DataEng S23: Data Storage In-class Assignment

This week you'll gain experience with various ways to load data into a 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. The United States conducts a full census of every household every 10 years (last one was in 2020), 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 (part 1)
> ACS Census Tract Data for 2015 (part 2)

Your job is to load the 2015 data set (approximately 74000 rows divided into two parts). You'll configure a postgres DBMS on a new GCP virtual machine, and then load the two parts of the data five different ways, comparing the cost of each method. Load the two parts of the data separately so that you gain experience with the issues related to incremental loading of data.

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 observe and understand a variety of data loading methods. If you start to run out of time, then skip to part I (copy_from) to be sure to get experience with what is probably the fastest way to load bulk data into a Postgres server.

Discussion Question (discuss as a group near the beginning of the week and note the various responses in this space):

> *Do you have any experience with ingesting bulk data into a database? If yes, then describe your experience, especially what method was used to input the data into the database. If no, then describe how you might ingest daily incremental breadcrumb data for your class project.*

My response: I have experience with ingesting bulk data into a database. I inserted around 50k records from a csv file into the database as part of my project for the DBMS course. We used the Postgres UI for importing the data from a csv file into the database table.

Kavitha Jajula: Yes, I do have experience in inserting bulk data in to database. When I was doing the project Employee Database Management System for Introduction to Database course, I have inserted around 2k records in to Database. This was done through \copy command from command line to copy the data from csv file to table. It was pretty fast.

Shatabdi Pal : Yes, I have experience with bulk data loading into a database. In the Intro database management course, I used PgAdmin UI for inserting data into postgres database schema from a csv file. For project work we used python script for inserting data into tables. In this project, we used health data which had 12 csv files (each having 120 rows and 10 columns.)

Shreshtha Siddhi: Yes I have experience of ingesting bulk data into a database. In my final project for Intro to DBMS course, I have used python script to perform data loading in Postgres database. I added rows via python command to all the table schemas related to Movies data with about 8000 records.

Submit: By this Friday at 10pm submit your assignment using the in-class assignment submission form found on the Materials page on the class website.

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 a 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_part1.csv > Oregon2015.csv
   grep Oregon acs2015_census_tract_data_part1.csv >> Oregon2015.csv
   grep Oregon acs2015_census_tract_data_part2.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. Use this sample file to save time during testing.
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

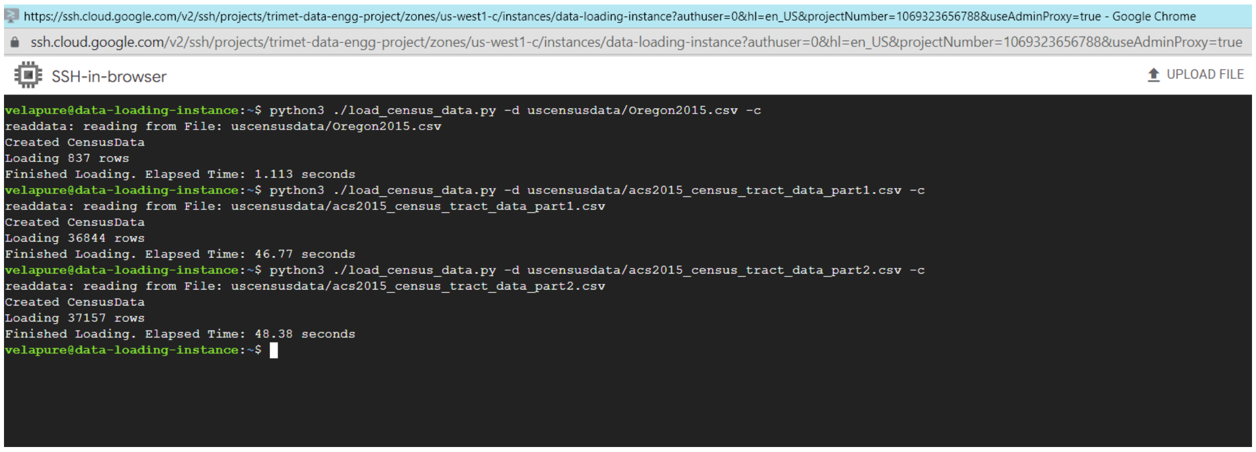
C. Baseline - Simple INSERT

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 60 seconds to load each part of the acs data.

Take the program and try it with both the Oregon sample and both parts of the ACS data set. Fill in the appropriate information in the table below.

