Spis treści

Parametry bazy

Parametry bazy danych wpływające na jej wydajność

Zanim zaczniemy, musisz wiedzieć że podczas instalacji wszelkie parametry wpływające na wydajność są konfigurowane dosyć „minimalistycznie”, tak by tylko PostgreSQL był w stanie działać. Jeśli więc dysponujemy np. zapasem pamięci lub przestrzeni dyskowej ponad ustawienia domyślne, mamy pole do popisu w optymalizacji. Wiele też zależy od specyfiki i zastosowania konkretnej bazy danych, a domyślna konfiguracja jest powiedzmy – uniwersalna.

Bufory

shared_buffers

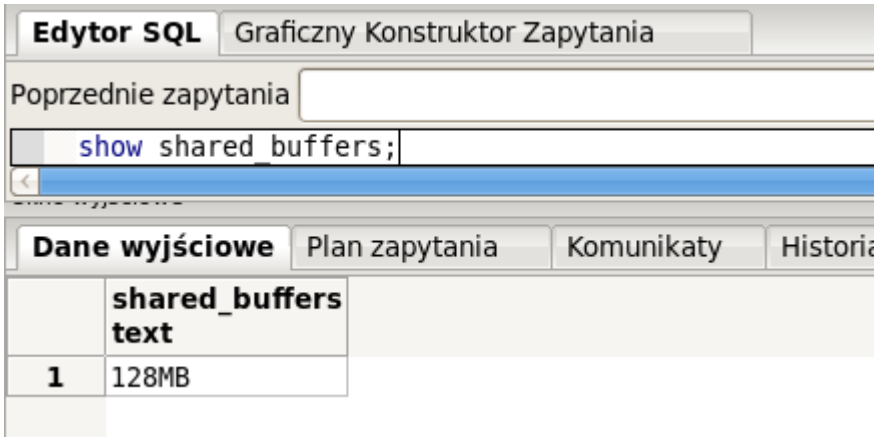
Parametr ten dotyczy wielkości podstawowego bufora pamięciowego PostgreSQL. Służy on buforowaniu danych podczas operacji zapisu i odczytu w bazie. Jest to jeden z najważniejszych parametrów wpływających na wydajność bazy danych, a jego właściwe ustawienie pozwoli nam uzyskać odczuwalną poprawę wydajności.

Jeśli posiadasz więcej niż 1GB pamięci na serwerze (nie to nie jest pomyłka. Dla wielu serwerów produkcyjnych jest to ułamek wielkości pamięci, ale weź też pod uwagę że PostgreSQL jest też często stosowaną bazą na „mini-serwerach” postawionych np. na Raspberry Pi) wskazane jest ustawienie wielkości shared_buffers na 25% wielkości pamięci. Na serwerach mających mniej niż 1GB pamięci musimy odnaleźć złoty środek między wydajnością serwera PostgreSQL, a innym oprogramowaniem – choćby sam system operacyjny również potrzebuje pamięci! Przy wartościach niższych niż 1GB wskazane jest ustawienie na 15% wielkości pamięci. Pamiętaj że PostgreSQL będzie również wykorzystywał bufor systemu operacyjnego ponad ustawienie shared_buffers.

Zmiana wartości tego parametru na wyższą jest jednym z najprostszych sposobów na optymalizację wydajności PostgreSQL. Zasadniczo shared_buffer służy jako cache, więc im jest większy tym więcej może być w nim składowane. Im więcej jest w nim składowane tym rzadziej będzie potrzeba odczytywania danych z dysku, a jak powszechnie wiadomo – odczyty z dysku są jednym z najczęstszych przyczyn spadku wydajności, ponieważ są znacznie wolniejsze od odczytów z pamięci.

Sprawdzić wielkość tego bufora możemy zwykłym poleceniem SHOW:

show shared_buffer;

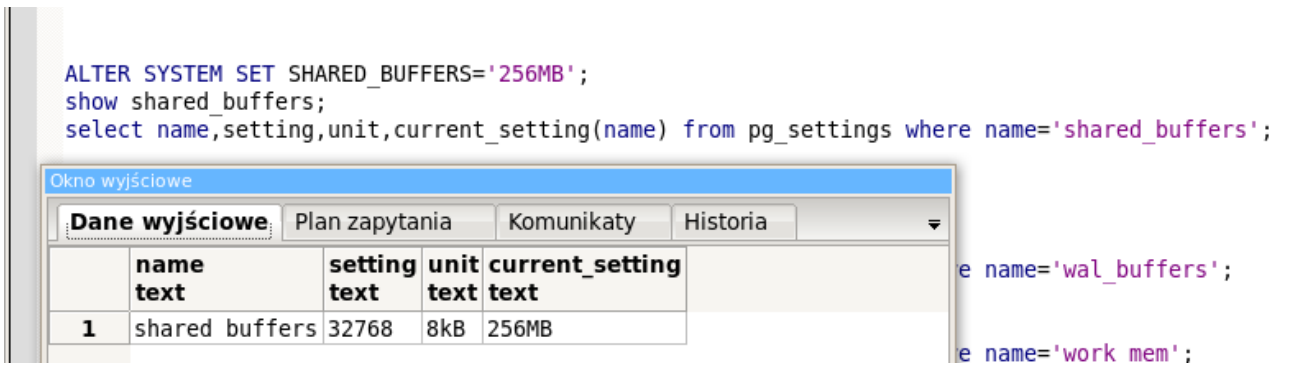


Zmianę wartości wielkości tego bufora możemy albo edytując bezpośrednio plik postgresql.conf, albo stosując instrukcję alter system (dostępną od wersji 9.4). Zmiana z użyciem alter system spowoduje dopisanie nowej wartości parametru do pliku postgresql.auto.conf który jest czytany dodatkowo razem z postgresql.conf przy starcie systemu. W podobny sposób możemy modyfikować każdy omawiany tutaj parametr. Należy pamiętać o kolejności przeładowania konfiguracji – przez restart klastra. Sprawdzać wartość parametrów możemy też sięgając do słownika pg_settings, będziemy dzięki temu znali również jednostkę oraz zastosowanie (jest dodatkowy opis) ustawienia:

ALTER SYSTEM SET SHARED_BUFFERS='256MB';

show shared_buffers;

select name,setting,unit,current_setting(name) from pg_settings where name='shared_buffers';



work_mem

Parametr work_mem określa wielkość pamięci zarezerwowaną na operacje sortowania, operacje arytmetyczne czy łączenie tabel. W starszych wersjach PostgreSQL nosił on nazwę sort_mem. Wartość ta określa pamięć dostępną dla każdej operacji osobno – co oznacza że jeśli damy work_mem np. 20MB, a jednocześnie pracować będą 3 zapytania używające sortowania, zaalokowane na ich potrzeby będzie 60MB! Trzeba zwrócić szczególną uwagę na ilość jednocześnie działających sesji, aby nie zostało zaalokowane nadspodziewanie dużo pamięci. Sprawdzić wielkość work_mem możemy używając komendy:

```
show work_mem;
```

Parametr work_mem można ustawić dla klastra z użyciem komendy ALTER SYSTEM, lub zmieniając wpis w pliku postgresql.conf. Będzie to ustawienie domyślne dla wszystkich nowo nawiązywanych sesji. W przeciwieństwie do shared_buffers, ten parametr możemy też zmienić dla naszej sesji z użyciem komendy set:

```
set work_mem='10MB';
```

Na co wpływa zmiana wielkości tego bufora? Głównie na ilość operacji I/O podczas np. sortowania czy grupowania. Czemu? Wyobraź sobie, że masz do posortowania dużą tabelę. Jej całkowita wielkość to np. 100MB. Dostępny work_mem to domyślne 4MB. Nie ma szans na to, aby operacja sortowania została przeprowadzona na poziomie pamięci. W związku z powyższym podczas sortowania będzie potrzebne cache'owanie danych na dysku, a to spowoduje zwiększenie I/O operacji odczytu i zapisu z użyciem HDD, co istotnie wpływa negatywnie na wydajność nie tylko przeprowadzanej operacji, ale również „ogólnoustrojowo”.

Przeprowadziłem mały eksperyment na potrzeby zbadania wpływu tego parametru na wydajność. W jednym z naszych systemów mamy tabelkę „zlecenia” o wielkości ok 2 milionów wierszy. Sprawdziłem jej wielkość:

```
SELECT  relname ,
   pg_size_pretty(pg_total_relation_size(C.oid)) AS "total_size"
 FROM pg_class C where relname='zlecenia';
```

i jak widać zajmuje ona ponad 3 GB:

```
SELECT  relname ,
    pg_size_pretty(pg_total_relation_size(C.oid)) AS "total_size"
  FROM pg_class C where relname='zlecenia';
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	relname name	total_size text
1	zlecenia	3140 MB

Sprawdziłem też wartość parametru work_mem:

```
show work_mem;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	work_mem text
1	4MB

No nie zmieści się, choćby założyła gorset ;) Na tej tabeli uruchomię za chwilę zapytanie wykorzystujące sortowanie, więc operacja ta będzie musiała przebiegać „na raty” w kawałkach po 4MB co da nam w sumie 3140/4 = 785 „rat”. Po zapełnieniu dostępnej pamięci, system będzie musiał za każdym razem zrzucić dane na dysk by zrobić miejsce na nowe dane. W zasadzie będzie to ciągła jazda po dysku. Zobaczmy teraz jak wielkość work_mem wpływa na wydajność operacji wymagających sortowania. Zanim rozpocząłem to badanie zadbałem o to by przefetchować całą zawartość tabeli „zlecenia”, by na tyle na ile to możliwe została ona zcache'owana w cached_buffers. Sam serwer jest serwerem testowym i w czasie badania inne operacje nie mogły mieć wpływu na wydajność moich operacji.

Aby mieć jakiś punkt wyjścia, pobrałem zawartość tej tabeli bez użycia sortowania, tak by sprawdzić o ile sortowanie spowalnia całą operację.

