Intro to Stored Procedures in PostgreSQL

Edwin Grubbs
May 12, 2008
edwin@grubbs.org

Why use Stored Procedures?

- Reusability
  - Avoid rewriting subqueries and improve readability.
  - If you can't store a query in a library that all the applications can access, you can put that query in a stored procedure.
- Separation of duties
  - You don't trust non-DBA's to write queries.
- Data integrity
  - Use triggers or constraints to prevent bad data from entering.
  - Run several interdependent queries in a transaction in a single stored procedure.
- Event handling
  - Log changes.
  - Notify other systems of new data.

Why NOT use Stored Procedures?

- Views may be all you need.
- An object-relational mapper (ORM) can help write queries safely.
- Difficult to version control stored procedures.
- Software rollouts may require more db changes.
- Could slow software development process.

Writing a Stored Procedure in SQL

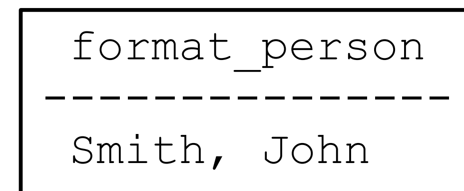
```
CREATE TABLE person (
   id SERIAL PRIMARY KEY,
   first_name TEXT,
   last_name TEXT);
```

```
CREATE OR REPLACE FUNCTION insert_person(text, text)
RETURNS void AS
$delimiter$
   INSERT INTO person (first_name, last_name)
   VALUES ($1, $2);
$delimiter$
LANGUAGE SQL;
```

```
CREATE FUNCTION format_person(person)
RETURNS text AS
$$
   SELECT $1.last_name || ', ' || $1.first_name;
$$
LANGUAGE SQL;
```

```
SELECT insert_person('John', 'Smith');

SELECT format_person(person.*)
FROM person;
```



```
 format_person
----------------
 Smith, John
```

Writing Stored Procedure in PL/pgSQL

```
CREATE LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION get_last_name(first_name text)
RETURNS text AS $body$
DECLARE
    table TEXT;
    query TEXT;
    row RECORD;
BEGIN
    IF first_name = 'John' THEN
        table := 'person';
    ELSIF first_name = 'Moe' THEN
        table := 'loser';
    ELSE
        RETURN 'unknown table';
    END IF;

    RAISE NOTICE 'Table is "%"', table; -- print debug message

    query := 'SELECT last_name FROM ' || table
        || $$ WHERE first_name = '$$ || first_name || $$'$$;

    FOR row IN EXECUTE query LOOP
        RETURN row.last_name;
    END LOOP;
    RETURN 'not found';
END
$body$ LANGUAGE plpgsql;
```

Executing get_last_name()

```
$ psql -E test

test=# SELECT get_last_name('John');
NOTICE:  Table is "person"
 get_last_name
---------------
 Smith
(1 row)

test=# SELECT get_last_name('Bob');
 get_last_name
---------------
 unknown table
(1 row)

test=# SELECT get_last_name('Moe');
NOTICE:  Table is "loser"
ERROR:  relation "loser" does not exist
CONTEXT:  SQL statement "SELECT last_name FROM loser WHERE first_name =
'Moe'"
PL/pgSQL function "get_last_name" line 19 at for over execute statement
test=#
```

Triggers

- Triggers can be used to call a stored procedure before or after an INSERT, UPDATE, or DELETE statement on a table.
- Triggers can be called once per each row affected or once per each INSERT, UPDATE, or DELETE statement.
- Triggers on many different tables can share the same stored procedure.

Automatic Variables in Triggers

- NEW: The new row for INSERT/UPDATE statements. It is NULL in statement-level triggers.
- OLD: The old row for UPDATE/DELETE statements. It is NULL in statement-level triggers.
- TG_NAME: Name of the trigger.
- TG_WHEN: 'BEFORE' or 'AFTER'.
- TG_LEVEL: 'ROW' or 'STATEMENT'.
- TG_OP: 'INSERT', 'UPDATE', or 'DELETE'
- TG_TABLE_NAME: Name of the table that invoked the trigger.
- TG_TABLE_SCHEMA: Schema for the table in TG_TABLE_NAME.

