DataEng S24: 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 using multiple loading methods, 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.

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.

Please highlight your responses so that we can more easily see them

A. [MUST] Discussion Question (discuss as a group near the beginning of the week. Note your own response in this space):

> *Do you have any experience with ingesting bulk data into a DBMS? 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*.

Yes, I do have experience with ingesting bulk data into a DBMS. In the past, I worked on a project that involved importing a large dataset into a Oracle relational database. The method we used to input the data was through SQLLoader utility using bulk Load operations for Oracle Database and then we wrote a Stored Procedure to transform the data with multiple Key table data to the way we need for our operations.

For ingesting daily incremental breadcrumb data for a class project. The approach would involve:

Data Extraction: Extract the daily breadcrumb data from the source to the Staging Table in DB.
Data Transformation: Take the data once and then do Clean, filter, and format the data as needed to match the schema of the target database. This step might involve removing duplicates, handling missing values, and ensuring data consistency.
Data Loading: Load the data post transformation to the main tables in the schema.
Monitoring and Validation: Continuously monitor the data ingestion process for errors and validate the loaded data to confirm its accuracy and completeness.

Submit: submit your assignment by this Friday at 10pm

B. [MUST] Configure the 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. To keep your project work clean, use a new, different vm and delete the vm when finished with this assignment.
3. Also running the following commands on your VM 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
```

We have created a VM for the Class Activity and installed PostgreSQL in the VM and ran the commands mentioned.

C. [MUST] Prepare Data for Loading

1. Copy/upload both ACS Census Tract data files to your VM

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 > AL2015_1.csv
head -1 acs2015_census_tract_data_part2.csv > OR2015_2.csv
grep Alabama acs2015_census_tract_data_part1.csv >> AL2015_1.csv
grep Oregon acs2015_census_tract_data_part2.csv >> OR2015_2.csv
```

   The first two commands copy the headers to the sample files and the next command appends Alabama and Oregon 2015 data to the sample files. This should produce files with fewer than 1500 records per file. Use these sample files 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](). For example:

```
python3 load_inserts.py -d ./AL2015_1.csv -c
python3 load_inserts.py -d ./OR2015_2.csv
```

```
python3 load_inserts.py -d ./acs2015_census_tract_data_part1.csv -c
python3 load_inserts.py -d ./acs2015_census_tract_data_part2.csv
```

D. [MUST] 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 ACID properties 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 of the test samples and both parts of the ACS data set. Fill in the appropriate information in the table below (see part J).

After loading each part of the ACS data, try out a few validations to check that the data is loaded correctly. Use the following SQL queries and/or create more of your own:

