PostgreSQL Conference East 2009

Introduction to PL/pgSQL

Josh Williams

All you can do in a database...

- CREATE
- READ
- UPDATE
- DELETE
- ... Complete and utter CRUD.

Intro to PL/pgSQL

- All About PL/pgSQL
- Anatomy of a Language
- Interacting with the Inside World
- --Variable Data In and Variable Data Out
- If, Then, Elsif, Else, Etc, Etc, Etc, End If
- Stop Doing It Manually: Writing Triggers
- Even More Languages?!

About PL/pgSQL

- Close in structure to SQL
- Access to any custom data types, operators
- Write functions, Encapsulate complex logic,
  - Write triggers, Return sets, Create operators,
    - Custom aggregates, ~~Custom data types~~ ... not quite yet
- Not loaded by default, but...
  - createlang plpgsql database-name (outside)
  - or ... CREATE LANGUAGE plpgsql; (inside)

Anatomy of a Language

```
CREATE FUNCTION function_name
   (param1 int, param2 numeric)
   RETURNS text
   AS $dollarquote$
      << blocklabel >>
      ...;
   $dollarquote$
   LANGUAGE plpgsql
   STRICT IMMUTABLE SECURITY INVOKER;
```

CREATE (or replace) FUNCTION

- Optimizer hints:
  - Immutable: Same arguments return same result.
  - Stable: Can change across SQL statements.
  - Volatile: Can change between any two calls.
- Security options:
  - SECURITY INVOKER: Execute as calling user
  - SECURITY DEFINER: Execute as creating user

CREATE (or replace) FUNCTION

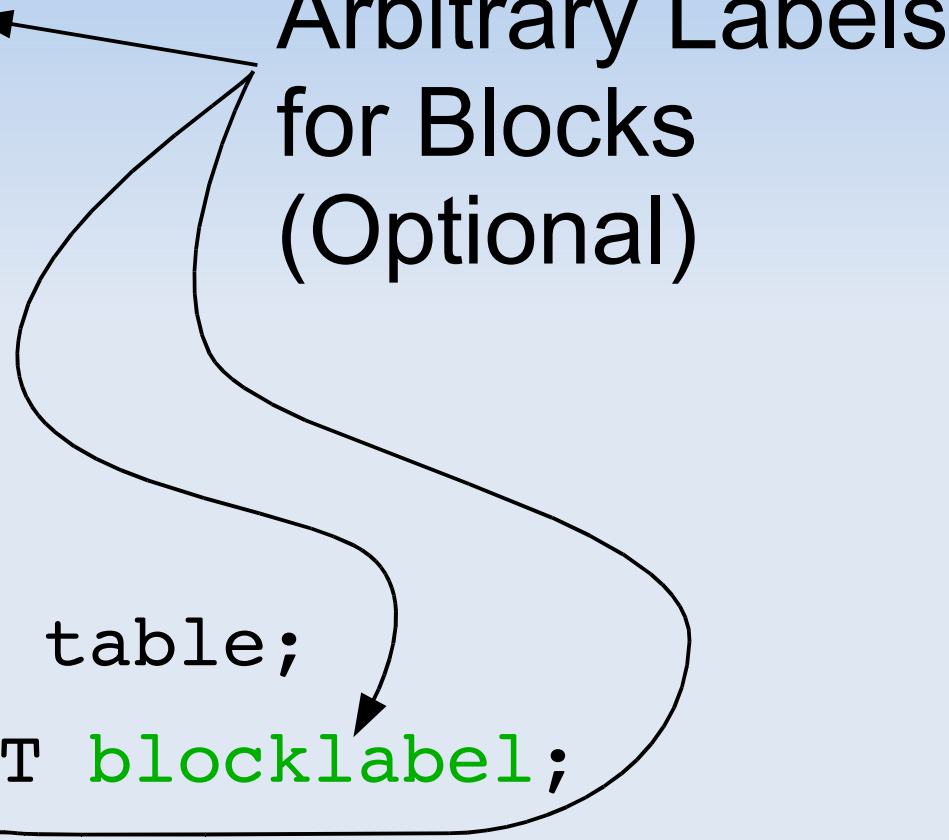
- Behavior options:
  - CALLED ON NULL INPUT
  - RETURNS NULL ON NULL INPUT (or STRICT)
- Lesser-used options:
  - COST (cpu_cost): Per row, default to 100
  - ROWS (returned_rows): Only for sets, default 1000

