to fetch user & permissions

Execute query
Return result

If not error: store session id for use in all future requests

If result: create session & return session id
Else: return error

Indiscriminate

/anyApiEndpoint
session id
parameters

Validate session id
Check user permissions, error if insufficient
Decide what to do & construct sql queries

Execute query
Return result

x1000

Access Control

Use result

Return result

Multiple application users identified by session id

Single DB user with sum of all users’ privileges

Application Users: Better Access Control using RLS

Application Users: Better Access Control using RLS

Requirements:

1. Custom user table (multiple application users)
2. One database connection (one database user)
3. Secure from vulnerable application code (database performs access control)
4. Users local to database (usernames reusable over multiple applications/databases)
5. Easily manage dynamic permissions (custom permission scheme)

For database access control the DB requires reliable knowledge of current application user

Naive solution is to store identification (e.g. user id) in session variable, but this is easily forged, so how can this value be made trustworthy?

Possible solutions:

SECURITY DEFINER functions to sign/validate the variable using secret key
(https://blog.2ndquadrant.com/application-users-vs-row-level-security/)

SECURITY DEFINER functions to create/validate an unguessable session id which is stored in the variable

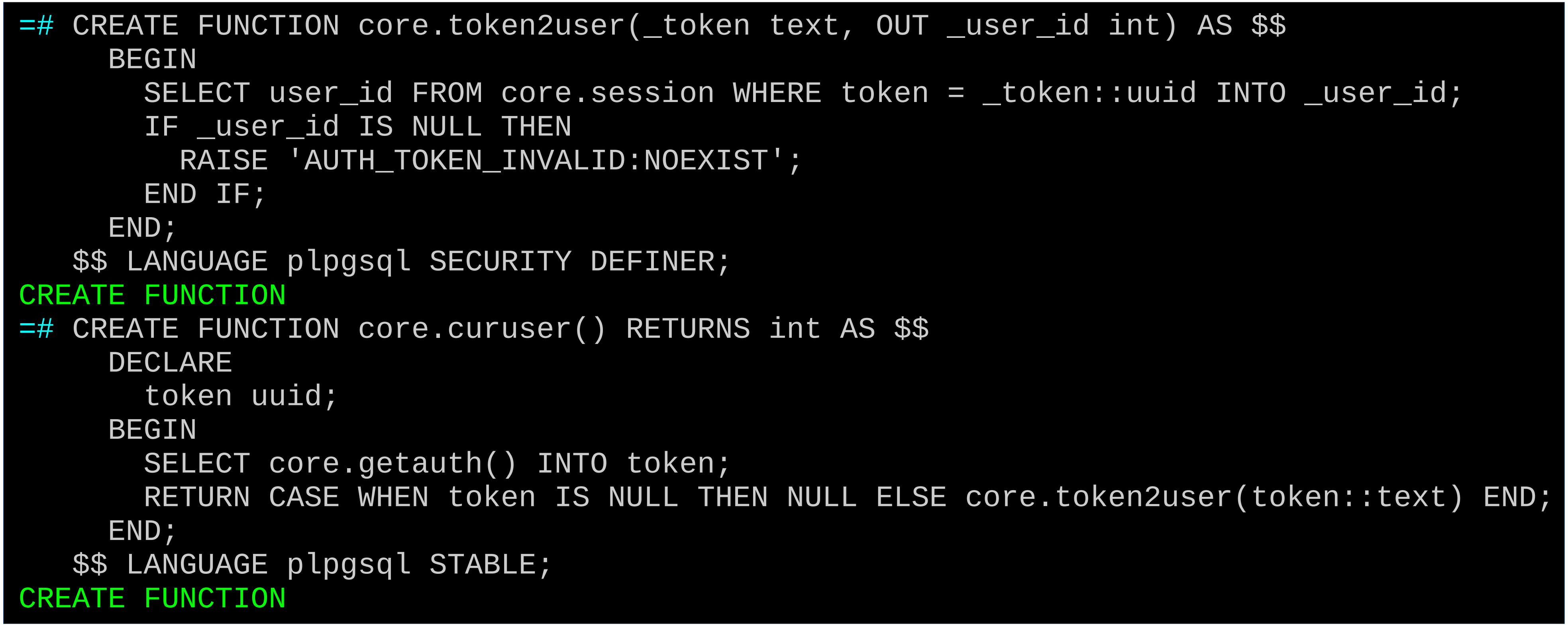
Application Users: Better Access Control using RLS

```
$ psql -U postgres postgres
=# CREATE EXTENSION pgcrypto;
CREATE EXTENSION
=# CREATE SCHEMA core;
CREATE SCHEMA
=# CREATE TABLE core.user (
     id       serial PRIMARY KEY,
     username text NOT NULL UNIQUE,
     password text NOT NULL);
CREATE TABLE
=# CREATE TABLE core.session (
     id      serial PRIMARY KEY,
     user_id int NOT NULL REFERENCES core.user (id),
     token   uuid NOT NULL DEFAULT gen_random_uuid() UNIQUE);
CREATE TABLE
```

Application Users: Better Access Control using RLS

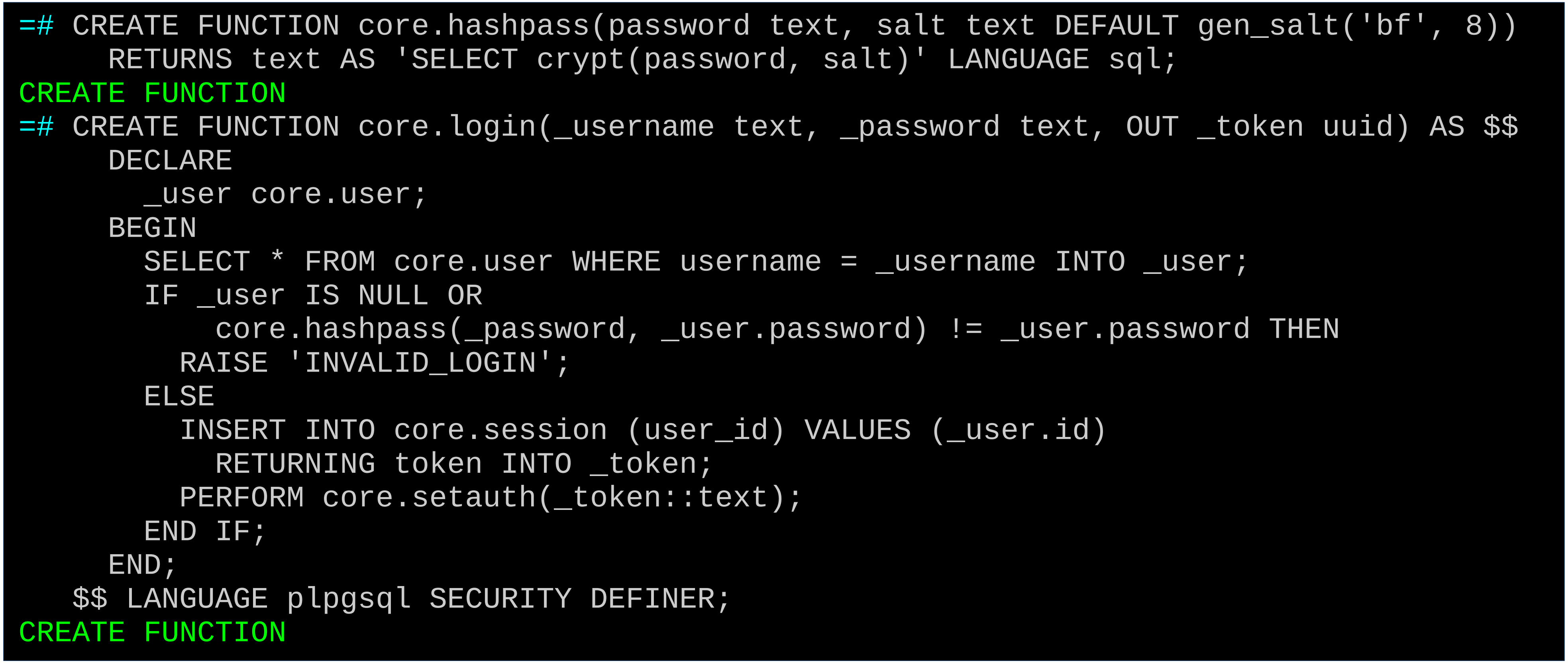
```
=# CREATE FUNCTION core.getauth(OUT token uuid) AS $$
     BEGIN
       SELECT nullif(current_setting('core.auth_token'), '') INTO token;
     EXCEPTION WHEN undefined_object THEN
     END;
   $$ LANGUAGE plpgsql STABLE;
CREATE FUNCTION
=# CREATE FUNCTION core.setauth(token text) RETURNS uuid AS $$
     BEGIN
       PERFORM set_config('core.auth_token', token, false);
       RETURN core.getauth();
     END;
   $$ LANGUAGE plpgsql;
CREATE FUNCTION
```