Below is the screenshot of loading the sample, part1 and part2 data:

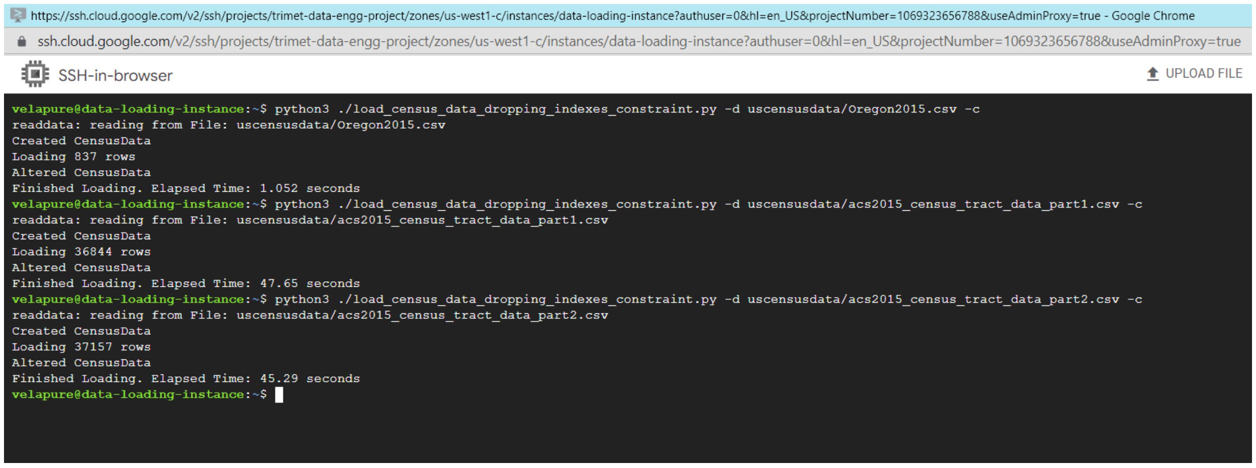

https://ssh.cloud.google.com/v2/ssh/projects/trimet-data-engg-project/zones/us-west1-c/instances/data-loading-instance?authuser=0&hl=en_US&projectNumber=1069323656788&useAdminProxy=true - Google Chrome
ssh.cloud.google.com/v2/ssh/projects/trimet-data-engg-project/zones/us-west1-c/instances/data-loading-instance?authuser=0&hl=en_US&projectNumber=1069323656788&useAdminProxy=true
SSH-in-browser
UPLOAD FILE

```
velapure@data-loading-instance:~$ python3 ./load_census_data.py -d uscensusdata/Oregon2015.csv -c
readdata: reading from File: uscensusdata/Oregon2015.csv
Created CensusData
Loading 837 rows
Finished Loading. Elapsed Time: 1.113 seconds
velapure@data-loading-instance:~$ python3 ./load_census_data.py -d uscensusdata/acs2015_census_tract_data_part1.csv -c
readdata: reading from File: uscensusdata/acs2015_census_tract_data_part1.csv
Created CensusData
Loading 36844 rows
Finished Loading. Elapsed Time: 46.77 seconds
velapure@data-loading-instance:~$ python3 ./load_census_data.py -d uscensusdata/acs2015_census_tract_data_part2.csv -c
readdata: reading from File: uscensusdata/acs2015_census_tract_data_part2.csv
Created CensusData
Loading 37157 rows
Finished Loading. Elapsed Time: 48.38 seconds
velapure@data-loading-instance:~$
```

D. Disabling Indexes and Constraints

You might notice that the CensusData table has a Primary Key constraint and an additional index on the state name column. Indexes and constraints are helpful for query performance but these features can slow down 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?



https://ssh.cloud.google.com/v2/ssh/projects/trimet-data-engg-project/zones/us-west1-c/instances/data-loading-instance?authuser=0&hl=en_US&projectNumber=1069323656788&useAdminProxy=true - Google Chrome

