Slovenská technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

Protokol zápisu importu tweetov do PostgreSQL

Martin Schön, Bc.

AIS ID: 103121

E-mail: xschon@stuba.sk

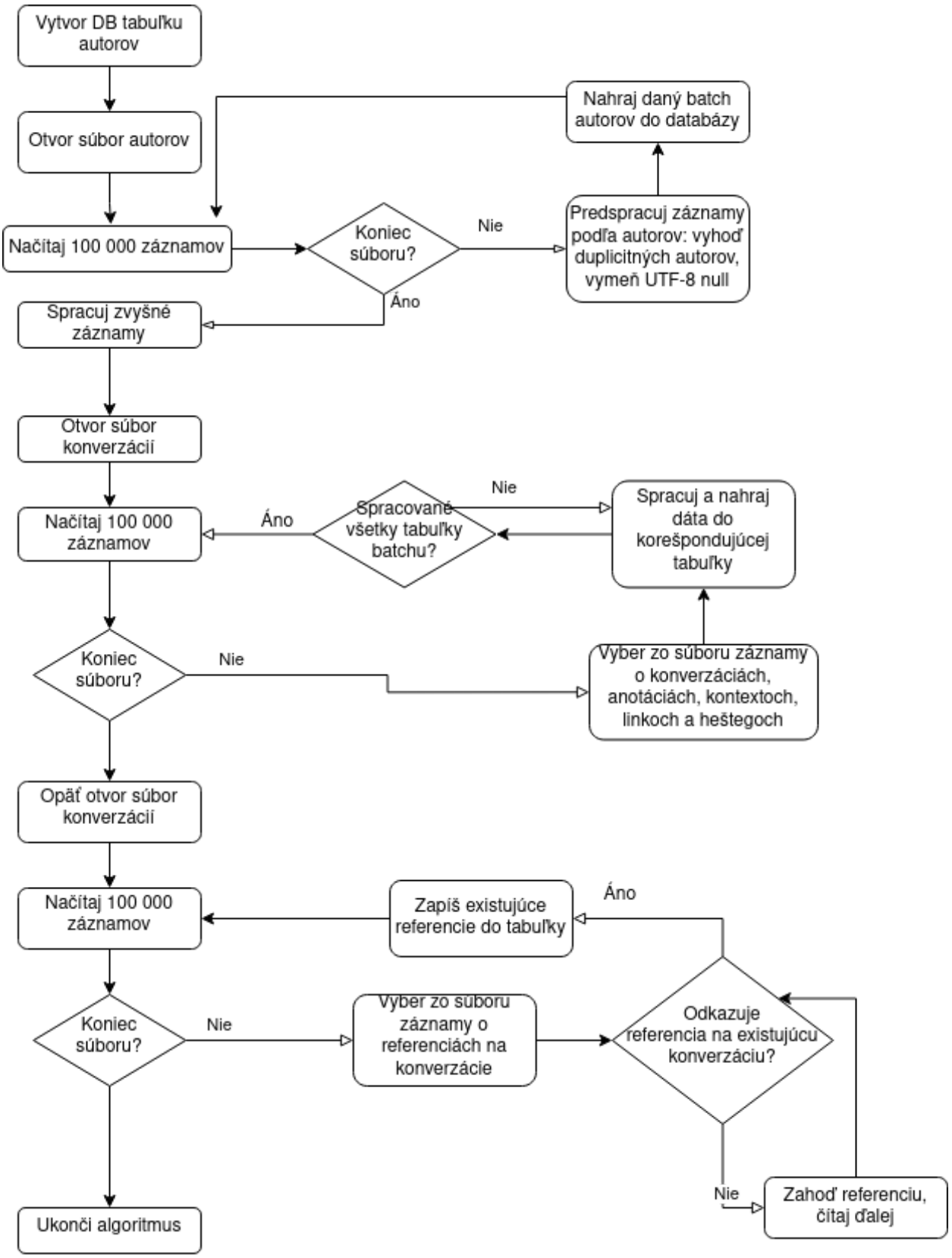
GitHub repozitár: https://github.com/FIIT-DBS/zadanie-pdt-xSchon

