

psycopg

September 18, 2026



pyscopg2, pyscopg3 et curseurs

Prérequis: 1 tube d'Aspirine

Marc Rehtë - DBA à M tro France

22/09/2026

psycopg est le standard de facto pour acc der   une base PostgreSQL   partir du langage Python.

2 versions de ce module sont actives. Sans casser la baraque, la version 3 modifie la fa on de dialoguer avec PostgreSQL (objet de cette pr sentation) et apporte d'autres fonctionnalit s (mode asynchrone, pipeline de commandes SQL, pooler de connexions).

1 Ce qu'on va faire dans cet atelier

L'exercice consiste   mettre   jour 2 lignes d'une table, en utilisant le langage Python.

On va d'abord utiliser pyscopg2 puis pyscopg3 et observer l'interaction avec PostgreSQL.

1.1 Jeu de donn es

Notre jeu de donn es de test ressemble  :

```
id | name | bal
---+-----+---
```

```

1 | toto | 0.00
2 | marco | 4093.00
(2 rows)

```

2 Préparatifs

2.1 Instance PostgreSQL

Créer une instance PostgreSQL (déjà fait):

```
initdb data18
```

La démarrer (normalement c'est déjà fait):

```
pg_ctl -D data18 \
-o '-c port=5433 -c unix_socket_directories=/tmp -c listen_addresses='' \
-c logging_collector=on -c log_filename=postgresql-%a.log -c log_statement=mod -c log_checkpoints=on' start
```

Créer la base de de test jupyter (déjà fait):

```
createdb -h /tmp -p 5433 jupyter
```

Si besoin de se connecter en psql:

```
psql -h /tmp -p 5433
```

2.2 Terminal pour suivre le log PostgreSQL

Un terminal SSH est ouvert avec un `tail -f ~/data18/log/postgresql-Tue.log`

2.3 Création du jeu de données

C'est l'occasion d'écrire notre premier programme Python.

3 commandes SQL sont exécutées: CREATE TABLE, TRUNCATE, INSERT

Puis une dernière: ANALYZE

```
[2]: from random import randint

from psycopg2 import connect

# DSN = (host, port, user, db, application)
DSN = ('/tmp', 5433, 'jupyter', 'jupyter', 'jupyter')

SQL = f"""
CREATE TABLE IF NOT EXISTS myusers (
    id INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    name VARCHAR(100),
    bal NUMERIC(10,2)
);
TRUNCATE TABLE myusers;
```

```

INSERT INTO myusers (name, bal) VALUES
    ('toto', 1),
    ('marco', {randint(0, 10000)})
"""

print("Ouverture d'une connexion à la base")
con = connect("host=%s port=%s user=%s dbname=%s application_name=%s" % DSN)

con.autocommit = True

cur = con.cursor()

print("Eventuelle création de la table myusers, vidage et insertion")
cur.execute(SQL)

print("Bonne pratique, lancement d'un ANALYZE sur la table")
cur.execute("ANALYZE myusers")

cur.close()
con.close()

```

Ouverture d'une connexion à la base

Eventuelle création de la table myusers, vidage et insertion

Bonne pratique, lancement d'un ANALYZE sur la table

3 Pyscopg2

3.1 2 appels à execute()

ID	NAME	BAL	
1	TOTO	1	←----- EXECUTE(SQL, {"ID": 1, "BAL": 1000})
2	MARCO	137	←----- EXECUTE(SQL, {"ID": 2, "BAL": {}})

```

LOG: instruction : UPDATE myusers SET bal = bal + 1000 WHERE id = 1
LOG: instruction : UPDATE myusers SET bal = bal + 1 WHERE id = 2

```

Le code d'initialisation, vu ci-dessus, utilise déjà un curseur. Dans psycopg, pour executer tout ordre SQL il faut un "curseur".

```

[7]: from psycopg2 import connect

con = connect("host=%s port=%s user=%s dbname=%s application_name=%s" % DSN)

```

```

con.autocommit = True

cur = con.cursor()

SQL = "UPDATE myusers SET bal = bal + %(bal)s WHERE id = %(id)s"

cur.execute(SQL, {"id": 1, "bal": 1000})
cur.execute(SQL, {"id": 2, "bal": 1})

cur.close()
con.close()

```

Dans le log PostgreSQL, on voit quelque chose de la sorte:

```

2026-01-14 18:14:06.928 CET [15446] LOG:  instruction : UPDATE myusers SET bal = bal + 1000 WH
2026-01-14 18:14:06.940 CET [15446] LOG:  instruction : UPDATE myusers SET bal = bal + 1 WHERE

```

pyscopg2 a envoyé deux requêtes SQL au serveur. Ce dernier a dû analyser 2 fois (parsing, droits d'accès, etc.).

