PDT - Indexy

https://github.com/MennoCoehoorn/PDT_2.git

Marek Štrba

October 2021

Contents

1 Vyhľadajte v accounts screen_name s presnou hodnotou 'realDonaldTrump' a analyzujte daný select. Akú metódu vám vybral plánovač a prečo - odôvodnite prečo sa rozhodol tak ako sa rozhodol?

Plánovač vybral **Parallel Seq Scan**. Seq Scan vybral preto, že som robil dopyt na iba neindexovaný stĺpec. Zároveň musel tento select prejsť všetky záznamy tabuľky, čo znamená veľa I/O operácií. Tie sa pri seq scan dopyte vykonávajú sekvenčne a nie náhodne, preto je vykonanie rýchlejšie. Dôvodom môže byť aj to, že postgres má seq scan ako fallback možnosť, ktorá je vykonateľná vždy.

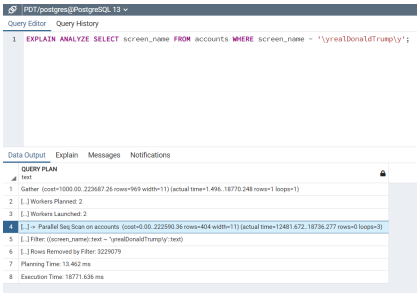


Figure 1

2 Koľko workerov pracovalo na danom selecte a na čo slúžia? Zdvihnite počet workerov a povedzte ako to ovplyvňuje čas. Je tam nejaký strop? Ak áno, prečo? Od čoho to závisí?

Na danom selecte pracovali 2 workeri (Fig 1). Worker je samostatný proces, ktorý sa podieľa na vykonaní rovnakej query. Neparalelné časti query sú vykonané "leader" procesom. Počet workerov je limitovaný parametrami "max_parallel_workers", "max_worker_processes" a "max_parallel_workers_per_gather". Zvýšil som počet "max_parallel_workers_per_gather" (nemôže byť väčší ako počet "max_worker_processes") a následne zbehol znova ten istý select. Ten sa vykonal v neporovnateľne kratšom čase. Limity pre "max_parallel_workers" a "max_parallel_workers_per_gather" sú 1024 a pre "max_worker_processes" je to 262143 (https://postgresqlco.nf). Nastavenia týchto premenných by mali brať do úvahy dostupné HW zdroje, "max_worker_processes" by nemalo byť viac ako je počet jadier na danom stroji
.

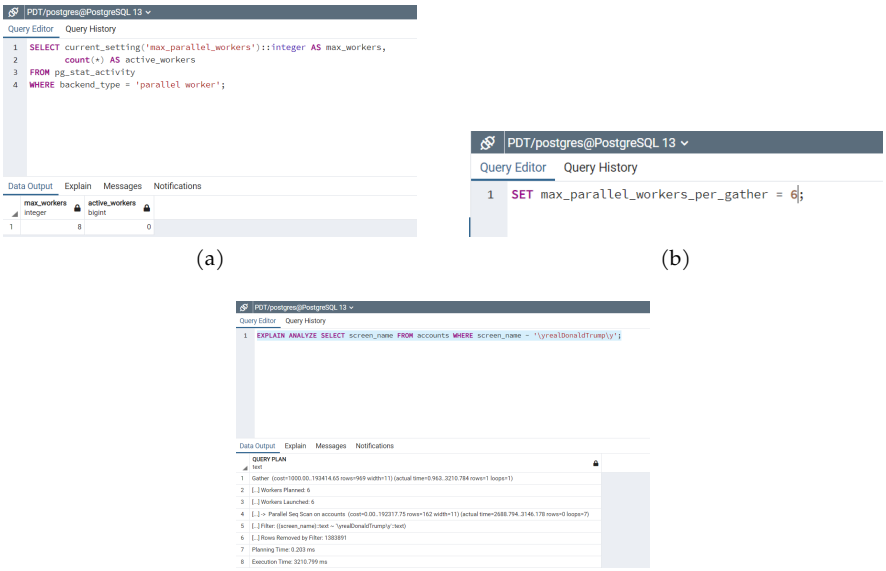
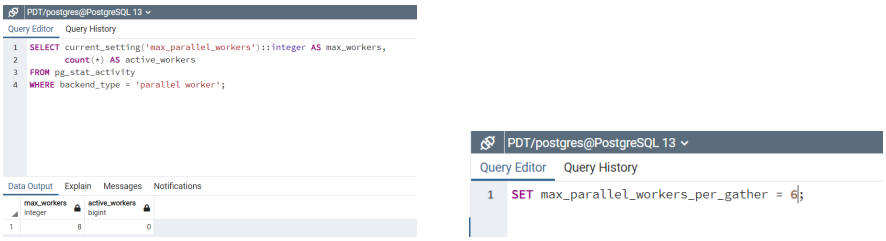


Figure 2: (a) - Aktuálny počet ”max_worker_processes” , (b) - Nastavenie ”max_parallel_workers_per_gather” na 6, (c) - výsledok selectu z predošlej úlohy so 6 workermi

3 Vytvorte btree index nad screen_name a pozrite ako sa zmenil čas a porovnajte výstup oproti požiadavke bez indexu. Potrebuje plánovač v tejto požiadavke viac workerov? Čo ovplyvnilo zásadnú zmenu času?