Predmet: Pokročilé databázové technológie

Zimný semester 2022/2023

Obsah

1. Opis algoritmu a postupu pri importe



Na diagrame vyššie je popísaný tok práce pri spracovaní údajov a ich následnom nahrávaní do databázy. Čítanie údajov zo súboru a ich zapisovanie prebieha v troch blokoch. Každý z týchto blokov generuje vlastný súbor csv, v ktorom sú zapísané časy spracovania jednotlivých batchov (várok) dát. Pre prvý blok (autorov): *time_tracker_authors_filling.csv* súbor drží tieto informácie. Pre druhý blok (hlavnú časť programu, spracovanie konverzácií) je to *time_tracker_main_filling.csv* a nakoniec *time_tracker_references.csv* slúži pre tretí, finálny blok (referencií).

Vo všetkých troch častiach programu je používané rovnaké načítanie súboru. Súbor je otvorený pomocou knižnice gzip, konkrétne cez funkciu *gzip.open* - máme dva rôzne súbory authors.json.gz použitý v prvom bloku a conversations.json.gz pre druhý a tretí blok. Následne program preiteruje cez všetky riadky súboru, pomocou *enumerate* súboru. Každý riadok je dekódovaný pomocou utf-8 dekódera a načítaný vo formáte json, potom je uložený do poľa riadkov.

```
data_rows = []
with gzip.open(config.USERS_PATH, 'rb') as f:
        for row_number, row in enumerate(f):
            data_rows.append(json.loads(row.decode(encoding='utf-8')))
```

Takto najskôr prebieha načítanie požadovaného počtu riadkov pre batch. V programe je veľkosť batchu voliteľná, ale základná (a mnou použitá) veľkosť batchu je 100 000. Takáto veľkosť zvyšuje efektivitu programu, keďže umožňuje spracovanie a nahrávanie väčšieho množstva dát naraz, čím sa znižuje časová náročnosť. Ak by sme nastavili väčšiu veľkosť batchu mohlo by to byť ešte výhodnejšie, avšak bola by potrebná aj väčšia veľkosť pamäte RAM (na 100 000 stačí 16GB).

Potom, čo bolo načítané požadované množstvo dát prebieha jeho spracovanie. Z poľa veľkosti 100k je jednoducho možné vytvoriť pandas DataFrame, so stĺpcami podľa json kľúčov a ich hodnotami. Po vytvorení takéhoto DataFrame je potrebné dáta upraviť a nahrať do databázy. To sa deje pri každej časti programu inak.

Pri **autoroch** (prvý blok) prebieha celý tento proces v súbore *fill_authors.py*. Autori sú na spracovanie relatívne jednoduchí - stačí kontrolovať duplicitné výskyty autorov (podľa id) a odstraňovať UTF-8 nulls, čo robím pomocou regexu. Duplicitné id autorov kontrolujem pomocou numpy array-u, v ktorom si držím vložených autorov a nemusím tak opakovane spúšťat query na ich výskyt. Potom pomocou to_sql funkcie nahrám autorov do ich databázy.

V **hlavnej časti** (blok dva) je hlavná časť programu. Spoločne s tretím blokom je uložený v súbore *fill_database.py*. Ešte pred začiatkom načítania batchov je deklarovaných 5 pomocných polí, ktoré získajú z databázy id autorov, rovnako ako (spravidla prázdne) polia id-čiek existujúcich užívateľov, id-čiek context entít a context domén a tagy hashtagov. Tieto polia následne slúžia na kontrolu duplicít pri vkladaní do jednotlivých tabuliek. Rovnako ako pri autoroch, je to zrýchlenie programu s tým, že polia držíme v pamäti RAM. V tejto časti prebieha načítanie tweetov, kde sú preskočené tie, ktoré sa už nachádzajú v tabuľke conversations, na základe id. Táto časť naplní 8 tabuliek: conversations, links, annotations, conversation_hashtags, hashtags, contex_annotations, contex_entities a context_domains. Toto napĺňanie prebieha vždy s miernymi rozdielmi - podľa formátu a limitácií dát. Uvediem jeden príklad za všetky, nakoľko je tento princíp využívaný skoro vo všetkých prípadoch.

