# DataEng: Data Storage Activity

Genevieve LaLonde
Data Engineering Winter 2021
Bruce Irvin
12 Feb 2021

## 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** | **acs2017** |
|---|---|---|---|
| Simple inserts | load_inserts.py | Elapsed Time: 90.66 seconds | Elapsed Time: 93.73 seconds |
| Drop Indexes and Constraints | no_index.py<br>delay_index.py | Elapsed Time when indexes are simply removed: 90.07 seconds<br>Elapsed Time when delaying adding indexes till after load: 131.5 seconds | Elapsed Time when indexes are simply removed: 96.5 seconds<br>2nd run when indexes are simply removed: Elapsed Time: 90.64 seconds<br>Elapsed Time when delaying adding indexes till after load: 119.8 seconds |
| Use UNLOGGED table | unlogged_set_logged.py<br>unlogged_insert_select.py | Time for unlogged, set as logged, add indexes: 17.78 s<br>Time for unlogged, insert select to CensusData: 18.57 s | Time for unlogged, set as logged, add indexes: 17.79 s<br>Time for unlogged, insert select to CensusData: 17.78 s |

| | | | |
|---|---|---|---|
| Temp Table with memory tuning | temp_default_mem.py temp_extended_mem.py | Time with default memory: 17.59 s<br>Extended memory: 17.62 seconds<br><br>Finished Loading.<br>Total elapsed Time: 16.95 seconds<br>Load time: 16.15 seconds<br>Append and drop stage time: 0.8058 seconds | Time with default memory: 17.64 s<br>Extended memory:<br><br>Finished Loading.<br>Total elapsed Time: 18.65 seconds<br>Load time: 17.76 seconds<br>Append and drop stage time: 0.884 seconds |
| Execute with parameters | execute_params.py | Total elapsed Time: 21.18 seconds<br>Load time: 20.78 seconds<br>Append and drop stage time: 0.4042 seconds | Total elapsed Time: 21.13 seconds<br>Load time: 20.74 seconds<br>Append and drop stage time: 0.3901 seconds |
| Batching | executemany_params.py | Total elapsed Time: 21.47 seconds<br>Load time: 21.17 seconds<br>Append and drop stage time: 0.3036 seconds | Total elapsed Time: 20.88 seconds<br>Load time: 20.48 seconds<br>Append and drop stage time: 0.3924 seconds |
| Batching as execute_batch | execute_batch_params.py | Total elapsed Time: 8.257 seconds<br>Load time: 7.934 seconds<br>Append and drop stage time: 0.3225 seconds<br><br>Total elapsed Time: 7.658 seconds<br>Load time: 7.378 seconds<br>Append and drop stage time: 0.2801 seconds | Total elapsed Time: 8.006 seconds<br>Load time: 7.65 seconds<br>Append and drop stage time: 0.3562 seconds<br><br>Total elapsed Time: 7.68 seconds<br>Load time: 7.286 seconds<br>Append and drop stage time: 0.3943 seconds |

|  |  |  |  |
|---|---|---|---|
| copy_from | copy_from.py | Total elapsed Time: 0.7366 seconds<br>Total elapsed Time: 0.5988 seconds | Total elapsed Time: 0.6462 seconds<br>Total elapsed Time: 0.6202 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 and RDBMS functions and why various loading approaches produce varying performance results?

Temp and unlogged tables performed the best, and were comparable to each other, even given variations like memory allocation and whether an unlogged table is subsequently set as logged, or used as the source of an INSERT SELECT statement.

Not creating indexes, or delaying their creation did not significantly impact performance. Perhaps this is related to the fact that the primary key was numeric data types, and the values of the text index is only 2 characters. So they would be performant to create. Might be interesting to repeat, with indexes on text columns with longer values and see if there is a larger impact.

By far the built in method copy_from is the fastest, returning in less than 1 s. It is also very easy to use. However as it is based on separators, it is appropriate for csv but not for json as in the project. To use the copy_from() command in the project, we would first need to convert the json, as shown in Haki Benita's blog.

