Introduction to Database Systems

Stored Procedures, Functions and Triggers in PostGreSQL

Stéphane Bressan

National University of Singapore

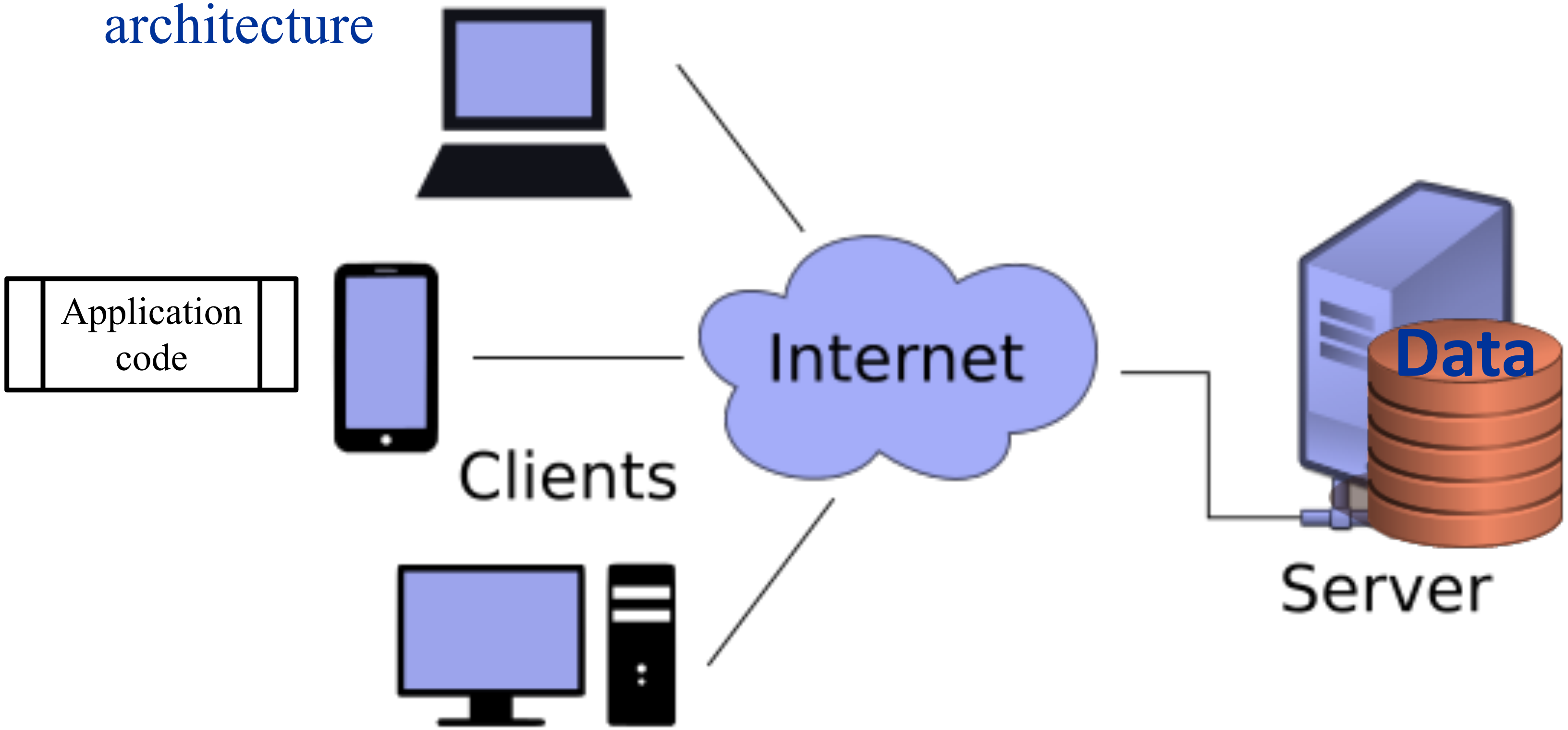
Disclaimer

The objective of this lecture is neither to offer a exhaustive presentation of the SQL standard for stored procedures and triggers or a comprehensive presentation of the underlying concepts, nor to serve as a complete tutorial for PostgreSQL stored procedures and functions in PL/pgSQL and triggers.

