2ndQuadrant

Professional PostgreSQL

# PL/Python – Python inside the PostgreSQL RDBMS


## Peter Geoghegan
## (Rhymes with “Ronald Reagan”)

peter@2ndQuadrant.com

http://www.2ndQuadrant.com/


EUROPYTHON 2011

FLORENCE, JUNE 20–26

PostgreSQL

# Licence

- Creative Commons:
  - Attribution-Non-Commercial-Share Alike 3.0
  - You are free:
    - to Share — to copy, distribute and transmit the work
    - to Remix — to adapt the work
  - Under the following conditions:
    - Attribution: You must attribute the work in the manner specified by the author or licensor
    - Non-Commercial: You may not use this work for commercial purposes
    - Share Alike: If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one

Source: http://creativecommons.org/licenses/by-nc-sa/3.0/

# About me

- PostgreSQL hacker
- Work for 2ndQuadrant as a PostgreSQL consultant
- Python enthusiast. Most of our internal tools, and many of our opensource tools such as pgtune are written in Python.

# What is a PL?

A language for creating user-defined functions in the database, that are called directly from SQL queries.

# In the context of a RDBMS, what is a function?

- Traditionally, a thin wrapper on a snippet of SQL that the parser may inline:

```
-- Create a function from the SQL command line
-- (or, potentially, from within a regular Python app)
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    STRICT;

-- Now, to call our newly defined function:
SELECT add(2, 2) AS result;
 result
--------
      4
(1 row)
```

# More advanced examples

```
--Aggregates:

SELECT count(*) from europython_attendees WHERE id IN
(SELECT attendees_id FROM attending_talks WHERE talk =
'PL/Python – Python inside the PostgreSQL RDBMS');

--General functions:

SELECT add_user('Peter Geoghegan',
'peter@2ndquadrant.com');

SELECT upper(name) from users;
```

# ...and most interestingly

Functions that are:

- External to the database
- Do not operate within the restricted execution environment of the database.
- Things like writing to sockets and files (with the permissions of the postgres operating system user) are possible:

```
SELECT send_remote_confirmation();

SELECT order_food('Eggs, Bacon & spam', 'greasy spoon');
```

# More about functions

**Functions can be written in:**

- C
- SQL
- A procedural language (PL)

**Dozens of PLs, some niche:**

- PL/R
- PL/OpenCL
- PL/LOLCODE

**Some mainstream:**

- PL/PgSQL
- PL/Python

# Topical stuff

- In last year's 9.0 release, great strides were made in advancing PL/Perl as a procedural language.
- In the upcoming 9.1 release, due out in about September, it's PL/Python's turn.

# Overview of PL/Python

- Supports Python 2 and Python 3
- Acts as a seemless wrapper between the SQL interpreter and Python, mapping datatypes and concepts as appropriate.

```
# Conceptually:
pl_python_mapping = {'null':None,'integer':int,'anyarray':list}
# and so on
```

# Simple example: Trigger

- Sales table represents each sale
- Line items table represents line items sold that comprise those sales
- We want to maintain stock quantity of line items accurately and across applications.

```
-- represents individual line items for each sale.
CREATE TABLE line_items_sold (
    line_items_sold_id serial PRIMARY KEY,
    sales_id integer NOT NULL REFERENCES sales,
    line_items_id integer NOT NULL REFERENCES line_items,
    qty integer NOT NULL
);
```

# Trigger example continued

```
CREATE OR REPLACE FUNCTION line_items_sold_stock_qty() RETURNS TRIGGER
AS $$
        # Work out the increment/decrement amount(s).
        if TD["event"] == 'DELETE':
            delta_qty = TD["old"].qty * -1;
            line_items_id = TD["old"].line_items_id
        elif TD["event"] == 'UPDATE':
            delta_qty = TD["new"].qty - TD["old"].qty;
            line_items_id = TD["old"].line_items_id
        else: # insert
            delta_qty = TD["new"].qty;
            line_items_id = TD["new"].line_items_id

        # Update "line_items" so that the stock qty is accurate

         plan = plpy.prepare(
    """UPDATE line_items SET stock_qty = stock_qty + $1 WHERE line_items_id = $2""", ["integer", "integer"])
        plpy.execute(plan, [delta_qty, line_items_id])
$$ LANGUAGE plpythonu;
```

# Finally, define trigger itself

```
CREATE TRIGGER line_items_sold_changed
AFTER INSERT OR UPDATE OR DELETE ON line_items_sold
    FOR EACH ROW EXECUTE PROCEDURE
line_items_sold_stock_qty();
```

# Finished!

Now, stock levels are maintained regardless of how, when or where the line items table is modified. We've baked a business rule into the database.