Application Users: Better Access Control using RLS



```
=# CREATE FUNCTION core.token2user(_token text, OUT _user_id int) AS $$
     BEGIN
       SELECT user_id FROM core.session WHERE token = _token::uuid INTO _user_id;
       IF _user_id IS NULL THEN
         RAISE 'AUTH_TOKEN_INVALID:NOEXIST';
       END IF;
     END;
   $$ LANGUAGE plpgsql SECURITY DEFINER;
CREATE FUNCTION
=# CREATE FUNCTION core.curuser() RETURNS int AS $$
     DECLARE
       token uuid;
     BEGIN
       SELECT core.getauth() INTO token;
       RETURN CASE WHEN token IS NULL THEN NULL ELSE core.token2user(token::text) END;
     END;
   $$ LANGUAGE plpgsql STABLE;
CREATE FUNCTION
```

Application Users: Better Access Control using RLS

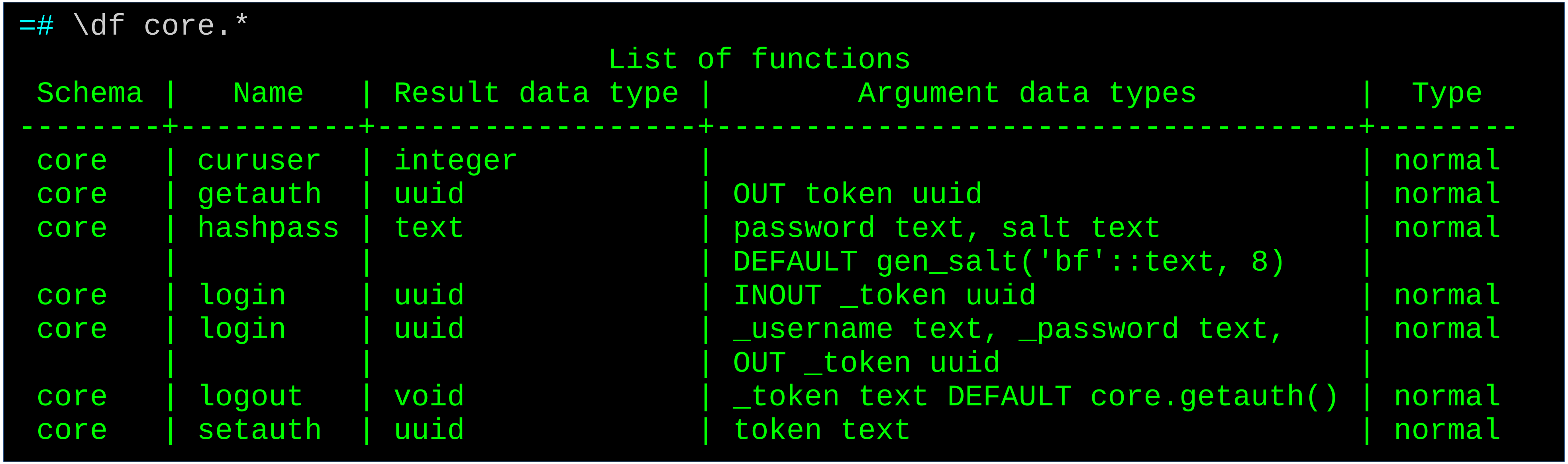


```
=# CREATE FUNCTION core.hashpass(password text, salt text DEFAULT gen_salt('bf', 8))
     RETURNS text AS 'SELECT crypt(password, salt)' LANGUAGE sql;
CREATE FUNCTION
=# CREATE FUNCTION core.login(_username text, _password text, OUT _token uuid) AS $$
     DECLARE
       _user core.user;
     BEGIN
       SELECT * FROM core.user WHERE username = _username INTO _user;
       IF _user IS NULL OR
           core.hashpass(_password, _user.password) != _user.password THEN
         RAISE 'INVALID_LOGIN';
       ELSE
         INSERT INTO core.session (user_id) VALUES (_user.id)
           RETURNING token INTO _token;
         PERFORM core.setauth(_token::text);
       END IF;
     END;
   $$ LANGUAGE plpgsql SECURITY DEFINER;
CREATE FUNCTION
```

Application Users: Better Access Control using RLS

```
=# CREATE FUNCTION core.login(INOUT _token uuid) AS $$
     BEGIN
       PERFORM core.setauth(NULL);
       PERFORM core.token2user(_token); -- Validate token.
       PERFORM core.setauth(_token::text);
     END;
   $$ LANGUAGE plpgsql SECURITY DEFINER;
CREATE FUNCTION
=# CREATE FUNCTION core.logout(_token text DEFAULT core.getauth()) RETURNS VOID AS $$
     BEGIN
       BEGIN
         DELETE FROM core.session WHERE token = _token::uuid;
       EXCEPTION WHEN OTHERS THEN
       END;
       PERFORM core.setauth(NULL);
     END;
   $$ LANGUAGE plpgsql SECURITY DEFINER;
CREATE FUNCTION
```

