09 JULY 2019 / PYTHON, POSTGRESQL, PERFORMANCE

Fastest Way to Load Data Into PostgreSQL Using Python

From two minutes to less than half a second!

. . .

As glorified data plumbers, we are often tasked with loading data fetched from a remote source into our systems. If we are lucky, the data is serialized as JSON or YAML. When we are less fortunate, we get an Excel spreadsheet or a CSV file which is always broken in some way, can't explain it.

Data from large companies or old systems is somehow always encoded in a weird way, and the Sysadmins always think they do us a favour by zipping the files (please gzip) or break them into smaller files with random names.

Modern services might provide a decent API, but more often that not we need to fetch a file from an FTP, SFTP, S3 or some proprietary vault that works only on Windows.

In this article we explore the best way to import messy data from remote source into PostgreSQL.

To provide a real life, workable solution, we set the following ground roles:

- The Data is fetched from a remote source.
- The Data is dirty and needs to be transformed.
- Data is big.

Table of Contents

Speedy Gonzales

. . .

Setup: A Beer Brewery

I found this great public API for beers, so we are going to import data to a beer table in the database.

The Data

A single beer from the API looks like this:

```
$ curl https://api.punkapi.com/v2/beers/?per_page=1&page=1
[
    {
        "id": 1,
        "name": "Buzz",
        "tagline": "A Real Bitter Experience.",
        "first_brewed": "09/2007",
        "description": "A light, crisp and bitter IPA ...",
        "image_url": "https://images.punkapi.com/v2/keg.png",
        "abv": 4.5,
        "ibu": 60,
        "target_fg": 1010,
        "target_og": 1044,
        "ebc": 20,
        "srm": 10,
        "ph": 4.4,
        "attenuation_level": 75,
        "volume": {
            "value": 20,
            "unit": "litres"
        },
        "contributed_by": "Sam Mason <samjbmason>"
        "brewers_tips": "The earthy and floral aromas from...",
        "boil_volume": {},
        "method": {},
        "ingredients": {},
        "food_pairing": [],
    }
]
```

I trimmed the output for brevity, but there is a lot of information about beers here. In this article we want to import all of the fields before `brewers_tips` to a table in the database.

The field `volume` is nested. We want to extract only the `value` from the field, and save it to a field called `volume` in the table.

```python
volume = beer['volume']['data']
```

The field `first_brewed` contains only year and month, and in some cases, only the year. We want to transform the value to a valid date. For example, the value `09/2007` will be transformed to date `2007-09-01`. The value `2006` will be transformed to date `2016-01-01`.

Let's write a simple function to transform the text value in the field, to a Python `datetime.date`:

```python
import datetime

def parse_first_brewed(text: str) -> datetime.date:
    parts = text.split('/')
    if len(parts) == 2:
        return datetime.date(int(parts[1]), int(parts[0]), 1)
    elif len(parts) == 1:
        return datetime.date(int(parts[0]), 1, 1)
    else:
        assert False, 'Unknown date format'
```

Let's quickly make sure that it works:

```
>>> parse_first_brewed('09/2007')
datetime.date(2007, 9, 1)

>>> parse_first_brewed('2006')
datetime.date(2006, 1, 1)
```

In real life, transformations can be much more complicated. But for our purpose, this is more than enough.

Fetch the Data

The API provides paged results. To encapsulate the paging, we create a generator that yields beers one by one:

```
from typing import Iterator, Dict, Any
from urllib.parse import urlencode
import requests


def iter_beers_from_api(page_size: int = 5) -> Iterator[Dict[str, Any]]:
    session = requests.Session()
    page = 1
    while True:
        response = session.get('https://api.punkapi.com/v2/beers?' + urlencode({
            'page': page,
            'per_page': page_size
        }))
        response.raise_for_status()
        data = response.json()
        if not data:
            break

        yield from data

        page += 1
```

And to use the generator function, we call and iterate it:

```
>>> beers = iter_beers_from_api()
>>> next(beers)
{'id': 1,
 'name': 'Buzz',
 'tagline': 'A Real Bitter Experience.',
 'first_brewed': '09/2007',
 'description': 'A light, crisp and bitter IPA brewed...',
 'image_url': 'https://images.punkapi.com/v2/keg.png',
 'abv': 4.5,
 'ibu': 60,
 'target_fg': 1010,
...
}
>>> next(beers)
{'id': 2,
 'name': 'Trashy Blonde',
 'tagline': "You Know You Shouldn't",
 'first_brewed': '04/2008',
 'description': 'A titillating, ...',
 'image_url': 'https://images.punkapi.com/v2/2.png',
 'abv': 4.1,
 'ibu': 41.5,
```

You will notice that the first result of each page takes a bit longer. This is because it does a network request to fetch the page.