# Enforcing datatype level constraints

# Popular barcode formats

- UPC
- EAN-8
- EAN-13
- GTIN-14

All of these formats are valid GTINs. In each case, the last digit is a checkdigit, that verifies the integrity of the barcode, based on a simple formula.

# Here's how we do it in PL/Python

```
CREATE OR REPLACE FUNCTION is_gtin(barcode bigint)
RETURNS BOOLEAN
IMMUTABLE STRICT
LANGUAGE plpythonu
AS $$
    if barcode is None:
        # Function is strict, so technically
        # this isn't necessary
        return None

    chars = str(barcode)
    total = 0
    for i, c in enumerate(chars):
        total += int(c) if i % 2 == 0 else int(c) * 3
    return total % 10 == 0
$$;
```

# Domain with check constraint

```
CREATE DOMAIN gtin AS BIGINT
    CHECK ( is_gtin(VALUE) );

CREATE TABLE barcodes
(
  barcode gtin primary key,
  products_id integer NOT NULL REFERENCES products
);
```

# Which we'll now try and violate

```
INSERT INTO barcodes
(
  barcode,
  products_id
)
VALUES
(
  5060193210737, --Transposition error. Actually “...773”
  17
);
ERROR:  value for domain gtin violates check constraint
"gtin_check"
```

# In practice...

- Functions initially created with SQL script that contains a string of python
- Python code actually resides on database server
  - There is no .py file
- More complex systems often just have simple forwarding functions in PL/Python, that import a custom module call the real functions
- That way, we avail of existing tools
- Unit testing with simple mock objects that emulate pl/python stuff

# Why not use an ORM/Client side code?

- No approach fundamentally better.
- When low-level business rules need to be enforced, you can't really beat doing it in the database.
- To each his/her own.
- ORMs can work well when they're good, and when the DB isn't treated as a total black box.
- When you treat the DB + ORM as a black box, you risk making performance tank.
- Combine both approaches!

# Words of wisdom

“Let the database be good at what it's good at, including smart processing of bulk data...Avoid shipping data to an external client program just to process and ship stuff back to the database. Only fetch data from the database if you need it to display or send to another system.”

- Andrew Dunstan's dictum (PostgreSQL major contributor).

# It may interest you to know...

- Python is embeddable
- Postgres is extensible
- All of this is brought to you by plpython.c
- Less than 5000 lines of C (read: hardly any)


The
sandboxing
issue

# So I lied...

- Well, fibbed
- There is no such thing as PL/Python
- Well, there was, but not anymore
- Problems with restricted execution environment
- PL/Python is today more correctly referred to as PL/PythonU
- The U stands for untrusted
- Can't be imprisoned in DB's restricted execution environment
- Inherently hard problem though
- General trend towards richer languages being hard to sandbox

# More topical stuff – features in 9.1

- Table function support
- Syntax checking at function creation time
- Traceback information
  - Yes, we should have had that a long time ago
- Fix exception handling with Python 3
  - Exception classes are now available

# New features (continued)

- Explicit substransactions
- Implements Python context manager

```
CREATE FUNCTION make_fund_transfer() RETURNS void AS $$
try:
    with plpy.subtransaction():
        plpy.execute("UPDATE acc SET bal = bal - 100 WHERE name = 'Peter'")
        plpy.execute("UPDATE acc SET bal = bal + 100 WHERE name = 'Marco'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

# The database as application server

## Case study – Lustre POS

- POS application – traditional business app
- Lots of clients, including fat client, embedded handset and embedded POS terminals, all based on native code.
- Installed at lots of remote sites – simple, low-maintenance infrastructure required.

# Wholesaler interface

- App had to interface with multiple third party wholesalers
- CSV-ish flat text file format over FTP
- Horrible, ill-specified
- Orders, product updates, special offers, sales figures
- Found a new way to break every single week
- Had to be able to run from every client, including scheduler
- PostgreSQL was already available to us
- They don't care about data integrity, but I do!

# Questions?