Počet workerov som znížil späť na 2 aby boli podmienky rovnaké ako pri prvom spustení. Preto boli rovnako použitý dvaja workeri, čiže nepotrebuje viac. Dopyt zbehol kratšie kvôli vytvorenému indexu, plánovač však znova vybral parallel seq scan na vykonanie dopytu.



(a) (b)

Figure 3: (a) - Tvorba indexu , (b) - Dopyt vykonaný s indexom

4 Vyberte používateľov, ktorý majú followers_count väčší, rovný ako 100 a zároveň menší, rovný 200. Je správanie rovnaké v prvej úlohe? Je správanie rovnaké ako v tretej úlohe? Prečo?

Správanie v tejto úlohe je iné ako správanie v prvej a tretej úlohe, keďže namiesto parallel seq scan bol použitý iba seq scan. Toto rozhodnutie s najväčšou pravdepodobnosťou kvôli tomu, že komunikáciu medzi jednotlivými workermi vyhodnotil ako náročnejšiu na zdroje než samotné porovnanie medzi hodnotami followers_count.

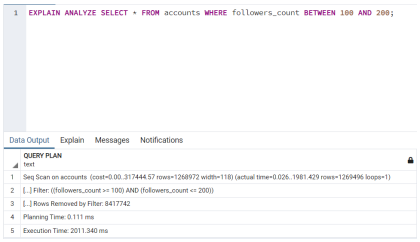
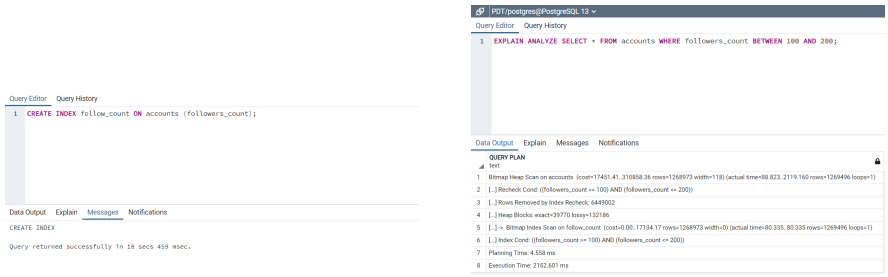


Figure 4

5 Vytvorte index nad 4 úlohou a popíšte prácu s indexom. Čo je to Bitmap Index Scan a prečo je tam Bitmap Heap Scan? Prečo je tam recheck condition?

Tým, že je stĺpec followers_count zindexovaný, tak je zoradený podľa hodnoty, to znamená, že riadky s rovnakou/podobnou hodnotou budú pri sebe, čo urýchly

načítanie. Bitmap Index Scan vytvorí bitmapu tuples - pozícií konkrétnych riadkov (stránok ak to pamäť nedovolí), kde sa na základe indexu nachádzajú dáta, ktoré chceme z databázy získať. Bitmap Heap Scan následne použije túto vytvorenú bitmapu na zistenie, ktoré stránky má kontrolovať kvôli daným záznamom a ktoré môže preskočiť. Takto sa minimalizuje celkový počet i/o operácií a zvýši sa rýchlosť. Pokiaľ sa ale bitmapa tvorená Bitmap Index Scanom stane priveľkou tak sa stane stratovou (pokiaľ to pamäť dovoluje tak sa bitom označuje konkrétny záznam), ak je bitmapa stratová, tak teba prekontrolovať celú stránku, ktorá je bitom reprezentovaná. V EXPLAIN ANALYZE je počet "presných" bitov v čísle Heap Blocks Exact a počet stratových "nepresných" stránkových bitov v čísle Heap Blocks lossy.



(a) (b)

Figure 5: (a) - Tvorba indexu , (b) - Explain analyze nad selectom

6 Vyberte používateľov, ktorí majú followers_count väčší, rovný ako 100 a zároveň menší, rovný 1000? V čom je rozdiel, prečo?