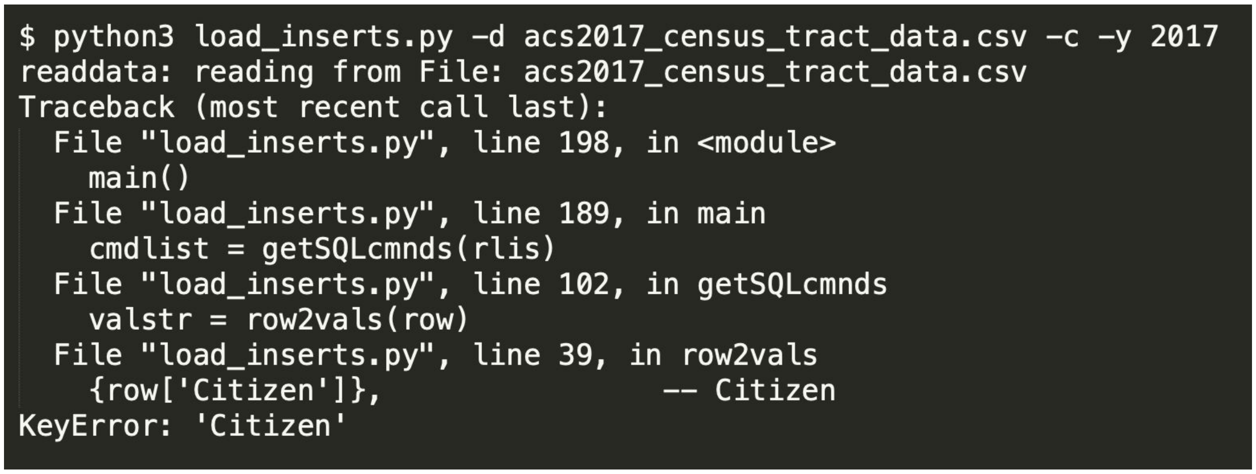
# Harder Parts

**Obstacle #1:**
2015 matched expected header names
CensusTract,State,County,TotalPop,Men,Women,Hispanic,White,Black,Native,Asian,Pacific,Citizen,Income,IncomeErr,IncomePerCap,IncomePerCapErr,Poverty,ChildPoverty,Professional,Service,Office,Construction,Production,Drive,Carpool,Transit,Walk,OtherTransp,WorkAtHome,MeanCommute,Employed,PrivateWork,PublicWork,SelfEmployed,FamilyWork,Unemployment

2017 had a changed name for the first column
TractId,State,County,TotalPop,Men,Women,Hispanic,White,Black,Native,Asian,Pacific,VotingAgeCitizen,Income,IncomeErr,IncomePerCap,IncomePerCapErr,Poverty,ChildPoverty,Professiona

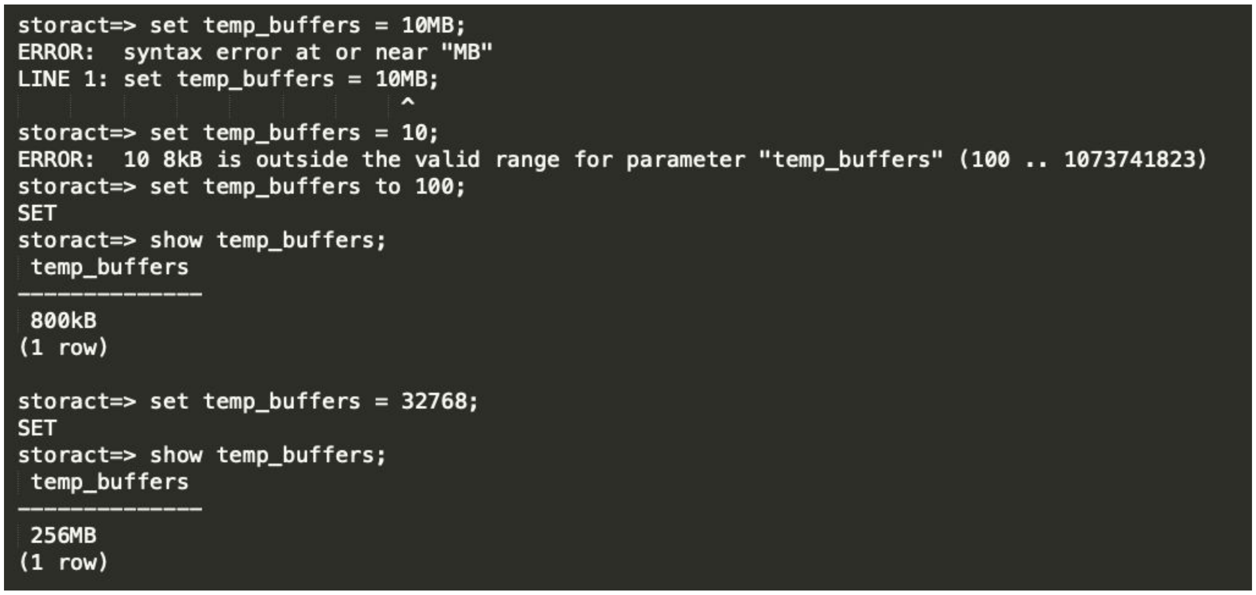
l,Service,Office,Construction,Production,Drive,Carpool,Transit,Walk,OtherTransp,WorkAtHome,MeanCommute,Employed,PrivateWork,PublicWork,SelfEmployed,FamilyWork,Unemployment