Chcem získať hashtagy z načítaného batchu. Tie sú uložené v jsone zo stĺpca entities. Pomocou pandas funkcie to_list() rozložím stĺpec entities na nový pandas DataFrame, ktorý bude obsahovať stĺpce podľa kľúčov jsonov stĺpca entities. Z nového DataFrame ma zaujíma stĺpec hashtags, ktorý opäť obsahuje jsony, tentokrát priamo k hashtagom. V každom jsone je ľubovoľný počet hashtagov od 1 do N (tie, ktoré majú NULL dropnem). Môžem teda aplikovať to_list nad stĺpcom hashtags a spraviť tak nový dataframe, ktorého počet stĺpcov bude korešpondovať maximálnemu zapísanému počtu hashtagov v jednom jsone.

```
hashtags=pd.DataFrame(entities.hashtags.to_list(),index=entities.index)
```

Jsony, ktoré sú kratšie naplnia zvyšné stĺpce NULLmi. Tak mi vznikol indexovaný DataFrame hashtagov, kde v každom okne tabuľky je buď hashtag , alebo NULL. Prejdem cez každý stĺpec DataFramu a namapujem ich (pomocou DataFrame indexu) na conversation_ids z pôvodného poľa tweetov. Po spracovaní všetkých hashtagov v batchi, program skontroluje unikátne použité tagy hashtagov, porovná ich už s existujúcimi a nahrá nové hashtagy do databázy. Následne nahráva aj *conversation_hashtags*, ktoré referujú na existujúce hashtagy, ako aj konverzácie.
Podobným štýlom funguje spracovanie pre všetky tabuľky, samozrejme so svojimi vlastnými špecifikáciami. Každá skupina tabuliek má v programe svoju vlastnú funkciu, kde je možné pozrieť sa, ako je konkrétny údaj spracovaný.

V **poslednej** (tretej) **časti** programu prebieha spracovanie referencií na konverzácie. Táto časť v podstate funguje veľmi podobne ako časť druhá, avšak je možné ju spustiť až potom, čo už je tabuľka konverzácií naplnená. To je spôsobené tým, že tabuľka *conversation_references* odkazuje cudzím kľúčom na záznamy v konverzáciách, ak takáto konverzácia existuje. Tieto dáta sú získané z *conversations.jsonl.gz*.

Po tom, čo prebehne celé toto vkladanie pre jeden batch, načíta sa ďalší, až pokiaľ nie je súbor prečítaný.

2. Použité technológie

Program som naprogramoval v jazyku **Python 3.10** na OS Ubuntu 22.04.1 LTS. Python som si vybral preto, že je to vhodný jazyk na prácu s dátami a ponúka veľké množstvo knižníc na ich spracovanie. Taktiež umožňuje vhodné pripojenie na databázu postgreSQL a získavanie, či zapisovanie údajov.

Zadanie je vyhotovené ako Python projekt. Ten obsahuje viaceré časti kódu rozdelené do jednotlivých súborov, podľa ich zamerania. Pre správny beh je potrebné najskôr spustiť súbor setup.py, na Linuxe pomocou príkazu:

```
sudo python3 setup.py install
```

a nainštalovať knižnice pomocou requirements.txt, čo sa dá vykonať pomocou:

```
sudo pip install -r requirements.txt
```

2.1. gzip

Na načítanie súborov využívam knižnicu **gzip.** Tá ponúka možnosť načítať bitové súbory typu .gzip, preto je na túto úlohu ideálna. Takýto súbor je následne možné čítať po riadkoch pomocou funkcie enumerate.

```
with gzip.open(config.TWEETS_PATH, 'rb') as f:
    for row_number, current_tweet in enumerate(f):
```