Anatomy of a Language

```
<< blocklabel >>
DECLARE
   variable text;
   declarations int;
BEGIN
   PERFORM col1 FROM table;
   IF FOUND THEN EXIT blocklabel;
END blocklabel;
```

Anatomy of a Language

```
<< blocklabel >>
DECLARE
   variable text;
   declarations int;
BEGIN
   PERFORM col1 FROM table;
   IF FOUND THEN EXIT blocklabel;
END blocklabel;
```



Anatomy of a Language

```
<< blocklabel >>
DECLARE
   variable text;
   declarations int;
BEGIN
   PERFORM col1 FROM table;
   IF FOUND THEN EXIT blocklabel;
END blocklabel;
```

Structure-Defining Statements

Anatomy of a Language

```
<< blocklabel >>
DECLARE
   variable text;
   declarations int;
BEGIN
   PERFORM col1 FROM table;
   IF FOUND THEN EXIT blocklabel;
END blocklabel;
```

Semicolon Positioning...

Anatomy of a Language

```
<< blocklabel >>
DECLARE
   variable text;
   declarations int;
BEGIN
   -- SQL-style Comments
   /* Block Comments */
END blocklabel;
```

Comment Styles

Anatomy of a Language

```
<< blocklabel >>
DECLARE
   variable text;
   declarations int;
   referenced-type table.column%TYPE;
   composite-type table-name;
   structure-less RECORD;
BEGIN
```

Variable Declaration (Optional)

Talking to the Database: Data In

- Assignment operator: `:=`
  - `variable[.field] := EXPRESSION`
  - Result of `SELECT EXPRESSION`
- `SELECT EXPRESSION INTO variable;`
  - Store the result of arbitrarily-complex queries.
  - Supports simple variables or record/row variables.
  - Can be used within curors (But that's another talk.)

Talking to the Database: Data In

- (8.2+) `INSERT` / `UPDATE` / `DELETE`
  - ... `RETURNING` *`EXPRESSION`* `INTO variable;`
  - Same syntax as `SELECT`.
- The `FOUND` variable, Boolean, set to true...
  - SELECT, PERFORM, FETCH... if row(s) produced.
  - UPDATE, INSERT, DELETE... if row(s) affected.
  - MOVE or FOR... if it does anything interesting.
- `GET DIAGNOSTICS var = ROW_COUNT;`

Talking to the Database: Data Out

- Of course `INSERT`, `UPDATE`, `DELETE`...
  - But be careful about variable naming:

```
CREATE FUNCTION newstat (acctnum int,
 status boolean) RETURNS boolean AS $$
  BEGIN
  UPDATE acct SET status = status
   WHERE acctnum = acctnum;
  RETURN status;
  END;
$$ VOLATILE LANGUAGE plpgsql;
```

Talking to the Database: Data Out

- Of course `INSERT`, `UPDATE`, `DELETE`...
  - Table and block-qualify column and variables:

```
CREATE FUNCTION newstat (acctnum int,
 status boolean) RETURNS boolean AS $$
  BEGIN
  UPDATE acct SET acct.status = newstat.status
   WHERE acct.acctnum = newstat.acctnum;
  RETURN status;
  END;
$$ VOLATILE LANGUAGE plpgsql;
```

Talking to the Database: Data Out

- Also `PERFORM`...
  - SELECT (without INTO) is not allowed.
  - PERFORM has the same syntax as SELECT...
    - `PERFORM col1, col2 FROM tbl WHERE active;`
  - Useful for calling functions (without return values)
    - `PERFORM newstat(1, true);`
    - `PERFORM newstat(acctnum, true) FROM acct;`
  - It also sets FOUND.