- After loading part1,
  - The number of states in the database should be 24

        - select count(distinct state) from censusdata ;
    - The state of Portland is not found in the database
        - select 1 from censusdata where state = 'Portland' limit 1 ;
    - The state of Oregon is not found in the database
        - select count(*) from censusdata where state = 'Oregon' ;
    - The state of Iowa is found in the database
        - select 1 from censusdata where state = 'Iowa' limit 1 ;
    - There are 99 counties in Iowa
        - select count(distinct county) from censusdata where state = 'Iowa' ;
- After loading part2,
    - The number of states in the database should be 52
        - select count(distinct state) from censusdata ;
    - The state of Portland is not found in the database
        - select 1 from censusdata where state = 'Portland' limit 1 ;
    - The state of Oregon is found in the database
        - select count(*) from censusdata where state = 'Oregon' ;
    - There are 36 counties in Oregon
        - select count(distinct county) from censusdata where state = ‘Oregon’ ;
    - The state of Iowa is found in the database
        - select 1 from censusdata where state = 'Iowa' limit 1 ;
    - There are 99 counties in Iowa
        - select count(distinct county) from censusdata where state = 'Iowa' ;

```
srirams@classactivity:~$ python3 load_inserts.py
usage: load_inserts.py [-h] -d DATAFILE [-c]
load_inserts.py: error: the following arguments are required: -d/--datafile
srirams@classactivity:~$ python3 load_inserts.py -d ./AL2015_1.csv -c
python3 load_inserts.py -d ./OR2015_2.csv
readdata: reading from File: ./AL2015_1.csv
Created CensusData
Loading 1181 rows
Finished Loading. Elapsed Time: 1.341 seconds
readdata: reading from File: ./OR2015_2.csv
Loading 837 rows
Finished Loading. Elapsed Time: 0.8899 seconds
srirams@classactivity:~$ python3 load_inserts.py -d ./acs2015_census_tract_data_part1.csv -c
python3 load_inserts.py -d ./acs2015_census_tract_data_part2.csv
readdata: reading from File: ./acs2015_census_tract_data_part1.csv
Created CensusData
Loading 36844 rows
Finished Loading. Elapsed Time: 37.55 seconds
readdata: reading from File: ./acs2015_census_tract_data_part2.csv
Loading 37157 rows
Finished Loading. Elapsed Time: 36.34 seconds
srirams@classactivity:~$
```

```
postgres=# select count(distinct state) from censusdata;
 count
-------
    24
(1 row)

postgres=# select 1 from censusdata where state = 'Portland' limit 1 ;
 ?column?
----------
(0 rows)

postgres=# select count(*) from censusdata where state = 'Oregon' ;
 count
-------
     0
(1 row)

postgres=# select 1 from censusdata where state = 'Iowa' limit 1 ;
 ?column?
----------
        1
(1 row)

postgres=# select count(distinct county) from censusdata where state = 'Iowa' ;
 count
-------
    99
(1 row)
```

```
postgres=# select count(distinct state) from censusdata ;
 count
-------
    52
(1 row)

postgres=# select 1 from censusdata where state = 'Portland' limit 1 ;
 ?column?
----------
(0 rows)

postgres=# select count(*) from censusdata where state = 'Oregon' ;
 count
-------
   834
(1 row)
```

```
postgres=# select count(distinct county) from censusdata where state = 'Oregon' ;
 count
-------
    36
(1 row)

postgres=# select 1 from censusdata where state = 'Iowa' limit 1 ;
 ?column?
----------
        1
(1 row)

postgres=# select count(distinct county) from censusdata where state = 'Iowa' ;
 count
-------
    99
(1 row)
```

E. [MUST] 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.

Modify your load_inserts.py program to delay creation of constraints/indexes until after the data set is loaded. Enter the resulting load time into the results table below. How much does this technique improve load performance?

dropping indexes and constraints before loading results in a substantial and consistent improvement of about 19% for both parts of the data. This indicates that this technique significantly enhances the efficiency of data loading by reducing the overhead associated with managing indexes and constraints during the insertion process. Such improvements are crucial for optimizing the performance of bulk data operations in database systems.

F. [SHOULD] Disabling Autocommit

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 (e.g., write ahead logging). It also allows readers of the database to view an incomplete set of data during the load.

Modify your load_inserts.py program to avoid setting autocommit=True
Enter the resulting load time into the results table below.

```
srirams@classactivity:~$ python3 load_inserts_WithoutAutoCommit.py -d ./acs2015_census_tract_data_part1.csv -c
readdata: reading from File: ./acs2015_census_tract_data_part1.csv
Created CensusData
Dropped indexes and Constraints .
Loading 36844 rows
Finished Loading. Elapsed Time: 5.928 seconds
Added indexes and constraints.
srirams@classactivity:~$ python3 load_inserts_WithoutAutoCommit.py -d ./acs2015_census_tract_data_part2.csv
readdata: reading from File: ./acs2015_census_tract_data_part2.csv
Dropped indexes and Constraints .
Loading 37157 rows
Finished Loading. Elapsed Time: 6.242 seconds
Added indexes and constraints.
srirams@classactivity:~$
```

G. [SHOULD] UNLOGGED table

By default, RDBMS tables incur overheads of write-ahead logging (WAL) such that the database outputs extra metadata about each row insert to a log file known as the

Transaction Recovery Log (sometimes just called the WAL or "Write-Ahead Log"). The RDBMS uses that WAL data to recover the contents of the table if/when the RDBMS crashes. Crash Recovery is a great feature but it can slow down load performance.

