DataEng: Data Storage Activity

This week you'll gain experience with various ways to load data into a Postgres database.

Make a copy of this document and use it to record your results. Store a PDF copy of the document in your git repository along with any needed code before submitting for this week.

The data set for this week is US Census data from 2015 ~~and 2017~~. The United States conducts a full census of every household every 10 years (we just finished one last year), but much of the detailed census data comes during the intervening years when the Census Bureau conducts its detailed American Community Survey (ACS) of a randomly selected sample of approximately 3.5 million households each year. The resulting data gives a more detailed view of many factors of American life and the composition of households.

> ACS Census Tract Data for 2015
> ~~ACS Census Tract Data for 2017~~ (ignore the 2017 data)

Your job is to load the 2015 data set (approximately 74000 rows each) into your database. You'll configure a postgres DBMS on a new GCP virtual machine, and then load the data five different ways, comparing the cost of each method.

We hope that you make it all the way through to the end, but regardless, use your time wisely to gain python programming experience and learn as much as you can about bulk loading of data. Note that the goal here is not to achieve the fastest load times. Instead, your goal should be to gain knowledge about how a data storage server (such as PostgreSQL) works and why various data loading approaches produce differing performance results.

Submit: In-class Activity Submission Form

A. Configure Your Database

1. Create a new GCP virtual machine for this week's work (medium size or larger).
2. Follow the steps listed in the Installing and Configuring and PostgreSQL server instructions provided for project assignment #2. To keep things separate from your project work we suggest you use a separate vm, separate database name, separate user name, etc. Then each/all of these can then be updated or deleted whenever you need without affecting your project.
3. Also the following commands will help to configure the python module "psycopg2" which you will use to connect to your postgres database:

```
sudo apt install python3 python3-dev python3-venv
sudo apt-get install python3-pip
pip3 install psycopg2-binary
```

B. Connect to Database and Create Your Main Data Table

1. Copy/upload your data to your VM and uncompress it
2. Create a small test sample by running a linux command similar to the following. The small test sample will help you to quickly test any code that you write before running your code with the full dataset.

   ```
   head -1 acs2015_census_tract_data.csv > Oregon2015.csv
   grep Oregon acs2015_census_tract_data.csv >> Oregon2015.csv
   ```

   The first command copies the headers to the sample file and the second command appends all of the Oregon 2015 data to the sample file. This should produce a file with approximately 800 records which is a bit more than 1% of the 2015 data set.

3. Write a python program that connects to your postgres database and creates your main census data table. Start with this example code: load_inserts.py. If you want to run load_inserts.py, then run it with -d <data file> -c -y 2015 Note that you must input a year value because the census data does not explicitly include the calendar year within each data file.

C. Baseline - Simple INSERT

The tried and true SQL command INSERT INTO ... is the most basic way to insert data into a SQL database, and often it is the best choice for small amounts of data, production databases and other situations in which you need to maintain performance and reliability of the updated table.

The load_inserts.py program shows how to use simple INSERTs to load data into a database. It is possibly the slowest way to load large amounts of data. For me, it takes approximately 1 second for the Oregon sample and nearly 120 seconds for the full acs 2015 data set.

Take the program and try it with both the Oregon sample and the full data sets. Fill in the appropriate table rows below.

D. The Effects of Indexes and Constraints

You might notice that the CensusData table has a composite Primary Key constraint and an additional index on the state name column. Indexes and constraints are helpful for query performance, but do not work well for load performance.

Try delaying the creation of these constraints/indexes until after the data set is loaded. Enter the resulting load time into the results table. Did this technique improve load performance?

Answer: Yes, this technique did improve load performance. Although, the improvement in performance was not too much. Load time went from 92.47s to 87.77s.

E. The Effects of Logging

By default, RDBMS tables incur overheads of write-ahead logging (WAL) such that the database logs extra metadata about each update to the table and uses that WAL data to recover the contents of the table in case of RDBMS crash. This is a great feature but can get in your way when trying to bulk load data into the database.

Try loading to a “staging” table, a table that is declared as UNLOGGED. This staging table should have no constraints or indexes. Then use a SQL query to append the staging data to the main CensusData table.

By the way, you might have noticed that the load_inserts.py program sets autocommit=True on the database connection. This makes loaded data available to DB queries immediately after each insert. But it also implies a great amount of transaction overhead. It also allows readers of the database to view an incomplete set of data during the load. How does load performance change if you do not set autocommit=True and instead explicitly commit all of the loaded data within a transaction?

Answer: Load performance improves significantly and load time goes down to 12.06s from 16s when auto-commit is turned off and the changes are committed as a single transaction at the end (i.e. only 1 commit made).

F. Temp Tables and Memory Tuning

Next compare the above approach with loading the data to a temporary table (and copying from the temporary table to the CensusData table). Which approach works best for you.