Resolution:
Rename TractId->CensusTract in 2017 file.
**Obstacle #2:**
Same as complication #1 for column "Citizen" which is called "VotingAgeCitizen" in the 2017 data.



```
$ python3 load_inserts.py -d acs2017_census_tract_data.csv -c -y 2017
readdata: reading from File: acs2017_census_tract_data.csv
Traceback (most recent call last):
  File "load_inserts.py", line 198, in <module>
    main()
  File "load_inserts.py", line 189, in main
    cmdlist = getSQLcmnds(rlis)
  File "load_inserts.py", line 102, in getSQLcmnds
    valstr = row2vals(row)
  File "load_inserts.py", line 39, in row2vals
    {row['Citizen']},                -- Citizen
KeyError: 'Citizen'
```

**Obstacle #3:**
Setting the temp_buffer requires math:

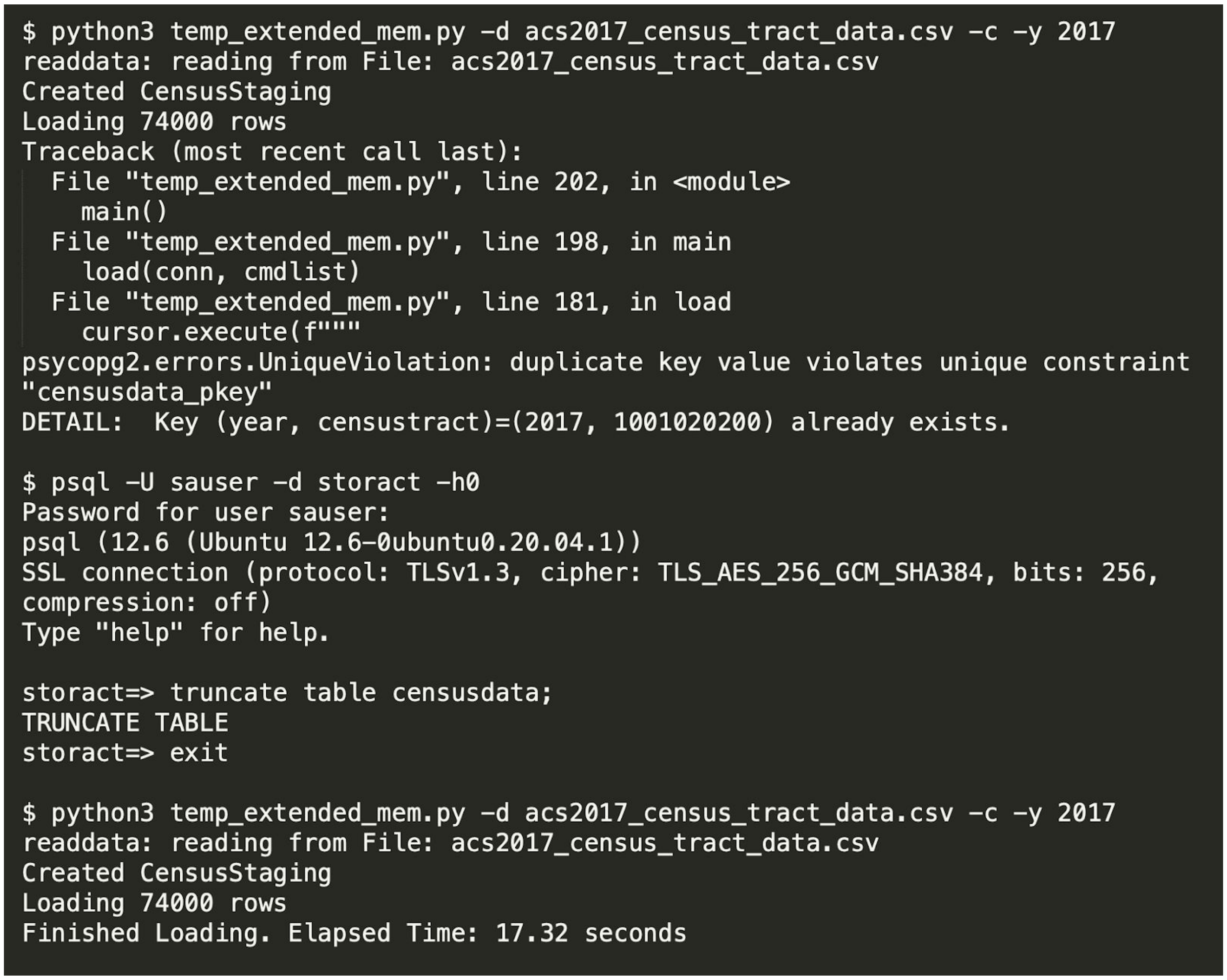


```
storact=> set temp_buffers = 10MB;
ERROR:  syntax error at or near "MB"
LINE 1: set temp_buffers = 10MB;
                             ^
storact=> set temp_buffers = 10;
ERROR:  10 8kB is outside the valid range for parameter "temp_buffers" (100 .. 1073741823)
storact=> set temp_buffers to 100;
SET
storact=> show temp_buffers;
 temp_buffers
--------------
 800kB
(1 row)

storact=> set temp_buffers = 32768;
SET
storact=> show temp_buffers;
 temp_buffers
--------------
 256MB
(1 row)
```