You can avoid this extra WAL overhead by loading to an UNLOGGED table. Modify your load_inserts.py program to load data to an UNLOGGED table. Then enhance load_inserts.py to use a SQL query to append the staging data to the main CensusData table. Then create the needed index and constraint.

H. [ASPIRE] 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?

Yes, Using the Temporary Table approach, I noticed Loading the data was faster than Direct approach.

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?

Yes, increasing the temp_buffers setting did make a positive difference in reducing the data load times when using temporary tables. This approach, therefore, is more efficient in this context compared to direct loading for larger datasets, likely due to the more efficient use of memory and reduced disk I/O.

```
srirams@classactivity:~$ python3 load_inserts_TempTables.py -d ./acs2015_census_tract_data_part1.csv -c
readdata: reading from File: ./acs2015_census_tract_data_part1.csv
Created CensusData
TEMPORARY table tempCensusData set up successfully
Dropped indexes and Constraints .
Loading 36844 rows
Finished Loading. Elapsed Time: 5.386 seconds
Added indexes and constraints.
srirams@classactivity:~$ python3 load_inserts_TempTables.py -d ./acs2015_census_tract_data_part2.csv
readdata: reading from File: ./acs2015_census_tract_data_part2.csv
TEMPORARY table tempCensusData set up successfully
Dropped indexes and Constraints .
Loading 37157 rows
Finished Loading. Elapsed Time: 5.978 seconds
Added indexes and constraints.
srirams@classactivity:~$
```

I. [MUST] 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](). See the last section of [this blog post]() for an example.

```
srirams@classactivity:~$ python3 copy_form_LoadData.py -d ./acs2015_census_tract_data_part1.csv -c ./acs2015_census_tract_data_part1.csv
Created censusdata
Data loaded using COPY.
Finished Loading. Elapsed Time: 0.3622 seconds
srirams@classactivity:~$ python3 copy_form_LoadData.py -d ./acs2015_census_tract_data_part2.csv ./acs2015_census_tract_data_part2.csv
Data loaded using COPY.
Finished Loading. Elapsed Time: 0.3485 seconds
srirams@classactivity:~$
```

J. [MUST] Results

Use this table to present your results. List only the methods that you actually completed. List only load times for the full amount of Census data (not the small test sample of OR data created in part C). For each loading method, load both parts of the Census data and measure/report the load time for each part separately in the corresponding column of the table.

Method	Time to load part1	Time to load part2
D. Simple inserts	37.55 seconds	36.34 seconds
E. Drop Indexes and Constraints	30.42 seconds	29.4 seconds
F. Disable Autocommit	5.92 seconds	6.24 seconds
G. Use UNLOGGED table	6.67 seconds	6.44 seconds
H. Temp Table with memory tuning	5.39 seconds	5.98 seconds
I. copy_from	0.36 seconds	0.35 seconds

K. [SHOULD] 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?

In the performance evaluation of various data loading methods, the most efficient technique is using the copy_from method, which minimizes parsing and bypasses most transaction mechanisms, significantly reducing load time to under a second. Disabling autocommit, utilizing UNLOGGED tables, and employing temporary tables with memory tuning also considerably enhance performance by reducing transaction overhead and disk I/O operations. These methods, recording load times from 5.39 to 6.67 seconds, optimize by grouping multiple operations into fewer transactions and minimizing write-ahead logging. The drop indexes and constraints method further improves efficiency to around 30 seconds by eliminating the need to update indexes and check constraints during each insert. In contrast, simple inserts, which treat each operation as a separate transaction, are the slowest, demonstrating the substantial impact of transaction overhead on performance. These insights highlight the importance of tailoring data loading strategies to balance efficiency and system capabilities.