Talking to the Database: Data Out

- If PERFORM isn't versatile enough...
- `EXECUTE 'SQL';` -- any arbitrary command
- ... `INTO` *`variable`*
- No caching of plans (good for DDL, pre-8.3)
- No variable substitution either
  - Either build your SQL statement in a string variable
  - And/Or concatenate the command after EXECUTE

Talking to the Database: Data Out

```
CREATE FUNCTION sampletable (tablename TEXT)
RETURNS void AS $stfunc$
   DECLARE createcmd TEXT;
   BEGIN
   createcmd := 'CREATE TABLE ' || tablename
     || ' (id SERIAL, val TEXT)';
   EXECUTE createcmd;
   EXECUTE 'INSERT INTO '||tablename||' (val)
     VALUES ($$PostgreSQL Rocks!$$)';
   END;
$stfunc$ LANGUAGE plpgsql VOLATILE;
```

Talking to the Database: Data Out

```
testdb=# SELECT sampletable('tbl');
NOTICE:  CREATE TABLE will create implicity
  sequence ”tbl_id_seq” (etc, etc, etc...)
(1 row)
testdb=# SELECT * FROM tbl;
 id |          val
----+----------------------
  1 | PostgreSQL Rocks!
(1 row)
```

Talking to the Outside World

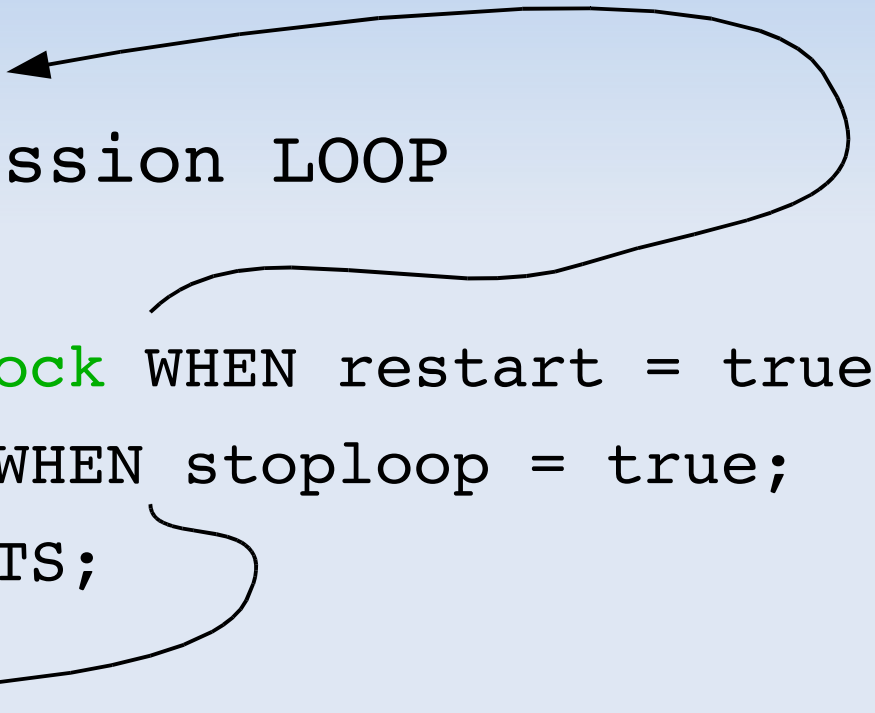
- Exiting function, returning a single value:
  - Scalar: `RETURN` *`expression`*`;`
  - Row/record type: `RETURN` *`record_variable`*`;`
- Set Returning Functions
  - Functions declared `RETURNS SETOF` *`type`*
  - `RETURN NEXT` *`variable_of_return_type`*`;`
  - `RETURN QUERY` *`SELECT`* `...;`
  - Follow up with an empty `RETURN` to end execution.

Logic Control Structures

- Industry Standard `IF` statement
  - `IF true-expression THEN`
    - `-- STATEMENTS;`
  - `ELSIF true-expression THEN`
    - `-- STATEMENTS;`
  - `ELSE`
    - `-- STATEMENTS;`
  - `END IF;`

Logic Control Structures

- Industry Standard looping structures: WHILE
  - `<< WhileBlock >>`
  - `WHILE true-expression LOOP`
    - `-- STATEMENTS;`
    - `CONTINUE WhileBlock WHEN restart = true;`
    - `EXIT WhileBlock WHEN stoploop = true;`
    - `-- MORE STATEMENTS;`
  - `END LOOP;`