Počet vyhovujúcich záznamov vzrástol z 1 269 496 (13%) na 4 382 646 (45%). Plánovač pre účel tohoto selectu vybral seq scan. To sa z najväčšej pravdepodobnosti stalo preto, že potenciálna bitmapa vytvorená Bitmap Index Scanom by bola lossy do takej miery, že náročnosť réžie (rozdiel vidno na dĺžke planning time oproti Figure 5 (b)) by prekonala náročnosť jednoduchšieho Seq Scanu aj za cenu väčšieho počtu i/o operácií. Parallel Seq Scan nebol vybratý z dôvodu pomerne veľkého počtu vyhovujúcich záznamov.

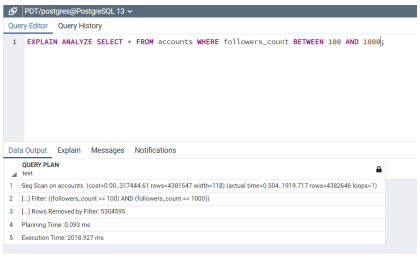
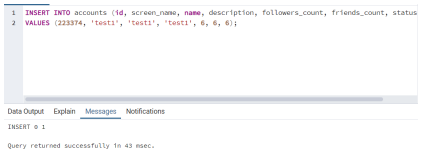


Figure 6

7 Vytvorte daľšie 3 btree indexy na name, friends_count, a description a insertnite si svojho používateľa (to je jedno aké dáta) do accounts. Koľko to trvalo? Dropnite indexy a spravte to ešte raz. Prečo je tu rozdiel?

Insert s indexami trval dlhšie, keďže btree index drží dáta v usporiadanom binárnom strome. To znamená, že každý insert vyžaduje prerátanie týchto indexov, lebo sa musí záznam vložiť na správne miesto podľa poradia na základe stĺpca nad ktorým je daný index vytvorený.

(a)



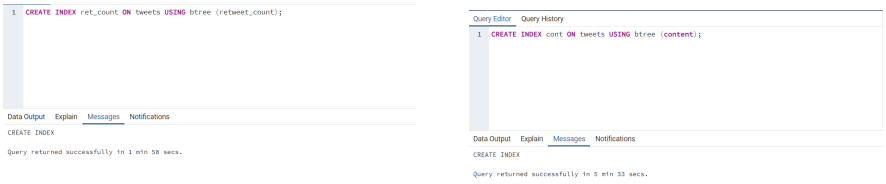
(b)

Figure 7: (a) - Insert s indexami , (b) - Insert bez indexov

8 Vytvorte btree index nad tweetami pre retweet_count a pre content. Porovnajte ich dĺžku vytvárania. Prečo je tu taký rozdiel? Čím je ovplyvnená dĺžka vytvárania indexu a prečo?

Dlhšie sa vytváral index nad contenom keďže pri tvorbe btree indexu sa porovnáva vždy hodnota záznamu s hodnotami, ktoré sú už v strome a porovnať dve čísla trvá menej ako porovnať dva stringy. Zároveň sa narozdieľ od retweet_count

nedá použiť deduplicita. To znamená viac stránok a hlbší btree. Hlbší btree znamená dlhšiu cestu pre záznam pri triedení počas tvorby.



(a) (b)

Figure 8: (a) - Index nad retweetmi , (b) - Index nad contentom

9 Porovnajte indexy pre retweet_count, content, followers_count, screen_name,... v čom sa líšia a prečo (opíšte výstupné hodnoty pre všetky indexy)?

Výsledky dopytov pre retweet_count index a follower_count index mi dali rovnaký výsledok vo všetkých ohľadoch. Túto skutočnosť si neviem vysvetliť, keďže napriek tomu, že sa jedná o rovnaké dátové typy tak sú nad stĺpcami tabuliek s výrazne odlišným počtom záznamov. Zároveň mi príde málo pravdepodobné, že root node by bol rovnaký pre dva rôzne btree.

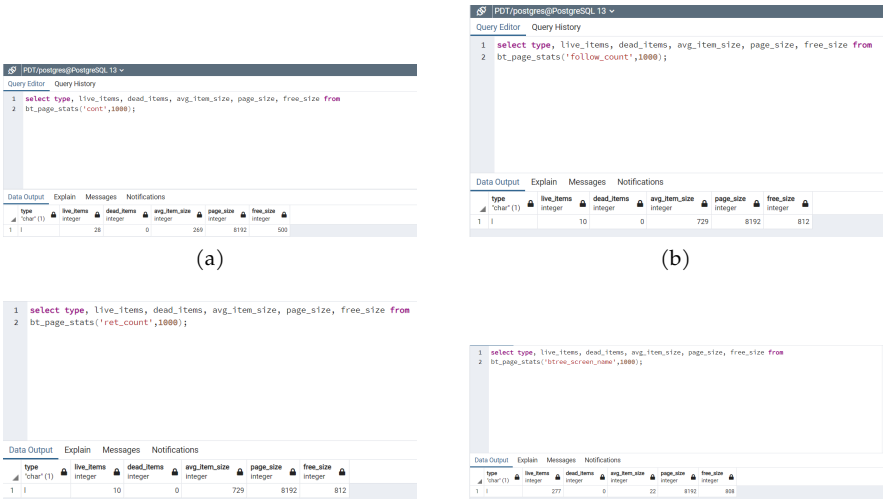
select * from bt_metap('idx')

