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 H (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*.
>
> Response 1 –
>
> Yes, I have had a brief experience with ingesting data in a HMS (Hospital Management Setting) setting, where the data provided by the Hospitals were centrally stored in a primary database, from where, it was consumed by the development teams to build a software system to consume, analyze and transfer the data to the next section. It was based on C# - PostgreSQL.
>
> Response 2 –

I've worked on bulk data ingesting into databases before. I had to weekly insert customer information from an Excel file into a MySQL database while working on a customer relationship management project.
Before ingesting the data, I used to perform preprocessing each time. This included eliminating duplicates, handling missing values, and making sure the data is formatted correctly for the database schema to prevent any errors. I developed a function that converts an Excel file submitted by a user on the front end into a CSV file. I passed the CSV file to a function that establishes a database connection and inserts the data into a table. To insert the data into the table, I used the "load data infile" query.

Response 3 –

I have uploaded max 1.3 Gb of csv data to Microsoft SQL server and that was just a simple import from csv statement.

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.

```
upasana@instance-1:~$ python3 load_inserts_C.py -d acs2015_census_tract_data_part1.csv -c
readdata: reading from File: acs2015_census_tract_data_part1.csv
Created CensusData
Loading 36844 rows
Finished Loading. Elapsed Time: 40.51 seconds
upasana@instance-1:~$ python3 load_inserts_C.py -d acs2015_census_tract_data_part2.csv -c
readdata: reading from File: acs2015_census_tract_data_part2.csv
Created CensusData
Loading 37157 rows
Finished Loading. Elapsed Time: 41.37 seconds
upasana@instance-1:~$ python3 load_inserts_C.py -d Oregon2015.csv -c
readdata: reading from File: Oregon2015.csv
Created CensusData
Loading 837 rows
Finished Loading. Elapsed Time: 0.8527 seconds
upasana@instance-1:~$
```

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?

```
upasana@instance-1:~$ python3 load_inserts_D.py -d Oregon2015.csv -c
readdata: reading from File: Oregon2015.csv
Created CensusData
Loading 837 rows
Finished Loading. Elapsed Time: 0.9436 seconds
upasana@instance-1:~$ python3 load_inserts_D.py -d acs2015_census_tract_data_part1.csv -c
readdata: reading from File: acs2015_census_tract_data_part1.csv
Created CensusData
Loading 36844 rows
Finished Loading. Elapsed Time: 39.33 seconds
upasana@instance-1:~$ python3 load_inserts_D.py -d acs2015_census_tract_data_part2.csv -c
readdata: reading from File: acs2015_census_tract_data_part2.csv
Created CensusData
Loading 37157 rows
Finished Loading. Elapsed Time: 42.77 seconds
upasana@instance-1:~$
```

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?

```
upasana@instance-1:~$ python3 load_inserts_E.py -d Oregon2015.csv -c
readdata: reading from File: Oregon2015.csv
Created CensusData
Loading 837 rows
Finished Loading. Elapsed Time: 0.1864 seconds
upasana@instance-1:~$ python3 load_inserts_E.py -d acs2015_census_tract_data_part1.csv -c
readdata: reading from File: acs2015_census_tract_data_part1.csv
Created CensusData
Loading 36844 rows
Finished Loading. Elapsed Time: 7.862 seconds
upasana@instance-1:~$ python3 load_inserts_E.py -d acs2015_census_tract_data_part2.csv -c
readdata: reading from File: acs2015_census_tract_data_part2.csv
Created CensusData
Loading 37157 rows
Finished Loading. Elapsed Time: 8.098 seconds
upasana@instance-1:~$
```

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.

```
upasana@instance-1:~$ python3 load_inserts_H.py -d Oregon2015.csv -c
Created CensusData
Finished Loading. Elapsed Time: 0.01068 seconds
upasana@instance-1:~$ python3 load_inserts_H.py -d acs2015_census_tract_data_part1.csv -c
Created CensusData
Finished Loading. Elapsed Time: 0.4481 seconds
upasana@instance-1:~$ python3 load_inserts_H.py -d acs2015_census_tract_data_part2.csv -c
Created CensusData
Finished Loading. Elapsed Time: 0.5705 seconds
upasana@instance-1:~$ 
```

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	40.51 secs	41.37 secs
D. Drop Indexes and Constraints	39.33 secs	42.77 secs
E. Disable Autocommit	7.862 secs	8.098 secs
F. Use UNLOGGED table		
G. Temp Table with memory tuning		

H. copy_from	0.4481 secs	0.5705 secs

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?

- The varying performance results among different loading approaches can be attributed to factors such as the size of the dataset being loaded, the presence of indexes and constraints on the target table, the use of auto commit and transaction logging, memory usage optimizations, and the choice of bulk loading methods.
- Larger datasets, complex indexes, and constraints can slow downloading times, while disabling auto commit and skipping transaction logging can improve performance.
- Leveraging memory tuning and using temporary tables can reduce disk I/O operations and speed up loading.
- Additionally, the `copy_from` command provides fast performance by directly loading data from a file into the table.
- The specific characteristics of the data and the database environment influence the performance outcomes, highlighting the importance of selecting the appropriate approach based on the specific requirements.