Rather, the objective of this lecture is to illustrate some selected important concepts and to give a handful of examples of PostgreSQL stored procedures and functions in PL/pgSQL and of PostgreSQL triggers that you can use and extend in your project.

Stored Procedures

Recall that most DBMS, including PostgreSQL, have a client/server architecture



Stored Procedures

The SQL 1999 standard proposes a language for stored procedures and triggers. PostgreSQL (and other DBMS) can store, share and execute code on the server. PL/pgSQL is PostgreSQL's language specifically designed to seamlessly embed SQL code and interact with the database SQL code. PL/pgSQL partially complies with the SQL standard. PL/pgSQL is similar to Oracle PL/SQL, SQLServer Transact-SQL and DB2 SQL Procedural Language and other languages implementing variants of the Persistent Stored Modules portion of the SQL standard.

Stored Procedures

Performance: Stored procedures are compiled. The code is cached and shared by all users. The code is executed on the server's side, usually a powerful machine. They incur generally fewer data transfer across the network. The optimization of the SQL code they contain is not adaptive. The clients' computing power is underutilized.

Productivity: Stored procedures are shared and reused under access control. The application logic that they encode is implemented and maintained in a single place. The code is not portable from one DBMS to the next. The languages and concepts are complicated.

Security: Data manipulation can be restricted to calling stored procedures under strict access control. It is easy to make tragic mistake while coding.

Example Function

`CREATE FUNCTION` creates a function. The function definition is between single quotes. The language is PLPGSQL (can also be C, Perl, Python or tcl).

```
CREATE OR REPLACE FUNCTION
gst(val NUMERIC) RETURNS
NUMERIC AS
'BEGIN
RETURN val * 1.07;
END;'
LANGUAGE PLPGSQL;
```

```
SELECT gst(1);
```

```
SELECT g.name,
gst(g.price)
FROM games g
WHERE gst(g.price) < 5;
```

Example Function

The function definition is between `$$` or `$<name>$` otherwise quotes in the function definition need to be escaped. Available types include: `NUMERIC, VARCHAR()` (SQL domains), `tablename%ROWTYPE, tablename.columnname%TYPE, RECORD`.

```
CREATE OR REPLACE FUNCTION
hello()
RETURNS CHAR(5) AS $$
BEGIN
RETURN 'Hello World';
END; $$
LANGUAGE PLPGSQL;

SELECT hello();

SELECT hello() FROM games;
```

Example Function

`RAISE NOTICE` prints on the database console.

```
DROP FUNCTION hello();

CREATE OR REPLACE FUNCTION
hello()
RETURNS BOOLEAN AS $$
BEGIN
RAISE NOTICE 'hello';
RETURN TRUE;
END; $$
LANGUAGE PLPGSQL;

SELECT hello() FROM games;
```

Example Function: Reading from the Database

SQL seamlessly integrates with PL/pgSQL.

```
CREATE TABLE gst (gst
NUMERIC);
INSERT INTO gst VALUES
(7);
```

```
CREATE OR REPLACE FUNCTION
gst(val NUMERIC)
RETURNS NUMERIC AS $$
DECLARE gst1 NUMERIC;
BEGIN
SELECT g.gst/100 INTO gst1
FROM gst g;
RETURN val * (1 + gst1);
END; $$
LANGUAGE PLPGSQL;

SELECT a.name, gst(g.price)
FROM games g;
```

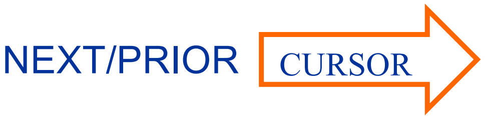
Example Function: Control Structures

```
IF condition
      THEN …
      ELSIF condition
      THEN …
      ELSE … END IF;
FOR somevariable
      IN (number …
number)
      LOOP …
      EXIT
      EXIT WHEN
      END LOOP;
```

```
CREATE OR REPLACE FUNCTION gst(val
NUMERIC)
RETURNS NUMERIC AS $$
DECLARE gst1 NUMERIC;
BEGIN
SELECT g.gst/100 INTO gst1 FROM
gst g;
IF val * (1 + gst1) > 5 THEN
RETURN val * 1 + gst1;
ELSE
RETURN 5;
END IF;
END; $$
LANGUAGE PLPGSQL;

SELECT g.name, gst(g.price)
FROM games g;
```