Create a Table in the Database

The next step is to create a table in the database to import the data into.

Create a database:

```
$ createdb -O haki testload
```

Change `haki` in the example to your local user.

To connect from Python to a PostgreSQL database, we use psycopg:

```
$ python -m pip install psycopg2
```

Using psycopg, create a connection to the database:

```
import psycopg2

connection = psycopg2.connect(
    host="localhost",
    database="testload",
    user="haki",
    password=None,
)
connection.autocommit = True
```

We set `autocommit=True` so every command we execute will take effect immediately. For the purpose of this article, this is fine.

Now that we have a connection, we can write a function to create a table:

```
def create_staging_table(cursor) -> None:
    cursor.execute("""
        DROP TABLE IF EXISTS staging_beers;
        CREATE UNLOGGED TABLE staging_beers (
            id                  INTEGER,
            name                TEXT,
            tagline             TEXT,
            first_brewed        DATE,
            description         TEXT,
            image_url           TEXT,
            abv                 DECIMAL,
            ibu                 DECIMAL,
            target_fg           DECIMAL,
            target_og           DECIMAL,
            ebc                 DECIMAL,
            srm                 DECIMAL,
            ph                  DECIMAL,
            attenuation_level   DECIMAL,
            brewers_tips        TEXT,
            contributed_by      TEXT,
            volume              INTEGER
        );
    """)
```

The function receives a cursor and creates a unlogged table called `staging_beers`.

UNLOGGED TABLE

Data written to an unlogged table will not be logged to the write-ahead-log (WAL), making it ideal for intermediate tables. Note that `UNLOGGED` tables will not be restored in case of a crash, and will not be replicated.

Using the connection we created before, this is how the function is used:

```
>>> with connection.cursor() as cursor:
>>>     create_staging_table(cursor)
```

We are now ready to move on to the next part.

· · ·

Metrics

Throughout this article we are interested in two main metrics: time and memory.

Measuring Time

To measure time for each method we use the built-in `time` module:

```
>>> import time
>>> start = time.perf_counter()
>>> time.sleep(1) # do work
>>> elapsed = time.perf_counter() - start
>>> print(f'Time {elapsed:0.4}')
Time 1.001
```

The function `perf_counter` provides the clock with the highest available resolution, which makes it ideal for our purposes.

Measuring Memory

To measure memory consumption, we are going to use the package memory-profiler.

```
$ python -m pip install memory-profiler
```

This package provides the memory usage, and the incremental memory usage for each line in the code. This is very useful when optimizing for memory. To illustrate, this is the example provided in PyPI:

```
$ python -m memory_profiler example.py

Line #    Mem usage  Increment   Line Contents
==============================================
     3                           @profile
     4      5.97 MB    0.00 MB   def my_func():
     5     13.61 MB    7.64 MB       a = [1] * (10 ** 6)
     6    166.20 MB  152.59 MB       b = [2] * (2 * 10 ** 7)
     7     13.61 MB -152.59 MB       del b
     8     13.61 MB    0.00 MB       return a
```

The interesting part is the `Increment` column that shows the additional memory allocated by the code in each line.

In this article we are interested in the peak memory used by the function. The peak memory is the difference between the starting value of the "Mem usage" column, and the highest value (also known as the "high watermark").

To get the list of "Mem usage" we use the function `memory_usage` from `memory_profiler`:

```
>>> from memory_profiler import memory_usage
>>> mem, retval = memory_usage((fn, args, kwargs), retval=True, interval=1e-7)
```

When used like this, the function `memory_usage` executes the function `fn` with the provided `args` and `kwargs`, but also launches another process in the background to monitor the memory usage every `interval` seconds.

For very quick operations the function `fn` might be executed more than once. By setting `interval` to a value lower than 1e-6, we force it to execute only once.

The argument `retval` tells the function to return the result of `fn`.

`profile` Decorator

To put it all together, we create the following decorator to measure and report time and memory:

```
import time
from functools import wraps
from memory_profiler import memory_usage

def profile(fn):
    @wraps(fn)
    def inner(*args, **kwargs):
        fn_kwargs_str = ', '.join(f'{k}={v}' for k, v in kwargs.items())
        print(f'\n{fn.__name__}({fn_kwargs_str})')

        # Measure time
        t = time.perf_counter()
        retval = fn(*args, **kwargs)
        elapsed = time.perf_counter() - t
        print(f'Time   {elapsed:0.4}')

        # Measure memory
        mem, retval = memory_usage((fn, args, kwargs), retval=True, timeout=200, ir

        print(f'Memory {max(mem) - min(mem)}')
        return retval

    return inner
```

To eliminate mutual effects of the timing on the memory and vice versa, we execute the function twice. First to time it, second to measure the memory usage.

