Forbes Miyasato

DataEng: Data Storage Activity

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.

```
readdata: reading from File: /home/miyasato/acs2015_census_tract_data.csv
Loading 74000 rows
Finished Loading. Elapsed Time: 197.3 seconds
```

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?

It did not, which is unexpected for me.

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?

- Loading data into unlogged table for 2015 data

```
readdata: reading from File: /home/miyasato/acs2015_census_tract_data.csv
Created CensusData2
Loading 74000 rows
Finished Loading. Elapsed Time: 17.68 seconds
```

- Loading data into unlogged table for 2015 data (Explicitly commit)

```
readdata: reading from File: /home/miyasato/acs2015_census_tract_data.csv
Created CensusData2
Loading 74000 rows
Finished Loading. Elapsed Time: 15.09 seconds
```

The load performance improved after explicitly committing all the loaded data instead of setting autocommit to true.

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?

```
readdata: reading from File: /home/miyasato/acs2015_census_tract_data.csv
Created CensusDataTemp
Loading 74000 rows
Finished Loading. Elapsed Time: 16.72 seconds
```

Yes, a lot faster than the baseline and around the same performance as unlogged tables.

G. Batching

So far our load performance has been held back by the fact we are using individuals 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.

```
miyasato@instance-2:~/dataeng-w21/Week_5_Data_Storage$ python3 load_inserts_batch.py -d ~/acs2015_census_tract_data
.csv
readdata: reading from File: /home/miyasato/acs2015_census_tract_data.csv
Loading 74000 rows
Finished Loading. Elapsed Time: 11.61 seconds
```

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.

```
miyasato@instance-2:~/dataeng-w21/Week_5_Data_Storage$ python3 load_inserts_copy.py -d ~/acs2015_census_tract_data.
csv
Loading 74001 rows
Finished Loading. Elapsed Time: 0.0008602 seconds
[(0,)]
```

Way too fast, it isn't actually loading data into the database.

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	197.3 seconds
Drop Indexes and Constraints	I altered the table in postgres	197.2 seconds
Use UNLOGGED table	load_inserts_unlogged.py	17.68 seconds
Temp Table with memory tuning	load_inserts_temp.py	16.72 seconds
Batching	load_inserts_batch.py	11.61 seconds
copy_from	load_inserts_copy.py	0.0008602 seconds (Not working properly)

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 and RDBMS functions and why various loading approaches produce varying performance results?

Dropping indexes and constraints surprisingly didn't reduce the load time. My assumption is that the PostgreSQL database doesn't index the columns and enforce constraints as you load, and instead loads all the data into the DB then starts processing the indexes and constraints after.

I learn that in order to improve the loading performances, the approaches either increase the usage of memory or reduce the network operations.

My main takeaway is that RDBMS offers so many powerful tools, and it's important for us as developers to be aware of all the options available and evaluate the constraints (space, run time and reliability...etc) to decide what is the best approach for solving our problem.