Si nous avons de nombreuses modifications à effectuer, ceci ne va pas être très efficace.

3.2 executemany()

On regardant la doc, on voit qu'il y a la méthode de cureur `executemany`. Essayons-là:

ID	NAME	BAL
1	TOTO	1
2	MARCO	137

←----- EXECMANY(SQL, ({"ID": 1, "BAL": 2000}, {"ID": 2, "BAL": 2}))

```

LOG:  instruction : UPDATE myusers SET bal = bal + 2000 WHERE id = 1
LOG:  instruction : UPDATE myusers SET bal = bal + 2 WHERE id = 2

```

```

[8]: from psycopg2 import connect

con = connect("host=%s port=%s user=%s dbname=%s application_name=%s" % DSN)

con.autocommit = True

cur = con.cursor()

SQL = "UPDATE myusers SET bal = bal + %(bal)s WHERE id = %(id)s"
prm = ({ "id": 1, "bal": 2000}, {"id": 2, "bal": 2})

cur.executemany(SQL, prm);

```

```
cur.close()
con.close()
```

Et aucun changement dans le log:

```
2026-01-14 18:25:34.343 CET [17538] LOG:  instruction : UPDATE myusers SET bal = bal + 2000 WHERE
2026-01-14 18:25:34.351 CET [17538] LOG:  instruction : UPDATE myusers SET bal = bal + 2 WHERE
```

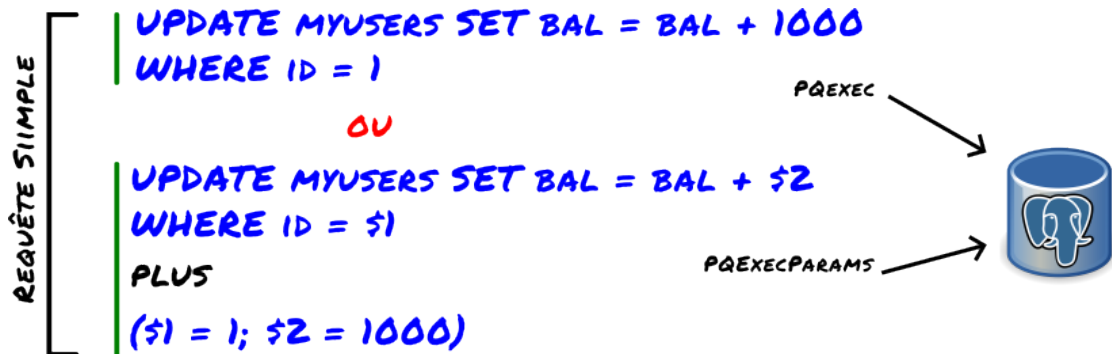
Ca n'arrange pas notre affaire.

3.3 Un peu de théorie

PostgreSQL propose deux méthodes pour faire une requête SQL de type DML (Data Manipulation Language = SELECT, INSERT, UPDATE, DELETE):

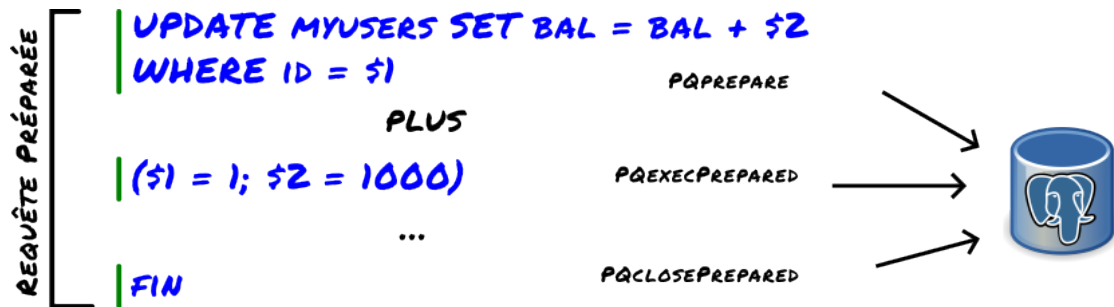
- **Requête simple:** on envoie la requête (cf. PQexec) ou la requête avec des paramètres (cf. PQexecParams)
- **Requête avec binding de paramètres:** dans ce cas au moins trois appels sont requis, un pour la requête (cf. PQprepare), un ou plusieurs pour l'exécution avec les paramètres (cf. PQexecPrepared) et un pour la fermeture (cf. PQclosePrepared). Le lien entre les appels se fait par le nom de la requête donné dans le premier appel (stmtName). Il est possible de ne pas lui donner de nom, elle est "unnamed" et il ne peut y avoir qu'une seule "unnamed" par session (= connexion) PostgreSQL.

