Optimal Data Loads

PGConf NYC 2022

The talk in a nutshell

- How best to load data into PostgreSQL
- Intermediate level but beginner friendly
- Tips, techniques, examples with timings
- COPY, Network, DDL, TCL
- https://github.com/benjlis/optimal-data-loads: slides and scripts

About me

- Ben Lis
- Data Engineer for the History Lab at Columbia University
- Python, SQL, AWS, Shell scripting
- PostgreSQL since 2015; RDBMS since 1988
- Various technical, product & management roles
- GitHub, LinkedIn, Twitter

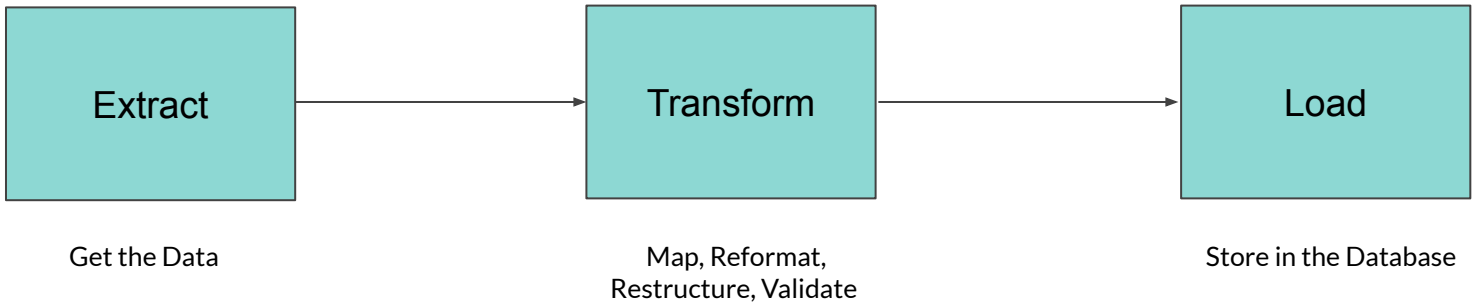
Preliminaries

We're optimizing for time, minimizing program runtime and your time.

A data load is the ‘L’ in ETL and ELT

ETL

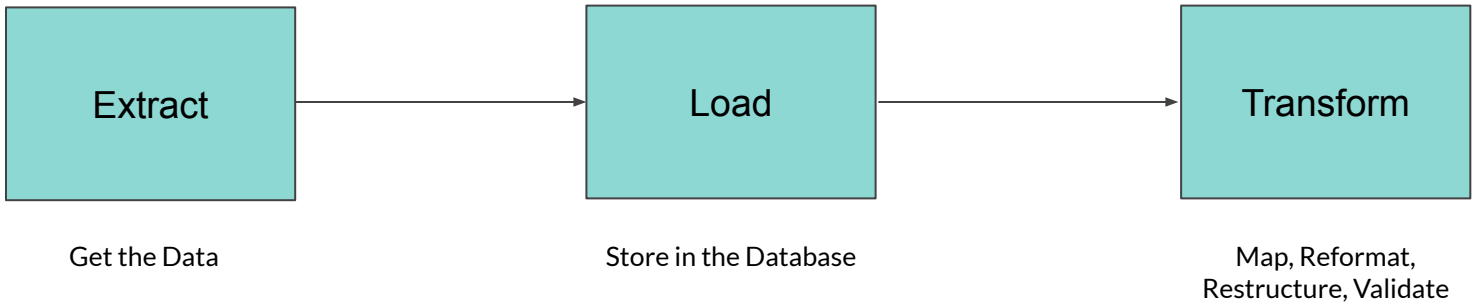
Extract, Transform & **Load**



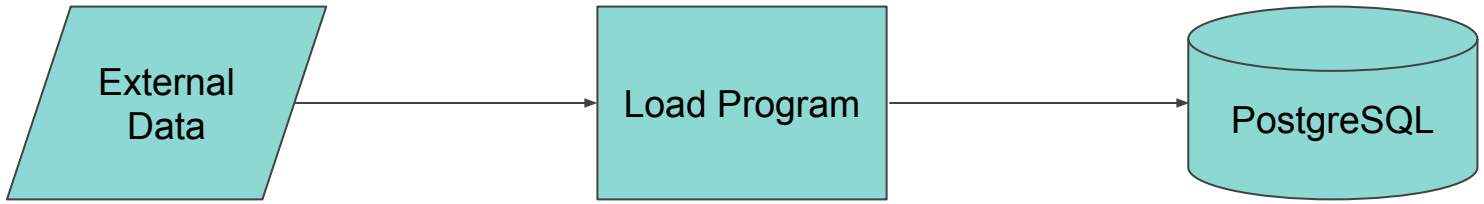
The original term used to describe loading data into a data warehouse

