

# Fast CSV and JSON Ingestion in PostgreSQL with COPY

April 03, 2018 **JONATHAN S. KATZ** | [Database Administration](#)

If you have been asked to provide a CSV that someone can open up in their favorite spreadsheet editor, chances are you have used the PostgreSQL [COPY](#) command. COPY has been around since the early open source releases of PostgreSQL back in the late 1990s and was designed to quickly get data in and out of PostgreSQL.

COPY is also incredibly helpful for ingesting data into a table, especially if you have a lot of it to ingest, and will generally outperform INSERT. Let's explore a few ways to use COPY to load some data into a table.

## The Setup

We will be creating two different tables, one that will store the results from ingesting data from a CSV and another for the JSON data.

For the CSV data:

```
CREATE TABLE blog_feed (  
  id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,  
  ingested_at timestamp DEFAULT CURRENT_TIMESTAMP,  
  author text NOT NULL,  
  content text NOT NULL  
);
```

For the JSON data:

```
CREATE TABLE news_feed (  
  id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,  
  ingested_at timestamp DEFAULT CURRENT_TIMESTAMP,  
  data jsonb NOT NULL  
);
```

Note: We will be using the [JSONB](#) data type for the JSON data. JSONB stores JSON data in a binary format which enables features such as [advanced indexing](#).

Note 2: If you are not using PostgreSQL 10 or later, you can substitute the `id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY` with `id serial PRIMARY KEY`.

## Ingesting a CSV

In order to demonstrate loading a CSV with COPY, it would help to have data in a CSV to load! Fortunately, this can be solved with a simple Python script to generate some random data. In order to generate the data, we will use the [Faker](#) library, which you can install with pip or your favorite Python packaging manager.

The code below will generate 1,000,000 author/content pairs and output them in a file `data.csv`:

```
import csv
import datetime
from faker import Faker

fake = Faker()
MAX_RANGE = 1000000

with open('data.csv', 'w') as csvfile:
    field_names = ['author', 'content']
    writer = csv.DictWriter(csvfile, fieldnames=field_names)
    writer.writeheader()
    for i in range(0, MAX_RANGE):
        writer.writerow({
            'author': fake.name(),
            'content': fake.sentence(nb_words=16, variable_nb_words=True),
        })
```

You can change how many rows are generated by adjusting the **MAX\_RANGE** variable. Save this file as **generator\_csv.py** and generate the data by running:

```
python generator_csv.py
```

Generating the CSV can take a few minutes depending on how fast your system is.

Now the fun part: ingesting the CSV file. The code below will read in the CSV file and put it into the **blog\_feed** table within a database named **feeds**:

```
cat data.csv | psql -h localhost -p 5432 feeds -c "COPY blog_feed (author, content) FROM STDIN WITH (FORMAT CSV, HEADER TRUE);"
```

Substitute **feeds** with the name of the database you created the **blog\_feed** table in.

What does the above command do? The rows from the **data.csv** file are piped into a connection which then invokes the **COPY** command. We pass in options to let the **COPY** command know that we have passed in a CSV file that contains a header, and to only load the **author** and **content** columns from the CSV.

This operation should run relatively quickly: within a matter of seconds you will have all 1,000,000 rows loaded. If you inspect the table, you will see that the data from the CSV made it into the table:

```
feeds=# SELECT * FROM blog_feed LIMIT 10 OFFSET 1000;
```

id	ingested_at	author	content
1001	2018-04-02 10:20:10.048562	Kevin Barron	Feeling without really here hot popular order think law all main behind hold production significant yes dark blue.
1002	2018-04-02 10:20:10.048562	Rodney Robinson	Get security seven finally minute she name majority anyone.
1003	2018-04-02 10:20:10.048562	Ryan Jones	Build professor exist six market soon yes car dream area government test benefit agent return space good style weight.
1004	2018-04-02 10:20:10.048562	Kristin Burke	Media likely know sound clear score perform buy child section blue save.
1005	2018-04-02 10:20:10.048562	Haley Bowman	Control become represent west his during until article middle reason agreement television real risk kind form these politics control begin white everything.
1006	2018-04-02 10:20:10.048562	Thomas Young	Evidence mouth ask picture time issue safe floor phone hot full.
1007	2018-04-02 10:20:10.048562	Edward Bates	Record system each appear most onto environment until sea likely particularly red seem.
1008	2018-04-02 10:20:10.048562	Angela Reyes	Partner job beat social point western address especially relate interesting alone.
1009	2018-04-02 10:20:10.048562	Michelle Glenn	Sound already save glass particular certain idea course cause agent onto improve exist lose physical star little note officer you director someone.
1010	2018-04-02 10:20:10.048562	James Walker Jr.	Physical word course high product best girl blood west continue environmental major hand.

