batch_inserts

October 19, 2023

1 RDB lab: Batch inserts

This lab intends to demonstrate to you the benefits of batch inserts.

For this, we will insert into a PostgreSQL database a substantial amount of data, first using INSERTs for each new record, then using batch inserts.

This lab works best if you run it in a jupyter notebook, but you can also copy-paste the code into individual scripts.

1.1 Prerequisites

The only new thing here is the psycopg library. Here is how to install it:

```
pip install --upgrade pip       # upgrade pip to at least 20.3
pip install "psycopg[binary]"
```

1.2 Step 0: create a database

Let's get started. First create a new database called `rdb_lab_batch_inserts` using Beekeeper Studio:

```
CREATE DATABASE rdb_lab_batch_inserts
```

1.3 Step 1: insert 10k records, with individual INSERTs / commits

Now, run the script below. It will connect to the database (you might have to change the details in `conn_info`) and insert 10'000 random records.

I recommend you read the comments to understand it.

[1]:
```
import random
import string
from time import time
from psycopg import sql, connect

# connect to postgres
conn_info = "dbname=rdb_lab_batch_inserts host=localhost port=5432␣
 ↪user=postgres password=myverysecretpassword"
conn = connect(conn_info)
```

```
# create a new table
with conn.cursor() as cur:
    cur.execute("CREATE TABLE IF NOT EXISTS records (id serial PRIMARY KEY, num␣
 ↪BIGINT, txt VARCHAR);")
    conn.commit()

size_exp = 64  # the length of txt in each row, e.g. txt will be NEIDGW if␣
 ↪size_exp is 6

# let's create a lot of data. We start with 10'000 entries, wow!
with conn.cursor() as cur:

    start = time()  # let's keep track of how long things take
    for i in range(10000):
        # we create a random int btw 1 and 10**17 and a random string with␣
 ↪length of size_exp
        random_num = random.randint(1, 10**17) # so as to fit in BIGINT
        random_str = ''.join(random.choices(string.ascii_uppercase + string.
 ↪digits, k=size_exp))
        # insert it, one record at a time
        cur.execute(sql.SQL(f"INSERT INTO records (num, txt) VALUES (%s, %s)"),␣
 ↪(random_num, random_str))
        conn.commit()

print(f"Done, took {round(time() - start, 2)}s for 10,000 records")
```

```
Done, took 7.01s for 10,000 records
```

1.4 Step 2: one single commit

Note how long it took to insert the 10'000 records above. We can do better!

In the code above, we call `conn.commit()` after every INSERT. In the code below, there is a single difference: we call `conn.commit()` just once (because `conn.commit()` is **outside** the for-loop now).

[2]:
```
with conn.cursor() as cur:

    start = time()  # let's keep track of how long things take
    for i in range(10000):
        # we create a random int btw 1 and 10**17 and a random string with␣
 ↪length of size_exp
        random_num = random.randint(1, 10**17) # so as to fit in BIGINT
        random_str = ''.join(random.choices(string.ascii_uppercase + string.
 ↪digits, k=size_exp))
        # insert it, one record at a time
        cur.execute(sql.SQL(f"INSERT INTO records (num, txt) VALUES (%s, %s)"),␣
 ↪(random_num, random_str))
    conn.commit()  # <---- this is called just once
```

```
print(f"Done, took {round(time() - start, 3)}s for 10,000 records")
```

```
Done, took 1.47s for 10,000 records
```

That's already much faster! But we can still do better!

1.5 Step 3: Batch inserts

For this, we **batch the INSERTs**. That is, instead of sending and committing an INSERT request for each new record, we will send one large INSERT command every 10'000 records.

We will insert 1M records in total.

[3]:
```
# note: if you run this code as a standalone script, you will have to copy
#       the first part of the script above (until where size_exp is defined).


batch_size = 1000
nr_batches = 1000   # 10'000 * 100 = 1M records in total

total_time = 0
for batch in range(nr_batches):
    with conn.cursor() as cursor:
        new_rows = [(random.randint(1, 10**17), ''.join(random.choices(string.
 ↪ascii_uppercase + string.digits, k=size_exp))) for i in range(1, batch_size)]

        start = time()
        cursor.executemany('INSERT INTO records (num, txt) VALUES (%s, %s)',␣
 ↪new_rows)
        conn.commit()
        total_time += time() - start

print(f"inserted {format(batch * batch_size, ',')} records, took␣
 ↪{round(total_time, 2)}s for 1'000'000 records ")
```

```
inserted 999,000 records, took 14.41s for 1'000'000 records
```

Notice how much faster this is! In practice, you have to experiment with different `batch_size` to see which one performs best.

If for some reason you have to interrupt the script above, and then try to run it again, you might get this exception:

```
InFailedSqlTransaction: current transaction is aborted, commands ignored until end of transacti
```

One way to fix it is to commit the open transaction like this.

```
conn.commit()
```

1.6 Step 4: Faster with copy

There's even a faster way with psycopg's copy function:

[4]:
```
nr_records = 1000000

records = [(random.randint(1, 10**17), ''.join(random.choices(string.
 ↪ascii_uppercase + string.digits, k=size_exp))) for i in range(1, nr_records)]

start=time()
with conn.cursor() as cursor:
    with cursor.copy("COPY records (num, txt) FROM STDIN") as copy:
        for record in records:
            copy.write_row(record)
print(f"Done, took {round(time() - start, 3)}s for 1'000'000 records")
```

```
Done, took 1.438s for 1'000'000 records
```

Quite good: we just inserted a million records in a couple of seconds! I think it's the fastest we can do with psycopg.

Once you have inserted all records, it's good practice to close the db connection:

[5]:
```
conn.close()
```

[ ]:
```

```