### PDT 2022 - ZADANIE I.

## **Protokol**

Vypracoval:	Štefan Hajdú
GitHub - private	https://github.com/StefanHajdu/PDT-22/tree/ master/Assignment 1
GitHub – FIIT DBS	https://github.com/FIIT-DBS/zadanie-pdt- StefanHajdu

# Postupu importu

### Použitá technológia

V zadaní sme nepoužili žiadnu špeciálnu technológiu typu ORM alebo big data framework. Využili sme iba obyčajné SQL príkazy, konkrétne to bol optimalizovný príkaz COPY. Tento príkaz je odporúčaný pre importovanie väčšieho množstva dát v oficiálnej PostgreSQL dokomumntácií (1):

"Use COPY to load all the rows in one command, instead of using a series of INSERT commands. The COPY command is optimized for loading large numbers of rows; it is less flexible than INSERT, but incurs significantly less overhead for large data loads."

Túto metódu sme najprv otestovali na example dátach prejdením si tutorailu (2). Čím sa nám potvrdilo, že ide skutočne o metódu, ktorá dáta dokáže importovať v priebehu niekoľkých hodín.

Samotný program je implementovaný v jazyku *Python* s použitím adaptéra pre Postgres databázu *psycopg2*.

# **Opis algoritmu**

Import prebieha v troch hlavných fázach:

- 1. import do tabuľky authors
- 2. import do tabuliek: conversations, hashtags, conversation\_hashtags, annotations, links, context\_domains, context\_entities, context\_annotations
- import do tabul'ky conversation\_references

Importovaním v tomto poradí sa vyhneme run-time chybám spôsobených neexistujúcim foreignkey.

Algorimus je pre všetky tri fázy veľmi podobný, líši sa len tým aké obmedzenie treba ošetriť pre hodnotu fieldu.

#### Pseudokód:

Ako vidieť v pseudokóde najprv si definujeme iterátor pre JSONL súbor. Iterátor má každá fáza vlastný. Zo súboru si vytiahneme niekoľko riadkov, ktoré predstavujú chunk dát, ktoré budú do databázy v danej iterácii importované.

COPY\_FROM vyžaduje vstup vo forme CSV. My si vytvárame CSV z každého chunku v pamäti, pomocou StringIO. Ako separátor sa najlepšie osvedčil tabulátor.

Predtým ako hodnotu fieldu zapíšene do CSV je jeho hodnota upravená nasledovne pomocou *clean\_field*(data=field\_value) funkcie:

```
if data is None:
# same as NULL in Postgres
    return r"\N"
return (
    str(data)
        .replace("\n", "\\n")
        .replace("\t", "\\t")
        .replace("\r", "\\r")
        .replace("\x00", "")
        .replace("\\", "\\\\")
)
```

Ak je hodnota filedu None, potom sa do databázy zapíše NULL hodnota. Rovnako treba urobiť escape pre špeciálne znaky, ktoré by kazili textový CSV súbor: new line, tab, carrige return, UTF-8 NULL, backslash.

#### Duplicity a "slepé" foreign keys

Prevencia proti vloženiu duplicity je riešená pomocou hash tabuliek. Pred spracovaním akéhokoľvek item z chunku v pseudokóde, tak najprv overíme či už exituje (nachádza sa v hash tabuľke), ak áno preskakujeme. Ak nie, item sa spracuje a potom sa uloží do hash tabuľky.

Pre toto riešenie sme sa rozhodli, pretože máme k dispozícii dostatok RAM (16 GB), teda si môžeme dovoliť držať v pamäti veľké hash tabuľky. Tiež sme sa chceli vyhnúť spôsobu, ktorý by závisel od spätného dohľadávania existencie záznamu v databáze. Dokumentácia PostgreSQL uvádza, že nie je vhodné importovať dáta do tabuľky s exitujúcim indexom, teda by naše vyhľadávanie bolo veľmi pomalé (1):

"If you are adding large amounts of data to an existing table, it might be a win to drop the indexes, load the table, and then recreate the indexes. Of course, the database performance for other users might suffer during the time the indexes are missing."

Pre kontrolu duplicít si evidujeme tieto hash tabuľky:

```
id_convs = {key: conversation["id"], value: True}
tags_dict = {key: hashtag["tag"], value: True}
domain_dict = {key: domain["title"] + domain["description"], value:
true}
entity_dict = {key: entity["title"] + entity["description"], value:
True}
```

Pomocou hash tabuliek kontorolujeme aj exitenciu "slepých" foreign keys, ktoré ukazujú na neexistujúcu *conversation* alebo *author*. Ak sa tento foreign key nevyskytuje v hash tabuľke pridelí sa mu hodnota None a do databázy sa zapíše hodnota NULL.