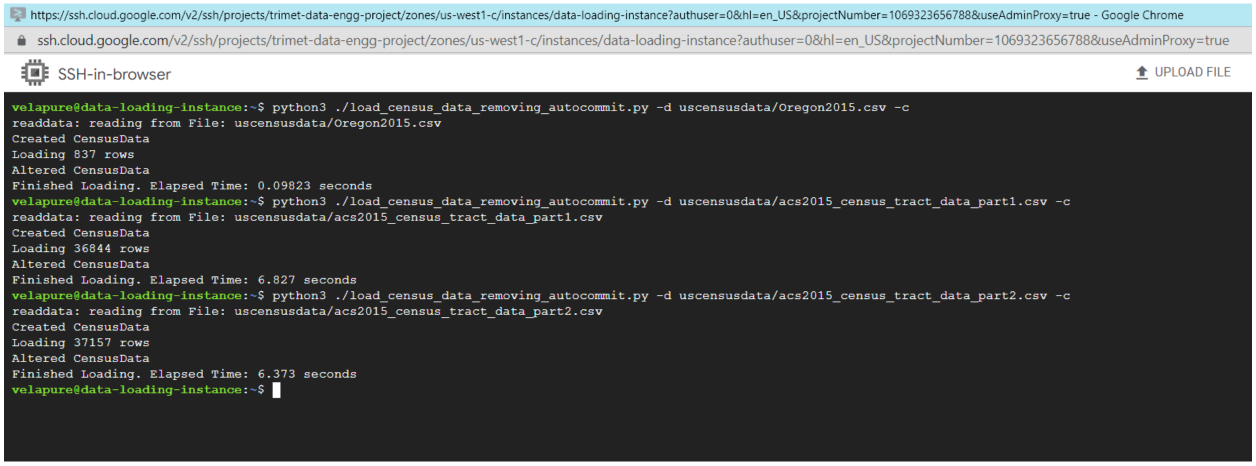
ssh.cloud.google.com/v2/ssh/projects/trimet-data-engg-project/zones/us-west1-c/instances/data-loading-instance?authuser=0&hl=en_US&projectNumber=1069323656788&useAdminProxy=true

```
velapure@data-loading-instance:~$ python3 ./load_census_data_dropping_indexes_constraint.py -d uscensusdata/Oregon2015.csv -c
readdata: reading from File: uscensusdata/Oregon2015.csv
Created CensusData
Loading 837 rows
Altered CensusData
Finished Loading. Elapsed Time: 1.052 seconds
velapure@data-loading-instance:~$ python3 ./load_census_data_dropping_indexes_constraint.py -d uscensusdata/acs2015_census_tract_data_part1.csv -c
readdata: reading from File: uscensusdata/acs2015_census_tract_data_part1.csv
Created CensusData
Loading 36844 rows
Altered CensusData
Finished Loading. Elapsed Time: 47.65 seconds
velapure@data-loading-instance:~$ python3 ./load_census_data_dropping_indexes_constraint.py -d uscensusdata/acs2015_census_tract_data_part2.csv -c
readdata: reading from File: uscensusdata/acs2015_census_tract_data_part2.csv
Created CensusData
Loading 37157 rows
Altered CensusData
Finished Loading. Elapsed Time: 45.29 seconds
velapure@data-loading-instance:~$
```

This technique did not increase the load performance for part1 data but it did increase the load performance for part2 data.

E. Disabling Autocommit

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 triggers transaction-related overhead operations. 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 single transaction?



https://ssh.cloud.google.com/v2/ssh/projects/trimet-data-engg-project/zones/us-west1-c/instances/data-loading-instance?authuser=0&hl=en_US&projectNumber=1069323656788&useAdminProxy=true - Google Chrome
ssh.cloud.google.com/v2/ssh/projects/trimet-data-engg-project/zones/us-west1-c/instances/data-loading-instance?authuser=0&hl=en_US&projectNumber=1069323656788&useAdminProxy=true
SSH-in-browser
UPLOAD FILE

```
velapure@data-loading-instance:~$ python3 ./load_census_data_removing_autocommit.py -d uscensusdata/Oregon2015.csv -c
readdata: reading from File: uscensusdata/Oregon2015.csv
Created CensusData
Loading 837 rows
Altered CensusData
Finished Loading. Elapsed Time: 0.09823 seconds
velapure@data-loading-instance:~$ python3 ./load_census_data_removing_autocommit.py -d uscensusdata/acs2015_census_tract_data_part1.csv -c
readdata: reading from File: uscensusdata/acs2015_census_tract_data_part1.csv
Created CensusData
Loading 36844 rows
Altered CensusData
Finished Loading. Elapsed Time: 6.827 seconds
velapure@data-loading-instance:~$ python3 ./load_census_data_removing_autocommit.py -d uscensusdata/acs2015_census_tract_data_part2.csv -c
readdata: reading from File: uscensusdata/acs2015_census_tract_data_part2.csv
Created CensusData
Loading 37157 rows
Altered CensusData
Finished Loading. Elapsed Time: 6.373 seconds
velapure@data-loading-instance:~$ 
```

The load performance improved drastically by removing autocommit from the connection setting and committing the data after inserting data and adding indexes and constraints.