Application Users: Better Access Control using RLS

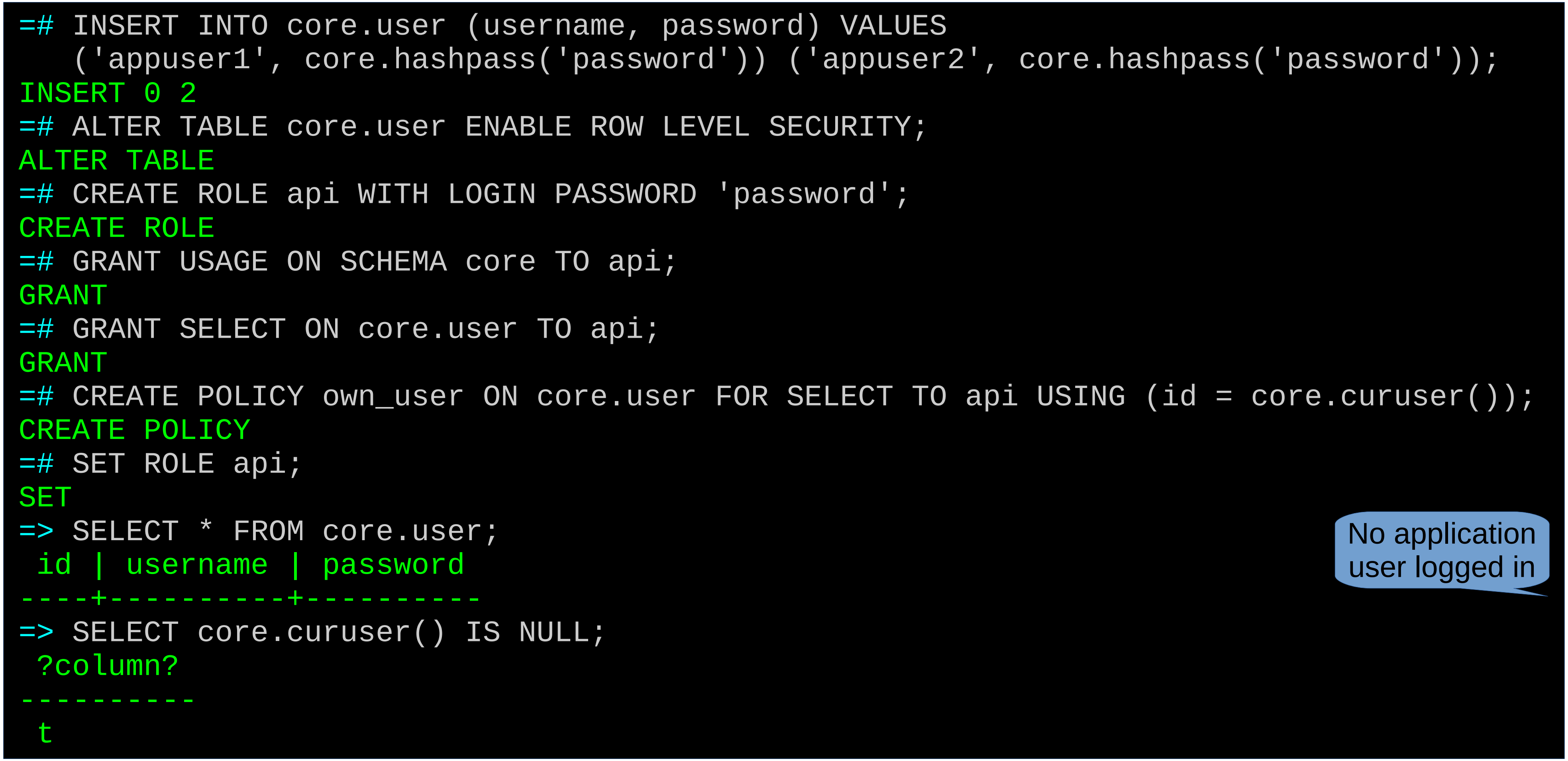


```
=# \df core.*
                                 List of functions
 Schema |   Name   | Result data type |        Argument data types         |  Type  
--------+----------+------------------+------------------------------------+--------
 core   | curuser  | integer          |                                    | normal
 core   | getauth  | uuid             | OUT token uuid                     | normal
 core   | hashpass | text             | password text, salt text           | normal
        |          |                  | DEFAULT gen_salt('bf'::text, 8)    |
 core   | login    | uuid             | INOUT _token uuid                  | normal
 core   | login    | uuid             | _username text, _password text,    | normal
        |          |                  | OUT _token uuid                    |
 core   | logout   | void             | _token text DEFAULT core.getauth() | normal
 core   | setauth  | uuid             | token text                         | normal
```

core.login(username text, password text): uuid	- Login and create new session
core.login(token uuid): uuid	- Login using existing session
core.curuser(): int	- ID of currently logged in user, or NULL
core.logout(): void	- Logout and delete current session

Current session token stored in core.auth_token variable

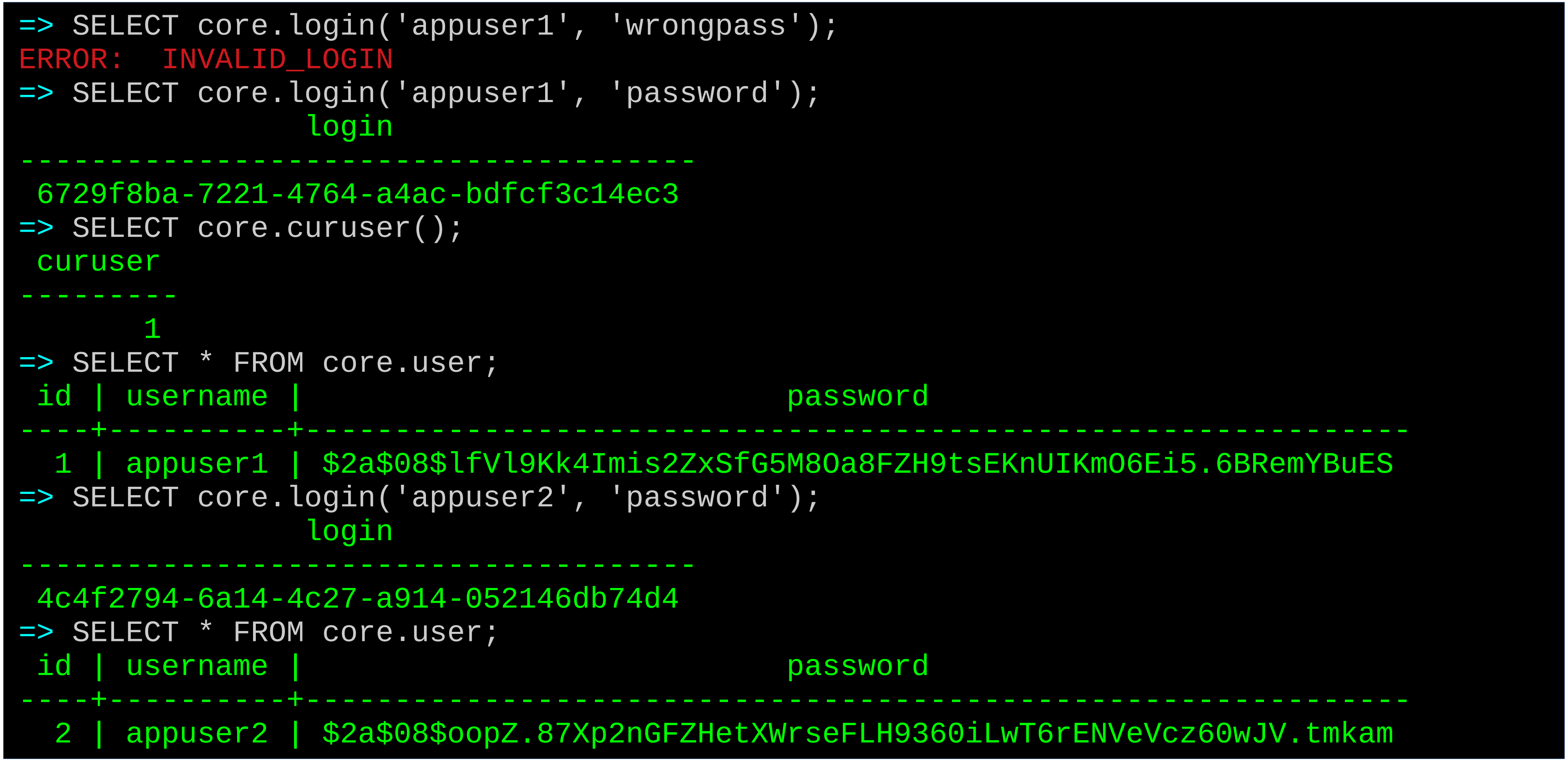
Application Users: Better Access Control using RLS



```
=# INSERT INTO core.user (username, password) VALUES
   ('appuser1', core.hashpass('password')) ('appuser2', core.hashpass('password'));
INSERT 0 2
=# ALTER TABLE core.user ENABLE ROW LEVEL SECURITY;
ALTER TABLE
=# CREATE ROLE api WITH LOGIN PASSWORD 'password';
CREATE ROLE
=# GRANT USAGE ON SCHEMA core TO api;
GRANT
=# GRANT SELECT ON core.user TO api;
GRANT
=# CREATE POLICY own_user ON core.user FOR SELECT TO api USING (id = core.curuser());
CREATE POLICY
=# SET ROLE api;
SET
=> SELECT * FROM core.user;
 id | username | password
----+----------+----------
=> SELECT core.curuser() IS NULL;
 ?column?
----------
 t
```