The decorator will print the function name and any keyword arguments, and report the time and memory used:

```
>>> @profile
>>> def work(n):
>>>     for i in range(n):
>>>         2 ** n

>>> work(10)
work()
Time   0.06269
Memory 0.0

>>> work(n=10000)
work(n=10000)
Time   0.3865
Memory 0.0234375
```

Only keywords arguments are printed. This is intentional, we are going to use that in parameterized tests.

. . .

Benchmark

At the time of writing, the beers API contains only 325 beers. To work on a large dataset, we duplicate it 100 times and store it in-memory. The resulting dataset contains 32,500 beers:

```
>>> beers = list(iter_beers_from_api()) * 100
>>> len(beers)
32,500
```

To imitate a remote API, our functions will accept iterators similar to the return value of `iter_beers_from_api`:

```
def process(beers: Iterator[Dict[str, Any]]) -> None:
    # Process beers...
```

For the benchmark, we are going to import the beer data into the database. To eliminate external influences such as the network, we fetch the data from the API in advance, and serve it locally.

To get an accurate timing, we "fake" the remote API:

```
>>> beers = list(iter_beers_from_api()) * 100
>>> process(beers)
```

In a real life situation you would use the function `iter_beers_from_api` directly:

```
>>> process(iter_beers_from_api())
```

We are now ready to start!

Insert Rows One by One

To establish a baseline we start with the simplest approach, insert rows one by one:

```
@profile
def insert_one_by_one(connection, beers: Iterator[Dict[str, Any]]) -> None:
    with connection.cursor() as cursor:
        create_staging_table(cursor)
        for beer in beers:
            cursor.execute("""
                INSERT INTO staging_beers VALUES (
                    %(id)s,
                    %(name)s,
                    %(tagline)s,
                    %(first_brewed)s,
                    %(description)s,
                    %(image_url)s,
                    %(abv)s,
                    %(ibu)s,
                    %(target_fg)s,
                    %(target_og)s,
                    %(ebc)s,
                    %(srm)s,
                    %(ph)s,
                    %(attenuation_level)s,
                    %(brewers_tips)s,
                    %(contributed_by)s,
                    %(volume)s
                );
            """, {
                **beer,
                'first_brewed': parse_first_brewed(beer['first_brewed']),
                'volume': beer['volume']['value'],
            })
```

Notice that as we iterate the beers, we transform the `first_brewed` to a `datetime.date` and extracted the volume value from the nested `volume` field.

Running this function produces the following output:

```
>>> insert_one_by_one(connection, beers)
insert_one_by_one()
Time   128.8
Memory 0.08203125
```

The function took 129 seconds to import 32K rows. The memory profiler shows that the function consumed very little memory.

Intuitively, inserting rows one by one does not sound very efficient. The constant context switching between the program and the database must be slowing it down.

Execute Many

Psycopg2 provides a way to insert many rows at once using `executemany`. From the docs:

> *Execute a database operation (query or command) against all parameter tuples or mappings found in the sequence vars_list.*

Sounds promising!

Let's try to import the data using `executemany`:

```
@profile
def insert_executemany(connection, beers: Iterator[Dict[str, Any]]) -> None:
    with connection.cursor() as cursor:
        create_staging_table(cursor)

        all_beers = [{
            **beer,
            'first_brewed': parse_first_brewed(beer['first_brewed']),
            'volume': beer['volume']['value'],
        } for beer in beers]
        cursor.executemany("""
            INSERT INTO staging_beers VALUES (
                %(id)s,
                %(name)s,
                %(tagline)s,
                %(first_brewed)s,
                %(description)s,
                %(image_url)s,
                %(abv)s,
                %(ibu)s,
                %(target_fg)s,
                %(target_og)s,
                %(ebc)s,
                %(srm)s,
                %(ph)s,
                %(attenuation_level)s,
                %(brewers_tips)s,
                %(contributed_by)s,
                %(volume)s
            );
        """, all_beers)
```

The function looks very similar to the previous function, and the transformations are the same. The main difference here is that we first transform all of the data in-memory, and only then import it to the database.

Running this function produces the following output:

```
>>> insert_executemany(connection, beers)
insert_executemany()
Time   124.7
Memory 2.765625
```

This is disappointing. The timing is just a little bit better, but the function now consumes 2.7MB of memory.