La doc est ici: <https://www.postgresql.fr/docs/current/libpq-exec.html>



Visiblement dans les essais précédents, pycpg2 a utilisé PQexec dans les 2 cas.

La méthode avec binding est intéressante si on souhaite envoyer la même requête avec des paramètres différents, car seuls les paramètres sont envoyés à chaque exécution et PostgreSQL a beaucoup moins de travail à effectuer à chaque exécution, puisque la requête a été préparée dans un premier temps.



Essayons autrement.

3.4 Server side cursors ?

En creusant la doc psycopg2, on trouve un chapitre sur “Server side cursors”.
<https://www.psycopg.org/docs/usage.html#server-side-cursors>

Pour activer ce mécanisme, on donne un nom (mycur) au curseur psycopg2:

```
[9]: from psycopg2 import connect

con = connect("host=%s port=%s user=%s dbname=%s application_name=%s" % DSN)

con.autocommit = True

cur = con.cursor("mycur")

SQL = "UPDATE myusers SET bal = bal + $2 WHERE id = $1"

cur.execute(SQL, (1, 3000))
cur.execute(SQL, (2, 3))

cur.close()
con.close()
```

```
-----
ProgrammingError                                Traceback (most recent call last)
Cell In[9], line 11
      7 cur = con.cursor("mycur")
      8
      9 SQL = "UPDATE myusers SET bal = bal + $2 WHERE id = $1"
     10
--> 11 cur.execute(SQL, (1, 3000))
     12 cur.execute(SQL, (2, 3))
     13
     14 cur.close()

ProgrammingError: can't use a named cursor outside of transactions
```

Rien dans le log postgres.

Soit, supprimons le autocommit:

```
[10]: from psycopg2 import connect

con = connect("host=%s port=%s user=%s dbname=%s application_name=%s" % DSN)

con.autocommit = False

cur = con.cursor("mycur")

SQL = "UPDATE myusers SET bal = bal + $2 WHERE id = $1"

cur.execute(SQL, (1, 3000))
cur.execute(SQL, (2, 3))

cur.close()
con.close()
```

```
-----
SyntaxError                                Traceback (most recent call last)
Cell In[10], line 11
      7 cur = con.cursor("mycur")
      8
      9 SQL = "UPDATE myusers SET bal = bal + $2 WHERE id = $1"
     10
--> 11 cur.execute(SQL, (1, 3000))
     12 cur.execute(SQL, (2, 3))
     13
     14 cur.close()

SyntaxError: syntax error at or near "UPDATE"
LINE 1: DECLARE "mycur" CURSOR WITHOUT HOLD FOR UPDATE myusers SET b...
                                         ^
```

Avec ce log:

```
2026-01-14 19:11:32.760 CET [24556] LOG:  instruction : BEGIN
2026-01-14 19:11:32.760 CET [24556] ERREUR:  erreur de syntaxe sur ou près de « UPDATE » au ca
2026-01-14 19:11:32.760 CET [24556] INSTRUCTION :  DECLARE "mycur" CURSOR WITHOUT HOLD FOR UPD
```

Ce qui ne colle pas, un DECLARE ce n'est pas pareil qu'un PREPARE !