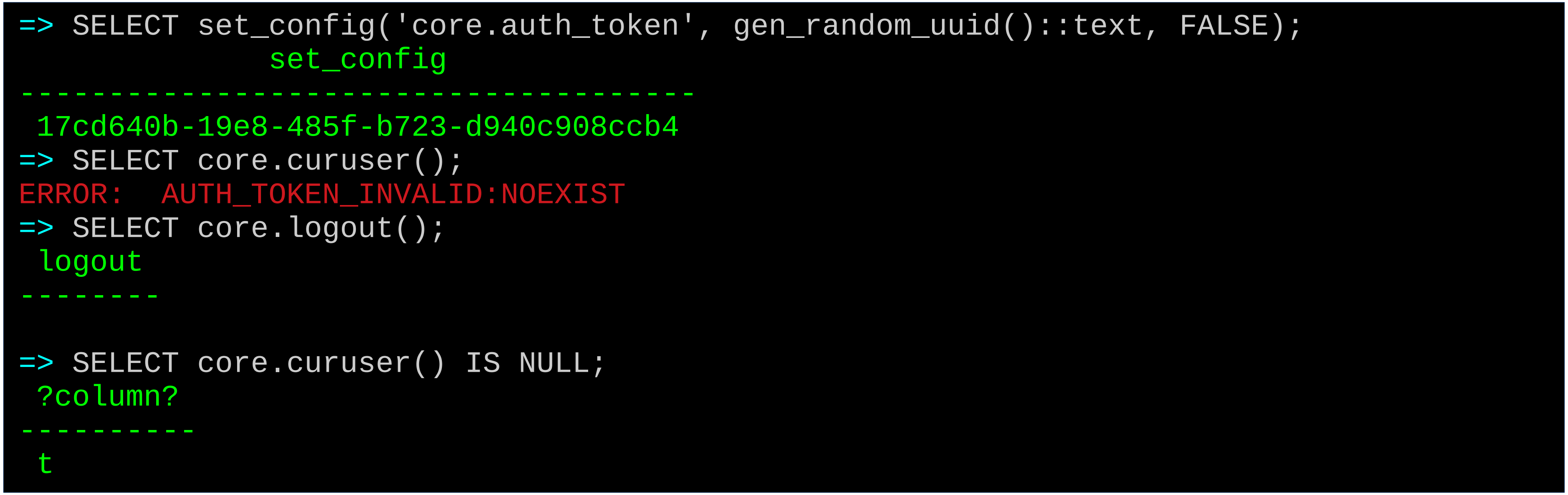
No application user logged in

Application Users: Better Access Control using RLS



```
=> SELECT core.login('appuser1', 'wrongpass');
ERROR:  INVALID_LOGIN
=> SELECT core.login('appuser1', 'password');
                login
--------------------------------------
 6729f8ba-7221-4764-a4ac-bdfcf3c14ec3
=> SELECT core.curuser();
 curuser
---------
       1
=> SELECT * FROM core.user;
 id | username |                           password
----+----------+--------------------------------------------------------------
  1 | appuser1 | $2a$08$lfVl9Kk4Imis2ZxSfG5M8Oa8FZH9tsEKnUIKmO6Ei5.6BRemYBuES
=> SELECT core.login('appuser2', 'password');
                login
--------------------------------------
 4c4f2794-6a14-4c27-a914-052146db74d4
=> SELECT * FROM core.user;
 id | username |                           password
----+----------+--------------------------------------------------------------
  2 | appuser2 | $2a$08$oopZ.87Xp2nGFZHetXWrseFLH9360iLwT6rENVeVcz60wJV.tmkam
```

Application Users: Better Access Control using RLS



```
=> SELECT set_config('core.auth_token', gen_random_uuid()::text, FALSE);
              set_config
--------------------------------------
 17cd640b-19e8-485f-b723-d940c908ccb4
=> SELECT core.curuser();
ERROR:  AUTH_TOKEN_INVALID:NOEXIST
=> SELECT core.logout();
 logout
--------

=> SELECT core.curuser() IS NULL;
 ?column?
----------
 t
```

Access Control: Moving Everything to the Database

Access Control: Moving Everything to the Database

Is it possible to move *all* access control into the database?