To put the memory usage in perspective, a JSON file containing only the data we import weighs 25MB on disk. Considering the proportion, using this method to import a 1GB file will require 110MB of memory.

Execute Many From Iterator

The previous method consumed a lot of memory because the transformed data was stored in-memory before being processed by psycopg.

Let's see if we can use an iterator to avoid storing the data in-memory:

```
@profile
def insert_executemany_iterator(connection, beers: Iterator[Dict[str, Any]]) -> Nor
    with connection.cursor() as cursor:
        create_staging_table(cursor)
        cursor.executemany("""
            INSERT INTO staging_beers VALUES (
                %(id)s,
                %(name)s,
                %(tagline)s,
                %(first_brewed)s,
                %(description)s,
                %(image_url)s,
                %(abv)s,
                %(ibu)s,
                %(target_fg)s,
                %(target_og)s,
                %(ebc)s,
                %(srm)s,
                %(ph)s,
                %(attenuation_level)s,
                %(brewers_tips)s,
                %(contributed_by)s,
                %(volume)s
            );
        """, ({
            **beer,
            'first_brewed': parse_first_brewed(beer['first_brewed']),
            'volume': beer['volume']['value'],
        } for beer in beers))
```

The difference here is that the transformed data is "streamed" into `executemany` using an iterator.

This function produces the following result:

```
>>> insert_executemany_iterator(connection, beers)
insert_executemany_iterator()
Time   129.3
Memory 0.0
```

Our "streaming" solution worked as expected and we managed to bring the memory to zero. The timing however, remains roughly the same, even compared to the one-by-one method.

Execute Batch

The psycopg documentation has a very interesting note about `executemany` in the "fast execution helpers" section:

> *The current implementation of executemany() is (using an extremely charitable understatement) not particularly performing. These functions can be used to speed up the repeated execution of a statement against a set of parameters. By reducing the number of server roundtrips the performance can be orders of magnitude better than using executemany().*

So we've been doing it wrong all along!

The function just below this section is `execute_batch`:

> *Execute groups of statements in fewer server roundtrips.*

Let's implement the loading function using `execute_batch`:

```
import psycopg2.extras

@profile
```

```python
def insert_execute_batch(connection, beers: Iterator[Dict[str, Any]]) -> None:
    with connection.cursor() as cursor:
        create_staging_table(cursor)

        all_beers = [{
            **beer,
            'first_brewed': parse_first_brewed(beer['first_brewed']),
            'volume': beer['volume']['value'],
        } for beer in beers]

        psycopg2.extras.execute_batch(cursor, """
            INSERT INTO staging_beers VALUES (
                %(id)s,
                %(name)s,
                %(tagline)s,
                %(first_brewed)s,
                %(description)s,
                %(image_url)s,
                %(abv)s,
                %(ibu)s,
                %(target_fg)s,
                %(target_og)s,
                %(ebc)s,
                %(srm)s,
                %(ph)s,
                %(attenuation_level)s,
                %(brewers_tips)s,
                %(contributed_by)s,
                %(volume)s
            );
        """, all_beers)
```

Executing the function:

```
>>> insert_execute_batch(connection, beers)
insert_execute_batch()
Time   3.917
Memory 2.50390625
```

Wow! That's a huge leap. The function completed in just under 4 seconds. That's ~33 times faster than the 129 seconds we started with.

Execute Batch From Iterator

The function `execute_batch` used less memory than `executemany` did for the same data. Let's try to eliminate memory by "streaming" the data into `execute_batch` using an iterator:

```
@profile
def insert_execute_batch_iterator(connection, beers: Iterator[Dict[str, Any]]) -> N
    with connection.cursor() as cursor:
        create_staging_table(cursor)

        iter_beers = ({
            **beer,
            'first_brewed': parse_first_brewed(beer['first_brewed']),
            'volume': beer['volume']['value'],
        } for beer in beers)

        psycopg2.extras.execute_batch(cursor, """
            INSERT INTO staging_beers VALUES (
                %(id)s,
                %(name)s,
                %(tagline)s,
                %(first_brewed)s,
                %(description)s,
                %(image_url)s,
                %(abv)s,
                %(ibu)s,
                %(target_fg)s,
                %(target_og)s,
                %(ebc)s,
                %(srm)s,
                %(ph)s,
                %(attenuation_level)s,
                %(brewers_tips)s,
                %(contributed_by)s,
                %(volume)s
            );
        """, iter_beers)
```

Executing the function

```
>>> insert_execute_batch_iterator(connection, beers)
insert_execute_batch_iterator()
Time   4.333
Memory 0.2265625
```

We got roughly the same time, but with less memory.