ELT

Extract, **Load** & Transform

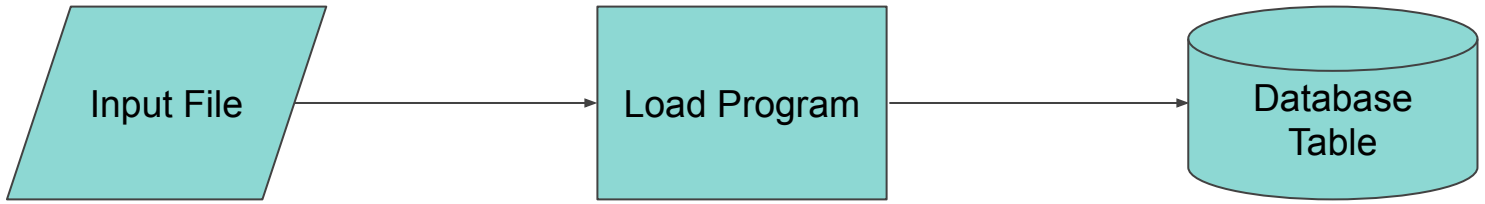


Data Load Definition



- External Data: not in PostgreSQL from an outside source
- Often need to ‘refresh’ data (i.e., re-run process)

Our Examples



Delimited Text File
CSV Format
UTF-8 Encoding

Python script OR psql commands

Our Input File

- Companies data
- UTF-8 character set
- CSV Format
- 295 MB
- 2,192,702 records
- 10 fields
- PK exists
- A subset of the public, open data published by GLEIF

Our Database Table

```
create table companies (
  lei                           text primary key,
  lname                         text not null,
  jurisdiction                  text,
  legal_form                    text not null,
  entity_status                 text not null,
  reg_status                    text not null,
  init_reg                      date not null,
  next_renewal                  date not null,
  last_update                   timestamp with time zone not null,
  managing_lou                  text not null);
```

companies.sql

Our Hardware Environment

Local

- iMac Late 2015
- 3.2 GHz Quad-Core Intel Core i5
- 16 GB 1867 MHz DDR3
- Network Speed (fast.com)
  - download: 270 Mbps
  - upload: 90 Mbps

Server

- t2.medium AWS EC2 instance
- 2 vCPU
- 3.3 GHz Intel Xeon Scalable processor
- 4 GB Memory
- us-east

Our Software Environment

Local

- macOS Monterey v12.5.1
- PostgreSQL 13.0
- Python 3.9.3
- psycopg 2.9.3

Server

- Ubuntu 20.04.2 LTS
- PostgreSQL 13.2
- Python 3.8.10
- psycopg 2.9.3

Our Specification

- Load must run in < 12 minutes, but the quicker, the better
- Load runs every 8 hours (i.e., 3x per day, 7 days a week)
- Load overwrites the existing content
- Data is read-only

On Timings

Timing vs. Benchmark

- TPC
- “ignore all benchmarks”
- YMMV
- Identify speed-ups > 25%

Reported Times

- Average of multiple runs
- Rounded to the nearest second
- Mechanism
  - time python si.py
  - \timing – command for SQL

Tips & examples with timings

Simple Inserts Script

```python
import csv
import psycopg2


insert_stmt = """
insert into companies
    (lei, lname, jurisdiction, legal_form, entity_status, reg_status, init_reg,
     next_renewal, last_update, managing_lou)
values
    (%s, %s, %s, %s, %s, %s, %s,
     %s, %s, %s)
"""

conn = psycopg2.connect("")
conn.autocommit = True
cursor = conn.cursor()
with open('companies.csv') as csvfile:
    creader = csv.reader(csvfile)
    for row in creader:
        cursor.execute(insert_stmt, row)
# conn.commit()
```

si.py

Script & DB Local	00:09:15
Script Local, DB Server	07:11:30

WHOA!!!!
The DB Server case is about 43x slower!