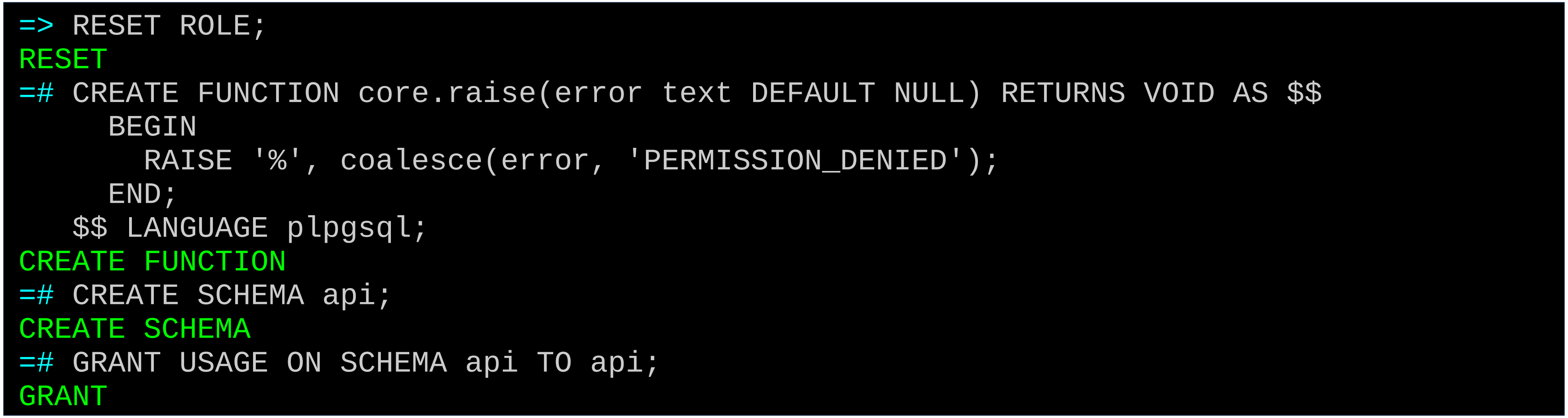
SQL Privilege System: control access to **columns** based on **database users**
Row Security Policies: control access to **rows** based on expressions (**application users**)

Is it possible to control access to **columns** based on **application users**?
How about controlling access to **individual cells**? Say what?

Use case: For core.user, allow anyone to select from id or username, but only allow selecting from password if you are logged in as that user

Solution: Abstraction using Views & Rules

Access Control: Moving Everything to the Database



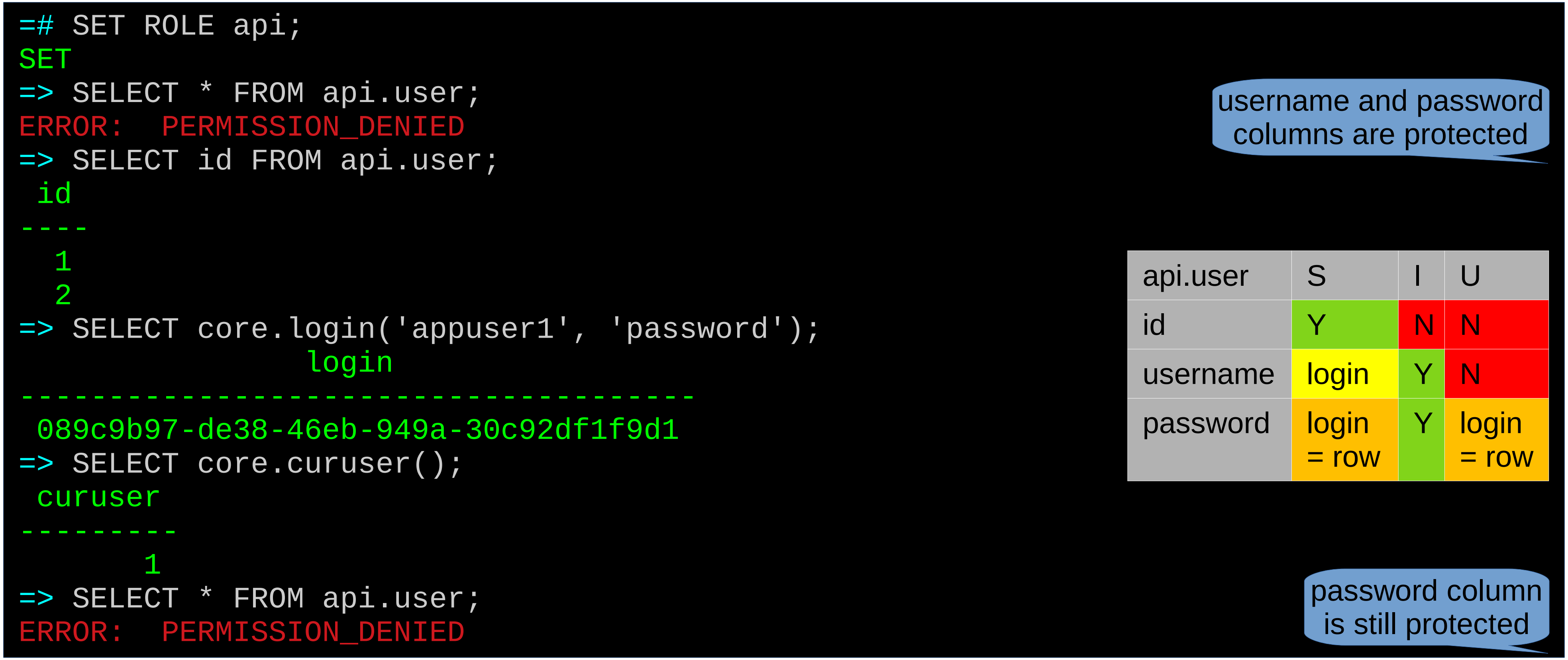
```
=> RESET ROLE;
RESET
=# CREATE FUNCTION core.raise(error text DEFAULT NULL) RETURNS VOID AS $$
     BEGIN
       RAISE '%', coalesce(error, 'PERMISSION_DENIED');
     END;
   $$ LANGUAGE plpgsql;
CREATE FUNCTION
=# CREATE SCHEMA api;
CREATE SCHEMA
=# GRANT USAGE ON SCHEMA api TO api;
GRANT
```

Access Control: Moving Everything to the Database

api.user	S	I	U
id	Y	N	N
username	login	Y	N
password	login = row	Y	login = row

```
=# CREATE VIEW api.user AS
     WITH curuser AS (SELECT core.curuser())
     SELECT
       id,
       CASE WHEN curuser IS NOT NULL THEN username
         ELSE core.raise()::text END AS username,
       CASE WHEN id = curuser THEN password
         ELSE core.raise()::text END AS password
     FROM core.user, curuser;
CREATE VIEW
=# CREATE RULE _INSERT AS ON INSERT TO api.user DO INSTEAD
   INSERT INTO core.user (username, password) VALUES (NEW.username, NEW.password);
CREATE RULE
=# CREATE RULE _UPDATE AS ON UPDATE TO api.user DO INSTEAD
   UPDATE core.user SET
     password = CASE WHEN OLD.id = core.curuser() THEN NEW.password
       ELSE core.raise()::text END
   WHERE id = OLD.id;
CREATE RULE
=# GRANT SELECT, INSERT (username, password), UPDATE (password) ON api.user TO api;
GRANT
=# GRANT USAGE ON SEQUENCE core.user_id_seq TO api;
GRANT
```

