DATABASE

Martin Brugnara

2{0,1}/11/2019

mb@disi.unitn.eu

wiki.postgresql.org/wiki/Logo

TODAY

1. Interacting with a DMBS
2. Non trivial features of PG
3. Database course Q&A

THE PSQL UTILITY

Command

```
psql -h <server> -U <user> <database>
```

Remote server

```
psql -h sci-didattica.unitn.it -U db_001 db_001
```

Local server

```
psql db_001
```

INSTALL

Client only

- Debian/Ubuntu: `postgresql-client`
- Brew: `libpq`

Server & Client

- Debian: `postgresql`
- Brew: `postgresql`
- macOS: `Potgres.app`

GUI

- web app: pgAdmin 4

VPN

The server is reachable only from the local network

`unitn`, `unitn-x`, (local) `eduroam`, `cable`

wiki.unitn.it/pub:conf-vpn

ACCOUNTS

gspreadsheets: 1627VkH7F01MueeWfsJ5j9V_gqCDs6SSxYeEyCmbzRf0

Structure

- Database: `db_ddd`
- Username: `db_ddd`
- Password: `secret_ddd`

Where `ddd` in [001, 250].

Change your password

```
# Connect via psql
PGPASSWORD=secret_001 psql \
    -h sci-didattica.unitn.it -U db_001 db_001
# Change password via query
ALTER USER "db_001" WITH PASSWORD 'newpass';
```

FAQ 1/2

Quotation marks

`""` : names of schemas, tables, columns, *etc.*

`''` : string constant.

Cast

```
-- SQL standard
CAST ( expression AS type )

-- Old style PG
expression::type
```

Auto increment

`SERIAL` – see documentation.

FAQ 2/2

Case Sensitivity

Tables, columns names: yes with ""

. . . without, transformed to lowercase.

String: case sensitive.

Naming things - doc.

Database Cluster (Server - User & Groups)

$\rightarrow$ Databases (Name Space) $\rightarrow$ Schemas $\rightarrow$ Tables.

JDBC & psycopg2

JDBC - CONNECT

```
/* JDBCExample.java */
import java.sql.*;

/* class JDBCExample > static void main () */
String url = "jdbc:postgresql://sci-didattica.unitn.it/db_001";
Properties props = new Properties();
props.setProperty("user","db_001");
props.setProperty("password","secret_001");
props.setProperty("ssl","false");
Connection conn = DriverManager.getConnection(url, props);
/* Do stuff. */
conn.close();

# Download jdbc.postgresql.org/download/postgresql-42.2.8.jar
java -cp "$(pwd)/postgresql-42.2.8.jar;$(pwd)/" JDBCExample
```

https://jdbc.postgresql.org/documentation/head

JDBC - QUERY

```
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM mytable WHERE column
while (rs.next())
{
    System.out.print("Column 1 returned ");
    System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

https://jdbc.postgresql.org/documentation/head/query.html

JDBC - PREPARED STATEMENT

```
String q = "INSERT INTO tbl (foo) VALUES (?)";
PreparedStatement st = conn.prepareStatement(q);
st.setObject(1, new_value);
st.executeUpdate();
st.close();
```

jdbc.postgresql.org/documentation/head/update.html
jdbc.postgresql.org/documentation/head/java8-date-time.html

PYTHON

```python
import psycopg2                           # initd.org/psycopg/

# Connect
conn = psycopg2.connect("dbname='db_001' user='db_001' " +
    "host='sci-didattica.unitn.it' password='secret_001'")
cur = conn.cursor()

# Create / Prepared statement
cur.execute("CREATE TABLE test ...")
cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)",
    (100, "abc'def"))
conn.commit()                              # Make changes permanent

# Query
cur.execute("SELECT * FROM test;")
cur.fetchone()                             # next row only
cur.fetchall()                             # all remaining ones

# Clean up
cur.close()
conn.close()
```

Nice types

RANGE

Types

numrange, tstzrange, daterange

Syntax

```
'(7,12]'::numrange
numrange(7, 12, '(]')
```

RANGE

Types

numrange, tstzrange, daterange

Syntax