Simple Inserts Analysis

```python
import csv
import psycopg2


insert_stmt = """
insert into companies
    (lei, lname, jurisdiction, legal_form, entity_status, reg_status, init_reg,
     next_renewal, last_update, managing_lou)
values
    (%s, %s, %s, %s, %s, %s, %s,
     %s, %s, %s)
"""

conn = psycopg2.connect("")
conn.autocommit = True
cursor = conn.cursor()
with open('companies.csv') as csvfile:
    creader = csv.reader(csvfile)
    for row in creader:
        cursor.execute(insert_stmt, row)
# conn.commit()
```

si.py

Performance Problems:

1. **Network Latency**
2. Commit Frequency
3. Parsing & Execution

COPY

- Fastest way to move data between OS files and PostgreSQL
- Optimized for loading large numbers of rows
- Minimizes network, commit and execution overhead
- Both a command and an internal message protocol
- Used by `pg_dump`
- Accessible in both psql (`\copy`) and psycopg2 (`cursor.copy_from()`)
- Easy to use (most of the time)

COPY

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | PROGRAM 'command' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]
    [ WHERE condition ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | PROGRAM 'command' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]

where option can be one of:

    FORMAT format_name
    FREEZE [ boolean ]
    DELIMITER 'delimiter_character'
    NULL 'null_string'
    HEADER [ boolean ]
    QUOTE 'quote_character'
    ESCAPE 'escape_character'
    FORCE_QUOTE { ( column_name [, ...] ) | * }
    FORCE_NOT_NULL ( column_name [, ...] )
    FORCE_NULL ( column_name [, ...] )
    ENCODING 'encoding_name'
```

PostgreSQL: Documentation: 14: COPY

COPY

```
\timing
\copy companies from 'companies.csv' CSV
```

cp.sql

	Simple Inserts	COPY	
Script & DB Local	00:09:15	00:00:51	*10x faster!!!*
Script Local, DB Server	07:11:30	00:00:32	*860x faster!!!*

COPY On DB Server

COPY Local & DB Server	00:00:32
COPY & DB Server	00:00:21

1.5x faster

- mainly attributable to input file on server
- `COPY` vs. `\copy`
- `COPY` can only be run on the DB server by the postgres OS user

COPY’s shortcomings

- Loads all columns in the input file
- Data must match the format required by the relevant PostgreSQL data type
- COPY stops operation at the first error (and ROLLBACKs)
- Embedded delimiters, backslashes, and carriage returns can cause problems
- `with null as '"'` will not insert a null, inserts an empty string
- `\copy` (psql) must fit on a single line

Addressing COPY’s shortcoming

- Preprocessing the input file with tools like tr, sed and awk
- Use a staging table
- pgloader - Still uses COPY under the covers

Drop Indices & PK/FK, Load, Recreate

```
-- drop PK/FK constraints....and any additional indexes
alter table companies drop constraint companies_pkey;
-- COPY companies data
\copy companies from 'companies.csv' CSV
-- recreate PK/FK constraints
alter table companies add primary key (lei);
```

COPY	00:00:21
Drop, COPY & Recreate	00:00:18

1.2 x faster

- \copy: 7.5 seconds
- PK recreate: 10.5 seconds
- YMMV

Understanding UNLOGGED

- PostgreSQL supports UNLOGGED tables
- Specified in CREATE or ALTER table statements
- Changes to UNLOGGED tables are not written to the WAL (write-ahead log)
- DML executed against UNLOGGED tables is faster
- **WARNING:** UNLOGGED tables are **NOT** crash or standby safe
- After a crash restart or standby failover, the UNLOGGED table is truncated
- Think through your application before using it!!!

UNLOGGED

```
-- make table unlogged, after understanding consequences
alter table companies set unlogged;
alter table companies drop constraint companies_pkey;
\copy companies from 'companies.csv' CSV
alter table companies add primary key (lei);
```

Drop, COPY & Recreate	00:00:18
UNLOGGED	00:00:13

1.3 x faster

- \copy: 6.5 seconds
- PK recreate: 6.5 seconds
- Don't bother if you intend to `set logged` after

Key Takeaways

1. Use COPY whenever possible
2. Drop-Load-Recreate Indexes, PK and FK constraints
3. If appropriate, consider NOLOGGING
4. If not using COPY, look to minimize
    a. network latency
    b. commit frequency
    c. parsing and execution overhead
5. If possible, develop with a DB environment similar to production
6. Spend time identifying the actual problem source

What Didn’t We Cover?

- Triggers
- Parallelism (Citus, Greenplum, TimescaleDB)
- PostgreSQL configuration parameters
- Advanced methods for bulk inserts in psycopg2

Questions & Comments

Thank You!