Execute Batch From Iterator with Page Size

When reading though the documentation for `execute_batch`, the argument `page_size` caught my eye:

> *page_size – maximum number of argslist items to include in every statement. If there are more items the function will execute more than one statement.*

The documentation previously stated that the function performs better because it does less roundtrips to the database. If that's the case, a larger page size should reduce the number of roundtrips, and result in a faster loading time.

Let's add an argument for page size to our function so we can experiment:

```
@profile
def insert_execute_batch_iterator(
    connection,
    beers: Iterator[Dict[str, Any]],
    page_size: int = 100,
) -> None:
    with connection.cursor() as cursor:
        create_staging_table(cursor)

        iter_beers = ({
            **beer,
            'first_brewed': parse_first_brewed(beer['first_brewed']),
            'volume': beer['volume']['value'],
        } for beer in beers)
        psycopg2.extras.execute_batch(cursor, """
            INSERT INTO staging_beers VALUES (
                %(id)s,
                %(name)s,
                %(tagline)s,
                %(first_brewed)s,
                %(description)s,
                %(image_url)s,
                %(abv)s,
                %(ibu)s,
                %(target_fg)s,
                %(target_og)s,
                %(ebc)s,
                %(srm)s,
                %(ph)s,
                %(attenuation_level)s,
                %(brewers_tips)s,
                %(contributed_by)s,
                %(volume)s
            );
        """, iter_beers, page_size=page_size)
```

The default page size is 100. Let's benchmark different values and compare the results:

```
>>> insert_execute_batch_iterator(connection, iter(beers), page_size=1)
insert_execute_batch_iterator(page_size=1)
Time   130.2
Memory 0.0

>>> insert_execute_batch_iterator(connection, iter(beers), page_size=100)
insert_execute_batch_iterator(page_size=100)
Time   4.333
Memory 0.0

>>> insert_execute_batch_iterator(connection, iter(beers), page_size=1000)
insert_execute_batch_iterator(page_size=1000)
Time   2.537
Memory 0.2265625

>>> insert_execute_batch_iterator(connection, iter(beers), page_size=10000)
```

```
insert_execute_batch_iterator(page_size=10000)
Time   2.585
Memory 25.4453125
```

We got some interesting results, let's break it down:

- 1: The results are similar to the results we got inserting rows one by one.
- 100: This is the default `page_size`, so the results are similar to our previous benchmark.
- 1000: The timing here is about 40% faster, and the memory is low.
- 10000: Timing is not much faster than with a page size of 1000, but the memory is significantly higher.

The results show that there is a tradeoff between memory and speed. In this case, it seems that the sweet spot is page size of 1000.

Execute Values

The gems in psycopg's documentation does not end with `execute_batch`. While strolling through the documentation, another function called `execute_values` caught my eye:

> *Execute a statement using VALUES with a sequence of parameters.*

The function `execute_values` works by generating a huge VALUES list to the query.

Let's give it a spin:

```
import psycopg2.extras

@profile
def insert_execute_values(connection, beers: Iterator[Dict[str, Any]]) -> None:
    with connection.cursor() as cursor:
        create_staging_table(cursor)
        psycopg2.extras.execute_values(cursor, """
            INSERT INTO staging_beers VALUES %s;
        """, [(
            beer['id'],
            beer['name'],
            beer['tagline'],
            parse_first_brewed(beer['first_brewed']),
            beer['description'],
            beer['image_url'],
            beer['abv'],
            beer['ibu'],
            beer['target_fg'],
            beer['target_og'],
            beer['ebc'],
            beer['srm'],
            beer['ph'],
            beer['attenuation_level'],
            beer['brewers_tips'],
            beer['contributed_by'],
            beer['volume']['value'],
        ) for beer in beers])
```

Importing beers using the function:

```
>>> insert_execute_values(connection, beers)
insert_execute_values()
Time   3.666
Memory 4.50390625
```

So right out of the box we get a slight speedup compared to `execute_batch`. However, the memory is slightly higher.

Execute Values From Iterator

Just like we did before, to reduce memory consumption we try to avoid storing data in-memory by using an iterator instead of a list:

```
@profile
def insert_execute_values_iterator(connection, beers: Iterator[Dict[str, Any]]) ->
    with connection.cursor() as cursor:
        create_staging_table(cursor)
        psycopg2.extras.execute_values(cursor, """
            INSERT INTO staging_beers VALUES %s;
        """, ((
            beer['id'],
            beer['name'],
            beer['tagline'],
            parse_first_brewed(beer['first_brewed']),
            beer['description'],
            beer['image_url'],
            beer['abv'],
            beer['ibu'],
            beer['target_fg'],
            beer['target_og'],
            beer['ebc'],
            beer['srm'],
            beer['ph'],
            beer['attenuation_level'],
            beer['brewers_tips'],
            beer['contributed_by'],
            beer['volume']['value'],
        ) for beer in beers))
```

Executing the function produced the following results:

```
>>> insert_execute_values_iterator(connection, beers)
insert_execute_values_iterator()
Time   3.677
Memory 0.0
```

So the timing is almost the same, but the memory is back to zero.