Access Control: Moving Everything to the Database



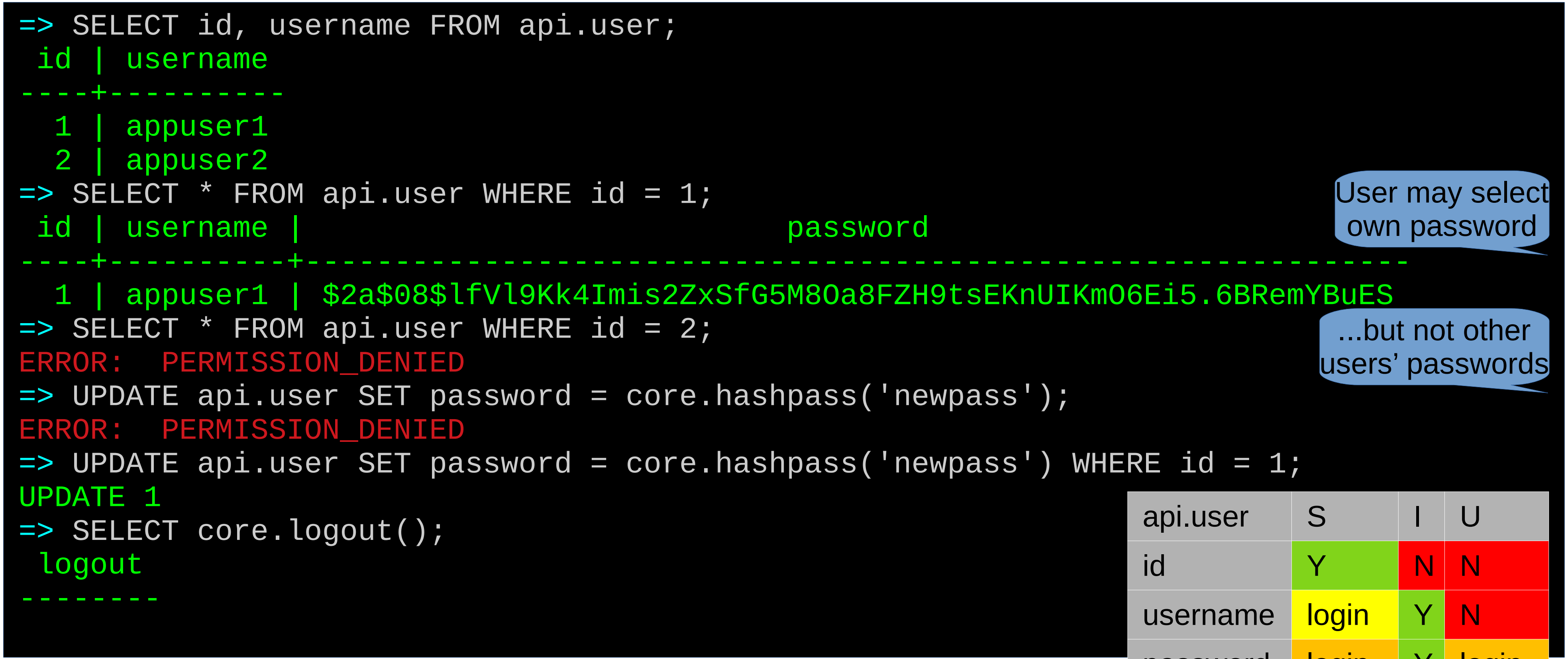
```
=# SET ROLE api;
SET
=> SELECT * FROM api.user;
ERROR:  PERMISSION_DENIED
=> SELECT id FROM api.user;
 id
----
  1
  2
=> SELECT core.login('appuser1', 'password');
                login
--------------------------------------
 089c9b97-de38-46eb-949a-30c92df1f9d1
=> SELECT core.curuser();
 curuser
---------
       1
=> SELECT * FROM api.user;
ERROR:  PERMISSION_DENIED
```

username and password columns are protected

api.user	S	I	U
id	Y	N	N
username	login	Y	N
password	login = row	Y	login = row

password column is still protected

Access Control: Moving Everything to the Database



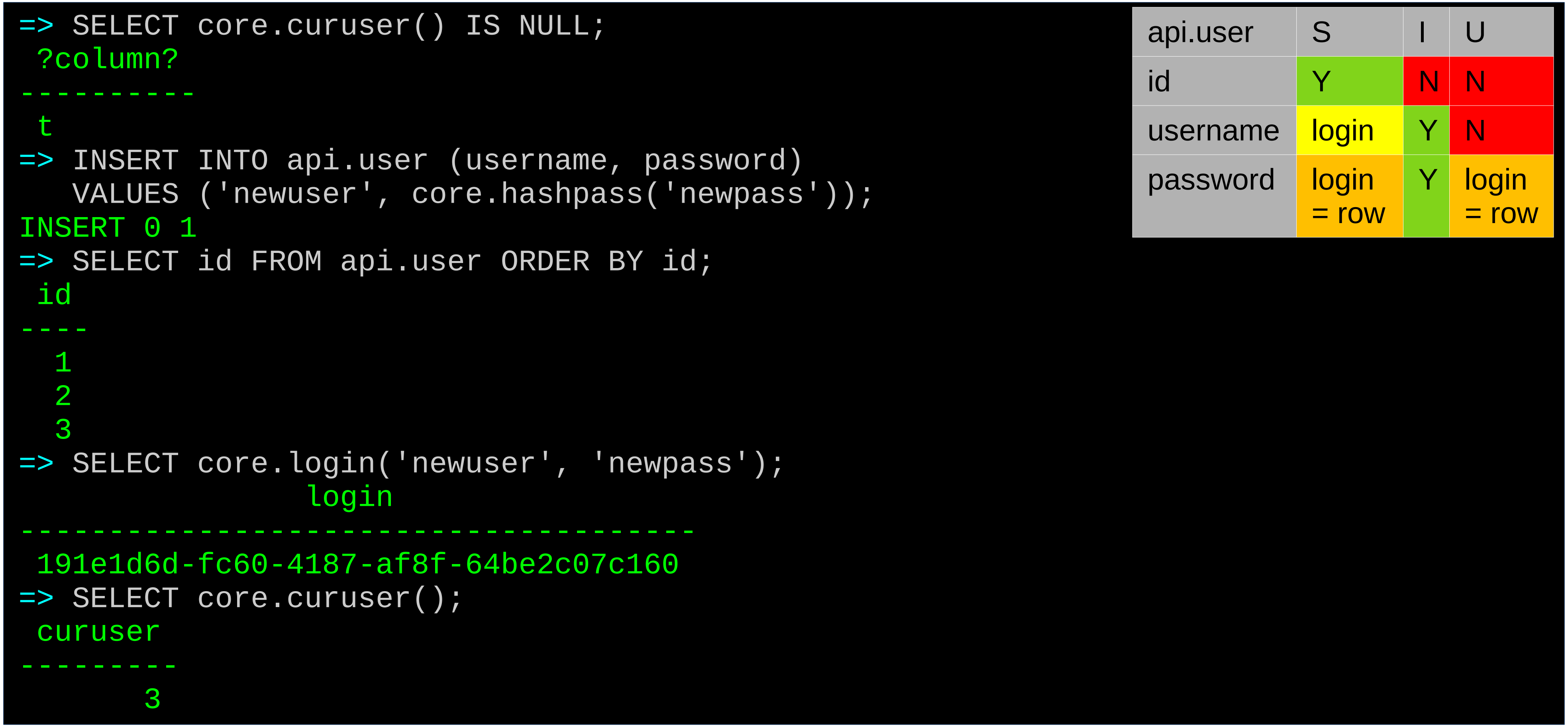
```
=> SELECT id, username FROM api.user;
 id | username
----+----------
  1 | appuser1
  2 | appuser2
=> SELECT * FROM api.user WHERE id = 1;
 id | username |                           password
----+----------+--------------------------------------------------------------
  1 | appuser1 | $2a$08$lfVl9Kk4Imis2ZxSfG5M8Oa8FZH9tsEKnUIKmO6Ei5.6BRemYBuES
=> SELECT * FROM api.user WHERE id = 2;
ERROR:  PERMISSION_DENIED
=> UPDATE api.user SET password = core.hashpass('newpass');
ERROR:  PERMISSION_DENIED
=> UPDATE api.user SET password = core.hashpass('newpass') WHERE id = 1;
UPDATE 1
=> SELECT core.logout();
 logout
--------
```