Uruchomiłem poniższą komendę:

explain analyze select * From zlecenia limit 10000;

By sprawdzić czas wykonania samego czytania danych. Ograniczyłem limit zwracanych wierszy do 10000, aby nie czekać do jutra ;) Poniżej wyniki szacowanego czasu wykonania operacji odczytu w zestawieniu z obowiązującą na dany moment wartością parametru work_mem:

Wartość work_mem	**Szacowany czas**
4MB	29ms
512MB	12ms
2GB	2ms

Zobaczmy teraz jak się przedstawia sytuacja w przypadku dodania sortowania:

explain analyze select * From zlecenia order by nazwa_zlecenia limit 10000;

Wartość work_mem	**Szacowany czas**
512kB	322541ms (322s)
4MB	146314ms (142s)
512MB	54023ms (54s)
2GB	53776ms (53s)

Już gołym okiem widać jak bardzo operacja sortowania spowalnia wszelakie zapytania. Pamiętajmy jednak że tutaj następuje odczyt sekwencyjny tabeli, nie jest wykorzystywany indeks. Z użyciem indeksu operacje te wyglądałyby całkiem inaczej – ale o tym w dalszej części tej książki.

Z badań wynika, że im większą część operacji sortowania PostgreSQL może wykonać na poziomie pamięci, tym szybciej ta operacja się odbywa. Istnieje też oczywiście limit, nie będzie ten wzrost wydajności postępował liniowo. Przyjrzyj się czasom wykonania przy wielkościach 512MB i 2GB dla parametru work_mem. Różnica jest w zasadzie niezauważalna. Najwyraźniej wszystkie sortowane dane zmieściły się już w pamięci. Dalsze zwiększanie wartości work_mem pod kątem tej konkretnej operacji mija się już z celem. W tej chwili jest już czas na zmniejszanie work_mem, aż do uzyskania wartości poniżej której wydajność zaczyna znów spadać.

Wysoka wartość work_mem jest wskazana np. w hurtowniach danych.

Taki mały „tip” dla programistów. Jeśli masz np. taką sytuację, że część sesji wykonuje jakieś obszerne raporty z wykorzystaniem sortowania i łączenia tabel (a więc intensywnie eksploatujacych przestrzeń work_mem) a część nie potrzebująca za dużej wartości tego parametru, możesz zrobić dwa osobne connectory do bazy danych, albo wstrzykiwać różne ustawienia tego parametru osobno na potrzeby takich sesji (w JDBC jest to np. bardzo proste). Sytuacja się nieco komplikuje przy zastosowaniu puli połączeń, ale zawsze możesz mieć dwie pule :)

Jak sprawdzić czy domyślne ustawienie work_mem jest wystarczające? Kolejny „tip” - tym razem dla administratorów baz danych. Zajrzyj do katalogu $PGDATA/base/OID_BAZY/pgsql_tmp. To tam właśnie powstają pliki tymczasowe w związku ze zrzucaniem danych na dysk podczas dużych operacji. Jeśli powstaje tam dużo sporych plików, to niezawodny znak, że należy zwiększyć work_mem.

A jak to właściwie działa? Kiedy uruchamiane jest zapytanie wymagające sortowania danych, PostgreSQL szacuje wielkość danych podlegających sortowaniu i porównuje z aktualnym ustawieniem work_mem. Jeśli ilość danych przekracza tę wielkość, sortowanie odbywa się z użyciem dysku. Szacowanie odbywa się na podstawie statystyk, dlatego warto jest je mieć zawsze aktualne.

maintance_work_mem

Do operacji typu VACUUM, czy zmiana definicji tabeli, odtwarzania bazy, tworzenie indeksu również niezbędna jest pamięć operacyjna. Owe działania podobnie jak w przypadku działań związanych z parametrem work_mem można wykonać korzystając z bufora lub dysku. Vacuum czy tworzenie indeksu będzie jednak z reguły potrzebować znacznie więcej pamięci niż to co ustawimy dla work_mem. Rzadko kiedy kilka operacji vacuum będzie uruchamianych równocześnie, więc na potrzeby tego typu działań możemy zarezerwować sobie znacznie większy obszar pamięci niż na potrzeby work_mem. Zależności wydajnościowe są podobne do work_mem, jednak maintance_work_mem odnosi się tylko do operacji wyżej wymienionych. Zwiększenie wartości tego parametru powinno przynieść wzrost wydajności operacji konserwacyjnych. Zalecane ustawienie to wartość 5% wielkości pamięci RAM.

effective_cache_size

Specyficzny parametr, ponieważ nie określa żadnych limitów a jedynie służy informacyjnie planerowi zapytania. Określa ile potencjalnie PostgreSQL ma dostępnej pamięci na cache, a jest to suma shared_buffers i cache systemu plików. Ustawienie tego parametru nie alokuje żadnych dodatkowych obszarów pamięci. PostgreSQL używa zarówno własnej pamięci – tej przydzielonej, jak i cache systemu plików.

Parametr ten jest bardzo istotny przy podejmowaniu decyzji o wykorzystaniu bądź nie skanu po indeksie zamiast odczytu sekwencyjnego tabeli. Im wyższa wartość parametru effective_cache_size tym większe prawdopodobieństwo wyboru indeksu. Jeśli ten parametr będzie ustawiony na zbyt niską wartość, istnieje duże prawdopodobieństwo wyboru skanu sekwencyjnego tabeli, nawet jeśli mogłyby z tego płynąć korzyści dla wydajności. Od wielkości tego parametru zależy też szacunkowy czas operacji, ponieważ planer zapytania może z pewnym prawdopodobieństwem określić czy uda mu się wykonać działania na poziomie pamięci, czy też będzie musiał użyć dysku. Wskazane jest ustawienie tego parametru na wartość 50-75% całkowitej wielkości pamięci RAM.

Teoria teorią, i zalecenia wszystkich świętych swoją drogą, ale jako Niewierny Tomasz zawsze lubię sprawdzić czy pismo prawdę mówi ;) No to sprawdzamy. Robimy przykładową tabelkę, zakładamy na niej indeks i ją analizujemy.

```
create table testowa (x integer);

insert into testowa select * from generate_series(1,3000000) order by random();

create index test_index on testowa(x);

analyze  testowa;
```

Po tych operacjach ustawiam parametr effective_cache_size na 2MB i sprawdzam koszt wykonania odczytu z testowej tabeli:

```
set effective_cache_size='2MB';

explain analyze select * from testowa order by x limit 100;
```

```
show effective_cache_size;
create table testowa (x integer);
insert into testowa select * from generate_series(1,3000000) order by random();
create index test_index on testowa(x);
analyze  testowa;

set effective_cache_size='2MB';
explain analyze select * from testowa order by x limit 100;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	QUERY PLAN text
1	Limit (cost=0.43..398.24 rows=100 width=4) (actual time=0.016..0.115 rows=100 loops=1)
2	-> Index Only Scan using test index on testowa (cost=0.43..11934178.25 rows=3000000 width=4) (ac
3	Heap Fetches: 100
4	Planning time: 0.055 ms
5	Execution time: 0.128 ms

WUT?

Koszt jest absurdalnie wysoki, a wynika to z faktu że mamy effective_cache_size ustawiony na marginalnie małą wartość 2MB, co z dużym prawdopodobnieństwem może skutkować koniecznością wykorzystania dysku.

Zmieniam teraz wielkość parametru na 4GB (mam 8GB RAMu na tym serwerze) i ponawiam próbę:

```
set effective_cache_size='4GB';
explain analyze select * from testowa order by x limit 100;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	QUERY PLAN text
1	Limit (cost=0.43..4.80 rows=100 width=4) (actual time=0.039..0.261 rows=100 loops=1)
2	-> Index Only Scan using test index on testowa (cost=0.43..131012.40 rows=3000000 width=4) (actu
3	Heap Fetches: 100
4	Planning time: 0.108 ms
5	Execution time: 0.313 ms

Szacunkowy koszy jest 10x mniejszy... Planer przyjął, że prawdopodobnie całość danych zmieści się w cache'u.

wal_buffers

Określa wielkość bufora wykorzystywanego na potrzeby zapisów w plikach WAL. Według dokumentacji domyślna wartość 16kB nie powinna być zmieniana do czasu aż pojedyncza transakcja nie przekracza tej wielkości. Ponieważ jednak pliki WAL są intensywnie tworzone i aktualizowane, a ten proces może być „opóźniaczem” w przypadku dużych operacji ładowania, warto jest ustawić ten parametr na wielkość jednego segmentu plików WAL (16MB domyślnie).

Dostęp do dysku

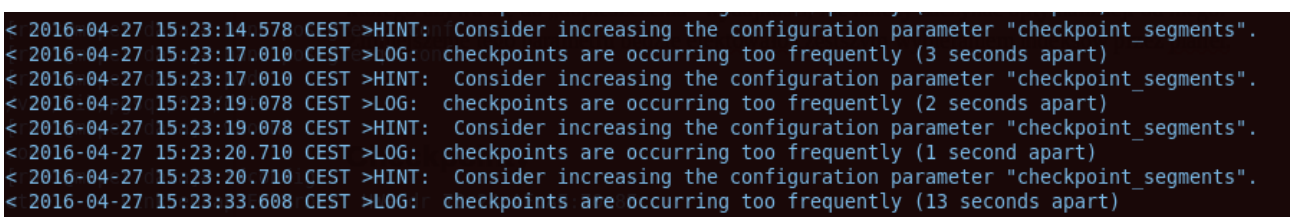
random_page_cost

Ten parametr jest ściśle powiązany z parametrem seq_page_cost. Wskazuje on koszt odczytu niesekwencyjnego odczytu danych z dysku tj. odczytu nieliniowego np. w wyniku dostępu do danych z użyciem adresów wierszy pobranych z indeksu. Im wyższy jest ten koszt, tym rzadziej planer zapytania będzie decydował o wyborze dostępu do danych z użyciem indeksów. Jeśli masz szybkie dyski, możesz zmniejszyć wartość tego parametru do 2-3 (domyślnie 4). To jest parametr którego używa się do „dopieszczenia” wydajności, nie należy zaczynać optymalizacji od niego. Zmniejszenie tego parametru będzie efektowało częstszym wybieraniem indeksów przez planer.