Execute Values From Iterator with Page Size

Just like `execute_batch`, the function `execute_values` also accept a `page_size` argument:

```
@profile
def insert_execute_values_iterator(
    connection,
    beers: Iterator[Dict[str, Any]],
    page_size: int = 100,
) -> None:
    with connection.cursor() as cursor:
        create_staging_table(cursor)
        psycopg2.extras.execute_values(cursor, """
            INSERT INTO staging_beers VALUES %s;
        """, ((
            beer['id'],
            beer['name'],
            beer['tagline'],
            parse_first_brewed(beer['first_brewed']),
            beer['description'],
            beer['image_url'],
            beer['abv'],
            beer['ibu'],
            beer['target_fg'],
            beer['target_og'],
            beer['ebc'],
            beer['srm'],
            beer['ph'],
            beer['attenuation_level'],
            beer['brewers_tips'],
            beer['contributed_by'],
            beer['volume']['value'],
        ) for beer in beers), page_size=page_size)
```

Executing with different page sizes:

```
>>> insert_execute_values_iterator(connection, iter(beers), page_size=1)
insert_execute_values_iterator(page_size=1)
Time   127.4
Memory 0.0

>>> insert_execute_values_iterator(connection, iter(beers), page_size=100)
insert_execute_values_iterator(page_size=100)
Time   3.677
Memory 0.0

>>> insert_execute_values_iterator(connection, iter(beers), page_size=1000)
insert_execute_values_iterator(page_size=1000)
Time   1.468
Memory 0.0
```

```
>>> insert_execute_values_iterator(connection, iter(beers), page_size=10000)
insert_execute_values_iterator(page_size=10000)
Time   1.503
Memory 2.25
```

Just like `execute_batch`, we see a tradeoff between memory and speed. Here as well, the sweet spot is around page size 1000. However, using `execute_values` we got results ~20% faster compared to the same page size using `execute_batch`.

Copy

The official documentation for PostgreSQL features an entire section on Populating a Database. According to the documentation, the best way to load data into a database is using the `copy` command.

To use `copy` from Python, psycopg provides a special function called `copy_from`. The `copy` command requires a CSV file. Let's see if we can transform our data into CSV, and load it into the database using `copy_from`:

```
import io

def clean_csv_value(value: Optional[Any]) -> str:
    if value is None:
        return r'\N'
    return str(value).replace('\n', '\\n')

@profile
def copy_stringio(connection, beers: Iterator[Dict[str, Any]]) -> None:
    with connection.cursor() as cursor:
        create_staging_table(cursor)
        csv_file_like_object = io.StringIO()
        for beer in beers:
            csv_file_like_object.write('|'.join(map(clean_csv_value, (
                beer['id'],
                beer['name'],
                beer['tagline'],
                parse_first_brewed(beer['first_brewed']),
                beer['description'],
                beer['image_url'],
                beer['abv'],
                beer['ibu'],
                beer['target_fg'],
                beer['target_og'],
                beer['ebc'],
                beer['srm'],
                beer['ph'],
                beer['attenuation_level'],
                beer['contributed_by'],
                beer['brewers_tips'],
                beer['volume']['value'],
            ))) + '\n')
        csv_file_like_object.seek(0)
        cursor.copy_from(csv_file_like_object, 'staging_beers', sep='|')
```

Let's break it down:

- `clean_csv_value`: Transforms a single value
  - **Escape new lines**: some of the text fields include newlines, so we escape `\n` → `\\n`.
  - **Empty values are transformed to `\N`**: The string `"\N"` is the default string used by PostgreSQL to indicate NULL in COPY (this can be changed using the `NULL` option).
- `csv_file_like_object`: Generate a file like object using `io.StringIO`. A `StringIO` object contains a string which can be used like a file. In our case, a CSV file.
- `csv_file_like_object.write`: Transform a beer to a CSV row
  - **Transform the data**: transformations on `first_brewed` and `volume` are performed here.
  - **Pick a delimiter**: Some of the fields in the dataset contain free text with commas. To prevent conflicts, we pick "|" as the delimiter (another option is to use `QUOTE`).