User may select own password

...but not other users' passwords

api.user	S	I	U
id	Y	N	N
username	login	Y	N
password	login = row	Y	login = row

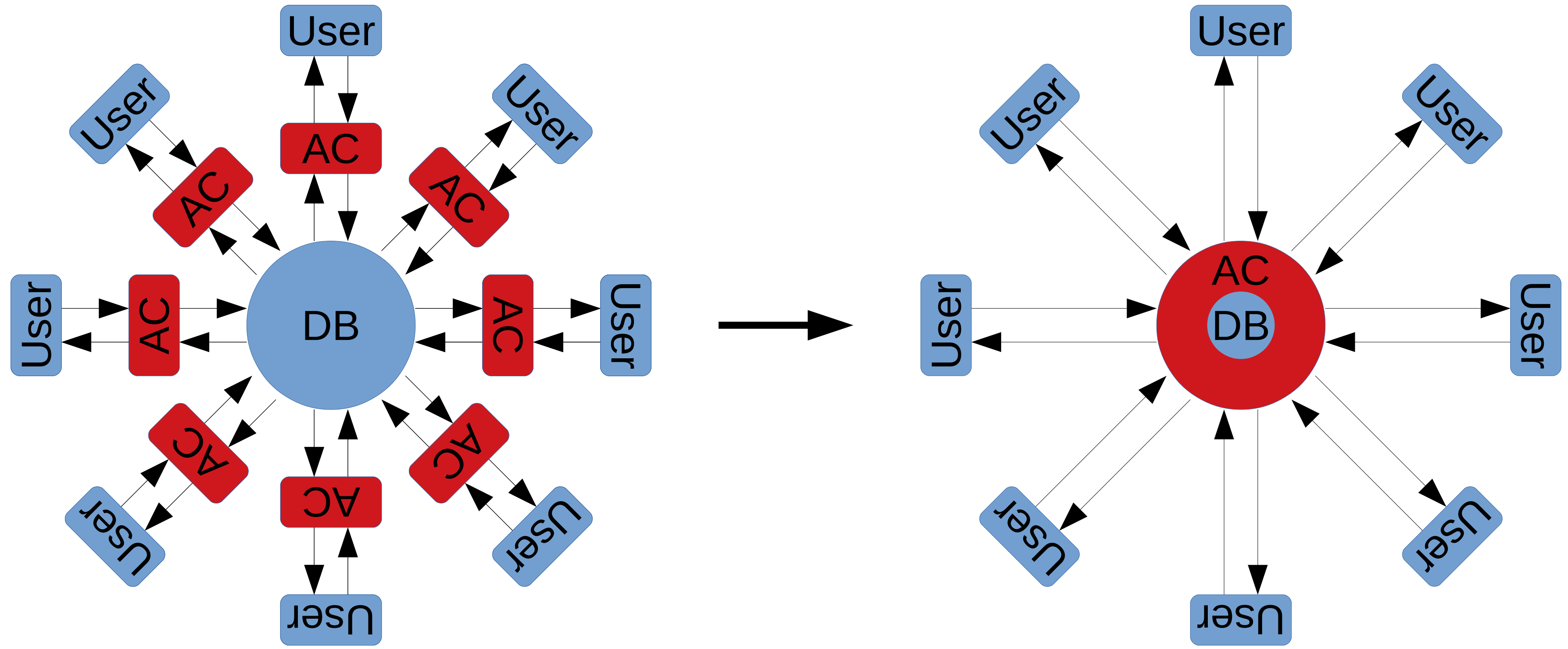
Access Control: Moving Everything to the Database



```
=> SELECT core.curuser() IS NULL;
 ?column?
----------
 t
=> INSERT INTO api.user (username, password)
   VALUES ('newuser', core.hashpass('newpass'));
INSERT 0 1
=> SELECT id FROM api.user ORDER BY id;
 id
----
  1
  2
  3
=> SELECT core.login('newuser', 'newpass');
                login
--------------------------------------
 191e1d6d-fc60-4187-af8f-64be2c07c160
=> SELECT core.curuser();
 curuser
---------
       3
```

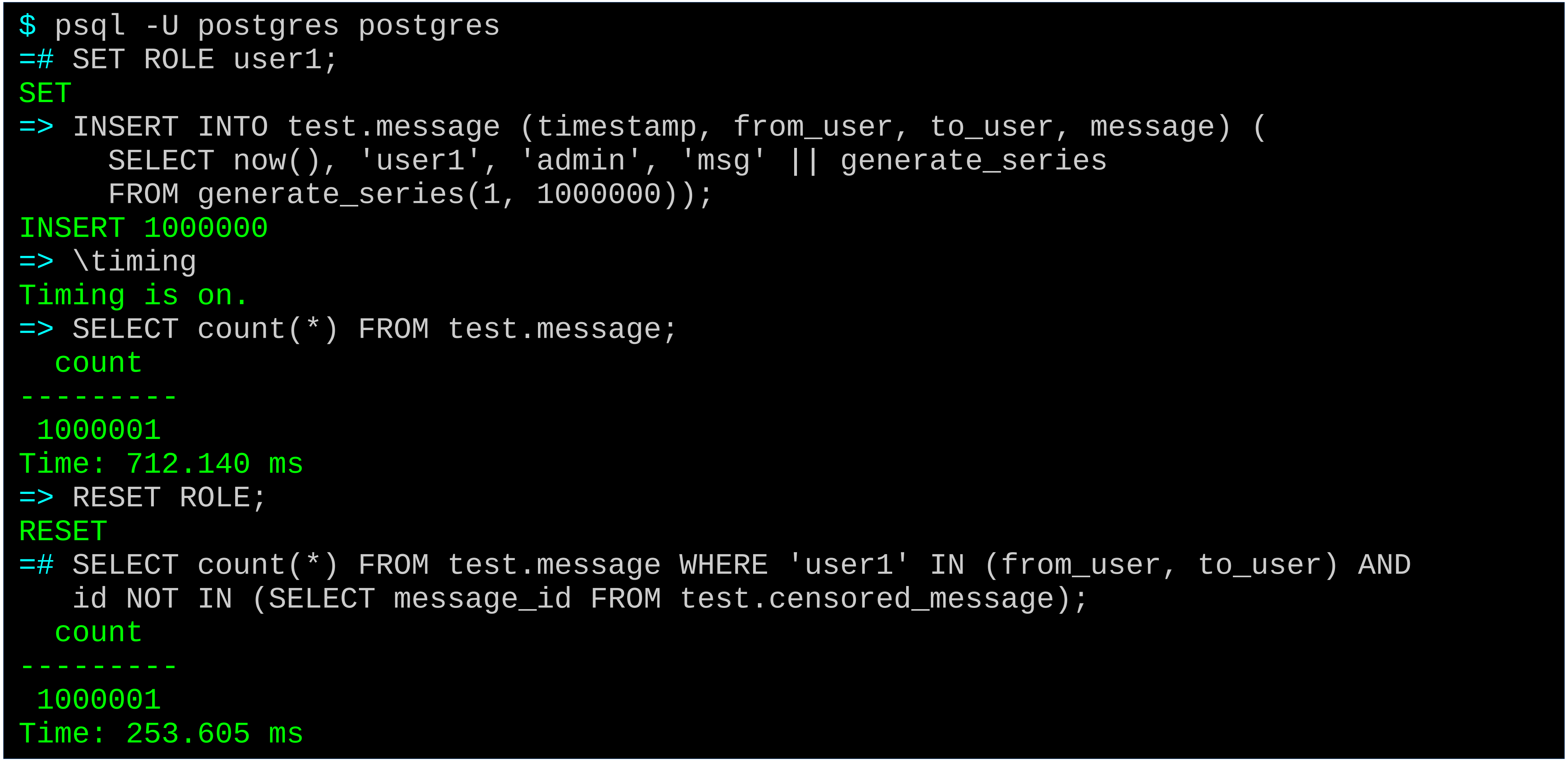
api.user	S	I	U
id	Y	N	N
username	login	Y	N
password	login = row	Y	login = row