Checkpointy

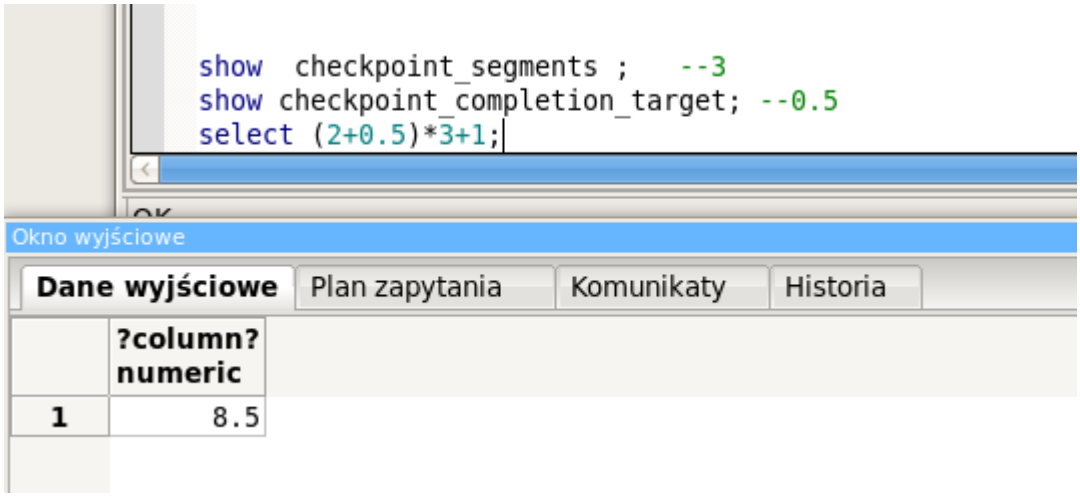
checkpoint_segments

Określa ilość segmentów WAL (domyślne 3) po których zapełnieniu jest tworzony checkpoint. Ten parametr ma olbrzymi wpływ na szybkość masowego ładowania danych. Domyślna wielkość segmentu – 16MB powoduje, że w domyślnej konfiguracji parametru checkpoint_segments po zapełnieniu 48MB plików WAL będzie chwila „wstrzymania” ładowania lub aktualizacji dużych ilości danych spowodowana tworzeniem punktu kontrolnego i zrzucaniem brudnych bloków. Jeśli wartość ta będzie osiągana zbyt szybko, zobaczysz w logach stosowny komunikat:



Na podstawie wzoru podanego w dokumentacji, maksymalną liczbę będących w użyciu w danym momencie segmentów WAL można obliczyć w taki sposób:

(2 + checkpoint_completion_target) * checkpoint_segments +1;



czyli domyślnie będzie to 8.5 (czyli wychodzi z tego 9 plików). Łatwo w ten sposób określić ile potencjalnie będą zajmowały na dysku pliki WAL (nie liczymy tych zarchiwizowanych) – 8.5 * 16MB = 136MB. Zwiększanie tego parametru będzie skutkowała m.in. wydłużeniem czasu odtwarzania bazy po awarii. Z drugiej strony zbyt mała wartość w stosunku do ilości ładowanych bądź modyfikowanych danych może być poważnym wąskim gardłem. Jeśli zwiększasz wartość parametru checkpoint_segments, weź też pod uwagę zmianę parametru checkpoint_timeout. Dokumentacja sugeruje ustawienie wartości parametru checkpoint_segments na wartość 10.

checkpoint_timeout

Ten parametr określa co jaki czas niezależnie od zapełnienia plików WAL powinien być tworzony checkpoint. Domyślnie ustawiony jest na 5 minut. Jeśli zwiększyłeś parametr checkpoint_segments na tyle że parametr checkpoint_timeout stał się podstawowym wyzwalaczem tworzenia punktów kontrolnych, rozważ również zwiększenie i tego parametru. Podobnie jak i zwiększanie wartości checkpoint_segments spowoduje dłuższy czas odtwarzania bazy po ewentualnej awarii, jednak zmniejszy częstotliwość „przestojów” spowodowanych tworzeniem punktu kontrolnego.

checkpoint_completion_target

Jeśli zwiększasz wartości w parametrach checkpoint_segments i checkpoint_timeout, warto zwiększyć również wartość w tym parametrze. Parametr ten określa część czasu pomiędzy checkpointami wyznaczanymi przez checkpoint_timeout co jaką PostgreSQL będzie usiłował tworzyć punkty kontrolne i w jakiej powinien się zmieścić. Aby zrozumieć działanie tego parametru wyobraźmy sobie taką sytuację. Ustawiliśmy checkpoint_segments i checkpoint_timeout na dość wysokie wartości i każdy checkpoint będzie powodował zrzut np. 8GB brudnych bloków na dysk. Jeśli nawet dzieje się to rzadziej, to jednocześnie taki checkpoint będzie powodował bardzo duże chwilowe opóźnienia działania bazy z powodu dużej ilości operacji I/O na raz. Zmiana tego parametru służy lepszemu rozłożeniu obciążenia wynikającego z tworzenia punktów kontrolnych. Domyślna wartość tego parametru to 0.5. Przyjmijmy że nie zmieniliśmy parametru checkpoint_timeout i pozostaje on ustawiony na 5 minut. Czyli na ten moment nasza konfiguracja wygląda tak:

checkpoint_timeout=5

checpoint_segments=512

checkpoint_completion_target=0.5

Taka konfiguracja sprawi że system będzie usiłował stworzyć punkt kontrolny co 0.5*5minut = 2,5 minuty by lepiej rozłożyć obciążenie I/O i bardziej je zrównoważyć na przestrzeni czasu. Jednocześnie oznacza to, że PostgreSQL będze usiłował w te 2,5 minuty zrzucić na dysk 8GB danych, co daje nam zapis na poziomie 8192MB/150s = 54,61 MB/s. Zestaw to teraz z szybkością swoich dysków, a dowiesz się ile transferu pozostaje dostępne dla innych operacji. Zalecane jest zwiększenie tego parametru do 0.9. Przyjmijmy więc taką zmianę dla wcześniejszych założeń. W tej chwili nasza konfiguracja przedstawiałaby się tak:

checkpoint_timeout=5

checpoint_segments=512

checkpoint_completion_target=0.9

No to liczymy. Próba wykonania punktu kontrolnego co 0.9 * 5 minut = 4,5 minuty. 8192MB/270s = 30,34 MB/s. Taka konfiguracja sprawi że pozostanie więcej „rezerwy” na operacje I/O inne niż związane z generowaniem punktów kontrolnych. Skutki dla wydajności mogą być bardzo różne, bo na częstotliwość punktów kontrolnych wpływa również choćby wielkość shared_buffers, więc najlepiej jest odnaleźć najlepszą wartość dla swojego systemu po prostu „organoleptycznie”.

Weźmy także pod uwagę, że wydłużenie czasu tworzenia punktu kontrolnego spowoduje też zwiększenie ilości danych przechowywanych w plikach WAL. Wartość tego parametru może się mieścić pomiędzy 0 a 0.9. Jednostką jest część czasu pomiędzy checkpointami.

checkpoint_warning

Parametr służy do ostrzegania administratora w sytuacji gdy tworzenie punktu kontrolnego w wyniku zapełnienia segmentów WAL odbywa się częściej niż czas określony w tym parametrze. Domyślnie ma on wartość 30s. Efekty jego ustawienia możemy zaobserwować na screenie użytym wcześniej w tym rozdziale:

```
< 2016-04-27 15:23:14.578 CEST >HINT:  Consider increasing the configuration parameter "checkpoint_segments".
< 2016-04-27 15:23:17.010 CEST >LOG:  checkpoints are occurring too frequently (3 seconds apart)
< 2016-04-27 15:23:17.010 CEST >HINT:  Consider increasing the configuration parameter "checkpoint_segments".
< 2016-04-27 15:23:19.078 CEST >LOG:  checkpoints are occurring too frequently (2 seconds apart)
< 2016-04-27 15:23:19.078 CEST >HINT:  Consider increasing the configuration parameter "checkpoint_segments".
< 2016-04-27 15:23:20.710 CEST >LOG:  checkpoints are occurring too frequently (1 second apart)
< 2016-04-27 15:23:20.710 CEST >HINT:  Consider increasing the configuration parameter "checkpoint_segments".
< 2016-04-27 15:23:33.608 CEST >LOG:  checkpoints are occurring too frequently (13 seconds apart)
```

Ten parametr sam w sobie nie wpływa w żaden sposób na wydajność, służy jedynie celom informacyjnym.