Logic Control Structures

- Industry Standard looping structures: FOR
  - `<< ForBlock >>`
  - `FOR var IN low..high [BY step] LOOP`
    - `-- STATEMENTS;`
  - `END LOOP;`
  - Lower and upper bounds are evaluated once.
  - `step`: how much `var` is incremented each loop
  - `IN REVERSE` ... decrement instead of increment

Logic Control Structures

- Industry Standard looping structures: FOR
  - `<< ForBlock >>`
  - `FOR rowvar/scalarvar IN query LOOP`
    - `-- STATEMENTS;`
    - `RETURN NEXT OuterBlock.rowvar;`
  - `END LOOP;`
  - Query can be `SELECT`, DML with `RETURNING`, `EXECUTE`, `EXPLAIN`?!

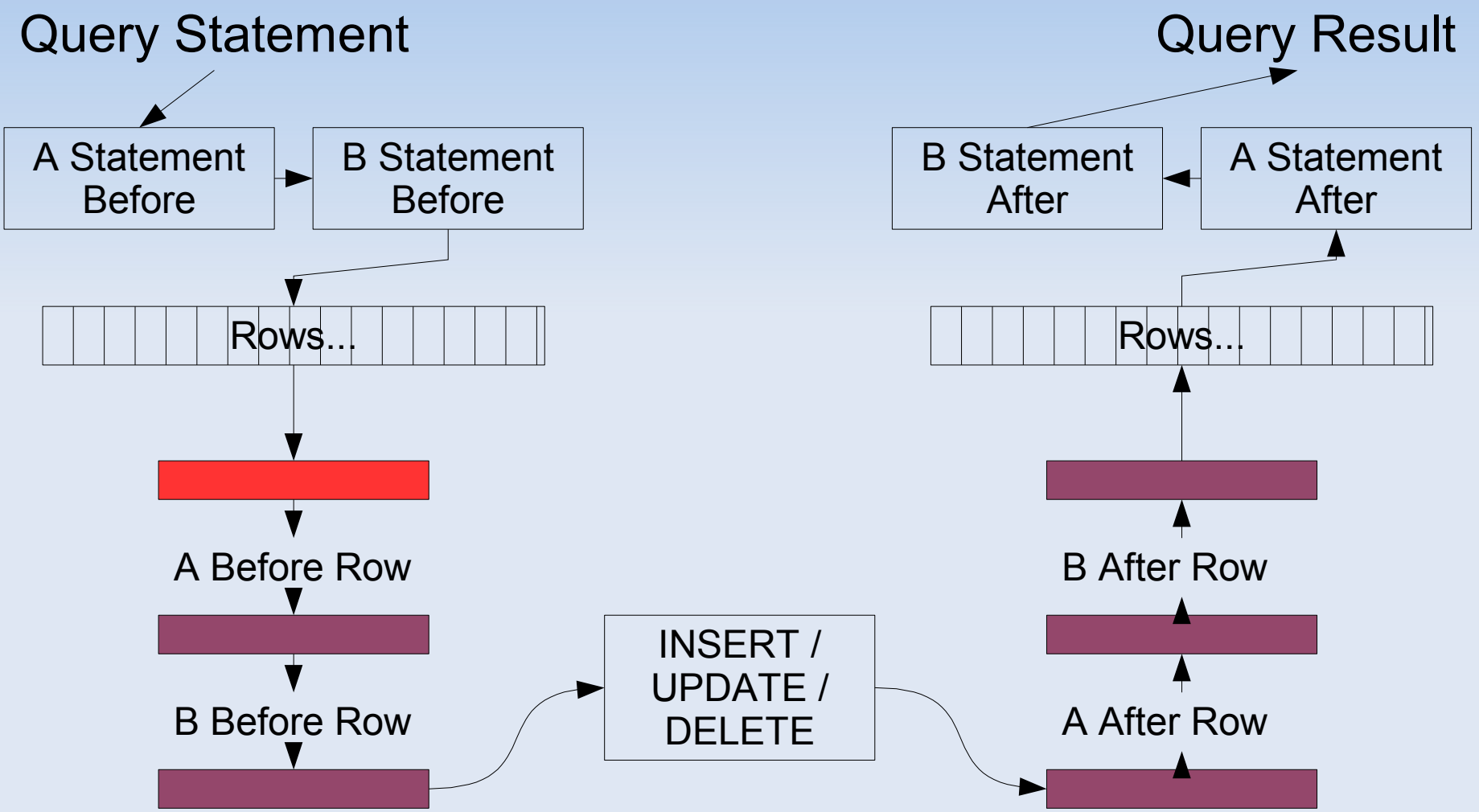
The Trigger Procedure

- Two steps to creating a trigger:

```
CREATE FUNCTION trig_fun() RETURNS trigger...
CREATE TRIGGER...EXECUTE PROCEDURE trig_fun();
```

