

# PDT 2022 - ZADANIE I.

## Protokol

Vypracoval:	Štefan Hajdú
GitHub - private	<a href="https://github.com/StefanHajdu/PDT-22/tree/master/Assignment_1">https://github.com/StefanHajdu/PDT-22/tree/master/Assignment_1</a>
GitHub – FIIT DBS	<a href="https://github.com/FIIT-DBS/zadanie-pdt-StefanHajdu">https://github.com/FIIT-DBS/zadanie-pdt-StefanHajdu</a>

## 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 dokumentácii (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 prejdéním 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 *Python3* 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`
3. import do tabuľky `conversation_references`

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

Algorismus je pre všetky tri fázy veľmi podobný, líši sa len tým, že pre niektoré hodnoty kontrolujem duplicity (bližšie je to opísané v časti 3) treba ošetriť pre hodnotu fieldu.

Pseudokód:

```
json_iter = create iterator for JSONL file using Python generator
for each chunk in load_chunk(json_iter, chunk_size=100000) do
    csv_input = StringIO()
    for each item in chunk:
        csv_input.write("\t".join(
            map(clean_field,
                item["field-1"],
                item["field-2"],
                item["field-3"],
                ...,
                item["field-N"])
        )
    )
    csv_input.seek(0)
    db_adapter.connection.cursor.copy_from(csv_input, "table-
name", separator="\t")
```

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 zapíšeme 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 fieldu 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`.

Špeciálne prípady riešime nasledovne:

- **záznam v conversations má autora s neznámym id:** vytvorí sa záznam v tabuľke authors s daným id, kde sú všetky ostatné fieldy s hodnotou NULL
- conversation references spracujeme všetky, nerozlišujeme či sa jedná o duplicitu konverzácie
- záznam v conversation references preskakujeme vtedy, ak parent\_id nie je v našej databáze. Pôvodne sme to riešili zápisom parent\_id=NULL, ale nové riešenie sa nám zdá logickejšie

## Duplicity a „slepé“ foreign keys

Prevenencia proti vloženiu duplicity je riešená pomocou hash tabuliek. Pred spracovaním akéhokoľvek item z chunku v pseudokóde, tak najprv overíme či už existuje (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 existujúcim 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 kontrolujeme aj existenciu „slepých“ foreign keys, ktoré ukazujú na neexistujúcu conversation alebo author. Ak sa tento foreign key nevyskytuje v hash tabuľke prideli 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 vkladaneho 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 [Chapter 30](#)), 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 intervaly pre 100,000 záznamov nie odporúčaných 10,000.

Teda každý riadok v [https://github.com/StefanHajdu/PDT-22/blob/master/Assignment\\_1/logs/log.csv](https://github.com/StefanHajdu/PDT-22/blob/master/Assignment_1/logs/log.csv) 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 trvalo môjmu PC (s i7-3770K) **140 minút**.

## Počet a veľkosť záznamov v každej tabuľke

Table: authors

	count bigint
1	5895176

	pg_size_pretty text
1	1071 MB

Table: conversations

	count bigint
1	32347011

	pg_size_pretty text
1	8659 MB

Table: annotations

	count bigint
1	19458972

	pg_size_pretty text
1	1721 MB

Table: context\_annotations

	count bigint
1	133941462

	pg_size_pretty text
1	10 GB

Table: context\_domains



	count bigint 	pg_size_pretty text 
1	89	80 kB

Table: context\_entities



	count bigint 	pg_size_pretty text 
1	26940	3320 kB

Table: conversation\_hashtags



	count bigint 	pg_size_pretty text 
1	54613745	3888 MB

Table: conversation\_references



	count bigint 	pg_size_pretty text 
1	27950190	2402 MB

Table: links





	count bigint 	pg_size_pretty text 
1	11540704	2022 MB

Table: hashtags

	count bigint 	pg_size_pretty text 
1	773865	88 MB

## Human readable

```
SELECT COUNT(*) FROM authors -- 5'895'176
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 -- 27'950'190
SELECT COUNT(*) FROM links -- 11'540'704
SELECT COUNT(*) FROM hashtags -- 773'865

SELECT pg_size_pretty(pg_total_relation_size('public.authors')); -- 1071 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')); -- 2402 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>