Statystyki

default_statistics_target

PostgreSQL określając algorytm wykonania zapytania bazuje na statystykach dotyczących tabel, powstałych przy użyciu komendy analyze lub w wyniku działania demona autovacuum. Ilość zbieranych w ten sposób danych określana jest przy użyciu parametru default_statistics_target. Zwiększenie wartości tego parametru będzie powodować wydłużenie czasu przeprowadzania analiz, ale też utworzenie bardziej precyzyjnych statystyk. Zmniejszenie skróci czas przeprowadzania analiz, ale też zmniejszy precyzję generowanych statystyk. Zbyt dogłębna analiza będzie zwiększała obciążenie generowane przez demona autovacuum, ale zbyt powierzchowna może doprowadzać do wyboru niewłaściwych planów zapytań. Parametr ten działa dla wszystkich kolumn wszystkich tabel, dla których ilość statystyk nie została indywidualnie określona z użyciem komendy ALTER TABLE SET STATISTICS. Wartość tego parametru może przyjąć dowolną liczbę z zakresu 1-1000. Parametr ten może być ustawiony na poziomie sesji. Zagadką pozostaje jednostka tego parametru. Dokumentacja wspomina jedynie że ma to być wartość integere z zakresu 1-1000.

inne

max_connections

Ten parametr określa maksymalną liczbę połączeń jaką ma przyjmować i obsługiwać PostgreSQL. Domyślne ustawienie – 100. Od ilości jednoczesnych połączeń zależy też wykorzystanie pamięci – mam tu na myśli możliwość zaalokowania nadspodziewanie dużej ilości pamięci na potrzeby work_mem. Ten parametr możesz konfigurować jak każdy innych, a jego właściwe ustawienie powinno być kompromisem pomiędzy wydajnością bazy, a ilością odrzucanych prób połączeń.

listen_address

Optymalizacja tego parametru to „dopieszczanie” wydajności. Za ten parametr zabieraj się po wykorzystaniu wszystkich innych efektywniejszych możliwości. Weryfikacja czy dany użytkownik łączący się do wskazanej bazy z określonego adresu przebiega przez weryfikację pliku pg_hba.conf. Zanim jednak dojdzie do sprawdzenia pg_hba.conf, połączenia są odfiltrowywane przez parametr listen_adress (zawartego w pliku postgresql.conf). Jeśli na tym poziomie połączenie nie zostanie odfiltrowane, PostgreSQL przechodzi do sprawdzenia pg_hba.conf i ewentualnie dopiero tutaj połączenie jest odrzucane. To powoduje małe bo małe, ale jednak zużycie zasobów serwera. Taka konfiguracja ułatwia też ataki typu DDoS. Lepiej filtrować połączenia już na pierwszym etapie – tj. postgresql.conf jeśli to możliwe.

log_min_duration_statement

Bardzo przydatny parametr przy określaniu problematycznych zapytań! Określa on minimalny czas trwania zapytania po którego przekroczeniu informacja o takim zapytaniu zostanie zapisana w logach. Jednostka tego parametru to milisekundy. Polecam zainteresowanie się tym logowaniem zwłaszcza administratorom stron internetowych. Wiemy przecież że użytkownik może się zniechęcić i opuścić naszą stronę internetową jeśli ta ładuje się np. 3 sekundy, a większość czasu ładowania strony to albo zasoby zewnętrzne – jak ciągnięte po sieci obrazki czy biblioteki JS, albo odczyt danych z bazy. Nie ma też co przesadzać w drugą stronę, bo zapisywanie dużej ilości zapytań w logach również odbije się negatywnie na wydajności bazy. Ustawienie parametru na -1 (czyli taka jak domyślna) spowoduje że nie będą zapisywane żadne zapytania do logów, wartość 0 spowoduje zapisywanie wszystkich zapytań.

log_statement

Ten parametr określa jakiego rodzaju instrukcje SQL będą zrzucane do logów (patrz parametr log_min_duration_statement). Może przyjąć wartości : none, ddl, mod, all. DDL spowoduje rejestrowanie wszystkich operacji DDL, MOD – wszystkich DDL i DML, ALL – wszystkich rodzajów operacji.

cpu_tuple_cost

Określa koszt odczytu sekwencyjnego jednej krotki w tabeli w trakcie zapytania. Domyślna wartość 0.01.

cpu_index_tuple_cost

Określa koszt odczytu sekwencyjnego jednej wartości w indeksie w trakcie zapytania. Domyślna wartość 0.005.

cpu_operator_cost

Określa koszt przetworzenia jednego operatora w trakcie wykonywania zapytania. Domyślna wartość 0.0025.

deadlock_timeout

Jeśli pojawia się blokada danych (np. w wyniku aktualizacji) PostgreSQL sprawdza czy nie jest to dead lock. Ten parametr określa (w milisekundach) po jakim czasie nastąpi to sprawdzanie. Ponieważ sprawdzanie zakleszczeń jest dosyć kosztowne, nie jest wykonywane od razu dla każdej blokady. Zwiększenie tego parametru zmniejszy ilość czasu poświęcaną na sprawdzanie takich blokad, ale z drugiej strony opóźni ich raportowanie. Kolejny parametr z kategorii „dopieszczaczy”. Domyślna wartość to 1000ms.

max_locks_per_transaction

Określa maksymalna ilość blokad na obiektach zakładanych przez pojedynczą transakcję. Domyślna wartość 64. Zbyt długie przeciąganie transakcji przy zablokowanych zasobach może prowadzić do eskalacji wzajemnych blokad. Możesz ten parametr zmniejszyć jeśli masz tego typu problem w swojej bazie danych.

Zalecane ustawienia parametrów

shared_buffers	RAM > 1GB – 25%, RAM<1GB – 15%
work_mem	Zmienne – zależne od specyfiki bazy danych
maintenance_work_mem	5%
effective_cache_size	50-75% RAM
wal_buffers	1 segment - 16MB
random_page_cost	Przy szybkich dyskach – 2-3
checkpoint_segments	10
Checkpoint_timeout	Zmienne – zależne od specyfiki bazy danych
checkpoint_completion_target	0.9
default_statistics_target	Zmienne – zależne od specyfiki bazy danych
max_connections	Zmienne – zależne od specyfiki bazy danych

Zalecenia konfiguracji wstępnej

- Zwiększenie wartości parametru shared_buffers
- Zwiększenie wartości parametru effective_cache_size
- Regularne uruchamianie operacji vacuum
- Regularne generowanie statystyk
- Przeniesienie plików WAL na osobny dysk twardy. Operacje zapisu w plikach WAL mają duży narzut na obciążenie I/O. Taka operacja pozwoli innym działaniom na swobodniejszy dostęp do dysku z plikami danych.
- Zwiększenie wartości parametru work_mem
- Dostosowanie parametrów związanych z wytwarzaniem punktów kontrolnych do specyfiki bazy danych.
- Przy szybkich dyskach dostrojenie parametru random_page_cost
- Jeśli to możliwe to przeniesienie treści długich zapytań SQL do procedur składowanych. Zmniejszy to czas poświęcany na transfer treści zapytania – ma to znaczenie tylko przy bardzo długich zapytaniach SQL.

Vacuum

Zwykły vacuum

W jaki sposób działa transakcyjność w PostgreSQL? Jak to jest możliwe że dwie sesje widzą zupełnie różne rzeczy w sytuacji gdy jedna z nich zmieni dane w ramach transakcji ale nie zatwiedzi?

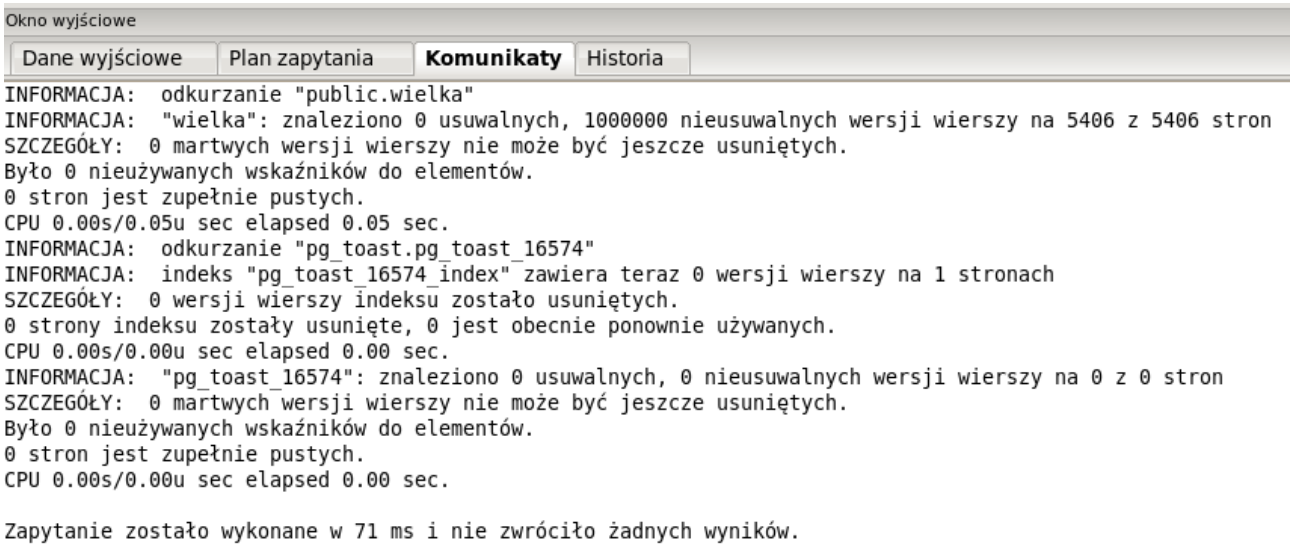
Gdy zmieniasz jakiś wiersz, PostgreSQL tworzy nową kopię wiersza na której Ty jako zmieniający operujesz. Kopia tego wiersza znajduje się w ramach tabeli w której oryginalny wiersz się znajduje. Dla zapytań odpytujących tę tabelę dostępne są stare wiersze. Po zakończeniu transakcji wszystkie zapytania wszystkich transakcji korzystają już z nowych wierszy. Taki sposób działania sprawia, że w plikach związanych z tabelami z czasem powstaje ogromna ilość nieużywanych wierszy wierszy do których już nawet nie ma dostępu. To z kolei powoduje rozrost plików danych. Miejsce zajmowane przez takie wiersze można odzyskać za pomocą polecenia VACUUM.

Polecenie to poza odzyskiwaniem wiersza może też odświeżać statystyki – ale nie jest to tematem tego rozdziału.

