PostgreSQL / Python Lecture Notes

In this lecture we will continue to look at ways to store complex data in PostgreSQL, create indexes on that data and then use the data. We will look at how we connect to a PostgreSQL database from within Python.

Additional Materials

- Sample SQL commands for this lecture
- URL for these notes: https://www.pg4e.com/lectures/06-Python

Connecting Python and PostgreSQL 📖 🏠 ⏩

While we can do a lot of data importing, processing and exporting, some problems are most simply solved by writing a bit of Python. Python can communicate with databases very naturally. In a sense, you use all the SQL skills that you have been learning but construct the SQL statements as strings in Python and send them to your PostgreSQL server.

It is important to note that Python is just another client like psql or pgadmin. It makes a network connection to the database server using login credentials and sends SQL commands and receives results from the server.

In order to connect to

```
$ python3
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
>>> import psycopg2
>>>
```

If the **import** fails you need to install the library using a command like:

```
pip install psycopg2      # or pip3
```

Pip is only one of many ways to manage Python "dependencies" / "add ons" depending on your operating system / virtual environment / python installation pattern.

References

- Python-PostgreSQL Database Adapter

SQL Commands in Python ⏪ 📖 🏠 ⏩

In this section we will be talking about simple.py and hidden-dist.py.

The sequence in Python to connect and log in to a PostgreSQL database is:

```
import psycopg2

conn = psycopg2.connect(
    host='35.123.23.37', database='pg4e',
    user='pg4e_user_42', password='pg4e_pass_42',
    connect_timeout=3)
```

The connection makes ure that your account and password are correct and there is truly a PostgreSQL server running on the specified host. If you notice in the code for simple.py, we store the actual secrets in a file called **hidden.py** and import them. You make your **hidden.py** file by copying hidden-dist.py and putting in your host, user, password, and database values.

Normally, we don't actually send SQL commands using the **connection**. For that we generally get a **cursor**. The cursor allows us to send an SQL command and then retrieve the results, perhaps in a loop. We can ue the cursor over and over in our program. You can think of the cursor as the equivalent of the "psql" command prompt but inside your Python program. You can have more than one cursor open at a time.

```
cur = conn.cursor()
cur.execute('DROP TABLE IF EXISTS pythonfun CASCADE;')

...
cur.execute('SELECT id, line FROM pythonfun WHERE id=5;')
row = cur.fetchone()
print('Found', row)
```

In the above example **cur** is just a commonly used variable name for a database cursor.

In the above example, when you send a **SELECT** using **cur.execute()** it does not retrieve the data. It primes the cursor to retrieve the data using methods like **fetchone()**.

References

- PEP 249 -- Python Database API Specification v2.0

Sample Code: Loading The Text of a Book ⏪ 📖 🏠 ⏩

In this sample code walkthrough, we will download the text of a book from [Project Gutenberg] and parse the contents of the book and put it into a PostgreSQL database and set up a full-text GIN index on the book text.

Download these files

- https://www.pg4e.com/code/loadbook.py
- https://www.pg4e.com/code/myutils.py

Make sure the **hidden.py** is set up as above and has your credentials.

Download a book from the Gutenberg project using **wget** or **curl**:

```
wget http://www.gutenberg.org/cache/epub/19337/pg19337.txt
```

Then run the loadbook code:

```
python3 loadbook.py
```

The program makes a document database, reads through the book text, breaking it into "paragraphs" and inserting each paragraph into a row of the database. The table is automatically *created by the code* and named the same as the book file so you can have more than one book in your database at one time.

```
CREATE TABLE pg19337 (id SERIAL, body TEXT);
```

You can watch the progress of the load using **psql** in another window:

```
pg4e=> select count(*) from pg19337;
 count
-------
    50
(1 row)

pg4e=> select count(*) from pg19337;
 count
-------
   150
(1 row)
```

You will notice that it always is a multiple of 50 until the load finishes because we are flushing the connection using a **conn.commit()** every 50 inserts.

Once the load is complete, you will create the **GIN** index and play with some queries:

```
CREATE INDEX pg19337_gin ON pg19337 USING gin(to_tsvector('english', body));

EXPLAIN ANALYZE SELECT body FROM pg19337  WHERE to_tsquery('english', 'goose') @@ to_tsvector('english', body);

                                                     QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on pg19337  (cost=12.03..24.46 rows=4 width=225) (actual time=0.027..0.029 rows=6 loops=1)
   Recheck Cond: ('''goos'''::tsquery @@ to_tsvector('english'::regconfig, body))
   Heap Blocks: exact=1
   ->  Bitmap Index Scan on pg19337_gin  (cost=0.00..12.03 rows=4 width=0) (actual time=0.016..0.016 rows=6 loops=1)
         Index Cond: ('''goos'''::tsquery @@ to_tsvector('english'::regconfig, body))
 Planning Time: 0.523 ms
 Execution Time: 0.070 ms
```

Sample Code: Loading Email Data ⏪ 📖 🏠 ⏩

In this example, we download some historical email data and do some parsing and cleanup of the data and insert it into a table. Then we use regular expressions to make an index that looks deep into the text field to allow indexed searches on data within the field.