The amount of memory used for temporary tables is default configured to only 8MB. Your VM has enough memory to allocate much more memory to temporary tables. Try

allocating 256 MB (or more) to temporary tables. So update the temp_buffers parameter to allow the database to use more memory for your temporary table. Rerun your load experiments. Did it make a difference?

Answer:

Loading the data into an unlogged table and then loading it to the main CensusData table with no auto-commit (which took 12.06 s) was far more performant than the temporary table approach (which took 16.05s even after memory tuning).

No memory tuning- 16.15 s

Memory tuning:
256MB - 16.05 s
750MB - 15.99 s
1500MB - 16.12 s

Updating the temp_buffers parameter to three different values as shown above didn't really make any difference at all.

G. Batching

So far our load performance has been held back by the fact we are using individual calls to the DBMS. As with many Computer Systems situations we can improve performance by batching operations. Haki Benita's great article about fast loading to Postgres notes that use of psycopg2's execute_batch() method can increase load rate by up to two orders of magnitude. The blog provides sample code, time measurements and memory measurements. Adapt his code to your case, rerun your experiments and note your results in the table below.

H. Built In Facility (copy_from)

The number one rule of bulk loading is to pay attention to the native facilities provided by the DBMS system implementers. As we saw with Joyo Victor's presentation last week, the DBMS vendors often put great effort into providing purpose-build loading mechanisms that achieve great speed and scalability.

With a simple, one-server Postgres database, that facility is known as COPY, \copy, or for python programmers copy_from. Haki Benita's blog shows how to use copy_from to

achieve another order of magnitude in load performance. Adapt Haki's code to your case, rerun your experiments and note your results in the table.

I. Results

Use this table to present your results. We are not asking you to do a sophisticated performance analysis here with multiple runs, warmup time, etc. Instead, do a rough measurement using timing code similar to what you see in the load_inserts.py code. Record your results in the following table.

Method	**Code Link**	**acs2015**
Simple inserts	load_inserts.py	92.47 seconds
Drop Indexes and Constraints	data-engineering/load_inserts_drop_constraints.py at main · pkaran57/data-engineering (github.com)	87.77 seconds
Use UNLOGGED table	Auto-commit on: https://github.com/pkaran57/data-engineering/blob/main/in-class-assignments/assignment-4/src/load_inserts_to_unlogged_table.py Auto-commit off: https://github.com/pkaran57/data-engineering/blob/main/in-class-assignments/assignment-4/src/load_inserts_to_unlogged_table-autocommit-off.py	With auto-commit on: 16 seconds With auto-commit off / explicit commit at the end of transaction: 12.06 seconds
Temp Table with memory tuning	Temp table with no memory tuning: https://github.com/pkaran57/data-engineering/blob/main/in-class-assignments/assignment-4/src/load_inserts_to_te	Temp table with no memory tuning: 16.15 seconds Temp table with memory tuning:

	mporary_table.py Temp table with memory tuning: https://github.com/pkaran57/data-engineering/blob/main/in-class-assignments/assignment-4/src/load_inserts_to_temporary_table-increased-temp_buffers.py	256MB - 16.05s 750MB - 15.99s 1500MB - 16.12s
Batching	https://github.com/pkaran57/data-engineering/blob/main/in-class-assignments/assignment-4/src/load_inserts_using_execute_batch.py	11.46 seconds
copy_from	https://github.com/pkaran57/data-engineering/blob/main/in-class-assignments/assignment-4/src/load_inserts_using_copy_from.py	1.304 seconds

J. Observations

Use this section to record any observations about the various methods/techniques that you used for bulk loading of the USA Census data. Did you learn anything about how an RDBMS functions and why various loading approaches produce varying performance results?

Answer:

Yes, I learned quite a few things about loading data into a relational database and got a sense of why performance differs between different loading methods.

In general, it seems like dropping constraints and indices on a given table to which data is to be loaded will speed up the loading process since the database does not have to perform any validations on the incoming data. However, dropping constraints and indices have their own downsides.

Bulk-loading data into an unlogged table also significantly improves the data loading performance since the database does not have to be resilient to failures. Once data is loaded to an unlogged table, it can be quickly loaded into the original table. Writing data from staging table to permanent table should be quick since all the data in the staging table is already in memory. I learned that databases have logging features to prevent data loss in case of a database crash.

I saw a decent data loading performance gain when auto-commit was turned off and data was committed in a single transaction. The improvement in performance was much more than I expected. Committing all data in a single transaction is what batching seems to be doing.

I was extremely surprised to see how quick and easy to use postgres's COPY command was (https://www.postgresql.org/docs/9.2/sql-copy.html). In the future, I will always look for built-in methods of loading bulk data. In terms of an analogy, why reinvent the wheel when someone has already spent a lot of time making one that is quick, easy and highly optimized?

Submit: In-class Activity Submission Form