Sprawdźmy więc działanie tego polecenia. Wykonaj sekwencję poniższych komend aby stworzyć przykładową tabelę i zapełnić ją danymi.

```
create table wielka (x integer,y varchar);
do
$$
begin
for x in 1..1000000 loop
        INSERT INTO WIELKA VALUES (X,'X='||X);
end loop;
end;
$$;

VACUUM VERBOSE WIELKA;
```

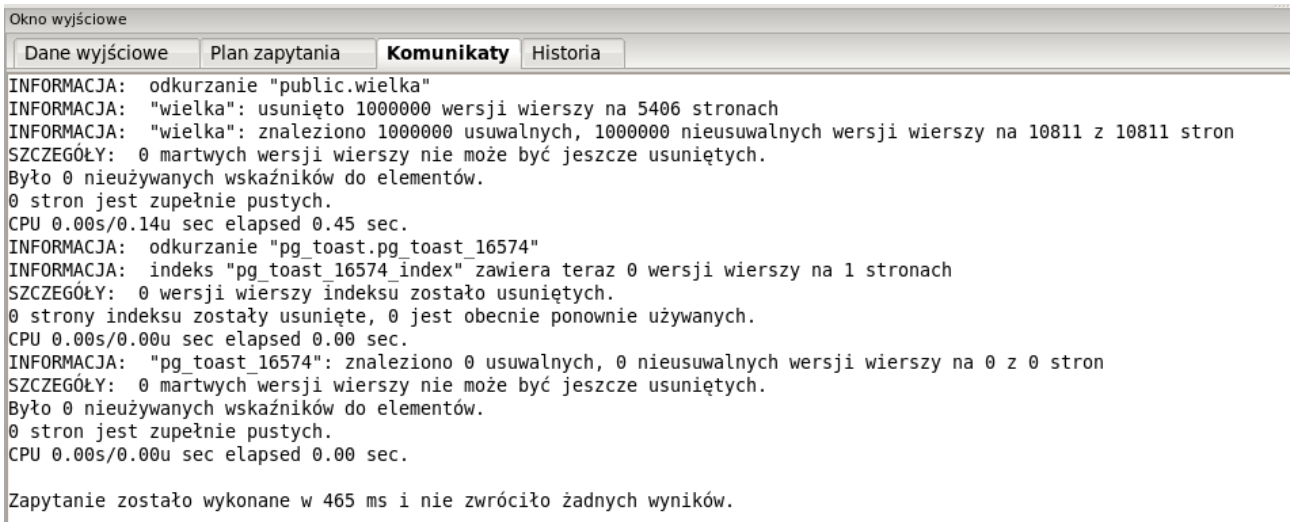
Na końcu wywoływany jest vacuum który powinien nam zwrócić mniej więcej taki wynik:



Jak widać mamy milion nieusuwalnych wierszy – to są te przed momentem wstawione, oraz 0 usuwalnych. Nie dokonaliśmy żadnych zmian jak dotąd, więc po prostu nie ma niepotrzebnych wierszy które można usunąć. Zmieńmy więc dane tak by takie wiersze powstały i ponownie przeprowadźmy czyszczenie:

UPDATE WIELKA SET Y='COS TAKIEGO';

VACUUM VERBOSE WIELKA;



Tym razem już było co usunąć, co też zostało wykonane. Zwolniło się sporo miejsca na nowe dane. Możesz też wykonać czyszczenie dla całej bazy danych:

vacuum verbose;

Proces vacuum skanuje tabele i indeksy w poszukiwaniu wpisów które już nie są widoczne. Informacja o nowo zwolnionym miejscu jest zapisywana w mapie wolnego miejsca FSM. Jeśli nastąpi jakaś operacja wymagająca wolnego miejsca – przykładowo aktualizacja danych, PostgreSQL w pierwszej kolejności będzie wykorzystywał miejsce oznaczone w FSM jako wolne, a więc przestrzeń odzyskaną.

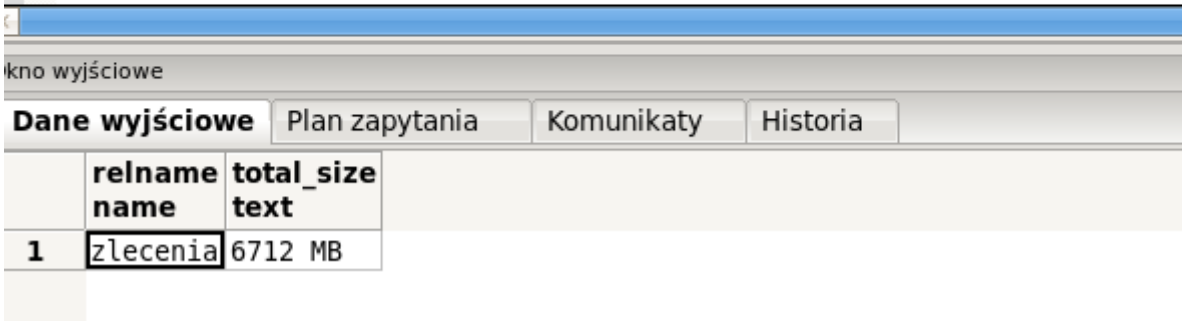
Ważna informacja – zwykły proces vacuum sam w sobie nie powoduje zmniejszenia plików danych! Używanie vacuum pozwala zaoszczędzić miejsce w tym sensie, że nie jest alokowana dodatkowa przestrzeń gdy istnieją dane które można by nadpisać. Operacja vacuum jest oczywiście bardzo zasobożerna, ale im częściej ją wykonujemy tym mniej będzie takie przestrzeni wymagającej oznaczenia jako wolna i tym w mniejszym stopniu będą nam się rozrastały pliki. Im będą one mniejsze tym operacja vacuum będzie się wykonywała szybciej. Paradoksalnie im częściej tym szybciej ;)

Zmniejszanie wielkości plików danych

Jeśli zechcielibyśmy zmniejszyć wielkość plików danych, musielibyśmy użyć polecenia VACUUM FULL. Działa ono w ten sposób, że poza czynnościami wykonywanymi przez zwykły vacuum, przenosi martwe wiersze na koniec tabeli a następnie zmniejsza jej wielkość odzyskując wolne miejsce z przestrzeni zwalnianej przez martwe wiersze. VACUUM FULL oczywiście nie jest rozwiązaniem idealnym i ma swoje wady. Przede wszystkim jest bardzo obciążające dla serwera i powoduje przy dużych tabelach ogromne ilości I/O. Ponadto zakłada blokadę na wyłączność. Z tych powodów operację tę powinniśmy przeprowadzać kiedy baza nie jest zbyt obciążona.

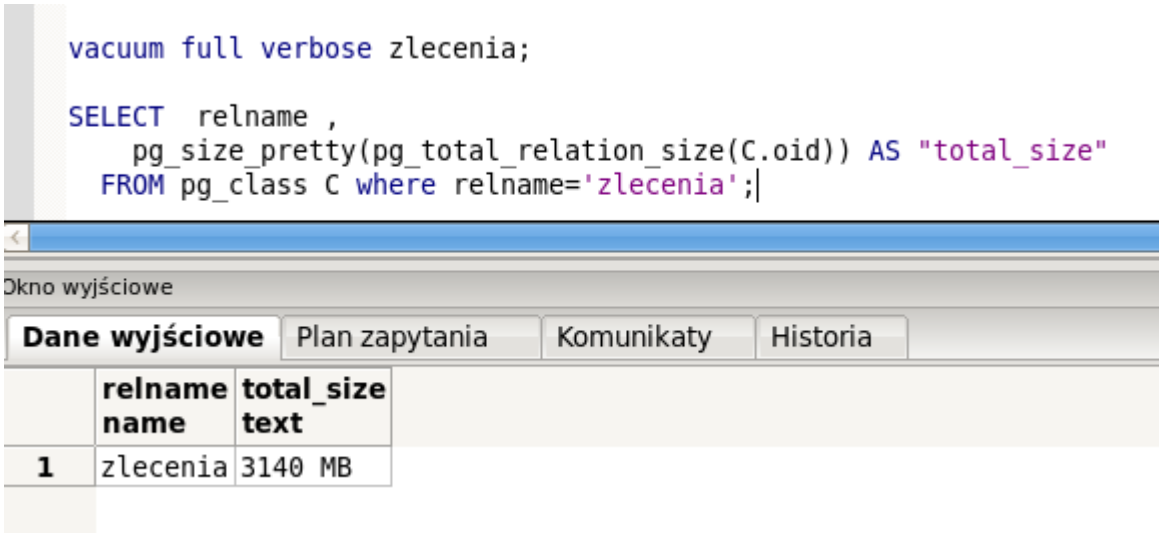
Przeprowadzimy mały eksperyment. Sprawdzimy ile miejsca uda się zwolnić z użyciem VACUUM FULL. Zaczynam od sprawdzenia ilości miejsca zajmowanego przez tabelę „zlecenia”. Tabela ta ma około 2 milionów wierszy i niedawno była aktualizowana jedna kolumna we wszystkich wierszach. Mamy więc pewien potencjał, na pewno powstało sporo „martwych” wierszy.

```
SELECT  relname ,
   pg_size_pretty(pg_total_relation_size(C.oid)) AS "total_size"
 FROM pg_class C where relname='zlecenia';
```



Na ten moment zajmuje ona niemal 7 gigabajtów! Uruchamiam więc polecenie vacuum full dla tej tabeli:

vacuum full verbose zlecenia;



Musisz liczyć się z tym, że operacja VACUUM powoduje dodawanie nowych wpisów do plików WAL, a więc jeśli Twoja baza działa w trybie archiwizacji ciągłej, powinieneś zachować nieco miejsca na potrzeby nowo powstałych plików. Uruchomienie operacji VACUUM FULL na naszej przykładowej tabeli „zlecenia” spowodowało wygenerowanie 195 (!!!!) zarchiwizowanych plików WAL o łącznej wadze ponad 3GB! Środowisko na którym uprawiam zabawy jest testowe, nic poza tym co sam puszczam tam nie jest uruchamiane, więc pliki mogły powstać wyłącznie w wyniku działania instrukcji vacuum full. W zamian wielkość tabeli zmniejszyła się do nieco ponad 3GB (z prawie 7).