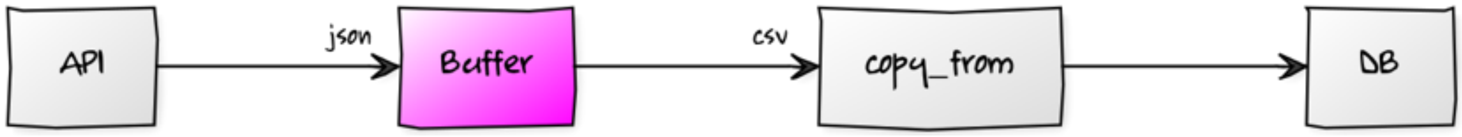
Now let's see if all of this hard work paid off:

```
>>> copy_stringio(connection, beers)
copy_stringio()
Time   0.6274
Memory 99.109375
```

The `copy` command is the fastest we've seen so far! Using `COPY`, the process completed in less than a second. However, it seems like this method is a lot more wasteful in terms of memory usage. The function consumes 99MB, which is more than twice the size of our JSON file on disk.

Copy Data From a String Iterator

One of the main drawbacks of using copy with `StringIO` is that the entire file is created in-memory. What if instead of creating the entire file in-memory, we create a file-like object that will act as a buffer between the remote source and the `COPY` command. The buffer will consume JSON via the iterator, clean and transform the data, and output clean CSV.



Copy Data From a String Iterator (yuml.me)

Inspired by this stack overflow answer, we created an object that feeds off an iterator, and provides a file-like interface:

```python
from typing import Iterator, Optional
import io

class StringIteratorIO(io.TextIOBase):
    def __init__(self, iter: Iterator[str]):
        self._iter = iter
        self._buff = ''

    def readable(self) -> bool:
        return True

    def _read1(self, n: Optional[int] = None) -> str:
        while not self._buff:
            try:
                self._buff = next(self._iter)
            except StopIteration:
                break
        ret = self._buff[:n]
        self._buff = self._buff[len(ret):]
        return ret

    def read(self, n: Optional[int] = None) -> str:
        line = []
        if n is None or n < 0:
            while True:
                m = self._read1()
                if not m:
                    break
                line.append(m)
        else:
            while n > 0:
                m = self._read1(n)
                if not m:
                    break
                n -= len(m)
                line.append(m)
        return ''.join(line)
```

To demonstrate how this works, this is how a CSV file-like object can be generated from a list of numbers:

```
>>> gen = (f'{i},{i**2}\n' for i in range(3))
>>> gen
<generator object <genexpr> at 0x7f58bde7f5e8>
>>> f = StringIteratorIO(gen)
>>> print(f.read())
0,0
1,1
2,4
```

Notice that we used `f` like a file. Internally, it fetched the rows from `gen` only when its internal line buffer was empty.

The loading function using `StringIteratorIO` looks like this:

```
@profile
def copy_string_iterator(connection, beers: Iterator[Dict[str, Any]]) -> None:
    with connection.cursor() as cursor:
        create_staging_table(cursor)
        beers_string_iterator = StringIteratorIO((
            '|'.join(map(clean_csv_value, (
                beer['id'],
                beer['name'],
                beer['tagline'],
                parse_first_brewed(beer['first_brewed']).isoformat(),
                beer['description'],
                beer['image_url'],
                beer['abv'],
                beer['ibu'],
                beer['target_fg'],
                beer['target_og'],
                beer['ebc'],
                beer['srm'],
                beer['ph'],
                beer['attenuation_level'],
                beer['brewers_tips'],
                beer['contributed_by'],
                beer['volume']['value'],
            ))) + '\n'
            for beer in beers
        ))
        cursor.copy_from(beers_string_iterator, 'staging_beers', sep='|')
```

The main difference is that the beers CSV file is consumed on demand, and the data is not stored in-memory after it was used.

Let's execute the function and see the results:

```
>>> copy_string_iterator(connection, beers)
copy_string_iterator()
Time   0.4596
Memory 0.0
```

Great! Timing is low and memory is back to zero.

Copy Data From a String Iterator with Buffer Size

In an attempt to squeeze one final drop of performance, we notice that just like `page_size`, the `copy` command also accepts a similar argument called `size`:

size – size of the buffer used to read from the file.