2.2 pandas

Hlavným dôvodom pre môj výber jazyku Python bola knižnica **pandas**. Pandas je vynikajúci nástroj so slušnou rýchlosťou, ktorý prináša ľahkú a efektívnu prácu s načítanými dátami. V tomto projekte som využíval najmä pandas DataFrame, čo je dátová štruktúra, ktorá dokáže držať dáta v tabuľkách, podobným tým, ktoré poznáme z SQL. DataFrame umožňuje aj vytvorenie tejto tabuľky na základe listu json-ov, čo je taktiež výhodné, nakoľko pôvodné dáta sú uložené práve vo formáte json. Uvádzam aj nasledovný príklad toho, ako vyzerá vytváranie DataFramu na základe jsonu.

```
pd.DataFrame([{
'meno' : 'Jan',
'priezvisko' : 'Balazia',
'pozicia' : 'najlepsi ucitel'
},{
'meno' : 'Martin',
'priezvisko' : 'Schon',
'pozicia' : 'najlepsi ziak'
}])
```

✓ 0.3s

	meno	priezvisko	pozicia
0	Jan	Balazia	najlepsi ucitel
1	Martin	Schon	najlepsi ziak

Ďalšou, mnou využívanou výbornou funkciou pandas DataFramu je možnosť nahrania DataFramu do databázy, s pomocou sqlalchemy, s využitím *df.to_sql()* funkcie. Na základe testov a výpočtov, ktoré vznikli vychádza to_sql ako najlepšia možnosť pre nahrávanie väčšieho počtu záznamov do databázy s využitím pythonu.

```
authors_to_add.to_sql(
        'authors',
        engine,
        if_exists="append",
        index=False
    )
```

2.3 sqlalchemy

Knižnica sqlalchemy slúži v Pythone ako SQL nástroj, ktorý slúži ako Python ORM nástroj. V mojom konkrétnom riešení ho používam ako základný nástroj pre vytvorenie enginu, ktorý je potrebný na spustenie pandas funkcie *to_sql*. SQLalchemy engine je jednoduchý na vytvorenie a dobre zapadá do úlohy nahrávania DataFramu do databázy.

```
engine = create_engine(
            f"postgresql://{config.DATABASE['USER']}:"
            f"{config.DATABASE['PASSWORD']}@"
            f"{config.DATABASE['HOST']}:"
            f"{config.DATABASE['PORT']}/"
            f"{config.DATABASE['DBNAME']}"
        )
```

2.4 psycopg2

Na vykonanie jednoduchých SQL dopytov využívam knižnicu psycopg2. Na rozdiel od SQLAlchemy, táto knižnica umožňuje vytvorenie priameho pripojenia na SQL databázu, ktoré vykoná užívateľom napísané dopyty.

```
conn = psycopg2.connect(
            dbname=config.DATABASE['DBNAME'],
            host=config.DATABASE['HOST'],
            user=config.DATABASE['USER'],
            password=config.DATABASE['PASSWORD'],
            port=config.DATABASE['PORT'],
        )
```

2.5 Menej dôležité knižnice

2.5.1 json

Knižnica json je jednou zo základných knižníc Pythonu. Slúži na prácu s jsonmi - ich načítanie, spracovanie a ukladanie. Používam ju na parsovanie textu pri zápise do DataFramu, pretože práca s jsonom je jednoduchšia, ako práca so stringom vo formáte jsonu.

```
json.loads(current_tweet.decode(encoding='utf-8'))
```

2.5.2 dotenv

Dotenv knižnica slúži na načítanie citlivých prihlasovacích údajov databázy v podobe environmentálnych premenných zo súboru *.env*. Takýto súbor vznikne správnou úpravou súboru *.env.example* podľa inštrukcií uvedených v tomto súbore. Takýto prístup zvyšuje bezpečnosť a zabezpečuje citlivé údaje. V programe je takéto spracovanie environmentálnych premenných súčasťou konfiguračného súboru *config.py*.

```
load_dotenv()
'DBNAME' = os.getenv('DB_NAME')
```