Przy okazji, przydatna kwerenda która zwróci nam informację o największych tabelach w bazie danych:

```
SELECT nspname || '.' || relname AS "relation",
  pg_size_pretty(pg_total_relation_size(C.oid)) AS "total_size"
FROM pg_class C  LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
WHERE nspname NOT IN ('pg_catalog', 'information_schema')
  AND C.relkind <> 'i'
  AND nspname !~ '^pg_toast'
ORDER BY pg_total_relation_size(C.oid) DESC
LIMIT 20;
```

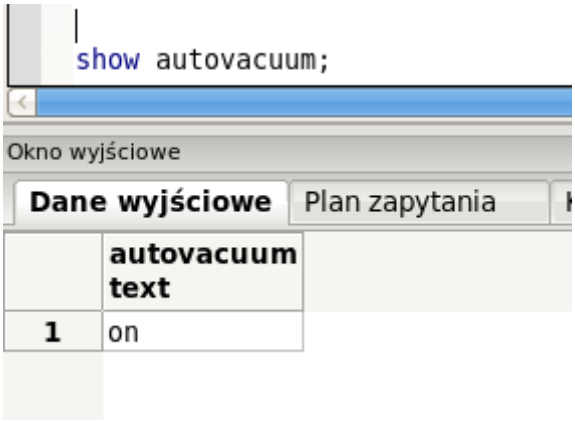
Można oczywiście wykonać vacuum full dla całej bazy:

vacuum full;

Automatyczny vacuum – autovacuum

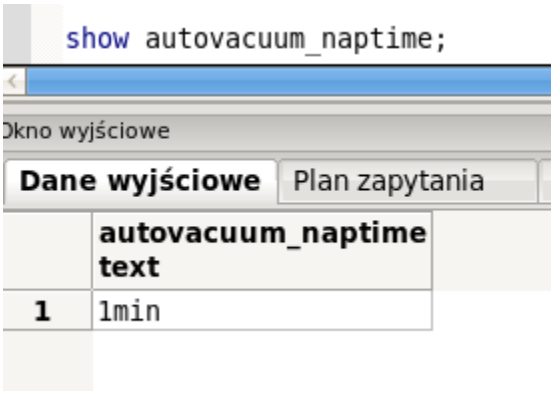
Od wersji 8.1 PostgreSQL wprowadzono nowy mechanizm – demona autovacuum. Domyślnie jest on włączony. Możemy się o tym przekonać wywołując komendę :

show autovacuum;



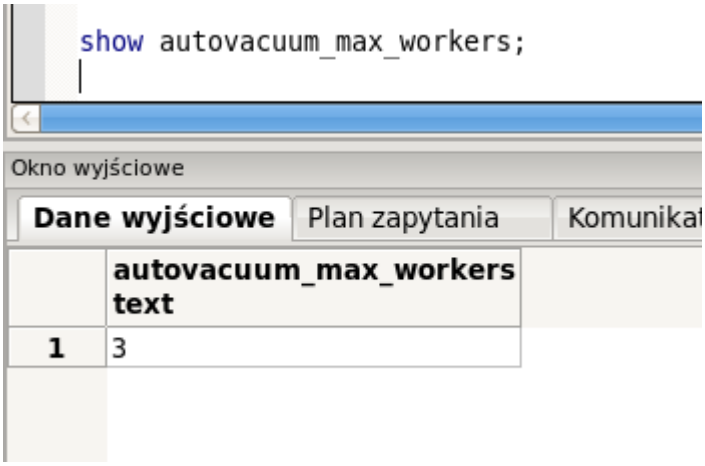
Mechanizm ten wykonuje te same czynności co vacuum, z tą różnicą że nie musimy go wywoływać ręcznie. Uruchamia się sam co czas określony w autovacuum_naptime domyślnie ustawionym na minutę:

show autovacuum_naptime;



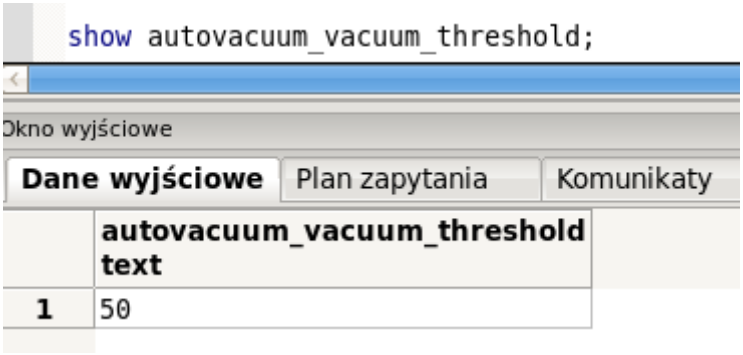
Autovacuum poszukuje tabel dla których została zmieniona, dodana lub skasowana znacząca ilość wierszy i przeprowadza w nich procesy czyszczące. Operacja wykonywana jest z użyciem takiej ilości procesów jaka jest ustawiona przez parametr autovacuum_max_workers:

show autovacuum_max_workers;



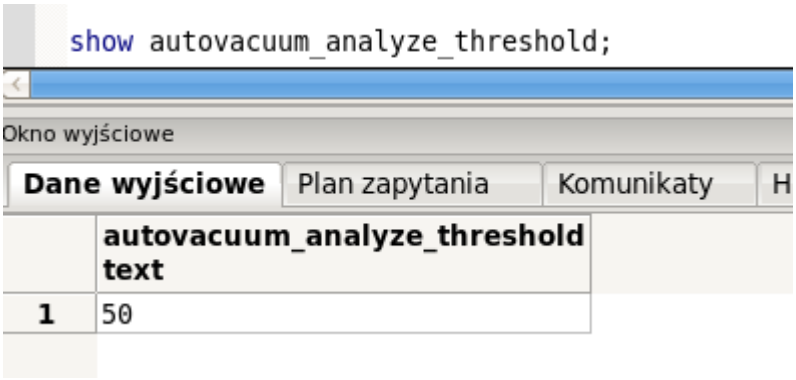
Ktoś dociekliwy mógłby zapytać – ile to jest „znacząca ilość wierszy” ? A to już określamy sami poprzez parametr autovacuum_vacuum_threshold domyślnie ustawiony na 50:

show autovacuum_vacuum_threshold;



Warto wiedzieć że autovacuum odświeża też statystyki tabel na potrzeby lepszego wybierania planów wykonania. Robi to dla tych tabel, dla których ilość zmienionych wierszy przekroczy wartość określoną w parametrze autovacuum_analyze_threshold:

show autovacuum_analyze_threshold;



który również jest domyślnie ustawiony na 50 wierszy.

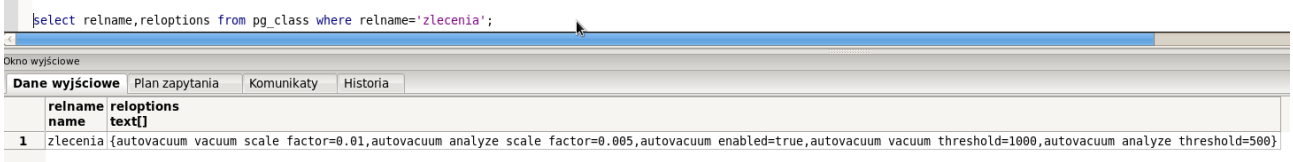
Jedna z pierwszych rzeczy jaka mi przyszła do głowy po zapoznaniu się z powyższymi parametrami było – a czy mogę ustawić te wartości indywidualnie dla jakiejś tabeli? Weźmy za przykład dużą tabelę – taką z 20 MLN wierszy. Czy dla niej zmiana albo skasowanie 50 wierszy to dużo? Marginalnie mało. Czy jest więc sens przeprowadzać czyszczenie i odświeżanie statystyk po takiej zmianie? Nie bardzo. Do tego to czyszczenie zjada nam zasoby, zwłaszcza na tak dużej tabeli. Gdyby ktoś sobie zadał podobne pytanie, to poniżej jest odpowiedź jak to zmienić dla wybranej tabeli:

ALTER TABLE zlecenia

SET (autovacuum_vacuum_threshold = 1000,autovacuum_analyze_threshold=500);

A jakbyśmy chcieli sprawdzić indywidualne ustawienia dla tabeli, wystarczy zajrzeć do słownika pg_class:

select relname,reloptions from pg_class where relname='zlecenia';



Monitorowanie działania vacuum i autovacuum

Jeśli chcesz sprawdzić kiedy dla wybranych tabel była ostatnio wykonana operacja vacuum lub autovacuum, możesz odwołać się do słownika pg_stat_all_tables:

select relname,last_vacuum,last_autovacuum, last_autoanalyze,last_analyze

from pg_stat_all_tables where schemaname='public';

```
select relname,last_vacuum,last_autovacuum, last_autoanalyze,last_analyze
from pg_stat_all_tables where schemaname='public';
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	relname name	last_vacuum timestamp with time zone	last_autovacuum timestamp with time zone
1	uzyszkodnicy		
2	zlecenia obserwatorzy		
3	zlecenia	2016-05-02 17:55:51.174533+02	2016-05-02 17:36:31.532494+02
4	t test		
5	kategorie		
6	podkategorie		
7	testowa		

Możesz też sprawdzić czy działa proces autovacuum z poziomu systemu operacyjnego. Na potrzeby autovacuum uruchamia osobny proces w systemie operacyjnym którego działanie możemy podejrzeć z konsoli choćby tak:

ps -axww | grep autovacuum

```
[mapet@mapet ~]$ ps -axww | grep autovacuum
Warning: bad syntax, perhaps a bogus '-'? See /usr/share/doc/procps-3.2.8/FAQ
 3243 ?        Ss     0:00 postgres: autovacuum launcher process