Access Control: Moving Everything to the Database



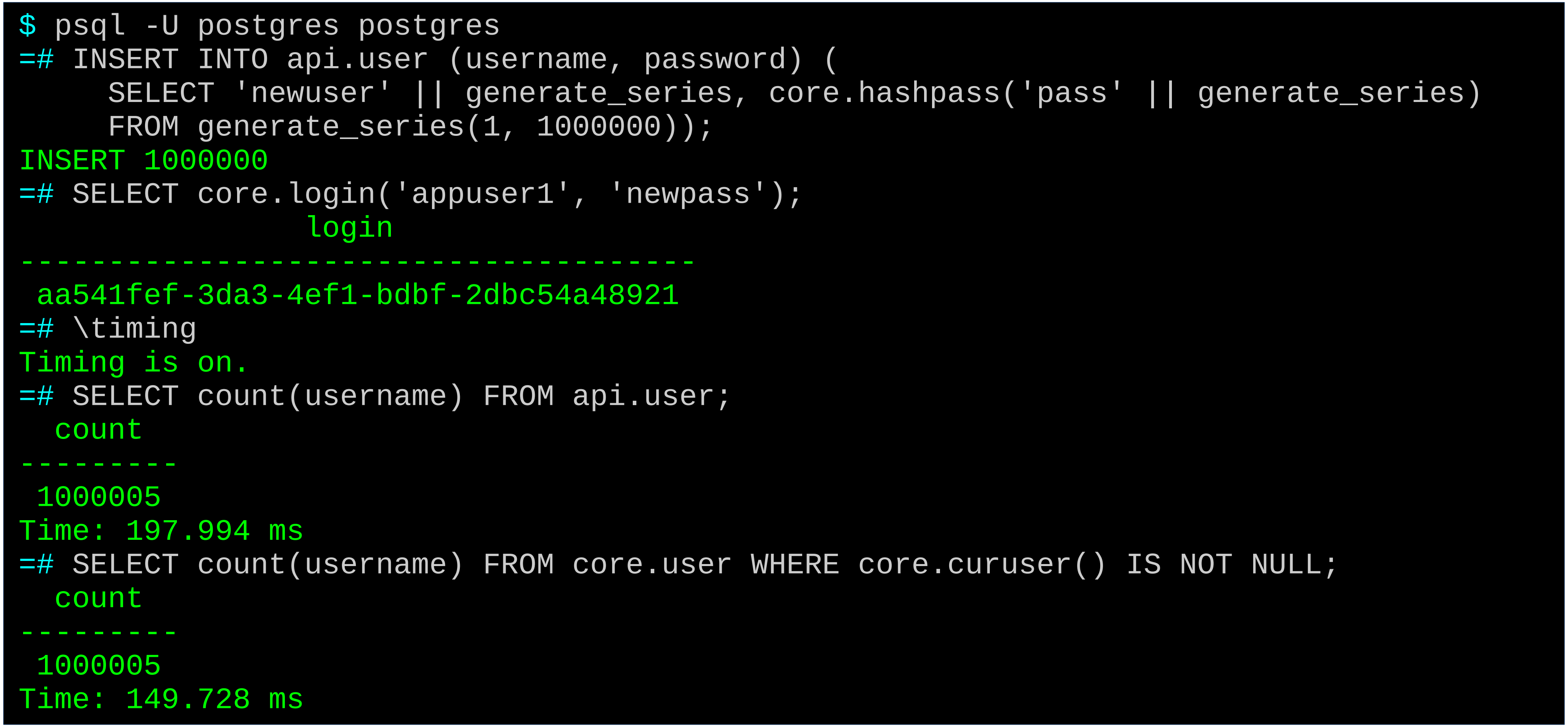
Performance Impacts

Performance Impacts



```
$ psql -U postgres postgres
=# SET ROLE user1;
SET
=> INSERT INTO test.message (timestamp, from_user, to_user, message) (
     SELECT now(), 'user1', 'admin', 'msg' || generate_series
     FROM generate_series(1, 1000000));
INSERT 1000000
=> \timing
Timing is on.
=> SELECT count(*) FROM test.message;
  count
---------
 1000001
Time: 712.140 ms
=> RESET ROLE;
RESET
=# SELECT count(*) FROM test.message WHERE 'user1' IN (from_user, to_user) AND
   id NOT IN (SELECT message_id FROM test.censored_message);
  count
---------
 1000001
Time: 253.605 ms
```

Performance Impacts



```
$ psql -U postgres postgres
=# INSERT INTO api.user (username, password) (
     SELECT 'newuser' || generate_series, core.hashpass('pass' || generate_series)
     FROM generate_series(1, 1000000));
INSERT 1000000
=# SELECT core.login('appuser1', 'newpass');
                login
--------------------------------------
 aa541fef-3da3-4ef1-bdbf-2dbc54a48921
=# \timing
Timing is on.
=# SELECT count(username) FROM api.user;
  count
---------
 1000005
Time: 197.994 ms
=# SELECT count(username) FROM core.user WHERE core.curuser() IS NOT NULL;
  count
---------
 1000005
Time: 149.728 ms
```

Performance Impacts

RLS and View Abstraction both slower than their manually constrained counterparts
This is expected – but how much slower?

Row Level Security - 2.8x
View Abstraction - 1.3x
} Very informal, so don't pay too much attention to this
Just note it's not *that* much slower

So if it is slower, then why do it?

Security:

- All access control performed by database – even if application code is compromised
- Essentially, users can be given freeform sql access – database is a Fort Knox and will not allow unauthorized operations

Developer productivity:

- No more time spent on access control and worrying about security
- Even the new guy can now safely work on applications, api's etc.
- Worst case, api breaks, but the data is perfectly safe

Thanks!

Comments and Questions