Pre kontrolu "slepých" foreign keys si evidujeme tieto hash tabuľky:

```
id_convs = {key: conversation["id"], value: True}
id_authors = {key: author["id"], value: True}
```

#### **SQLs**

Keďže sme vyriešili kontrolu duplicít pomocou hash tabuliek vytváraných počas behu importu. Nebolo potrebné vykonávať špeciálne select-y, ktoré by nám kontrolovali existenciu vkladaného fieldu. Preto v tejto časti nemôžeme uviesť analýzu dopytov.

Pri vytváraní tabuliek sme použili parameter UNLOGGED, pretože dokumentácia uvádza, že takéto tabuľky by mali byť rýchlejšie pre import:

"If specified, the table is created as an unlogged table. Data written to unlogged tables is not written to the write-ahead log (see <u>Chapter 30</u>), which makes them **considerably faster** than ordinary tables. However, they are not crash-safe..."

Všetky dopyty pre vytváranie tabuliek sú uvedené ako samostatná funkcia v súbore *db\_api.py*.

# Dĺžka trvania importu

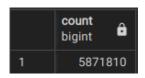
Pri testovaní importu sa nám ukázalo, že rýchlejšie vieme importovať chunk o veľkosti 100,000 záznamov, približne získame 10% času. Preto sme sa rozhodli importovať a monitorovať časové intervali pre 100,000 záznamov nie odporúčaných 10,000.

Teda každý riadok v <a href="https://github.com/StefanHajdu/PDT-22/blob/master/Assignment\_1/logs/run\_v3.csv">https://github.com/StefanHajdu/PDT-22/blob/master/Assignment\_1/logs/run\_v3.csv</a> predstavuje v poslednom stĺpci časový údaj pre importovanie 100,000 záznamov. V logu je pekne vidieť, kedy prebiehala **fáza 1 (1-2 sekundy)**, **fáza 2 (20 – 25 sekúnd)** a **fáza 3 (3-4 sekundy)**.

Dokopy to trvalu môjmu PC (s i7-3770K) 144 minút.

# Počet a veľkosť záznamov v každej tabuľke

Table: authors



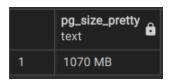


Table: conversations



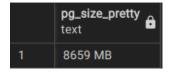
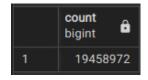


Table: annotations



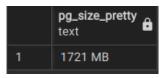
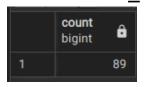


Table: context annotations





Table: context domains





## Table: context entities

	<b>count</b> bigint	â
1	269	40

	pg_size_pretty text
1	3312 kB

# Table: conversation\_hashtags

	<b>count</b> bigint	â
1	546137	45

	pg_size_pretty text
1	3888 MB

# Table: conversation\_references

	<b>count</b> bigint	â
1	28391	530

	pg_size_pretty text
1	2437 MB

#### Table: **links**

	count bigint
1	11540704

	pg_size_pretty text
1	2022 MB

## Table: hashtags

	count bigint
1	773865

	pg_size_pretty text
1	88 MB

#### Human readable:

```
SELECT COUNT(*) FROM authors -- 5'871'810
SELECT COUNT(*) FROM conversations -- 32'347'011
SELECT COUNT(*) FROM annotations -- 19'458'972
SELECT COUNT(*) FROM context_annotations -- 133'941'462
SELECT COUNT(*) FROM context_domains -- 89
SELECT COUNT(*) FROM context_entities -- 26'940
SELECT COUNT(*) FROM conversation_hashtags -- 54'613'745
SELECT COUNT(*) FROM conversation_references -- 28'391'530
SELECT COUNT(*) FROM links -- 11'540'704
SELECT COUNT(*) FROM hashtags -- 773'865
SELECT pg_size_pretty(pg_total_relation_size('public.authors')); -- 1070 MB
SELECT pg_size_pretty(pg_total_relation_size('public.conversations')); -- 8659 MB
SELECT pg_size_pretty(pg_total_relation_size('public.annotations')); -- 1721 MB
SELECT pg_size_pretty(pg_total_relation_size('public.context_annotations')); -- 10 GB
SELECT pg_size_pretty(pg_total_relation_size('public.context_domains')); -- 80 kB
SELECT pg_size_pretty(pg_total_relation_size('public.context_entities')); -- 3312 kB
SELECT pg_size_pretty(pg_total_relation_size('public.conversation_hashtags')); -- 3888 MB
SELECT pg_size_pretty(pg_total_relation_size('public.conversation_references')); -- 2437 MB
SELECT pg_size_pretty(pg_total_relation_size('public.links')); -- 2022 MB
SELECT pg_size_pretty(pg_total_relation_size('public.hashtags')); -- 88 MB
```

### Referencie

- (1) https://www.postgresql.org/docs/current/populate.html#POPULATE-COPY-FROM
- (2) https://hakibenita.com/fast-load-data-python-postgresql