Now you have access to all the wonderful features of PostgreSQL, such as finding out the top 10 authors that appeared in this blog feed:

```
feeds=# SELECT author,count(*) FROM blog_feed GROUP BY author ORDER BY count DESC LIMIT 10;
```

author	count
Michael Smith	455
David Smith	354
Michael Johnson	337
Michael Williams	317
James Smith	313
John Smith	312
Jennifer Smith	305
Christopher Smith	291
Robert Smith	284
Michael Jones	260

## Ingesting JSON Data

Now let’s perform the same exercise but for JSON data. We will be ingesting the JSON data into the “news\_feed” table. First, let’s generate some JSON data that can be placed into a file named **data.json**. The script below is similar to the CSV generation script: tweak the number of JSON objects generated by adjusting the value in **MAX\_RANGE**:

```
import csv

import datetime

import json

import uuid

from faker import Faker


fake = Faker()

MAX_RANGE = 1000000

datetime_end = datetime.datetime.now()

datetime_start = datetime_end - datetime.timedelta(days=365)


# generate JSON

with open('data.json', 'w') as f:
    for i in range(0, MAX_RANGE):
        f.write(json.dumps({
            'id': str(uuid.uuid4()),
            'author': fake.name(),
            'content': fake.sentence(nb_words=16, variable_nb_words=True),
            'source': fake.company(),
            'published_at': fake.date_time_between_dates(
                datetime_start=datetime_start,
                datetime_end=datetime_end,
            ).isoformat()
        }) + "\n")
```

```
python generator_json.py
```

```
cat data.json | psql -h localhost -p 5432 feeds -c "COPY news_feed (data) FROM STDIN;"
```

```
feeds=# SELECT * FROM news_feed LIMIT 2 OFFSET 1000;
```

id	ingested_at	data
1953	2018-04-02 10:44:59.838267	{"id": "17b70258-d2e6-4ae5-aa29-89e6568899c1", "author": "Civil stock current executive top stand effect near claim tell final east music option officer thing.", "source": "Williams-Pacheco", "content": "Foot tough bad meet third particularly share western expert air detail reality read option wind too while hotel your interest.", "published_at": "2018-02-07T19:19:56"}
1954	2018-04-02 10:44:59.838267	{"id": "5614ab36-3f8d-4382-b92f-3038b6bcce88", "author": "Spring worker book example your others wrong order better lose hard whatever well morning official.", "source": "Summers, Hall and Harrell", "content": "Space something six east happy impact dinner opportunity region.", "published_at": "2017-07-16T01:47:02"}

```
feeds=# SELECT data->>'author' AS author, count(*)
FROM news_feed
GROUP BY author
ORDER BY count DESC LIMIT 10;
```

author	count
Michael Smith	475
Michael Johnson	390
Jennifer Smith	321
John Smith	310
Michael Williams	308
James Smith	302
David Smith	299
Christopher Smith	294
Michael Brown	283
Robert Smith	278

## Bonus: Inspecting JSON Documents

PostgreSQL 9.4 introduced both the JSONB data type as well as the tools to quickly search for data stored within a JSON document via [GIN](#) indexes. To create a GIN index on the news\_feed column, execute the following command:

```
CREATE INDEX news_feed_data_gin_idx ON news_feed USING GIN(data);
```

This enables you to perform several key JSONB operations, such searching for elements inside the document. For example, let’s say I know there is a document with a [UUID](#) of `81865b56-4b76-4d33-86fe-660f96861ea0` and I want to pull all of its contents. I can do so quickly with the following query:

```
SELECT * FROM news_feed WHERE data @> '{ "id": "81865b56-4b76-4d33-86fe-660f96861ea0" }';
```

There are many other things you can do with JSON objects with PostgreSQL, including interacting with them via [procedural languages](#). Now you have the tools to ingest JSON data quickly and manipulate it!