F. UNLOGGED table

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. Crash Recovery is a great feature but it can slow down load performance.

Try loading part1 ACS data 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. Then create the needed index and constraint. Then do each of the operations again with part2 of the data.

G. Temp Tables and Memory Tuning

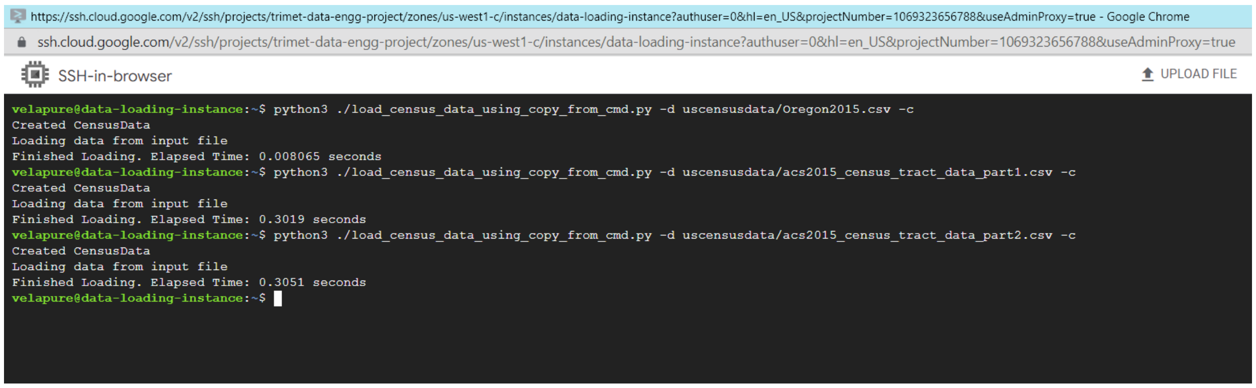
Next compare the above approach with loading the data to a temporary table (and copying from the temporary table to the CensusData table). Which approach works best for you?

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?

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. DBMS vendors often put great effort into providing purpose-built loading mechanisms that achieve high performance 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.


https://ssh.cloud.google.com/v2/ssh/projects/trimet-data-engg-project/zones/us-west1-c/instances/data-loading-instance?authuser=0&hl=en_US&projectNumber=1069323656788&useAdminProxy=true - Google Chrome
ssh.cloud.google.com/v2/ssh/projects/trimet-data-engg-project/zones/us-west1-c/instances/data-loading-instance?authuser=0&hl=en_US&projectNumber=1069323656788&useAdminProxy=true
SSH-in-browser
UPLOAD FILE

```
velapure@data-loading-instance:~$ python3 ./load_census_data_using_copy_from_cmd.py -d uscensusdata/Oregon2015.csv -c
Created CensusData
Loading data from input file
Finished Loading. Elapsed Time: 0.008065 seconds
velapure@data-loading-instance:~$ python3 ./load_census_data_using_copy_from_cmd.py -d uscensusdata/acs2015_census_tract_data_part1.csv -c
Created CensusData
Loading data from input file
Finished Loading. Elapsed Time: 0.3019 seconds
velapure@data-loading-instance:~$ python3 ./load_census_data_using_copy_from_cmd.py -d uscensusdata/acs2015_census_tract_data_part2.csv -c
Created CensusData
Loading data from input file
Finished Loading. Elapsed Time: 0.3051 seconds
velapure@data-loading-instance:~$
```

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	**Time to load part1**	**Time to load part2**
C. Simple inserts	46.77 seconds	48.38 seconds
D. Drop Indexes and Constraints	47.65 seconds	45.29 seconds
E. Disable Autocommit	6.827 seconds	6.373 seconds
F. Use UNLOGGED table		
G. Temp Table with memory tuning		
H. copy_from	0.3019 seconds	0.3051 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 why various loading approaches produce varying performance results?
Answer: I learnt different ways to load the data into the database and how each method gives a different load performance. I observed that using the DB built-in functionality such as the '\COPY' command for Postgres is very efficient and has a very good load performance. These methods of loading the data have varying load performance because the number of functions/tasks performed by DB internally and the way they are performed are different for each method.