Ce qu'on souhaite faire: <https://docs.postgresql.fr/current/sql-prepare.html>

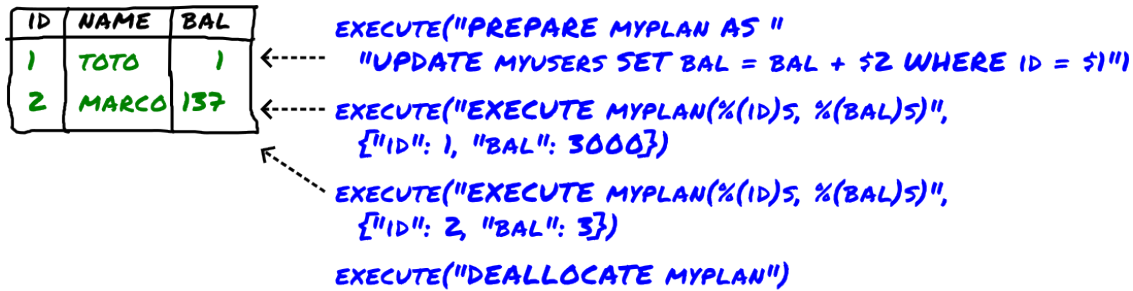
Ce que fait psycopg2: <https://docs.postgresql.fr/current/sql-declare.html>

Fausse piste donc.

Dans psychopg2, je n'ai pas trouvé de moyen implicite de faire une requête préparée.

3.5 PREPARE explicite

Ici on procède comme on le ferait avec `psql`: on utilise les commandes `PREPARE`, `EXECUTE` et `DEALLOCATE`.



```
LOG: instruction : PREPARE myplan AS UPDATE myusers SET bal = bal + $2 WHERE id = $1
LOG: instruction : EXECUTE myplan(1, 3000)
DÉTAIL: préparation : PREPARE myplan AS UPDATE myusers SET bal = bal + $2 WHERE id = $1
LOG: instruction : EXECUTE myplan(2, 3)
DÉTAIL: préparation : PREPARE myplan AS UPDATE myusers SET bal = bal + $2 WHERE id = $1
LOG: instruction : DEALLOCATE myplan
```

```
[11]: from psychopg2 import connect

con = connect("host=%s port=%s user=%s dbname=%s application_name=%s" % DSN)

con.autocommit = True

cur = con.cursor()

SQL1 = "PREPARE myplan AS UPDATE myusers SET bal = bal + $2 WHERE id = $1"

cur.execute(SQL1)

SQL2 = "EXECUTE myplan(%(id)s, %(bal)s)"

cur.execute(SQL2, {"id": 1, "bal": 3000})
cur.execute(SQL2, {"id": 2, "bal": 3})

# On fait le ménage
SQL3 = "DEALLOCATE myplan"
cur.execute(SQL3)

cur.close()
con.close()
```

Et le log associé:

```

2026-01-21 16:08:17.582 CET [23846] LOG:  instruction : PREPARE myplan AS UPDATE myusers SET bal = bal + $1 WHERE id = $2
2026-01-21 16:08:17.604 CET [23846] LOG:  instruction : EXECUTE myplan(1, 3000)
2026-01-21 16:08:17.604 CET [23846] DÉTAIL:  préparation : PREPARE myplan AS UPDATE myusers SET bal = bal + $1 WHERE id = $2
2026-01-21 16:08:17.622 CET [23846] LOG:  instruction : EXECUTE myplan(2, 3)
2026-01-21 16:08:17.622 CET [23846] DÉTAIL:  préparation : PREPARE myplan AS UPDATE myusers SET bal = bal + $1 WHERE id = $2
2026-01-21 16:08:17.626 CET [23846] LOG:  instruction : DEALLOCATE myplan

```

4 pyscopg3

Comme nous avons vu pour psycopg2, l'utilisation de requêtes avec binding de paramètres (requêtes préparées) n'est pas très simple. Psycopg3 remédie à cela.

4.1 Requête préparée (forcée)

On ajoute `prepare=True` à chaque exécution.

ID	NAME	BAL	
1	TOTO	1	←----- EXECUTE(SQL, {"id": 1, "bal": 1000}, PREPARE=TRUE)
2	MARCO	137	←----- EXECUTE(SQL, {"id": 2, "bal": 1}, , PREPARE=TRUE)

```

LOG:  exécute _pg3_0: UPDATE myusers SET bal = bal + $1 WHERE id = $2
DÉTAIL:  Parameters: $1 = '1000', $2 = '1'
LOG:  exécute _pg3_0: UPDATE myusers SET bal = bal + $1 WHERE id = $2
DÉTAIL:  Parameters: $1 = '1', $2 = '2'