Calculated as 256 * 1024 / 8 = 32768 (KB)

**Obstacle #4:**
When appending data to an existing table multiple times in multiple tests, it is necessary to truncate the table between tests. This prevents hitting primary key duplicates.



```
$ python3 temp_extended_mem.py -d acs2017_census_tract_data.csv -c -y 2017
readdata: reading from File: acs2017_census_tract_data.csv
Created CensusStaging
Loading 74000 rows
Traceback (most recent call last):
  File "temp_extended_mem.py", line 202, in <module>
    main()
  File "temp_extended_mem.py", line 198, in main
    load(conn, cmdlist)
  File "temp_extended_mem.py", line 181, in load
    cursor.execute(f"""
psycopg2.errors.UniqueViolation: duplicate key value violates unique constraint
"censusdata_pkey"
DETAIL:  Key (year, censustract)=(2017, 1001020200) already exists.

$ psql -U sauser -d storact -h0
Password for user sauser:
psql (12.6 (Ubuntu 12.6-0ubuntu0.20.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,
compression: off)
Type "help" for help.

storact=> truncate table censusdata;
TRUNCATE TABLE
storact=> exit

$ python3 temp_extended_mem.py -d acs2017_census_tract_data.csv -c -y 2017
readdata: reading from File: acs2017_census_tract_data.csv
Created CensusStaging
Loading 74000 rows
Finished Loading. Elapsed Time: 17.32 seconds
```

**Obstacle #5**
I had a lot of difficulty converting the command from the script example where the parameters are hard coded, into the proper way to run execute, which is required for its more advanced forms executemany and execute_batch, where the arguments (the inserted values) must be included in a parameter separate from the sql command. In building that code, I used the doc below for syntax for referring to the table.

https://www.psycopg.org/docs/usage.html#passing-parameters-to-sql-queries

```
>>> cur.execute(                                    # correct
...     SQL("INSERT INTO {} VALUES (%s)").format(Identifier('numbers')),
...     (10,))
```

As I worked on this issue I was getting these error messages below:

- TypeError: not all arguments converted during string formatting

- TypeError: dict is not a sequence
- psycopg2.errors.UndefinedTable: relation "CensusStaging" does not exist
  LINE 1: INSERT INTO "CensusStaging" VALUES ('2015','1001020200','Ala…

I ended up converting row2vals to return a list of tuples instead of letting it parse values out of the dictionary list items. And I ended up hard coding the table name into the insert statement because the formatting was adding quotes “ around the staging table name so it did not match the actual staging table name. This took the majority of my debug time for the second week.