```
'(7,12]'::numrange
numrange(7, 12, '(]')
```

Operators

`www.postgresql.org/docs/current/functions-range.html#`

`RANGE-OPERATORS-TABLE`

`@>`, `&&`, `>>` `&>`, `-|-`, `+`, `*`, `-`

RANGE

Types

numrange, tstzrange, daterange

Syntax

```
'(7,12]'::numrange
numrange(7, 12, '(]')
```

Operators

`www.postgresql.org/docs/current/functions-range.html#`

`RANGE-OPERATORS-TABLE`

`@>`, `&&`, `>>` `&>`, `-|-`, `+`, `*`, `-`

Functions

`www.postgresql.org/docs/current/functions-range.html#`

`RANGE-FUNCTIONS-TABLE`

lower, upper, isempty, lower_inc, lower_inf, range_merge.

ARRAYS

Declaration

```
field1   type[],
field2   type[][],
field3   type[n][m][k],
...
```

Syntax

```
'{1,2,3,4}'
'{{1,2,3},{4,5,6},{7,8,9}}'
ARRAY[1,2,3,4]
```

Navigation

```
field1[0],  field2[2:4][:3]
```

ARRAYS

Editing

```
field1[0] = 33
field2[1][:3] = '{2,3,4}'
arr_1 || arr_2
arr_1 || val || NULL
array_cat()
```

Search

```
ANY, ALL, &&,
array_position, array_positions
```

www.postgresql.org/docs/current/arrays.html

JSON

JSON & JSONb

Navigation

```
->      # descend one node, returns JSON node
->>     # descend one node, returns value/JSON as text
#>      # query path, returns JSON node
#>>     # query path, returns value/JSON as text
```

JSON

JSON & JSONb

Filter / Modify JSONb only.

```
Containment: <@, ?, ?|
Concat: ||
Delete: -, #-
Path query (& compare): @?, @@
```

HSTORE

Text only, dictionary.

Mostly replaced by JSONb.

Cool operators and functions

WINDOW FUNCTIONS

Like aggregation, but does not collapse rows.

```
func() [FILTER ( WHERE filter_clause )] OVER (window_def)
```

Additional functions [1]

`www.postgresql.org/docs/12/functions-window.html`

- row_number()
- lag, lead (expression, offset, default)
- first_value, last_value, (expression)
- nth_value (expression, nth)

WINDOW FUNCTIONS

Like aggregation, but does not collapse rows.

```
func() [FILTER ( WHERE filter_clause )] OVER (window_def)
```

Window definition [2]

- PARTITION BY
- ORDER BY
- frame_clause*

www.postgresql.org/docs/12/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS

WINDOW FUNCTIONS

Like aggregation, but does not collapse rows.

```
func() [FILTER ( WHERE filter_clause )] OVER (window_def)
```

Usage examples

"First *n* items for each group", "running sums", "stats", ...

Buy at least 4 and the 2 less expensive are free.

```
SELECT customer_id, SUM(price) FROM (
  SELECT customer_id, price,
    ROW_NUMBER() as r
      OVER (PARTITION BY customer_id ORDER BY price),
    COUNT() as m OVER (PARTITION BY customer_id),
) tmp WHERE m < 4 OR (m >= 4 AND r > 2)
```

SET RETURNING FUNCTIONS

Series Generating Functions