 7516 ?        Ss     0:05 postgres: autovacuum worker process   zlecenia

 7781 pts/0    S+     0:00 grep autovacuum
[mapet@mapet ~]$
```

Pamiętaj, że jeśli jakiejś tabeli dotyczy aktualnie trwająca transakcja, nie będzie można przeprowadzić dla niej procesu czyszczenia. Aby sprawdzić takie długo trwające transakcje możesz wywołać np.:

select current_timestamp-xact_start czas_trwania,datname, application_name,client_addr,query from pg_stat_activity where xact_start is not null;

Optymalizacja procesu VACUUM i AUTOVACUUM

Jeśli uruchamiamy proces vacuum na dużej ilości dużych tabel, lub na całej bazie szybko może się okazać że to właśnie VACUUM zużywa najwięcej zasobów z wszystkich operacji. Duże ilości I/O generowane przez vacuum spowodują ogólne spowolnienie działań na bazie w trakcie trwania czyszczenia. W tym kontekście interesujące się stają parametry vacuum_cost_limit i vacuum_cost_delay.

Vacuum_cost_limit określa maksymalny koszt operacji I/O po których przekroczeniu operacja zostaje wstrzymana na okres wskazany przez vacuum_cost_delay wyrażony w milisekundach (przyjmuje wartości 0-100). Domyślne ustawienie wartości 0 dla vacuum_cost_delay powoduje że operacja vacuum nie jest wstrzymywana wcale. Zwiększenie tej wartości spowoduje wydłużenie czasu trwania operacji vacuum, ale w zamian zmniejszy obciążenie podczas jej trwania.

Demon autovacuum ma swoje parametry: autovacuum_vacuum_cost_limit i autovacuum_vacum_cost_delay. Działają one w taki sam sposób. Domyślnym ustawieniem autovacuum_vacuum_cost_limit jest -1, co oznacza przeniesienie ustawień z normalnego vacuum.

Dane statystyczne bazy danych

W PostgreSQL mamy zbiór słowników dotyczących statystyk bazy danych. Możemy je wykorzystać w celu odnalezienia np. najczęściej używanych obiektów, tych które są najczęściej czytane z dysku, czy tych które zajmują najwięcej miejsca. Poniżej przedstawiam kilka wybranych.

pg_stat_all_tables, pg_stat_user_tables i pg_stat_sys_tables

W tym słowniku znajdziemy informacje na temat ilości odczytów obiektu z rozróżnieniem na odczyt sekwencyjny i odczyt z użyciem indeksu.

select * from pg_stat_all_tables where relname='przyklad_statystyki';

select relname,seq_scan,seq_tup_read,idx_scan, idx_tup_fetch

from pg_stat_all_tables where relname='przyklad_statystyki';

```
select * from pg_stat_all_tables;
select * from pg_stat_all_tables where relname='przyklad_statystyki';
select relname,seq_scan,seq_tup_read,idx_scan, idx_tup_fetch
 from pg_stat_all_tables where relname='przyklad_statystyki';
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	relname name	seq_scan bigint	seq_tup_read bigint	idx_scan bigint	idx_tup_fetch bigint
1	przyklad statystyki	7	8000000		

Kolumna seq_scan mówi o ilości odczytów sekwencyjnych danej tabeli, seq_tup_read o ilości wierszy odczytanych z użyciem skanu sekwencyjnego po tabeli. Idx_scan zawiera informacje o skanach z użyciem indeksu (a następnie dostępie do danych w tabeli), a idx_tup_fetch o ilości wierszy odczytach z tabeli odnalezionych z użyciem indeksu.

W słowniku pg_stat_all_tables znajdziemy dane o wszystkich tabelach. Jeśli chcemy zobaczyć informacje o wszystkich swoich obiektach wystarczy zajrzeć do słownika pg_stat_user_tables:

select * from pg_Stat_user_tables;

```
select * from pg_Stat_user_tables;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	relid oid	schemaname name	relname name	seq_scan bigint	seq_tup_read bigint	idx_scan bigint	idx_tup_fetch bigint	n_tup_ins bigint
1	47118	public	pgbench tellers	0	0	0	0	0
2	47124	public	pgbench branches	0	0	0	0	0
3	47121	public	pgbench accounts	0	0	0	0	0
4	47261	public	crazy stuff	9	20000			10000
5	47115	public	pgbench history	0	0			0
6	47256	public	some stuff	69	205398198	13	160012	100170000
7	47264	public	przyklad statystyki	7	8000000			2000000

Jeśli natomiast interesują nas wyłącznie obiekty systemowe jest też dostępny słownik pg_stat_sys_tables:

select * from pg_Stat_sys_tables;

```
select * from pg_Stat_sys_tables;
```

Okno wyjściowe

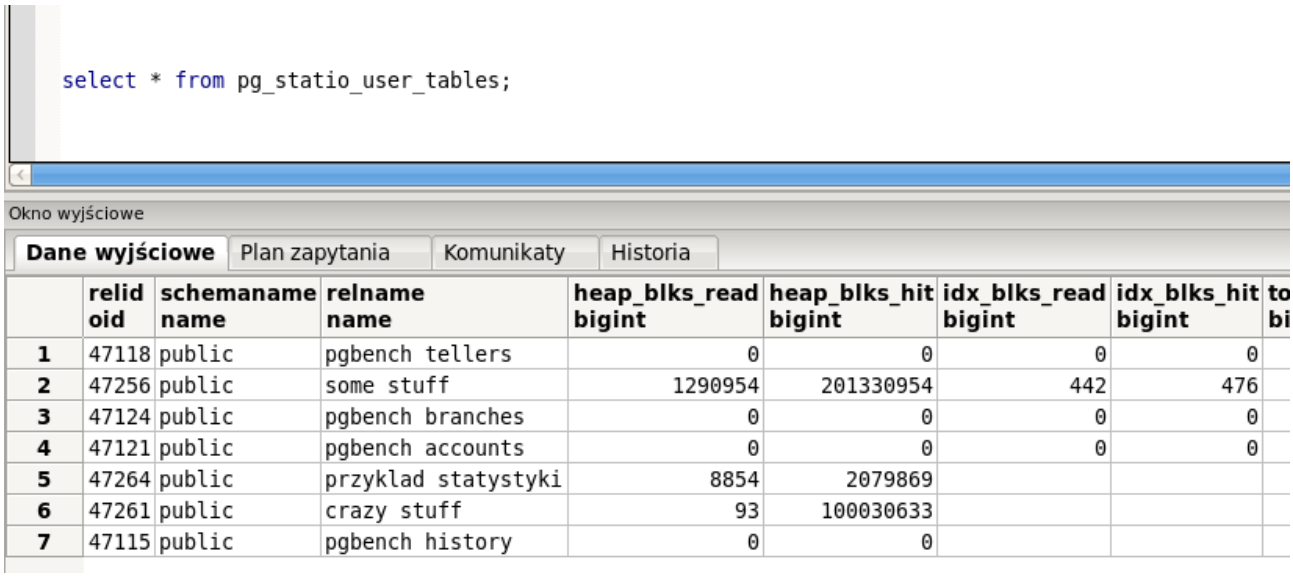
Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	relid oid	schemaname name	relname name	seq_scan bigint	seq_tup_read bigint
1	1261	pg catalog	pg auth members	0	0
2	2617	pg catalog	pg operator	0	0
3	2600	pg catalog	pg aggregate	0	0
4	12581	information schema	sql implementation info	0	0
5	1136	pg catalog	pg pltemplate	0	0
6	2966	pg toast	pg toast 2964	0	0
7	1262	pg catalog	pg database	2901	20183
8	2336	pg toast	pg toast 2620	0	0

pg_statio_user_tables i pg_statio_user_indexes

Dzięki informacjom z tego słownika uzyskamy informacje na temat tego, w jakim stopniu poszczególne tabele są czytane z dysku a jakim z bufora. Kolumna heap_blks_read mówi o ilości odczytów z użyciem dysku za pomocą skanu sekwencyjnego, heap_blks_hit z użyciem bufora shared_buffer za pomocą skanu sekwencyjnego. idx_blks_read to odczyt za pomocą dysku z użyciem indeksu, idx_blks_hit odczyt za pomocą bufora z użyciem indeksu.

select * from pg_statio_user_tables;



```
select * from pg_statio_user_tables;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	relid oid	schemaname name	relname name	heap_blks_read bigint	heap_blks_hit bigint	idx_blks_read bigint	idx_blks_hit bigint	to bi
1	47118	public	pgbench tellers	0	0	0	0	
2	47256	public	some stuff	1290954	201330954	442	476	
3	47124	public	pgbench branches	0	0	0	0	
4	47121	public	pgbench accounts	0	0	0	0	
5	47264	public	przyklad statystyki	8854	2079869			
6	47261	public	crazy stuff	93	100030633			
7	47115	public	pgbench history	0	0			

Na analogicznej zasadzie działa słownik pg_statio_user_indexes, z tą różnicą że odnosi się do indeksów:

select * from pg_statio_user_indexes;

```
select * from pg_statio_user_indexes;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_blks_read bigint	idx_blks_hit bigint
1	47256	47260	public	some stuff	turbo burbo	442	476
2	47124	47128	public	pgbench branches	pgbench branches pkey	0	0
3	47118	47130	public	pgbench tellers	pgbench tellers pkey	0	0
4	47121	47132	public	pgbench accounts	pgbench accounts pkey	0	0

pg_stat_database

W tym słowniku znajdziemy informacje bardzo użyteczne z punktu widzenia oceny wydajności bazy jako całości. Dla każdej bazy znajdziemy m.in. informacje o ilości commitów i rollbacków (xact_commit i xact_rollback), ilości odczytów z użyciem dysku i bufora (blks_read,blks_hit), ilości wierszy odczytanych, wstawionych i zaktualizowanych (tup_fetched,tup_inserted, tup_updated).

select * from pg_stat_database;

```
-- to samo tylko dla indeksów

select * from pg_stat_database;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