- Trigger options...
  - BEFORE or AFTER
  - Execute on INSERT, UPDATE, **and/or** DELETE
  - FOR EACH STATEMENT / ROW
  - String literal arguments passed to function

The Trigger Procedure



The Trigger Function

- All information is provided through variables
- Operation-related variables:
  - NEW and OLD – Record type variables (row-level)
  - TG_OP – String containing operation performed
    - ”INSERT”, ”UPDATE”, ”DELETE”
- Definition-related variables:
  - TG_NAME, TG_WHEN, TG_LEVEL
  - TG_RELNAME
  - TG_NARGS, TG_ARGV[]

The Trigger Function

- The RETURN value...
- Statement-level, or row-level after: Return `NULL`
- For `BEFORE`, `FOR EACH ROW` triggers,
  - Return NULL to abort operations on current row
  - Otherwise return a record-type matching NEW (or OLD for `ON DELETE` triggers)
- The returned row can be modified (or unmodified) NEW (or old), or a freshly-built row

Example: Generic Audit Trigger

```
=# CREATE TABLE changes (
   id SERIAL PRIMARY KEY,
   ts timestamp NOT NULL DEFAULT now(),
   tbl regclass NOT NULL,
   op TEXT NOT NULL,
   role TEXT NOT NULL
);
CREATE TABLE
```

Example: Generic Audit Trigger

```
CREATE FUNCTION tbl_audit() RETURNS
  trigger AS $$
   BEGIN
   -- Record who made what change to table
   INSERT INTO changes (tbl, op, role)
   VALUES (TG_RELID, TG_OP, SESSION_USER);
   RETURN NULL; -- Ignored
   END;
$$ LANGUAGE plpgsql;
```

The Trigger Definition

```
=# CREATE TRIGGER users_audit AFTER
  INSERT OR UPDATE OR DELETE ON users
  FOR EACH STATEMENT EXECUTE
  PROCEDURE tbl_audit();
```

```
CREATE TRIGGER
```

The Trigger Definition

```
=# UPDATE users SET active = false;
UPDATE 10
=# SELECT * FROM changes;
 id |         ts          |  tbl  |   op   | role
----+---------------------+-------+--------+------
  1 | 2009-02-31 01:23:45 | users | UPDATE | josh
(1 row)
```

Beyond PL/pgSQL

- C
- Distribution included languages
  - PL/Tcl
  - PL/Perl (PL/PerlU)
  - PL/Python
- Contributed languages
  - PL/sh, PL/Ruby, PL/R, PL/PHP, PL/Java
  - PL/LOLCODE

Beyond PL/pgSQL

```
CREATE OR REPLACE FUNCTION getblog(text)
RETURNS SETOF blog LANGUAGE plphp AS $$
   $blog = simplexml_load_file($args[0]);
   $r[0] = (string) $blog->channel[0]->title;
   foreach($blog->$channel[0]->item as $entry) {
      $r[1] = (string) $entry->title;
      $r[2] = (string) $entry->pubDate;
      $r[3] = (string) $entry->link;
      return_next($r);
   } $$;
```

(http://www.xzilla.net/blog/2007/Feb/The-web-IN-your-database.html)

Isn't This Where We Came In

- All About PL/pgSQL
- Anatomy of a Language
- Interacting with the Inside World
- --Variable Data In and Variable Data Out
- If, Then, Elsif, Else, Etc, Etc, Etc, End If
- Stop Doing It Manually: Writing Triggers
- Even More Languages?!

Further Reading

- Cursors
- Exceptions, Error Handling
- IN/OUT parameters
- Variable Scope & Qualification

Intro to PL/pgSQL

Slides...

http://dbahumor.com/plpgsql