```

```

[12]: from psycopg import connect

con = connect("host=%s port=%s user=%s dbname=%s application_name=%s" % DSN,
             ↪autocommit=True)

cur = con.cursor()

SQL = "UPDATE myusers SET bal = bal + %(bal)s WHERE id = %(id)s"

# On spécifie prepare=True afin de ne pas attendre prepare_threshold (=5 par
  ↪défaut) requêtes identiques pour préparer
cur.execute(SQL, {"id": 1, "bal": 1000}, prepare=True)
cur.execute(SQL, {"id": 2, "bal": 1}, prepare=True)

cur.close()
con.close()

```

Le log PostgreSQL indique:

```

2026-01-21 16:39:25.374 CET [25977] LOG:  exécute _pg3_0: UPDATE myusers SET bal = bal + $1 WH
2026-01-21 16:39:25.374 CET [25977] DÉTAIL:  Parameters: $1 = '1000', $2 = '1'

```

```
2026-01-21 16:39:25.375 CET [25977] LOG:  exécute _pg3_0: UPDATE myusers SET bal = bal + $1 WH
2026-01-21 16:39:25.375 CET [25977] DÉTAIL:  Parameters: $1 = '1', $2 = '2'
```

Pyscopg exécute deux fois une requête nommée `_pg3_0` (`_pg3` pour pyscopg3 et `_0` numéro unique identifiant la requête). Par contre, on n'a pas de trace de la phase de préparation, ni de déallocation. Je suppose que c'est le fait de l'utilisation de la libpq (et non les ordre SQL PREPARE/EXECUTE/DEALLOCATE).

4.2 Comportement par défaut

Prenons ce code (sans spécifier `prepare=True`):

```
[13]: from pyscopg import connect

con = connect("host=%s port=%s user=%s dbname=%s application_name=%s" % DSN,
             ↪autocommit=True)

cur = con.cursor()

# Noter qu'il s'agit d'une nouvelle requête SQL (+ => -)
SQL = "UPDATE myusers SET bal = bal - %(bal)s WHERE id = %(id)s"

cur.execute(SQL, {"id": 2, "bal": 1})
cur.execute(SQL, {"id": 2, "bal": 1})
cur.execute(SQL, {"id": 2, "bal": 1})
cur.execute(SQL, {"id": 2, "bal": 1})
cur.execute(SQL, {"id": 2, "bal": 1})
cur.execute(SQL, {"id": 2, "bal": 1})

cur.close()
con.close()
```

Et le log:

```
2026-01-21 16:54:17.853 CET [27108] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal - $1
2026-01-21 16:54:17.853 CET [27108] DÉTAIL:  Parameters: $1 = '1', $2 = '2'
2026-01-21 16:54:17.861 CET [27108] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal - $1
2026-01-21 16:54:17.861 CET [27108] DÉTAIL:  Parameters: $1 = '1', $2 = '2'
2026-01-21 16:54:17.863 CET [27108] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal - $1
2026-01-21 16:54:17.863 CET [27108] DÉTAIL:  Parameters: $1 = '1', $2 = '2'
2026-01-21 16:54:17.866 CET [27108] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal - $1
2026-01-21 16:54:17.866 CET [27108] DÉTAIL:  Parameters: $1 = '1', $2 = '2'
2026-01-21 16:54:17.868 CET [27108] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal - $1
2026-01-21 16:54:17.868 CET [27108] DÉTAIL:  Parameters: $1 = '1', $2 = '2'
2026-01-21 16:54:17.877 CET [27108] LOG:  exécute _pg3_0: UPDATE myusers SET bal = bal - $1 WH
2026-01-21 16:54:17.877 CET [27108] DÉTAIL:  Parameters: $1 = '1', $2 = '2'
```

On observe:

1. Les 5 premiers appels sont préparés, mais sans nommer l'instruction `<unnamed>`.

2. Le sixième appel est toujours préparé, mais avec un nom explicite `_pg3_0`.

Qu'est ce que ça change ?