(a) (b)

(c) (d)

Figure 9: (a) - Výsledok pre content, (b) - výsledok pre followers_count, (c) - výsledok pre retweet_count, (d) - výsledok pre screen_name

select type, live_items, dead_items, avg_item_size, page_size, free_size from bt_page_stats('idx',1000)



(a) (b)

(c) (d)

Figure 10: (a) - Výsledok pre content, (b) - výsledok pre followers_count, (c) - výsledok pre retweet_count, (d) - výsledok pre screen_name

select * from bt_page_items('idx',1) limit 1000

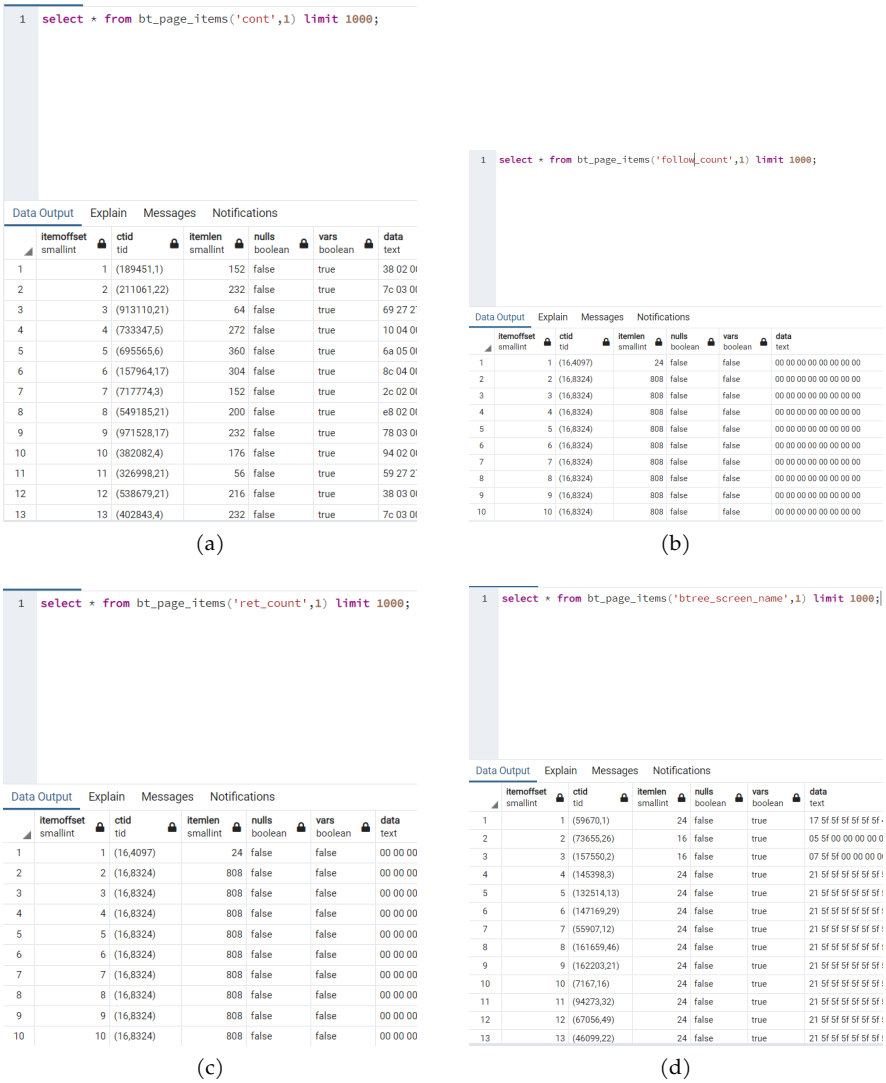


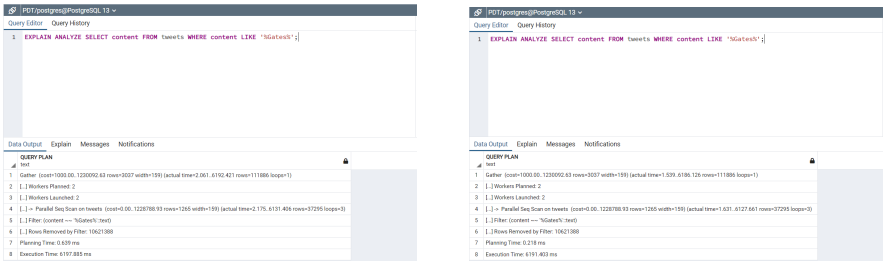
Figure 11: (a) - Výsledok pre content, (b) - výsledok pre followers_count, (c) - výsledok pre retweet_count, (d) - výsledok pre screen_name

Content index má na rovnakom počte záznamov väčšiu hĺbku ako retweet_count, lebo pre varchar a text sa nemôže použiť deduplication. Z toho istého dôvodu majú screen_name a content stĺpec tids nastavený na null (tretí dopyt). Screen_name má menšiu hĺbku ako content, lebo tabuľka accounts má výrazne menej záznamov ako tabuľka tweets. Rozdieľ bol aj v tom, že Screen_name a content mali nas-

tavený kvôli svojmu typu v treťom dopyte stĺpec vars na true. Všetky indexy mali nulls na null a žiadny neobsahoval dead tuples, keďže som nemazal žiadne záznamy. Tiež sa rovnali root a fastroot (rovnako ako level a fast level). Tieto hodnoty sa líšia iba pri veľkom počte deletov.

10 Vyhľadajte v tweets.content meno "Gates" na ľubovoľnom mieste a porovnajte výsledok po tom, ako content naindexujete pomocou btree. V čom je rozdiel a prečo?

V oboch prípadoch sa použil parallel seq scan, keďže sa mohlo slovo "Gates" nachádzať hocikde v rámci contentu tak zoradenie indexom nemalo žiadny pridaný efekt na urýchlenie query, keďže museli byť tak či tak prehľadané všetky záznamy. Rozdiel medzi query s indexom a bez indexu bol iba v čase plánovania. Podľa môjho názoru to bolo spôsobené tým, že plánovač musel najprv vyhodnotiť, či sa index oplatí použiť alebo nie a až potom vykonať rovnakú query, preto plánovanie s indexom trvalo dlhšie ako bez.



(a) (b)

Figure 12: (a) - Bez indexu , (b) - S btree indexom

11 Vyhľadajte tweet, ktorý začína "The Cabel and Deep State". Použil sa index?

Index nebol použitý, keďže podľa plánovača bol použitý parallel seq scan, ktorý nevyužíva index.

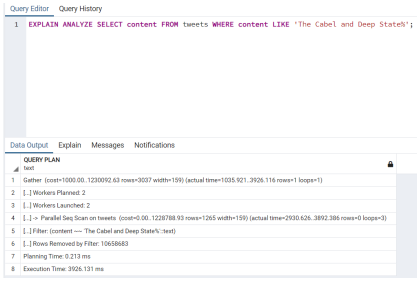
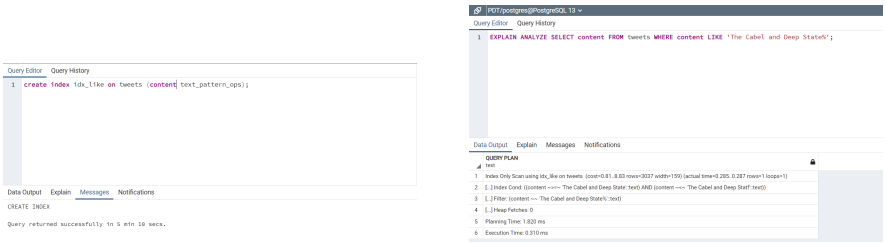


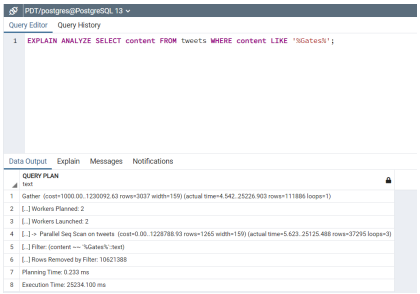
Figure 13

12 Teraz naindexujte content tak, aby sa použil btree index a zhodnoťte prečo sa pred tým nad “The Cabel and Deep State” nepoužil. Použije sa teraz na ”Gates“ na ľubovoľnom mieste? Zdôvodnite použitie alebo nepoužitie indexu?

Index som vytvoril s operátorom text_pattern_ops, v tomto prípade je obsah zoradený ozaj porovnaním písmena po písmene, na rozdieľ od indexu bez operátora, kde sú zoradené podľa lc_collate. Index sa pou6il iba pri “The Cabel and Deep State” a nie pri ”Gates” lebo pri ”Gates” bolo treba preskúmať celý content, pri ”The Cabel...” iba začiatok, čiže zoradenie char by char tam nemá význam.



(a) (b)

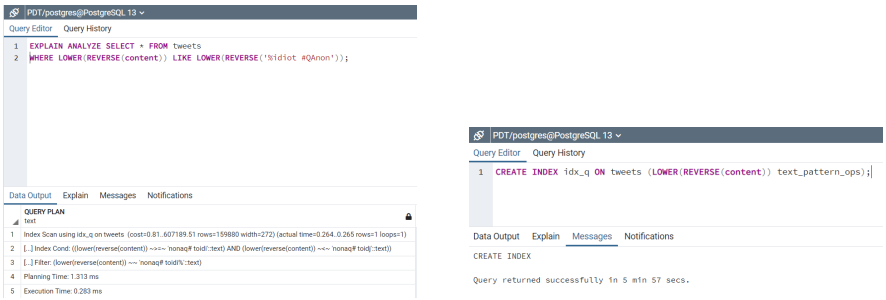


(c)

Figure 14: (a) - Tvorba indexu, (b) - výsledok pre “The Cabel and Deep State”, (c) - výsledok pre ”Gates”

13 Vytvorte nový btree index, tak aby ste pomocou neho vedeli vyhľadať tweet, ktorý končí reťazcom ”idiot #QAnon“ kde nezáleží na tom ako to napíšete. Popíšte čo jednotlivé funkcie robia.

Keďže pri btree indexe sa nedá použiť wildcard na začiatku tak som musel pomocou reverse funkcie otočiť obsah content stĺpca a wildcard znak sa tým pádom dostal na koniec. Zároveň aby som zaručil, že je jedno ako bude napísané, keďže aj content aj porovnávaný dopyt budú malými písmenami a je to rýchlejšie ako ILIKE, ktorý skúma všetky permutácie veľkých a malých písmen.



(a) (b)

Figure 15: (a) - Tvorba indexu , (b) - Explain analyze

14 Nájdite účty, ktoré majú follower_count menší ako 10 a friends_count väčší ako 1000 a výsledok zoraďte podľa statuses_count. Následne spravte jednoduché indexy a popíšte ktoré má a ktoré nemá zmysel robiť a prečo.

Postupne som pridával jednoduché indexy nad stĺpcami no s pridanými indexami stúpal aj planning time ale aj celkový execution time. Pri všetkých troch indexoch bola nutnosť vyrataťdokopy tri bit mapy (jedna naviac pre and) a až následne dokázal postgres použiť bitmap heap scan na vybratie potrebných záznamov. Hlavný dôvod prečo, z môjho pohľadu prekonal rýchlosťou vykonania dopyt bez indexov dopyt s tromi indexami bolo to, že podmienkam nad follower_count a friends_count vyhovovalo iba veľmi málo záznamov, preto vlastne dopyt efektívne musel operovať s prakticky všetkými záznamami a seq scan s menšou réžiou mal lepší celkový výkon.

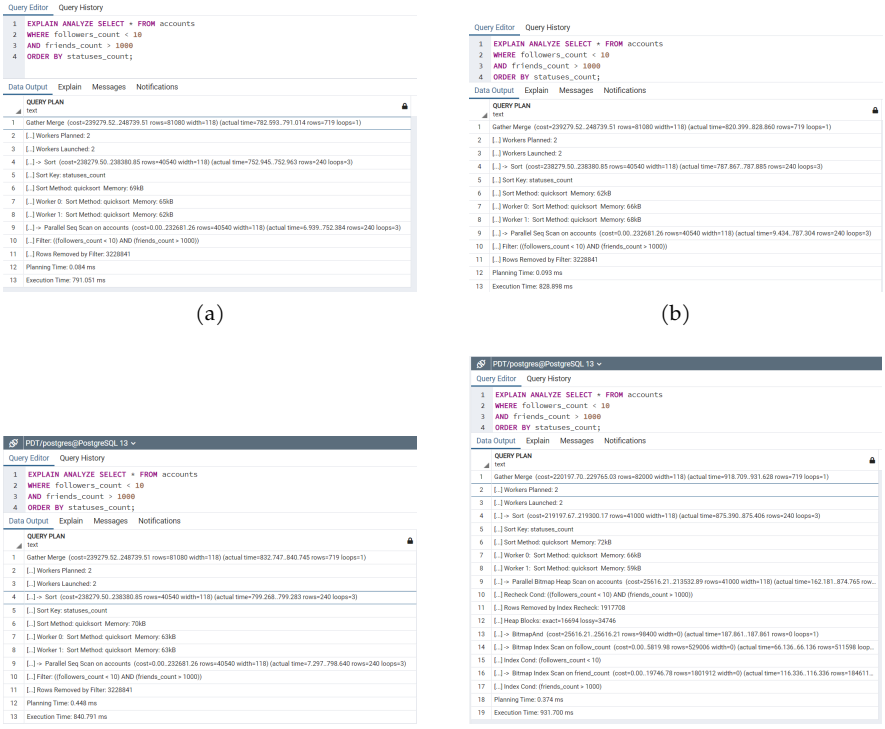


Figure 16: (a) - Výsledok bez indexov, (b) - Výsledok s indexom nad status_count, (c) - výsledok s indexom nad friends_count a status_count, (d) - s indexom nad všetkými 3 stĺpcami

15 Na predošlú query spravte zložený index a porovnajte výsledok s tým, keď sú indexy separátne. Výsledok zdôvodnite.

Po vytvorení zloženého indexu so stĺpcami v správnom poradí (Fig 17 (a)) bol dosiahnutý najlepší výsledok celkového vykonania query. Stĺpce som dal v poradí v akom boli v dopyte do indexu, čím som dosiahol zoradenie najpr podľa friends_count, následne podľa followers_count a na konci podľa statuses_count. Z tohto dôvodu bol výkon o veľa lepší než pri delenom indexe, kde boli záznami zoradené samostatne podľa jednotlivých stĺpcov a nie postupne ako tu.

(a) (b)

Figure 17: (a) - tvorba zloženého indexu , (b) - explain analyze

16 Upravte query tak, aby bol follower_count menší ako 1000 a friends_count väčší ako 1000. V čom je rozdiel a prečo?

Podmienku follower_count menší ako 1000 spĺňa celkovo 6 973 976 z 9 687 241 účtov. To sa dá považovať za značnú väčšinu, z toho dôvodu je pravdepodobné, že musí dopyt navštíviť väčšinu ak nie všetky stránky. Z toho dôvodu sa vybral jednoduchší parallel seq scan, čím sa odbúrala réžia potrebná pri použití indexu, ktorý by vlastne možno vôbec neznížil počet potrebných I/O operácií.

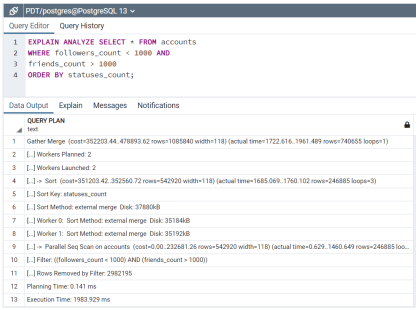


Figure 18

17 Vytvorte vhodný index pre vyhľadávanie písmen bez kontextu nad screen_name v accounts. Porovnajte výsledok pre vyhľadanie presne 'realDonaldTrump' voči btree indexu? Ktorý index sa vybral a prečo? Následne vyhľadajte v texte screen_name 'ldonaldt' a porovnajte výsledky. Aký index sa vybral a prečo?

Vytvoril som trigramový gin index nad stĺpcom screen_name. Porovnávaný bol s btree indexom s varchar ops operátorom. Pri vyhľadaní presne 'realDonaldTrump' to trvalo gin indexu dlhšie ako btree, keďže gin je založený na trigramoch a btree stačilo aby sa podľa zoradenia presunul na miesto, kde má byť 'realDonaldTrump'. Z tohoto dôvodu sa aj pri použití oboch indexov vybral btree. Pri hľadaní 'ldonaldt' v texte sa použil gin, keďže bolo treba prehľadať celý screen name na výskyty 'ldonaldt', v čom obyčajný btree nijako nepomôže.

```
CREATE INDEX gin_screen_name ON accounts USING gin (screen_name gin_trgm_ops);
```

Figure 19: Tvorba gin indexu

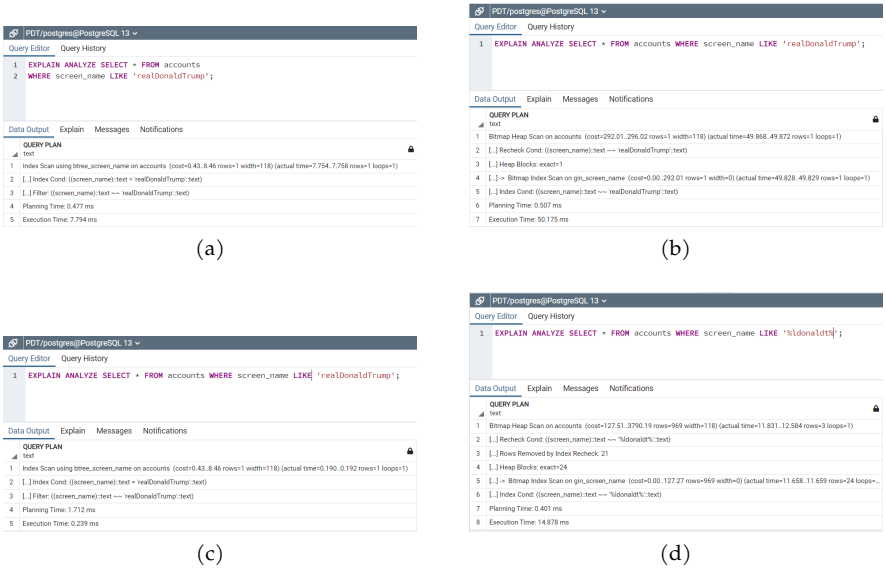
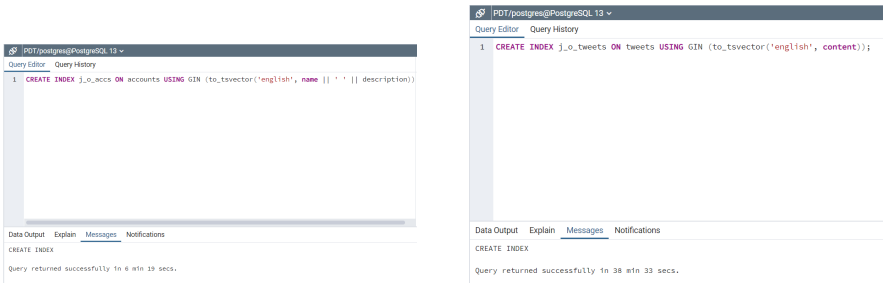