```
generate_series(start, stop, step)
```

```
generate_subscripts(array, dim, reverse)
```

www.postgresql.org/docs/current/functions-srf.html

Indexes

INDEXES OVERVIEW

- B-tree (default)
- Hash
- GiST
- SP-GiST
- GIN
- BRIN

`www.postgresql.org/docs/current/indexes-types.html`

B-TREE

Speeds up

- $<, <=, =, >=, >$

  $\rightarrow$ `BETWEEN`, `IN`
- LIKE

  constant pattern anchored to the beginning, *e.g.* 'foo%'
- `IS NULL`, `IS NOT NULL`

HASH

Speeds up

- $=$

GENERALIZED SEARCH TREE (GIST)

Balanced, tree-structured access method.
Template in which to implement arbitrary indexing schemes.

Natively supports

- Geometry: box, circle, point, polygon
- Internet Addresses: inet, cidr
- any range type
- tsquery, tsvector
  document representation to optimized full text search

Many more from the `contrib` collection.

`www.postgresql.org/docs/current/gist-intro.html`

SP-GIST

Similar to GiST but support non-balanced disk-based data structures.

e.g. quadtrees, k-d trees, and radix trees (tries)

GIN

Inverted indexes used for complex objects.

Natively supports

- array
- jsonb
- tsvector

`www.postgresql.org/docs/current/gin-builtin-opclasses.html`

BLOCK RANGE INDEXES (BRIN)

Used for very large tables.
Indexes range of consecutive table's physical blocks.

`www.postgresql.org/docs/current/brin-intro.html`

Explain!

THE EXPLAIN COMMAND

"show the execution plan of a statement"

For each query node shows

e.g. `(cost=0.00..0.24 rows=10 width=82)`

- `cost=f..a`

  `f`: Cost up to the first row

  `a`: Cost for computing all rows

- `rows`

  Expected number of returned rows

- `width`

  Expected width of the rows

`www.postgresql.org/docs/current/sql-explain.html`

EXPLAIN ANALYZE

Executes the command and
shows actual run times and other statistics.

For each query node shows
e.g. ...`(actual time=0.025..0.029 rows=10 loops=1)`

- `actual time`
  Average execution time for one loop
- `rows`
  Actual number of returned rows
- `loops`
  Times the node has been executed

EXPLAIN ANALYZE EXAMPLE 1

```
EXPLAIN ANALYZE
SELECT surname, name
FROM people_person LIMIT 10;
```

EXPLAIN ANALYZE EXAMPLE 1

```
EXPLAIN ANALYZE
SELECT surname, name
FROM people_person LIMIT 10;

Limit  (cost=0.00..0.24 rows=10 width=15)
    (actual time=0.016..0.022 rows=10 loops=1)
->  Seq Scan on people_person
    (cost=0.00..588.88 rows=24988 width=15)
    (actual time=0.014..0.017 rows=10 loops=1)
Planning time: 0.141 ms
Execution time: 0.043 ms
```

EXPLAIN ANALYZE EXAMPLE 2 - QUERY

```
EXPLAIN ANALYZE
SELECT p.surname, p.name, c.name,
    EXTRACT(YEAR FROM AGE(birth_date)),
FROM people_person p JOIN people_country c
    ON citizenship_id=c.id
WHERE EXTRACT(YEAR FROM AGE(birth_date)) > 25;
```

EXPLAIN ANALYZE EXAMPLE 2 - RESULT

```
Hash Join  (cost=8.46..1098.43 rows=8329 width=34)
    (actual time=0.294..69.545 rows=14518 loops=1)
Hash Cond: (p.citizenship_id = c.id)
->  Seq Scan on people_person p
    (cost=0.00..963.70 rows=8329 width=23)
    (actual time=0.019..39.535 rows=14596 loops=1)
        Filter: (date_part('year'::text, age((CURRENT_DATE)::
        Rows Removed by Filter: 10392
->  Hash  (cost=4.87..4.87 rows=287 width=15)
          (actual time=0.250..0.251 rows=287 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 22kB
        ->  Seq Scan on people_country c
            (cost=0.00..4.87 rows=287 width=15)
            (actual time=0.008..0.125 rows=287 loops=1)
Planning time: 0.550 ms
Execution time: 70.285 ms
```

Never ending

- Foreign data wrappers

  `www.postgresql.org/docs/current/ddl-foreign-data.html`
- PostGIS

  `postgis.net`
- Postgres Weekly

  `postgresweekly.com`
- ...

Q&A