4.3 Comportement par défaut (bis)

Faisons un mix de 2 requêtes SQL, la première incrémente, la deuxième décrémente. 7x2 appels cette fois-ci:

```
[14]: from psychopg import connect

con = connect("host=%s port=%s user=%s dbname=%s application_name=%s" % DSN,
             ↪autocommit=True)

cur = con.cursor()

# Noter qu'il s'agit d'une nouvelle requête SQL (+ => -)
SQL1 = "UPDATE myusers SET bal = bal + %(bal)s + 1 WHERE id = %(id)s"
SQL2 = "UPDATE myusers SET bal = bal + %(bal)s - 1 WHERE id = %(id)s"

# 7 appels
cur.execute(SQL1, {"id": 2, "bal": 1}); cur.execute(SQL2, {"id": 2, "bal": 1});
cur.execute(SQL1, {"id": 2, "bal": 1}); cur.execute(SQL2, {"id": 2, "bal": 1});
cur.execute(SQL1, {"id": 2, "bal": 1}); cur.execute(SQL2, {"id": 2, "bal": 1});
cur.execute(SQL1, {"id": 2, "bal": 1}); cur.execute(SQL2, {"id": 2, "bal": 1});
cur.execute(SQL1, {"id": 2, "bal": 1}); cur.execute(SQL2, {"id": 2, "bal": 1});
cur.execute(SQL1, {"id": 2, "bal": 1}); cur.execute(SQL2, {"id": 2, "bal": 1});
cur.execute(SQL1, {"id": 2, "bal": 1}); cur.execute(SQL2, {"id": 2, "bal": 1});

cur.close()
con.close()
```

Le log:

```
2026-01-21 17:11:51.957 CET [27872] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:11:51.957 CET [27872] DÉTAIL: Parameters: $1 = '1', $2 = '2'
2026-01-21 17:11:51.970 CET [27872] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:11:51.970 CET [27872] DÉTAIL: Parameters: $1 = '1', $2 = '2'
2026-01-21 17:11:51.972 CET [27872] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:11:51.972 CET [27872] DÉTAIL: Parameters: $1 = '1', $2 = '2'
2026-01-21 17:11:51.974 CET [27872] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:11:51.974 CET [27872] DÉTAIL: Parameters: $1 = '1', $2 = '2'
2026-01-21 17:11:51.975 CET [27872] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:11:51.975 CET [27872] DÉTAIL: Parameters: $1 = '1', $2 = '2'
2026-01-21 17:11:51.978 CET [27872] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:11:51.978 CET [27872] DÉTAIL: Parameters: $1 = '1', $2 = '2'
2026-01-21 17:11:51.980 CET [27872] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:11:51.980 CET [27872] DÉTAIL: Parameters: $1 = '1', $2 = '2'
2026-01-21 17:11:51.982 CET [27872] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal + $1
```

```

2026-01-21 17:11:51.982 CET [27872] DÉTAIL: Parameters: $1 = '1', $2 = '2'
2026-01-21 17:11:51.984 CET [27872] LOG: exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:11:51.984 CET [27872] DÉTAIL: Parameters: $1 = '1', $2 = '2'
2026-01-21 17:11:51.986 CET [27872] LOG: exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:11:51.986 CET [27872] DÉTAIL: Parameters: $1 = '1', $2 = '2'

2026-01-21 17:11:51.990 CET [27872] LOG: exécute _pg3_0: UPDATE myusers SET bal = bal + $1 +
2026-01-21 17:11:51.990 CET [27872] DÉTAIL: Parameters: $1 = '1', $2 = '2'
2026-01-21 17:11:51.998 CET [27872] LOG: exécute _pg3_1: UPDATE myusers SET bal = bal + $1 -
2026-01-21 17:11:51.998 CET [27872] DÉTAIL: Parameters: $1 = '1', $2 = '2'

2026-01-21 17:11:51.999 CET [27872] LOG: exécute _pg3_0: UPDATE myusers SET bal = bal + $1 +
2026-01-21 17:11:51.999 CET [27872] DÉTAIL: Parameters: $1 = '1', $2 = '2'
2026-01-21 17:11:52.001 CET [27872] LOG: exécute _pg3_1: UPDATE myusers SET bal = bal + $1 -
2026-01-21 17:11:52.001 CET [27872] DÉTAIL: Parameters: $1 = '1', $2 = '2'

```

On voit que sur les 5 premiers couples d'exécution, on alterne les requêtes SQL1 et SQL2 sur la préparation "unnamed". On perd le bénéfice de la préparation, puisque la requête doit être reparsée à chaque fois.

Les 2 derniers se comportent différemment: SQL1 porte le nom `_pg3_0` et SQL2 le nom `_pg3_1`. Ici on bénéficie de la préparation (`_pg3_0` et `_pg3_1` sont exécutées 2 fois chacune).

Au niveau de la libpq, à partir de la 6ème itération, psycopg a appelé, **pour chaque SQL**: `PQPrepare`, puis 2 fois `PQexecPrepared`, cette fois-ci avec des **noms distincts**.