This example shows some of the real-world challenges you will find when you have a historical data source that is not "perfect" in its formatting or approach and you must make use of the data regardless of its lack of consistency. As you look at it you quickly can see that Python was the only way to clean up this data.

Download these files

- https://www.pg4e.com/code/gmane.py
- https://www.pg4e.com/code/myutils.py
- https://www.pg4e.com/code/datecompat.py

Make sure the **hidden.py** is set up as above and has your credentials. The **datecompat.py** is needed because certain needed date parsing / conversion routines are only available in later versions of Python and is there to allow the code to run across many versions of Python.

The code creates a table to store the email messages:

```
CREATE TABLE IF NOT EXISTS messages
    (id SERIAL, email TEXT, sent_at TIMESTAMPTZ,
     subject TEXT, headers TEXT, body TEXT)
```

And then retrieves email messages from a copy of a message archive at http://mbox.dr-chuck.net/sakai.devel/. It turns out that email data is particularly wonky because so many different products send, receive, and process email and they treat certain fields ever so slightly differently.

The mail messages are in a format called Mbox. This format is a flat file where each message starts with a line "From ," followed by a set of headers, followed by one blank line, followed by the actual message text.

```
http://mbox.dr-chuck.net/sakai.devel/4/6

From news@gmane.org Tue Mar 04 03:33:20 2003
From: "Feldstein, Michael" <Michael.Feldstein@suny.edu>
Subject: RE: LMS/VLE rants/comments
Date: Fri, 09 Dec 2005 09:43:12 -0500

Yup, I think this is spot-on. Either/or, in reality, is actually neither
and both. We've had many discussions internally at SUNY about just how
loose our coupling can be. You start ...

From news@gmane.org Tue Mar 04 03:33:20 2003
From: John Norman <john@caret.cam.ac.uk>
Subject: RE: LMS/VLE rants/comments
Date: Fri, 9 Dec 2005 13:32:29 -0000

I should chip in here as Dan's PHB :)

Our strategy at Cambridge is to try and get the best of both worlds. I am
dismayed by the either/or tone of many discussions...
```

You can look through the output and see both how simple and how complex the mail messages can be. And how much fun we were having in the Sakai Project developing an Open Source Learning Management System in 2005.

This application is retrieving data across a slow network, talking to possibly overloaded servers with a possibility of Rate Limits on those servers. So we use a strategy similar to Web Crawlers where we make a restartable process that can be aborted part-way through and then next time the application runs, it picks up where it left off.

Like a web crawler, our goal is to make a complete copy of the data in our fast database, and clean it up and so we can repeatedly do very fast data analysis on our copy of the data.

This is a complex bit of sample code and took more than a week of trial and error to develop, so as you look at this code, don't feel like somehow every trick and technique to clean the data, recover form errors or build a restartable process is something you just write from scratch and it works perfectly the first time.

Code like this evolves based on your data analysis needs and the vagaries of your data and data source. You start to build code, run it and when it fails, you adapt and improvise. Eventually you transform the raw data into a pretty form in the database so the rest of your analysis works smoothly.

Ranking The Results ⏪ 📖 🏠

The key benefit of a GIN / Natural Language index is to speed up the look up and retrieval of the rows selected in the **WHERE** clause. When we have a set of rows, what we do with those rows is relatively inexpensive.

Ranking of the "how well" a row (ts_vector) matches the query (ts_query) is something we compute directly from the data in the **ts_query** and the **ts_vector** in each row. We can use different fields in the ranking computation than the fields we use in the **WHERE** clause. The **WHERE** clause dominates the cost of a query as it decides how to gather the matching rows.

```
SELECT id, subject, sender,
  ts_rank(to_tsvector('english', body), to_tsquery('english', 'personal & learning')) as ts_rank
FROM messages
WHERE to_tsquery('english', 'personal & learning') @@ to_tsvector('english', body)
ORDER BY ts_rank DESC;

 id |          subject           |           sender           | ts_rank
----+----------------------------+----------------------------+----------
  4 | re: lms/vle rants/comments | Michael.Feldstein@suny.edu | 0.282352
  5 | re: lms/vle rants/comments | john@caret.cam.ac.uk       |  0.09149
  7 | re: lms/vle rants/comments | john@caret.cam.ac.uk       |  0.09149

SELECT id, subject, sender,
  ts_rank_cd(to_tsvector('english', body), to_tsquery('english', 'personal & learning')) as ts_rank
FROM messages
WHERE to_tsquery('english', 'personal & learning') @@ to_tsvector('english', body)
ORDER BY ts_rank DESC;

 id |          subject           |           sender           |  ts_rank
----+----------------------------+----------------------------+-----------
  4 | re: lms/vle rants/comments | Michael.Feldstein@suny.edu |  0.130951
  5 | re: lms/vle rants/comments | john@caret.cam.ac.uk       | 0.0218605
  7 | re: lms/vle rants/comments | john@caret.cam.ac.uk       | 0.0218605
```

There are two at least two ranking functions **ts_rank** and **ts_rank_cd**. There is also the ability to weight different elements of a **ts_query** that influence how the relative ranking is computed.