Cursors



NEXT/PRIOR

CURSOR

name	version	price
'Aerified'	'1.0'	12
'Aerified'	'1.1'	3.99
'Aerified'	'1.2'	1.99
'Aerified'	'2.0'	5
'Aerified'	'2.1'	12
'Aerified'	'3.0'	3.99
'Alpha'	'1.0'	12
'Alpha'	'1.1'	3.99
'Alpha'	'1.2'	12
'Alpha'	'2.0'	12
'Alpha'	'2.1'	2.99
'Alpha'	'3.0'	5
'Alphazap'	'1.1'	5
'Alphazap'	'1.2'	3.99
'Alphazap'	'2.0'	5
'Alphazap'	'2.1'	3.99
'Alphazap'	'3.0'	12
'Andalax'	'1.0'	12
'Andalax'	'1.1'	12
'Andalax'	'2.0'	12

Example Function: Cursor

Cursors are the scalable way to process data from a query

SCROLL, NO SCROLL, indicates whether the cursor can be scrolled backwards, or nor, respectively. A cursor can move in different directions and modes: NEXT, LAST, PRIOR, FIRST, ABSOLUTE, RELATIVE, FORWARD, BACKWARD. Cursors must be closed.

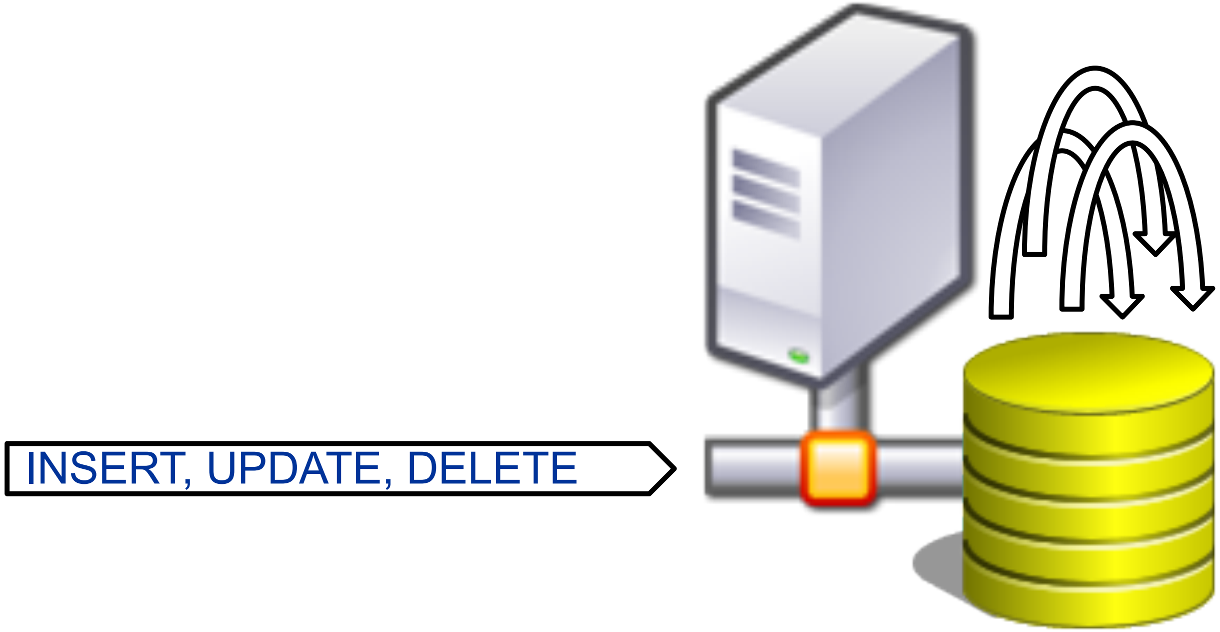
```
CREATE OR REPLACE FUNCTION avg1(appname VARCHAR(32))
RETURNS NUMERIC AS $$
DECLARE mycursor SCROLL CURSOR (vname VARCHAR(32))
FOR SELECT g.price FROM games g WHERE g.name=vname;
price NUMERIC; avgprice NUMERIC; count NUMERIC;
BEGIN
OPEN mycursor(vname:=appname);
avgprice:=0; count:=0; price:=0;
LOOP
           FETCH mycursor INTO price;
           EXIT WHEN NOT FOUND;
           avgprice:=avgprice + price;
           count:=count+1;
END LOOP;
CLOSE mycursor;
IF count<1 THEN RETURN null;
ELSE RETURN avgprice/count; END IF;
END; $$
LANGUAGE PLPGSQL;

SELECT avg1('Aerified');

SELECT name, avg1(name)
FROM games;
```