Let's add a `size` argument to the function:

```
@profile
def copy_string_iterator(connection, beers: Iterator[Dict[str, Any]], size: int = 8
    with connection.cursor() as cursor:
        create_staging_table(cursor)
        beers_string_iterator = StringIteratorIO((
            '|'.join(map(clean_csv_value, (
                beer['id'],
                beer['name'],
                beer['tagline'],
                parse_first_brewed(beer['first_brewed']).isoformat(),
                beer['description'],
                beer['image_url'],
                beer['abv'],
                beer['ibu'],
                beer['target_fg'],
                beer['target_og'],
                beer['ebc'],
                beer['srm'],
                beer['ph'],
                beer['attenuation_level'],
                beer['brewers_tips'],
                beer['contributed_by'],
                beer['volume']['value'],
            ))) + '\n'
            for beer in beers
        ))
        cursor.copy_from(beers_string_iterator, 'staging_beers', sep='|', size=size
```

The default value for size is 8192, which is `2 ** 13`, so we will keep sizes in powers of 2:

```
>>> copy_string_iterator(connection, iter(beers), size=1024)
copy_string_iterator(size=1024)
Time   0.4536
Memory 0.0
```

```
>>> copy_string_iterator(connection, iter(beers), size=8192)
copy_string_iterator(size=8192)
Time   0.4596
Memory 0.0

>>> copy_string_iterator(connection, iter(beers), size=16384)
copy_string_iterator(size=16384)
Time   0.4649
Memory 0.0

>>> copy_string_iterator(connection, iter(beers), size=65536)
copy_string_iterator(size=65536)
Time   0.6171
Memory 0.0
```

Unlike the previous examples, it seems like there is no tradeoff between speed and memory. This makes sense because this method was designed to consume no memory. However, we do get different timing when changing the page size. For our dataset, the default 8192 is the sweet spot.

Results Summary

A summary of the results:

FUNCTION	TIME (SECONDS)	MEMORY (MB)
`insert_one_by_one()`	128.8	0.08203125
`insert_executemany()`	124.7	2.765625
`insert_executemany_iterator()`	129.3	0.0
`insert_execute_batch()`	3.917	2.50390625
`insert_execute_batch_iterator(page_size=1)`	130.2	0.0
`insert_execute_batch_iterator(page_size=100)`	4.333	0.0
`insert_execute_batch_iterator(page_size=1000)`	2.537	0.2265625
`insert_execute_batch_iterator(page_size=10000)`	2.585	25.4453125
`insert_execute_values()`	3.666	4.50390625
`insert_execute_values_iterator(page_size=1)`	127.4	0.0

`insert_execute_values_iterator(page_size=100)`	3.677	0.0
`insert_execute_values_iterator(page_size=1000)`	1.468	0.0
`insert_execute_values_iterator(page_size=10000)`	1.503	2.25
`copy_stringio()`	0.6274	99.109375
`copy_string_iterator(size=1024)`	0.4536	0.0
`copy_string_iterator(size=8192)`	0.4596	0.0
`copy_string_iterator(size=16384)`	0.4649	0.0
`copy_string_iterator(size=65536)`	0.6171	0.0

· · ·

Summary

The big question now is *What should I use?* as always, the answer is *It depends*.

Each method has its own advantages and disadvantages, and is suited for different circumstances:

TAKE AWAY

Prefer built-in approaches for complex data types.

Execute many, execute values and batch take care of the conversion between Python data types to database types. CSV approaches required escaping.

TAKE AWAY

Prefer built-in approaches for small data volume.

The build-in approaches are more readable and less likely to break in the future. If memory and time is not an issue, keep it simple!

TAKE AWAY

Prefer copy approaches for large data volume.

Copy approach is more suitable for larger amounts of data where memory might become an issue.

. . .

SOURCE CODE

The source code for this benchmark can be found here.

. . .

. . .

. . .

SIMILAR ARTICLES

08 JUNE 2019 / PYTHON, DJANGO, PERFORMANCE

Improve Serialization Performance in Django Rest Framework

How we reduced serialization time by 99%!

11 JANUARY 2021 / POSTGRESQL, PERFORMANCE, SQL

Re-Introducing Hash Indexes in PostgreSQL

The Ugly Duckling of index types

20 OCTOBER 2020 / SQL, PERFORMANCE, POSTGRESQL

The Surprising Impact of Medium-Size Texts on PostgreSQL Performance

Why TOAST is the best thing since sliced bread

---

21 NOVEMBER 2019 / POSTGRESQL, SQL, PERFORMANCE

12 Common Mistakes and Missed Optimization Opportunities in SQL

Made by Developers and Non-Developers

---

22 DECEMBER 2018 / POSTGRESQL, SQL, PERFORMANCE

How We Solved a Storage Problem in PostgreSQL Without Adding a Single Byte of Storage

A short story about a storage-heavy query and the silver bullet that solved the issue

---