4.4 L'explication s'impose

La doc `pyscopg3` (<https://www.psycopg.org/psycopg3/docs/advanced/prepare.html#prepared-statements>) porte à confusion:

`prepare_threshold`: Number of times a query is executed before it is prepared.

Dans `pyscopg3`, **toutes** les requêtes DML sont préparées dès lors qu'on utilise la méthode `execute`. Donc si la majorité des requêtes sont uniques, alors **c'est moins performant** que le comportement `psycopg2` (2 aller-retours client/serveur contre 1 seul).

De plus, `pyscopg3` doit maintenir un cache des requêtes déjà exécutées, afin de détecter le franchissement du seuil `prepare_threshold`, ce qui implique consommation de RAM et ralentissement d'exécution (parcours du cache). Voir https://github.com/psycopg/psycopg/blob/master/psycopg/_preparing.py

La doc dit aussi qu'on peut désactiver ce mécanisme en mettant `prepare_threshold` à `None`. Testons:

4.5 Comportement par défaut (ter)

`prepare_threshold` à `None`

```

[15]: from psycopg import connect

con = connect("host=%s port=%s user=%s dbname=%s application_name=%s" % DSN,
             ↪autocommit=True, prepare_threshold=None)

```

```

cur = con.cursor()

SQL = "UPDATE myusers SET bal = bal + %(bal)s WHERE id = %(id)s"

# 6 exécutions de la même requête
cur.execute(SQL, {"id": 2, "bal": 1}); cur.execute(SQL, {"id": 2, "bal": 1});
↳ cur.execute(SQL, {"id": 2, "bal": 1})
cur.execute(SQL, {"id": 2, "bal": 1}); cur.execute(SQL, {"id": 2, "bal": 1});
↳ cur.execute(SQL, {"id": 2, "bal": 1})

cur.close()
con.close()

```

Le log:

```

2026-01-21 17:44:01.453 CET [29730] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:44:01.453 CET [29730] DÉTAIL:  Parameters: $1 = '1', $2 = '2'
2026-01-21 17:44:01.461 CET [29730] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:44:01.461 CET [29730] DÉTAIL:  Parameters: $1 = '1', $2 = '2'
2026-01-21 17:44:01.463 CET [29730] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:44:01.463 CET [29730] DÉTAIL:  Parameters: $1 = '1', $2 = '2'
2026-01-21 17:44:01.466 CET [29730] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:44:01.466 CET [29730] DÉTAIL:  Parameters: $1 = '1', $2 = '2'
2026-01-21 17:44:01.469 CET [29730] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:44:01.469 CET [29730] DÉTAIL:  Parameters: $1 = '1', $2 = '2'
2026-01-21 17:44:01.471 CET [29730] LOG:  exécute <unnamed>: UPDATE myusers SET bal = bal + $1
2026-01-21 17:44:01.471 CET [29730] DÉTAIL:  Parameters: $1 = '1', $2 = '2'

```

Donc ça ne désactive pas la préparation des requêtes, seulement leur mise en cache par pycopg3.

On sort l'aspirine ?

4.6 Par curseur ou par connexion ?

Ce principe de cache et transactions préparées est maintenu au niveau de la connexion.

Le nombre maximum de transactions préparées (`_pg3_0`, `_pg3_1`, ...) est limité par `prepared_max` (100 par défaut), qui est aussi la taille du cache. En cas de dépassement, pycopg relâche les plus anciennes (DEALLOCATE), et ne produit pas d'erreur...

4.7 Je veux faire du pycopg2 !

Et bien, c'est possible. pycopg3 met à disposition un `ClientCursor`

```

[16]: from pycopg import connect, ClientCursor

con = connect("host=%s port=%s user=%s dbname=%s application_name=%s" % DSN,
↳ autocommit=True)

```

```

con.cursor_factory = ClientCursor

cur = con.cursor()

SQL = "UPDATE myusers SET bal = bal + %(bal)s WHERE id = %(id)s"

cur.execute(SQL, {"id": 1, "bal": 2000})

cur.close()
con.close()

```

Et le log:

2026-04-14 17:03:12.192 CEST [212883] LOG: statement: UPDATE myusers SET bal = bal + 2000 WHE

4.8 Le bénéfice d'une requête préparée ?

Côté PostgreSQL, qu'on utilise pycopg, psycopg, libpq ou tout autre méthode d'accès, ceci reste valable.