2.5.3 logging

Ďalšou zo základných Python knižníc je logging. Táto knižnica zabezpečuje jednodné logovanie programu, podľa konfigurácie uvedenej v súbore *config.py*. V aktuálnom nastavení sú logy zapisované do súboru *uploader.log*.

```
logging.info('Logger succesfully initialized')
```

2.5.4 numpy

Knižnica numpy slúži na držanie vyššie spomínaných polí na prevenciu proti duplicitám. Numpy polia sú rýchlejšie a zaberajú menej miesta ako štandardné Python listy, preto sú na túto úlohu vhodnou voľbou.

3. Vysvetlenie SQL dopytov

V programe sú využívané 3 typy SQL dopytov. Ako prvé je DDL - Data Definition Language, ktorý slúži na vytvorenie tabuliek. SQL dopyty na vytvorenie jednotlivých tabuliek je možné nájsť v priečinku *sql_queries* v dvoch .sql súboroch.

```
CREATE TABLE IF NOT EXISTS conversations(
 id BIGINT PRIMARY KEY UNIQUE NOT NULL,
 author_id BIGINT NOT NULL,
 content TEXT NOT NULL,
 possibly_sensitive BOOLEAN NOT NULL,
 language VARCHAR(3) NOT NULL,
 source TEXT NOT NULL,
 retweet_count INT,
 reply_count INT,
 like_count INT,
 quote_count INT,
 created_at TIMESTAMPTZ NOT NULL,

 FOREIGN KEY (author_id)
   REFERENCES authors(id)
);
```

Vyššie je uvedený SQL dopyt na vytvorenie tabuľky konverzácií. Dodržiava všetky zadané a logické limitácie - ako napríklad dátové typy, hlavný kľúč, "nullable" vlastnosť, či referencie pomocou cudzieho kľúča. Takéto vytváranie tabuliek je bežne používané, nie je náročné a zaberie zopár milisekúnd.

Druhým typom sú jednoduché SQL dopyty na získanie hodnôt z databázy, pre budúcu kontrolu unikátnosti danej hodnoty (napríklad id).

```
SELECT id FROM conversations;
```

Na takomto príkaze nie veľmi čo pokaziť, ani optimalizovať z hľadiska SQL dopytu. Bolo by ale možné zefektívniť tento proces pridaním indexov, ktoré aktuálne v tabuľke nie sú (teším sa na zadanie 2 🙂).

Posledným a najviac používaným prístupom k databáze je ORM funkcia *to_sql* z knižnice SQLAlchemy popísanej vyššie. Táto funkcia vykonáva zápis údajov z batchu do databázy. Pomocou parametra ***echo=True*** sa dá spätne získať SQL tejto funkcie: ako príklad pri nahrávaní do tabuľky linkov slúži nasledujúci obrázok:

```
INFO:sqlalchemy.engine.Engine:INSERT INTO links (expanded_url, title,
description) VALUES (%(expanded_url)s, %(title)s, %(description)s)

2022-10-07 00:13:20,380 INFO sqlalchemy.engine.Engine [generated in 0.12715s]
({'expanded_url':
'https://twitter.com/MKGRDC/status/1496718618813370376/photo/1', 'title': Nor
'description': None}, {'expanded_url':
'https://twitter.com/SjWealth/status/1496732916784582659/photo/1', 'title':
None, 'description': None}, {'expanded_url':
'https://twitter.com/slickfinger_/status/1496732917434773505/photo/1', 'title
None, 'description': None}, {'expanded_url':
'https://twitter.com/elmastozlu/status/1496732919540367361/photo/1', 'title':
None, 'description': None}, {'expanded_url':
'https://twitter.com/maciej_zych23/status/1496732916965154818/photo/1',
'title': None, 'description': None}, {'expanded_url':
'https://twitter.com/AntisocialRaji/status/1496731581876162564/photo/1',
'title': None, 'description': None}, {'expanded_url':
'https://twitter.com/AyuwokiHee/status/1496716837102538755/photo/1', 'title':
None, 'description': None}, {'expanded_url':
'https://twitter.com/TobiasPetracca_/status/1496728398516740099/photo/1',
'title': None, 'description': None}  ... displaying 10 of 48423 total bound
parameter sets ...  {'expanded_url':
'https://twitter.com/VishwavaniNews/status/1496753265790181379/photo/1',
'title': None, 'description': None}, {'expanded_url':
'https://twitter.com/VishwavaniNews/status/1496753265790181379/photo/1',
'title': None, 'description': None})
```