Triggers

Triggers program the reaction to events happening to the database.



SQL 1999 Triggers

A trigger is a procedure or function that is executed when a database event occurs on a table (e.g. `INSERT`, `DELETE`, `UPDATE`, `CREATE TABLE`, etc.). Triggers are used to maintain integrity, propagate updates and repair the database (they are a generalization of `ON UPDATE/DELETE`). Their syntax and semantics vary from one DBMS to the next.

Triggers

Performance: Triggers are compiled. The code is cached and shared by all users. The code is executed on the server's side, usually a powerful machine. They incur no data transfer across the network. The optimization of the SQL code they contain is not adaptive. The clients' computing power is not utilized.

Productivity: Triggers are applied to all interactions. The application logic they encode is implemented and maintained in a single place. The languages and concepts are complicated. The code is not portable from one DBMS to the next. Interactions among triggers (chain reactions) and between triggers, constraints and transactions are difficult to control.

Security: Data manipulation can be restricted and transformations automatically propagated. It is easy to make tragic mistake while coding.

PostgreSQL Trigger Syntax

```
CREATE TRIGGER name
{BEFORE|AFTER|INSTEAD OF}{event [OR event]*}
ON table
[FOR [EACH] {ROW|STATEMENT}]% statement by default
[WHEN condition] % for row only
EXECUTE PROCEDURE function()
```

Variables: NEW(for row only), OLD(for row only), TG_WHEN('BEFORE', 'AFTER'),

TG_OP ('INSERT', 'DELETE, 'UPDATE', 'TRUNCATE')

Example Trigger: For Each Statement

In PostgreSQL a trigger execute a function of type trigger. The example shows a trigger that sends a message to the database console before every insertion and update (for each statement) on the table apps. The three rows are inserted. The message 'hello' is displayed once.

```
DROP FUNCTION hello() CASCADE;

CREATE OR REPLACE FUNCTION hello()
RETURNS TRIGGER AS $$
BEGIN
RAISE NOTICE 'hello';
RETURN NULL;
END; $$
LANGUAGE PLPGSQL;

CREATE TRIGGER hello
BEFORE INSERT OR UPDATE
ON games
FOR EACH STATEMENT
EXECUTE PROCEDURE hello();

INSERT INTO apps VALUES ('A', '5.1', 100),
('B', '3.0', 101), ('C', '3.0', 102);

SELECT * FROM games g WHERE g.name='A' OR
g.name='B' OR g.name='C';
```

Example Trigger: For each Row

The example shows a trigger that sends a message to the database console for each row of the table apps affected by an insertion or update if the new price is more than 100. `RETURN NULL` prevents the insertion or update. The row `('AAA', '5.1', 100)` is inserted. The message `'hello'` is displayed twice and the other rows are not inserted.

```
DROP FUNCTION hello() CASCADE;

CREATE OR REPLACE FUNCTION hello()
RETURNS TRIGGER AS $$
BEGIN
RAISE NOTICE 'hello';
RETURN NULL;
END; $$ LANGUAGE PLPGSQL;

CREATE TRIGGER hello
BEFORE INSERT OR UPDATE
ON games
FOR EACH ROW
WHEN (NEW.price > 100)
EXECUTE PROCEDURE hello();

INSERT INTO games VALUES ('AA', '5.1',
100), ('BB', '3.0', 101), ('CC', '3.0',
102);

SELECT * FROM games g WHERE g.name='AA' OR
g.name='BB' OR g.name='CC';
```