4.8.1 plan_cache_mode

Au niveau PostgreSQL, une transaction préparée consiste en: - La préparation: le client envoie la requête paramétrée ($\backslash \$1$, $\backslash \$2$, ...) au serveur. Ce dernier l'analyse (parsing, droits, ...), il ne calcule pas de plan d'exécution. - L'exécution: le client demande l'exécution de la requête avec la valeur des paramètres ($\backslash \$1$, $\backslash \$2$, ...). Le serveur calcule un plan d'exécution "custom" (avec connaissance des paramètres), puis exécute la requête - Si une deuxième exécution: le client passe un autre jeu de paramètres. Le serveur calcule un deuxième plan d'exécution "custom", puis exécute la requête - Jusqu'à la **cinquième exécution**, le principe reste le même. Le serveur calcule encore un plan d'exécution "custom", puis exécute la requête - A la sixième exécution, il fait la moyenne du cout des plans "custom" précédents. Si le cout du plan "generic" (c-à-d sans paramètres) est inférieur, il choisit le "generic" **pour toutes les exécutions suivantes**, sinon il reste en mode "custom" (c-à-d que le plan est calculé à chaque exécution).

Ici on parle bien de couts estimés (pas réels).

Le paramètre postgres `plan_cache_mode` (à 'auto' par défaut) permet de modifier ce comportement: - `force_custom_plan`: fait systématiquement une planification pour chaque exécution - `force_generic_plan`: la planification n'est faite qu'une seule fois (en ignorant la valeur des paramètres). Ce plan sera conservé pour toutes les exécutions suivantes.

4.8.2 Pourquoi planifier à chaque exécution ?

A cause de la sélectivité: nombre de lignes touchées par la requête / nombre de lignes de la table %.

Par exemple, ceci va déterminer si l'accès se fait en utilisant un index, plutôt qu'en parcourant toute la table.

Pour creuser le sujet, un article de Richard Yen: https://richyen.com/postgres/2026/03/30/plan_cache_mode.htm

L'exemple montre une table avec une colonne `flag` pouvant prendre que 2 valeurs (Y ou N), avec un gros déséquilibre entre la population de Y (999.000) et de N (1.000). Cette colonne est indexée.

Sur une requête du type:

```
SELECT * FROM pgbench_accounts WHERE flag = 'Y'
```

sera plus optimale en faisant un scan de la table, alors que:

```
SELECT * FROM pgbench_accounts WHERE flag = 'N'
```

aurait intérêt à utiliser l'index.

5 Conclusion

5.1 Vous avez dit curseur ?

Avec pycopg, la notion de curseur est floue: Cursor, ServerCursor, ClientCursor. A sa defense, il implémente la DB-API de Python. Voir <https://peps.python.org/pep-0249/#cursor-objects>

Dans le concept SQL, un curseur est un pointeur, côté serveur, sur le jeu de données manipulées par une commande (en général SELECT, ou INSERT, DELETE, UPDATE avec la clause RETURNING). Le curseur est positionné au début du jeu de données, puis se déplace vers la fin, au fur et à mesure que le client récupère ses lignes (commande FETCH). Son usage est réservé a des requêtes retournant un volume très important de données.

La durée de vie d'un curseur (DECLARE CURSOR FOR) est, le temps de la transaction, ou de la session (déconnexion), ou de sa fermeture explicite (commande CLOSE).

Donc, un curseur pycopg ne correspond pas à un curseur SQL, il faut le voir comme un objet maintenant l'état d'exécution d'une commande SQL.

Seul ServerCursor (curseur nommé, voir plus haut) déclare un curseur SQL. Voir <https://www.pycopg.org/pycopg3/docs/advanced/cursors.html>

5.2 Nouveau projet = pycopg3

Il me paraît évident que tout nouveau projet doit reposer sur pycopg3, on peut choisir son mode de fonctionnement des *curseurs* (Cursor ou ClientCursor) au cas par cas, et on dispose d'API sur les nouvelles fonctionnalités de PostgreSQL (pipeline, LISTEN/NOTIFY), et de Python (asyncio).

Je trouve son système de cache des requêtes SQL douteux (la paramètre `prepare_threshold`). Perso je le mettrais à 0 (désactivation du cache) et jouerais sur le choix ClientCursor / Cursor.

[]: