

Zadanie 2 – vyhľadávanie a indexovanie

Odovzdanie do 24.10.2021 23:59 – máte na to 2 týždne – dostanete za to 7,5 boda.

Otázky 2-17 sú dokopy za 5,5 boda (každá rovnako). Zadanie 18 je za 2 body. Teda good luck and have fun.

Zadania prosím neopisujte jednoslovné ale zmysluplnou vetou (nie slohová práca - teda vecne). Vždy priložte screenshot z explain analyse.

GITHUB = <https://github.com/jaruji/PDT-2>

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?

```
EXPLAIN ANALYSE SELECT screen_name FROM accounts WHERE screen_name = 'realDonaldTrump'
```

	QUERY PLAN
	text
1	Gather (cost=1000.00..223590.46 rows=1 width=11) (actual time=0.345..834....
2	[...] Workers Planned: 2
3	[...] Workers Launched: 2
4	[...] -> Parallel Seq Scan on accounts (cost=0.00..222590.36 rows=1 width=11)...
5	[...] Filter: ((screen_name)::text = 'realDonaldTrump'::text)
6	[...] Rows Removed by Filter: 3229079
7	Planning Time: 0.082 ms
8	Execution Time: 834.031 ms

Paralelný scan bol zvolený z dôvodu, že nad stĺpcom screen_name nebol nikdy vytvorený index. Taktiež je podmienka špecifická, čiže si môžeme dovoliť ju rozdeliť do viacerých workerov (preto par. seq. scan)

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 selecte pracovali 2 workeri - slúžia na paralelné spracovanie query pre lepšiu efektivitu (treba však mať na mysli že spotrebuje viac zdrojov, keďže sa spustí paralelne n procesov - preto je vhodné mať rozumne nastavenú hranicu, resp. strop pre počet workerov). Maximálny počet workerov (strop) vieme zmeniť príkazom

max_parallel_workers_per_gather = hodnota. Strop závisí od nastavenia tohoto parametra (default hodnota je 2). Ak chceme vypnúť paralelné spracovanie môžeme tento parameter nastaviť na hodnotu 0. Môžeme vidieť že zdvojnásobenie workerov malo za výsledok rýchlejšie vykonanie query - v mojom prípade z 800ms na 600ms.

```
SET max_parallel_workers_per_gather = 4;
EXPLAIN ANALYSE SELECT screen_name FROM accounts WHERE screen_name = 'realDonaldTrump';
```

	QUERY PLAN	
	text	
1	Gather (cost=1000.00..203408.72 rows=1 width=11) (actual time=0.423..607.119 rows=1 loops=1)	
2	[...] Workers Planned: 4	
3	[...] Workers Launched: 4	
4	[...] -> Parallel Seq Scan on accounts (cost=0.00..202408.62 rows=1 width=11) (actual time=416.674..536.782 rows=0 loops=5)	
5	[...] Filter: ((screen_name)::text = 'realDonaldTrump'::text)	
6	[...] Rows Removed by Filter: 1937447	
7	Planning Time: 0.063 ms	
8	Execution Time: 607.136 ms	

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

Index vytvoríme príkazom *CREATE INDEX*. Metódu netreba v tomto prípade zadávať, keďže default je nastavená btree (z postgres dokumentácie).

```
CREATE INDEX screen_name_btree_index ON accounts(screen_name);
```

```
CREATE INDEX
```

```
Query returned successfully in 1 min 42 secs.
```

	QUERY PLAN	
	text	
1	Index Only Scan using screen_name_btree_index on accounts (cost=0.43..4.45 rows=1 width=11) (actual time=0.341..0.342 rows=1 loops=1)	
2	[...] Index Cond: (screen_name = 'realDonaldTrump'::text)	
3	[...] Heap Fetches: 0	
4	Planning Time: 1.117 ms	
5	Execution Time: 0.354 ms	

Môžeme si všimnúť, že už nie je využívaný parallel seq. scan (z dôvodu existencie indexu) a tým pádom nie sú využívaný viacerí workeri. Požiadavka je omnoho rýchlejšia

(0.3ms) - dôvodom je opäťovne existencia indexu nad stĺpcom screen_name v tabuľke accounts.

4. Vyberte používateľov, ktorí 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 sa odlišuje aj od prvej a aj od tretej úlohy. V tejto situácii sa totižto využíva sekvenčný scan - nie paralelný sekv. scan a ani index only scan (neexistuje index). Dôvodom zmeny metódy je ich efektivita, ktorú vyhodnotil plánovač - pokiaľ query nie je dosť špecifická, klasický sekvenčný scan môže byť efektívnejší ako paralelný sekvenčný scan (z dôvodu presúvania veľkého množstva tuples medzi workerami - <https://www.2ndquadrant.com/en/blog/postgresql96-parallel-sequential-scan/>).

```
EXPLAIN ANALYSE SELECT * FROM accounts WHERE followers_count >= 100 AND followers_count <= 200;
```

QUERY PLAN	
	text
1	Seq Scan on accounts (cost=0.00..317444.57 rows=1248378 width=118) (actual time=0.052..1972.474 rows=1269496 loops=1)
2	[...] Filter: ((followers_count >= 100) AND (followers_count <= 200))
3	[...] Rows Removed by Filter: 8417742
4	Planning Time: 0.083 ms
5	Execution Time: 2014.596 ms

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?

```
CREATE INDEX followers_count_index ON accounts(followers_count);
```

```
CREATE INDEX
```

```
Query returned successfully in 15 secs 755 msec.
```

Bitmap Index Scan - Skonstruuje bitmapu potenciálnych lokácií záznamov (niečo medzi seq scan a index scan - vychádza z faktu že dáta sa lepšie spracúvajú v bulkoch)

Bitmap Heap Scan - dekoduje čo uložil index scan, použije recheck condition aby sa opätovne overila podmienka

recheck condition - tie záznamy ktoré sme našli treba opätovne preveriť, lebo nemusia spĺňať podmienku (neuložil som jednotlivé záznamy ale celé stránky).

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

```
EXPLAIN ANALYSE SELECT * FROM accounts WHERE followers_count >= 100 AND followers_count <= 1000;
```

```
Successfully run. Total query runtime: 6 secs 547 msec.  
4382646 rows affected.
```

	QUERY PLAN
1	Seq Scan on accounts (cost=0.00..317444.57 rows=4407069 width=118) (actual time=0.047..2009.612 rows=4382646 loops=1)
2	[...] Filter: ((followers_count >= 100) AND (followers_count <= 1000))
3	[...] Rows Removed by Filter: 5304592
4	Planning Time: 1.351 ms
5	Execution Time: 2138.100 ms

Sekvenčný scan sa zvolil lebo pracujeme s veľkým počtom záznamov a plánovač vyhodnotil, že tento scan bude efektívnejší ako index scan (operácia seq. scan môže byť rýchlejšia ako skákanie pomocou indexu).

7. Vytvorte ďalšie 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?


```
CREATE INDEX name_btree_index ON accounts(name);  
CREATE INDEX friends_count_btree_index ON accounts(friends_count);  
CREATE INDEX description_btree_index ON accounts(description);  
|
```

```
CREATE INDEX
```

```
Query returned successfully in 4 min 24 secs.
```




```
EXPLAIN ANALYSE INSERT INTO accounts(screen_name, name, description, followers_count, friends_count, statuses_count)
VALUES ('PDT', ':-)', ':->', 10, 5, 3);
```

QUERY PLAN		
	text	
1	Insert on accounts (cost=0.00..0.01 rows=1 width=888) (actual time=8.708..8.709 rows=0 loops=1)	
2	[...] -> Result (cost=0.00..0.01 rows=1 width=888) (actual time=3.314..3.314 rows=1 loops=1)	
3	Planning Time: 0.978 ms	
4	Execution Time: 8.749 ms	

```
DROP INDEX name_btree_index;
DROP INDEX friends_count_btree_index;
DROP INDEX description_btree_index;
|
```

```
DROP INDEX
```

```
Query returned successfully in 153 msec.
```

QUERY PLAN		
	text	
1	Insert on accounts (cost=0.00..0.01 rows=1 width=888) (actual time=1.145..1.145 rows=0 loops=1)	
2	[...] -> Result (cost=0.00..0.01 rows=1 width=888) (actual time=0.007..0.008 rows=1 loops=1)	
3	Planning Time: 0.029 ms	
4	Execution Time: 1.188 ms	

Vkladanie hodnôt do indexovaných stĺpcov je pomalšie pretože tieto indexy je potrebné dodatočne aktualizovať o pridávané hodnoty. Čím viac indexov má tabuľka, tým pomalšia bude operácia INSERT.

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?

```
CREATE INDEX retweet_count_btree_index ON tweets(retweet_count);
```

```
CREATE INDEX
```

```
Query returned successfully in 2 min 7 secs.
```

```
CREATE INDEX content_btree_index ON tweets(content);
```

Pre content je vytváranie indexu mnohonásobne pomalšie z dôvodu znakov/text (varchar) obsiahnutých v tomto atribúte - nejedná sa o integer, ktorého indexácia je rýchlejšia (varchar overhead je väčší).

```
CREATE INDEX
```

```
Query returned successfully in 17 min.
```

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

- create extension pageinspect;
- select * from bt_metap('idx_content');
- select type, live_items, dead_items, avg_item_size, page_size, free_size from bt_page_stats('idx_content',1000);
- select * from bt_page_items('idx_content',1) limit 1000;

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

Bez indexu:

```
EXPLAIN ANALYSE SELECT content FROM tweets WHERE content LIKE '%Gates%';
```

QUERY PLAN		text	
1	Gather (cost=1000.00..1270579.83 rows=3029 width=158) (actual time=0.397..6797.110 rows=111886 loops=1)		
2	[...] Workers Planned: 2		
3	[...] Workers Launched: 2		
4	[...] -> Parallel Seq Scan on tweets (cost=0.00..1269276.93 rows=1262 width=158) (actual time=1.679..6737.132 rows=37295 loops=3)		
5	[...] Filter: (content ~~ '%Gates%':text)		
6	[...] Rows Removed by Filter: 10621388		
7	Planning Time: 2.734 ms		
8	Execution Time: 6801.996 ms		

S indexom:

```
EXPLAIN ANALYSE SELECT content FROM tweets WHERE content LIKE '%Gates%';
```

	QUERY PLAN	
	text	
1	Gather (cost=1000.00..1270579.83 rows=3029 width=158) (actual time=0.489..6899.473 rows=111886 loops=1)	
2	[...] Workers Planned: 2	
3	[...] Workers Launched: 2	
4	[...] -> Parallel Seq Scan on tweets (cost=0.00..1269276.93 rows=1262 width=158) (actual time=0.425..6810.599 rows=37295 loops=3)	
5	[...] Filter: (content ~~ '%Gates%':text)	
6	[...] Rows Removed by Filter: 10621388	
7	Planning Time: 0.192 ms	
8	Execution Time: 6904.517 ms	

Výsledok bol v podstate totožný, z čoho vyplýva, že index v tejto situácii nemal na SELECT žiaden dopad (nevyužil sa, preto sa použil paralelný seq. scan)

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

```
EXPLAIN ANALYSE SELECT content FROM tweets WHERE content LIKE 'The Cabel and Deep State%';
```

1	Gather (cost=1000.00..1270579.83 rows=3029 width=158) (actual time=19879.799..44584.445 rows=1 loops=1)
2	[...] Workers Planned: 2
3	[...] Workers Launched: 2
4	[...] -> Parallel Seq Scan on tweets (cost=0.00..1269276.93 rows=1262 width=158) (actual time=36305.988..44539.287 rows=0 loops=3)
5	[...] Filter: (content ~~ 'The Cabel and Deep State%':text)
6	[...] Rows Removed by Filter: 10658683
7	Planning Time: 3.951 ms
8	Execution Time: 44584.464 ms

Index sa nepoužil (použil sa parallel seq. scan).

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?

```
CREATE INDEX content_btree_index ON tweets(content text_pattern_ops);
```

```
CREATE INDEX
```

```
Query returned successfully in 8 min 55 secs.
```



```
EXPLAIN ANALYSE SELECT content FROM tweets WHERE content LIKE '%Gates%';
```

	QUERY PLAN	
	text	
1	Gather (cost=1000.00..1270579.83 rows=3029 width=158) (actual time=1.918..44979.122 rows=111886 loops=1)	
2	[...] Workers Planned: 2	
3	[...] Workers Launched: 2	
4	[...] -> Parallel Seq Scan on tweets (cost=0.00..1269276.93 rows=1262 width=158) (actual time=1.537..44897.434 rows=37295 loops=3)	
5	[...] Filter: (content ~~ '%Gates%':text)	
6	[...] Rows Removed by Filter: 10621388	
7	Planning Time: 4.470 ms	
8	Execution Time: 44987.149 ms	

Pri tejto úlohe sa nový index opäťovne nepoužil (wildcard je na oboch stranách reťazca).

```
EXPLAIN ANALYSE SELECT content FROM tweets WHERE content LIKE 'The Cabel and Deep State%';
```

	QUERY PLAN	
	text	
1	Index Only Scan using content_btree_index on tweets (cost=0.81..8.83 rows=3029 width=158) (actual time=1.713..1.715 rows=1 loops=1)	
2	[...] Index Cond: ((content ~>= 'The Cabel and Deep State':text) AND (content ~<= 'The Cabel and Deep State':text))	
3	[...] Filter: (content ~~ 'The Cabel and Deep State':text)	
4	[...] Heap Fetches: 0	
5	Planning Time: 0.370 ms	
6	Execution Time: 1.728 ms	

Nový index sa použil lebo parameter text_pattern_ops je vhodný pre použitie pri operácii LIKE (porovnávanie striktné znak po znaku <https://www.postgresql.org/docs/9.4/indexes-opclass.html>). Vieme presne kde reťazec začína (znakom T)).

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.

Čítal som že sa to dá cez pg_trm

(<https://www.cybertec-postgresql.com/en/postgresql-more-performance-for-like-and-ilike-statements/>) ale to nie je btree index.

```
EXPLAIN ANALYSE SELECT content FROM tweets WHERE content ILIKE '%idiot #QAnon';
```

?

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

```
EXPLAIN ANALYSE SELECT * from accounts
WHERE followers_count < 10 AND friends_count > 1000
ORDER BY statuses_count DESC;
```

	QUERY PLAN	
	text	
1	Gather Merge (cost=239115.83..248314.47 rows=78840 width=118) (actual time=880.374..888.852 rows=719 loops=1)	
2	[...] Workers Planned: 2	
3	[...] Workers Launched: 2	
4	[...] -> Sort (cost=238115.80..238214.35 rows=39420 width=118) (actual time=832.038..832.059 rows=240 loops=3)	
5	[...] Sort Key: statuses_count DESC	
6	[...] Sort Method: quicksort Memory: 72kB	
7	[...] Worker 0: Sort Method: quicksort Memory: 61kB	
8	[...] Worker 1: Sort Method: quicksort Memory: 63kB	
9	[...] -> Parallel Seq Scan on accounts (cost=0.00..232681.25 rows=39420 width=118) (actual time=2.229..828.640 rows=240 loops=3)	
10	[...] Filter: (((followers_count < 10) AND (friends_count > 1000)))	
11	[...] Rows Removed by Filter: 3228840	
12	Planning Time: 1.695 ms	
13	Execution Time: 888.900 ms	

```
CREATE INDEX friends_count_btree_index ON accounts(friends_count);
CREATE INDEX followers_count_index ON accounts(followers_count);
CREATE INDEX statuses_count_index ON accounts(statuses_count);
```

```
CREATE INDEX
```

Query returned successfully in 25 secs 30 msec.

```
EXPLAIN ANALYSE SELECT * from accounts
WHERE followers_count < 10 AND friends_count > 1000
ORDER BY statuses_count DESC;
```

	QUERY PLAN text	
1	Gather Merge (cost=218164.91..227485.12 rows=79882 width=118) (actual time=1096.892..1110.260 rows=719 loops=1)	
2	[...] Workers Planned: 2	
3	[...] Workers Launched: 2	
4	[...] -> Sort (cost=217164.88..217264.74 rows=39941 width=118) (actual time=1057.581..1057.603 rows=240 loops=3)	
5	[...] Sort Key: statuses_count DESC	
6	[...] Sort Method: quicksort Memory: 73kB	
7	[...] Worker 0: Sort Method: quicksort Memory: 62kB	
8	[...] Worker 1: Sort Method: quicksort Memory: 62kB	
9	[...] -> Parallel Bitmap Heap Scan on accounts (cost=25839.87..211651.78 rows=39941 width=118) (actual time=156.754..1057.027 rows=24...	
10	[...] Recheck Cond: ((followers_count < 10) AND (friends_count > 1000))	
11	[...] Rows Removed by Index Recheck: 1917707	
12	[...] Heap Blocks: exact=16458 lossy=34522	
13	[...] -> BitmapAnd (cost=25839.87..25839.87 rows=95859 width=0) (actual time=182.026..182.027 rows=0 loops=1)	
14	[...] -> Bitmap Index Scan on followers_count_index (cost=0.00..5526.87 rows=502191 width=0) (actual time=63.582..63.582 rows=511595 lo...	
15	[...] Index Cond: (followers_count < 10)	
16	[...] -> Bitmap Index Scan on friends_count_btree_index (cost=0.00..20264.82 rows=1849118 width=0) (actual time=112.589..112.589 rows=...	
17	[...] Index Cond: (friends_count > 1000)	
18	Planning Time: 0.891 ms	
19	Execution Time: 1110.338 ms	

Indexy sa použili. Zmysel má vytvárať indexy pre klauzulu WHERE (teda columny followers_count a friends_count). Index pre statuses_count zmysel vytvárať nemá.

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

```
CREATE INDEX turbo_index ON accounts(friends_count, followers_count, statuses_count);
```

```
CREATE INDEX
```

```
Query returned successfully in 20 secs 587 msec.
```

```
EXPLAIN ANALYSE SELECT * from accounts
WHERE followers_count < 10 AND friends_count > 1000
ORDER BY statuses_count DESC;
```

	QUERY PLAN	
	text	
1	Gather Merge (cost=235742.23..244940.87 rows=78840 width=118) (actual time=100.274..105.020 rows=719 loops=1)	
2	[...] Workers Planned: 2	
3	[...] Workers Launched: 2	
4	[...] -> Sort (cost=234742.20..234840.75 rows=39420 width=118) (actual time=62.659..62.677 rows=240 loops=3)	
5	[...] Sort Key: statuses_count DESC	
6	[...] Sort Method: quicksort Memory: 83kB	
7	[...] Worker 0: Sort Method: quicksort Memory: 57kB	
8	[...] Worker 1: Sort Method: quicksort Memory: 56kB	
9	[...] -> Parallel Bitmap Heap Scan on accounts (cost=44567.27..229307.65 rows=39420 width=118) (actual time=57.450..62.503 rows=240 lo...	
10	[...] Recheck Cond: (((friends_count > 1000) AND (followers_count < 10)))	
11	[...] Heap Blocks: exact=333	
12	[...] -> Bitmap Index Scan on turbo_index (cost=0.00..44543.61 rows=94607 width=0) (actual time=94.485..94.485 rows=719 loops=1)	
13	[...] Index Cond: (((friends_count > 1000) AND (followers_count < 10)))	
14	Planning Time: 1.385 ms	
15	Execution Time: 105.124 ms	

Index sa opätovne použil. Vykonanie bolo rýchlejšie ako pri jednotlivých indexoch - vyhľadávame iba v jednom indexe namiesto troch.

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

```
EXPLAIN ANALYSE SELECT * from accounts
WHERE followers_count < 1000 AND friends_count > 1000
ORDER BY statuses_count DESC;
```

```
DROP INDEX friends_count_btree_index;
```

	QUERY PLAN	
	text	
1	Gather Merge (cost=354934.21..484445.58 rows=1110020 width=118) (actual time=1117.687..1353.667 rows=740655 loops=1)	
2	[...] Workers Planned: 2	
3	[...] Workers Launched: 2	
4	[...] -> Sort (cost=353934.18..355321.71 rows=555010 width=118) (actual time=1069.662..1146.162 rows=246885 loops=3)	
5	[...] Sort Key: statuses_count DESC	
6	[...] Sort Method: external merge Disk: 38856kB	
7	[...] Worker 0: Sort Method: external merge Disk: 35376kB	
8	[...] Worker 1: Sort Method: external merge Disk: 34064kB	
9	[...] -> Parallel Seq Scan on accounts (cost=0.00..232681.25 rows=555010 width=118) (actual time=0.230..862.017 rows=246885 loops=3)	
10	[...] Filter: (((followers_count < 1000) AND (friends_count > 1000)))	
11	[...] Rows Removed by Filter: 2982195	
12	Planning Time: 0.124 ms	
13	Execution Time: 1381.517 ms	

Pri zmene podmienky sa nám nepoužil index - máme omnoho viac záznamov, plánovač sa rozhodol, že paralel. seq. scan je efektívnejší (pokiaľ je počet záznamov menší ako 5-10%, oplatí sa používať index <https://thoughtbot.com/blog/why-postgres-wont-always-use-an-index> - inak je efektívnejšie získavať riadky priamo z tabuľky a nie prostredníctvom indexu).

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

```
CREATE EXTENSION pg_trgm;
```

```
CREATE INDEX screen_name_gist_index ON accounts USING gist (screen_name gist_trgm_ops);
CREATE INDEX screen_name_btree_index ON accounts(screen_name);
```

```
CREATE INDEX
```

```
Query returned successfully in 4 min 54 secs.
```

```
EXPLAIN ANALYSE SELECT * from accounts WHERE screen_name = 'realDonaldTrump'
```

QUERY PLAN		
	text	
1	Index Scan using screen_name_btree_index on accounts (cost=0.43..8.45 rows=1 width=118) (actual time=0.051..0.052 rows=1 loops=1)	
2	[...] Index Cond: ((screen_name)::text = 'realDonaldTrump'::text)	
3	Planning Time: 0.116 ms	
4	Execution Time: 0.070 ms	

Použil sa btree index.

```
EXPLAIN ANALYSE SELECT * from accounts WHERE screen_name LIKE '%realDonaldTrump%'
```

QUERY PLAN		
	text	
1	Bitmap Heap Scan on accounts (cost=51.93..3714.61 rows=969 width=118) (actual time=125.072..125.074 rows=1 loops=1)	
2	[...] Recheck Cond: ((screen_name)::text ~~ '%realDonaldTrump% '::text)	
3	[...] Heap Blocks: exact=1	
4	[...] -> Bitmap Index Scan on screen_name_gist_index (cost=0.00..51.68 rows=969 width=0) (actual time=125.008..125.008 rows=1 loops=1)	
5	[...] Index Cond: ((screen_name)::text ~~ '%realDonaldTrump% '::text)	
6	Planning Time: 0.141 ms	
7	Execution Time: 125.917 ms	

pg_trm indexy sú vhodné pre optimalizáciu LIKE a ILIKE queries (aj kde sú wildcardy na oboch stranách reťazca, resp. vyhľadávania substringov v texte), preto ich (gist indexy) pri takomto query plánovač preferuje. Pri obyčajnom =, t-j. pri hľadaní presnej zhody celého stringu je vhodnejší klasický btree index.

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 zoradíte 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.