Example Trigger: RETURN NEW

The insertion or update is done when the stored procedure returns `NEW`. The three rows are inserted. The message `'hello'` is displayed twice.

```
DROP FUNCTION hello() CASCADE;

CREATE OR REPLACE FUNCTION hello()
RETURNS TRIGGER AS $$
BEGIN
RAISE NOTICE 'hello';
RETURN NEW;
END; $$ LANGUAGE PLPGSQL;

CREATE TRIGGER hello
BEFORE INSERT OR UPDATE
ON games
FOR EACH ROW
WHEN (NEW.price > 100)
EXECUTE PROCEDURE hello();

INSERT INTO games VALUES ('AAA', '5.1',
100), ('BBB', '3.0', 101), ('CCC', '3.0',
102);

SELECT * FROM games WHERE g.name='AAA' OR
g.name='BBB' OR g.name='CCC';
```

Example Trigger: Logging Changes

```
CREATE TABLE glog (name VARCHAR(32)
NOT NULL, version CHAR(3)NOT NULL,
pricebefore NUMERIC, priceafter
NUMERIC NOT NULL, date DATE NOT
NULL);

CREATE OR REPLACE FUNCTION pricelog()
RETURNS TRIGGER AS $$
DECLARE delta NUMERIC;
DECLARE pb NUMERIC;
DECLARE now DATE;
BEGIN
now := now();
IF TG_OP ='INSERT'
THEN pb:=null; ELSE pb:=OLD.price;
END IF;
INSERT INTO glog VALUES (NEW.name,
NEW.version, pb, NEW.price, now);
RETURN NULL;
END; $$
LANGUAGE PLPGSQL;
```

```
CREATE TRIGGER pricelog
AFTER INSERT OR UPDATE
ON games
FOR EACH ROW
EXECUTE PROCEDURE pricelog();

INSERT INTO games VALUES ('AAAA',
'5.1', 100), ('BBBB', '3.0', 101),
('CCCC', '3.0', 102);

SELECT * FROM ganes WHERE name='AAAA'
OR name='BBBB' OR name='CCCC';

INSERT INTO games VALUES ('AAAA',
'5.1', 110);

SELECT * FROM games WHERE
name='AAAA';
```

Example Trigger: Logging Changes

Triggers interact! (in alphabetical order!). The `hello` trigger and the `hello()` function prevented the change. We the `hello` trigger and the `hello()` function. There is an integrity constraint violation.

```
DROP FUNCTION hello() CASCADE;

INSERT INTO apps VALUES ('AAAA',
'5.1', 110);

UPDATE games SET price=99 WHERE name
LIKE'C%';

SELECT * from glog;
```

A Note on Constraints and Transactions

Triggers interact with constraints but things are even more complicated when constraints are deferred. The foreign key constraint on table `test1` is not deferred; the foreign key constraint on table `test2` is deferred. The transaction on table `test1` is aborted. The transaction on `test2` succeeds and is committed.

```
CREATE TABLE test1 (
father NUMERIC primary key NOT DEFERRABLE,
son NUMERIC REFERENCES test1(father)NOT
DEFERRABLE);

CREATE TABLE test2 (
father NUMERIC primary key NOT DEFERRABLE,
son NUMERIC REFERENCES test2(father)DEFERRABLE
INITIALLY DEFERRED);

BEGIN;
INSERT INTO test1 VALUES (1,2);
INSERT INTO test1 VALUES (2,1);
END;

SELECT * FROM test1;

BEGIN;
INSERT INTO test2 VALUES (1,2);
INSERT INTO test2 VALUES (2,1);
END;

SELECT * FROM test2;
```

Credits

The content of this lecture is based on the book “Introduction to database Systems”
By
S. Bressan and B. Catania, McGraw Hill publisher

Images and clips used in this presentation are licensed from Microsoft Office Online Clipart and Media

For questions about the content of this course and about copyrights, please contact Stéphane Bressan

steph@nus.edu.sg