Writing a Trigger in PL/pgSQL

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);
```

```
CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- Who works for us when she must pay for it?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Watching Triggers in Action

```
test=# begin;
test=# insert into emp values('john', 15000, NULL, NULL);
INSERT 0 1
test=# select * from emp;
 empname | salary |         last_date          | last_user
---------+--------+----------------------------+-----------
 john    |  15000 | 2008-05-12 08:03:36.015859 | egrubbs
(1 row)

test=# update emp set salary = 30000 where empname = 'john';
UPDATE 1
test=# select * from emp;
 empname | salary |         last_date          | last_user
---------+--------+----------------------------+-----------
 john    |  30000 | 2008-05-12 08:03:36.015859 | egrubbs
(1 row)

test=# commit;
COMMIT
test=# update emp set salary = 40000 where empname = 'john';
UPDATE 1
test=# select * from emp;
 empname | salary |         last_date          | last_user
---------+--------+----------------------------+-----------
 john    |  40000 | 2008-05-12 08:05:45.579625 | egrubbs
(1 row)

test=# update emp set salary = -5 where empname = 'john';
ERROR:  john cannot have a negative salary
```

Views are Actually Rules

Views in PostgreSQL are implemented using the rule system. In fact, there is essentially no difference between:

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

compared against the two commands:

```
CREATE TABLE myview (same column list as mytab);

CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

Syntax for creating rules:

```
CREATE [ OR REPLACE ] RULE name AS ON {SELECT | INSERT | UPDATE | DELETE }
    TO table [ WHERE condition ]
    DO [ ALSO | INSTEAD ]
    { NOTHING | command | ( command ; command ... ) }
```

Example Rule for Updates

```
CREATE TABLE shoelace_data (
    sl_name    text,          -- primary key
    sl_avail   integer,       -- available number of pairs
    sl_color   text,          -- shoelace color
    sl_len     real,          -- shoelace length
    sl_unit    text           -- length unit
);
```

```
CREATE TABLE shoelace_log (
    sl_name    text,          -- shoelace changed
    sl_avail   integer,       -- new available value
    log_who    text,          -- who did it
    log_when   timestamp      -- when
);
```

```
CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

```
SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who | log_when
---------+----------+---------+----------------------------------
 sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

Granting Access through Views

```
test=# create user loser;
CREATE ROLE
test=# CREATE VIEW safe_emp AS select empname, last_date from emp;
CREATE VIEW
test=# GRANT SELECT ON safe_emp TO loser;
GRANT
test=# \connect - loser
You are now connected to database "test" as user "loser".
test=> select * from emp;
ERROR:  permission denied for relation emp
test=> select * from safe_emp;
 empname |         last_date
---------+----------------------------
 john    | 2008-05-12 08:05:45.579625
(1 row)
```

Views and Rules Summary

- Relations (tables/views) that are used due to rules get checked against the privileges of the rule owner, not the user invoking the rule.
- One of the things that cannot be implemented by rules are some kinds of constraints, especially foreign keys.
- A trigger that is fired on INSERT on a view can do the same as a rule: put the data somewhere else and suppress the insert in the view. But it cannot do the same thing on UPDATE or DELETE, because there is no real data in the view relation that could be scanned, and thus the trigger would never get called.
- A rule may be more efficient than row-level triggers for bulk updates or deletes.
- A rule may be easier to write than a statement-level triggers, since the rule will rewrite complex queries for you.

References

- http://mysql.meetup.com/284/
- http://www.postgresql.org/docs/8.3/interactive/server-programming.html
  - SQL stored procedures (trusted)
  - PL/pgSQL (trusted)
  - PL/Tcl (trusted and untrusted)
  - PL/Perl (trusted and untrusted)
  - PL/Python (untrusted)
  - Triggers
  - Rules
- PL/Java
  - http://wiki.tada.se/display/pljava/Home
- Other stored procedure languages and 3rd party tools:
  - http://www.postgresql.org/download/