Figure 20: (a) - Btree, (b) - GIN, (c) - výber po implementácií oboch indexov, (d) - vyhľadávanie ”%ldonaldt%

18 Vytvorte query pre slová ”John” a ”Oliver” pomocou FTS (tsvector a tsquery) v angličtine v stĺpcoch tweets.content, accounts.decription a accounts.name, kde slová sa môžu nachádzať v prvom, druhom ALEBO treťom stĺpci. Teda vyhovujúci záznam je ak aspoň jeden stĺpec má ”match“. Výsledky zoraďte podľa retweet_count zostupne. Pre túto query vytvorte vhodné indexy tak, aby sa nepoužil ani raz sekvenčný scan (správna query dobehne rádovo v milisekundách, max sekundách na super starých PC). Zdôvodnite čo je problém s OR podmienkou a prečo AND je v poriadku pri joine.

Ako prvé som si vytvoril GIN indexy nad tabuľkami accounts (stĺpce description a name) a tweets (stĺpec content) na prácu s tsvector a tsquery. Index nad tabuľkou tweets sa tvoril 38 minút.



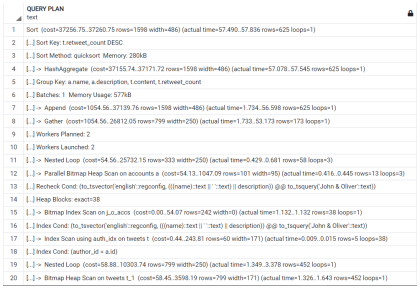
(a) (b)

Figure 21: (a) - Tvorba indexu tabuľka accounts, (b) - Tvorba indexu tabuľka tweets

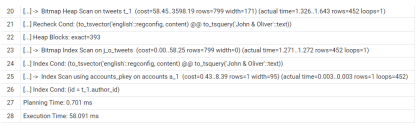
Na urýchlenie joinu som vytvoril jednoduchý btree index v tabuľke tweets nad stĺpcom author_id. Stĺpec id v accounts nebolo treba indexovať, keďže sa jedná o primary key. Pri samotnom selecte som postupoval následovne:

Select som rozdelil na dva samostatné selecty a tým som sa vyhol OR podmienke, ktorá nedokáže použiť indexy, ktoré som vytvoril na prácu s tsvectorom a tsquery. Pri čiastkových selectoch som vždy predfiltroval jednu z tabuliek za použitia podmienky s tsvectorom a tsquery, na ktorú som mal pripravený gin index. Takto prefiltrovaná tabuľka bola v joine vždy naľavo, lebo bola menšia. Z týchto joinov som vždy vybral všetky stĺpce, ktoré som potreboval (a.name, a.description, t.content, t.retweet_count). Výsledky daných čiastkových selectov som následne spojil pomocou príkazu UNION. Problém bol ale, že sa mohli vo výsledkoch objaviť duplicity (napríklad ak by sa slová "John" a "Oliver" nachádzali aj v description aj v content, vtedy by sa rovnaký tweet objavil v oboch čiastkových selectoch). Toto by bol problém iba ak by som použil UNION ALL, ale pri použití UNION by sa mali duplikáty samé vytriediť. Výsledok tejto query som následne zoradil cez ORDER BY DESC podľa stĺpca retweet_count, nad ktorým bol urobený jednoduchý btree index.

(a)



(b)



(c)

Figure 22: (a) - výsledná query, výsledok query plánu bol príliš dlhý pre jeden screenshot, preto je rozdelený na dve časti (b,c)

Podmienka AND pri joine nevadí, lebo sa na jej základe prefiltruje jedna z tabuliek za použitia indexu a tým pádom sa zníži počet riadkov, ktoré sa v joine spájajú a zrýchly sa samostný join.