Z porovnaní v článku spomenutom vyššie vyplýva, že takýto postup vkladania do databázy je rýchlejší, ako iné možnosti. To je možné vďaka tomu, že do databázy je vkladaný blok dát, namiesto toho, aby boli vložené záznamy jednotlivo. Nie je teda potrebné vykonávať dopyt pre každý riadok zvlášť. V mojom prípade používam veľkosť bloku 100 000, takže počet dopytov je niekoľko tisíc-násobne menší. Samozrejme, takýto dopyt beží dlhšie, ako by bežal pri jednom vloženom riadku, ale je rýchlejší ako jeden riadok * n-tisíc.

4. Dĺžka trvania importu a časový opis priebehu

Pre každú fázu (autori, hlavná časť, referencie) behu programu je vygenerovaný jeden .csv súbor opisujúci časový priebeh podľa požiadaviek, s jednou zmenou. Nakoľko vkladám bloky po 100 000 namiesto 10 000, značím aj časové trvanie jedného bloku dlhého 100k záznamov - vo formáte aktuálny čas; celkový čas; čas pre aktuálny blok.

Pre autorov je zápis v súbore ***time_tracker_authors_filling****.csv*. Pre jeden blok autorov je potrebných približne 5 sekúnd - pre import celého súboru sa jedná približne o 6 minút.

Pre hlavnú časť, teda všetky tabuľky okrem referencií (autorov sa dopĺňa iba zopár - v prípade, ak autor id neexistuje, ale konverzácia naň odkazuje, vytvorí sa nový záznam s ID, ale ostatnými NULL hodnotami) je využívaný súbor ***time_tracker_main_filling****.csv*. V tejto fáze je spracovaných približne 32 miliónov záznamov. Prvé bloky sú nahrávané za približne 30 sekúnd, posledné bloky zaberajú okolo minúty. Celkový čas bloku sa hýbe okolo 4 hodín 20 minút.

Posledná časť opäť prechádza spomenutými 32 miliónmi záznamov. Súbor pre referencie sa volá ***time_tracker_references.csv*****.** Tie sú spustené už v dobe, kedy už je tabuľka konverzácií naplnená a teda čas na blok je konzistentný - tesne pod 20 sekúnd. Tejto konzistencii pomáhajú aj polia načítané v RAM pamäti, ktoré sú mojím bežným použitím, aby sme nemuseli do DB dopytovať pričasto. Čas bloku je približne 100 minút.

Celkový čas behu programu je teda **približne 6 hodín.** Tento priebeh a proces jednotlivých častí programu je možné sledovať aj v logeri, pre predbežnú informáciu.

5. Počet a veľkosť záznamov

Celá databáza má spolu **30 GB dát.** Veľkosť jednotlivých tabuliek je možné vidieť na obrázku nižšie. Pre hlavnú časť riešenia (teda pre všetky tabuľky okrem conversation_references) som preskakoval údaje pri duplicitnom id konverzácie. Pri referenciách som si ale nebol istý, ktorá bola vložená do tabuľky, preto tieto duplicty nie sú preskakované a dát je teda o trochu viac.

table_name	table_size	table_row_count
annotations	1721 MB	19458972
authors	1069 MB	5895176
context_annotations	10 GB	134285948
context_domains	64 kB	88
context_entities	4128 kB	29438
conversation_hashtags	3888 MB	54613745
conversation_references	2402 MB	27950190
conversations	8632 MB	32347011
hashtags	88 MB	773865
links	2022 MB	11540704

(10 rows)