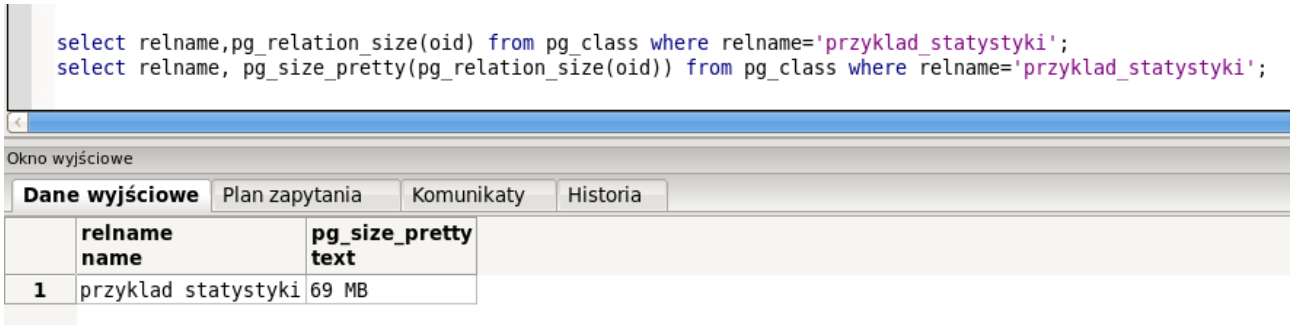
	datid oid	datname name	numbackends integer	xact_commit bigint	xact_rollback bigint	blks_read bigint	blks_hit bigint	tup_returned bigint	tup_fetched bigint	tup_inserted bigint	tup_updated bigint
1	1	template1	0	0	0	0	0	0	0	0	0
2	12998	template0	0	0	0	0	0	0	0	0	0
3	13003	postgres	1	2941	0	475	85571	1349059	17616	0	0
4	16699	zlecenia	0	0	0	0	0	0	0	0	0
5	44657	pgbench	2	4736	11	1301172	303544919	214988432	186848	102180050	20020
6	44734	results	0	0	0	0	0	0	0	0	0
7	47224	test indeksow	0	0	0	0	0	0	0	0	0

pg_class

W tym słowniku możemy znaleźć informacje o wielkości obiektów w bazie, co pozwala nam zweryfikować wielkość tych które nas interesują, lub odnaleźć te największe. Poniżej przykład w którym sprawdzam wielkość pojedynczej tabeli:

select relname,pg_relation_size(oid) from pg_class where relname='przyklad_statystyki';

select relname, pg_size_pretty(pg_relation_size(oid)) from pg_class where relname='przyklad_statystyki';

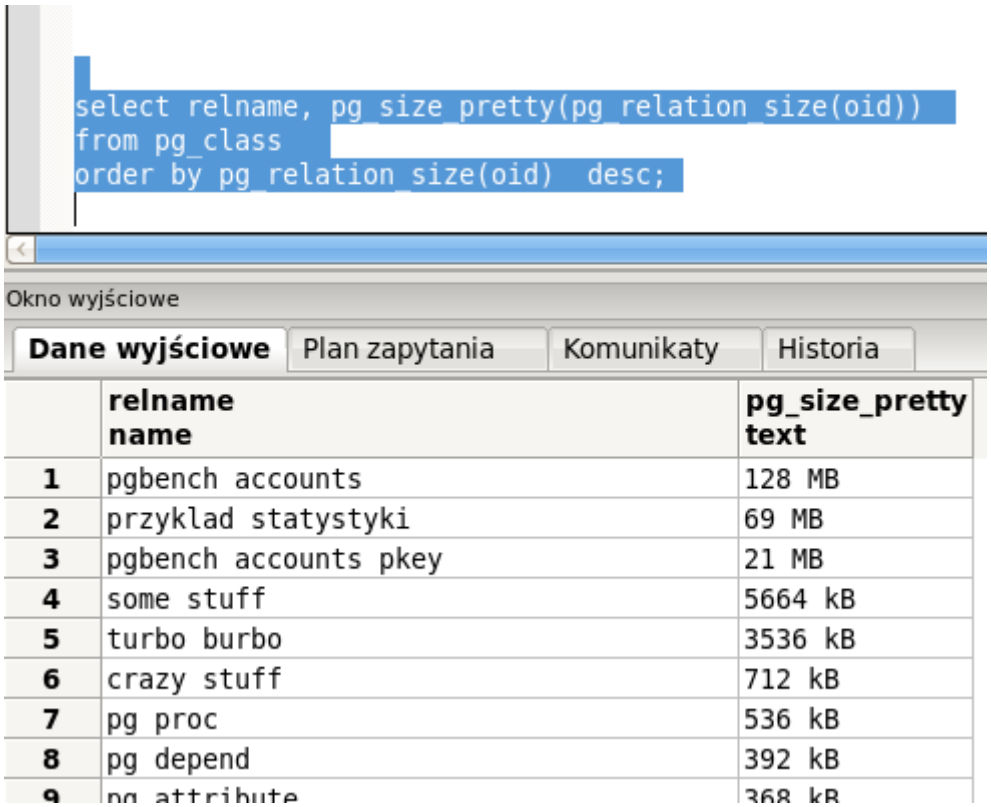


Przebudowując odpowiednio zapytanie, nietrudno również znaleźć największe tabele w bazie:

select relname, pg_size_pretty(pg_relation_size(oid))

from pg_class

order by pg_relation_size(oid) desc;



```
select relname, pg_size_pretty(pg_relation_size(oid))
from pg_class
order by pg_relation_size(oid)  desc;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	relname name	pg_size_pretty text
1	pgbench_accounts	128 MB
2	przyklad_statystyki	69 MB
3	pgbench_accounts_pkey	21 MB
4	some_stuff	5664 kB
5	turbo_burbo	3536 kB
6	crazy_stuff	712 kB
7	pg_proc	536 kB
8	pg_depend	392 kB
9	pg_attribute	368 kB

Plany wykonania zapytań i ich analiza

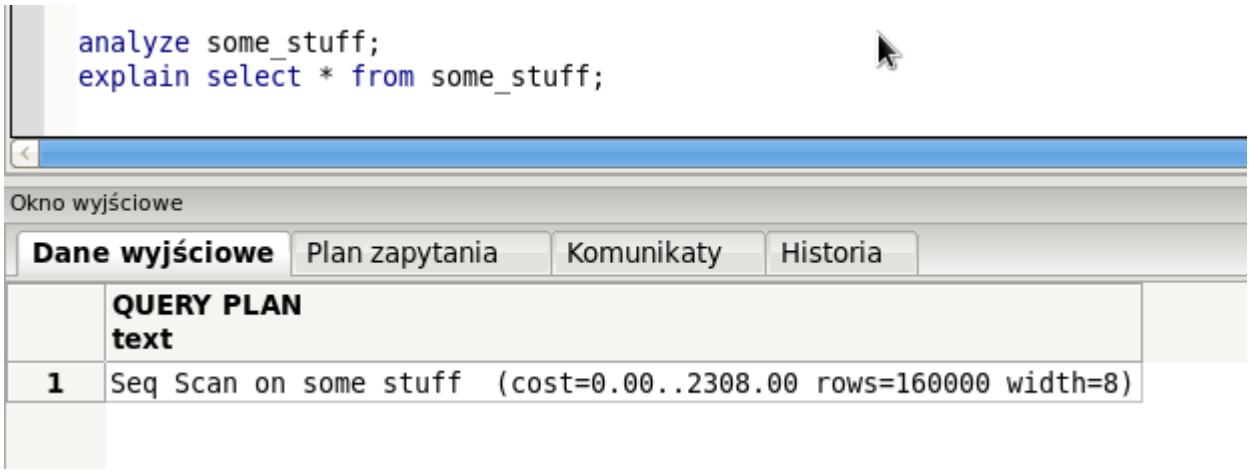
Sprawdzanie planu

Punktem wyjścia jeśli zapytanie wykonuje się wolno, jest sprawdzenie jego planu wykonania. Możemy się w tym celu posłużyć instrukcją EXPLAIN lub EXPLAIN ANALYZE. Ponieważ szacowanie kosztów wykonania zapytania oparte jest o statystyki tabel i indeksów, przez sprawdzeniem planu odśwież je dla tabel biorących udział w zapytaniu komendą ANALYZE.

Sama komenda EXPLAIN przestawi tylko plan wykonania zapytania.

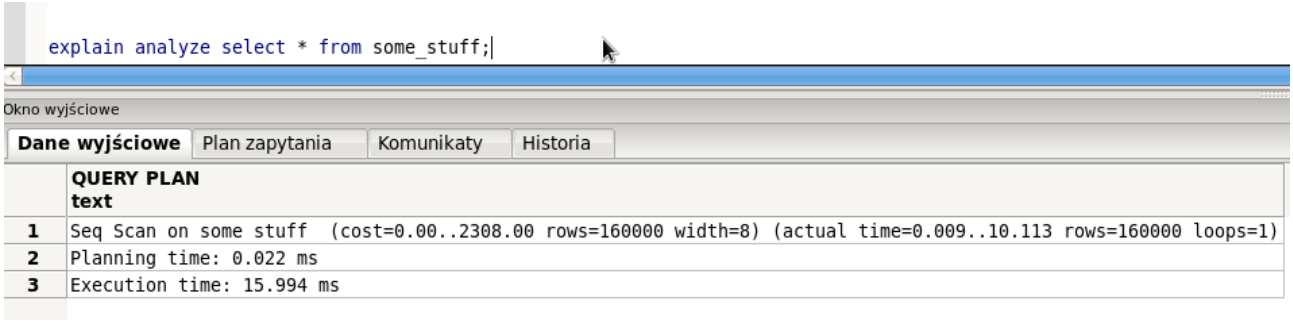
analyze some_stuff;

explain select * from some_stuff;



Jeśli skorzystasz z komendy EXPLAIN ANALYZE, poza wymyśleniem planu wykonania zapytania, zostanie ono jeszcze wykonane i zostanie wyświetlony czas zarówno planowania jak i wykonania zapytania.

explain analyze select * from some_stuff;



Analizując plany wykonania zapytania, weź pod uwagę że dane mogą być cache'owane w buforze. Najlepiej jest wykonać zapytanie kilkukrotnie i sprawdzić czy kolejne wykonania nie będą szybsze od pierwszego. Jeśli tak będzie, oznaczać to będzie że zapytanie było za pierwszym razem wykonywane przy zimnym buforze.

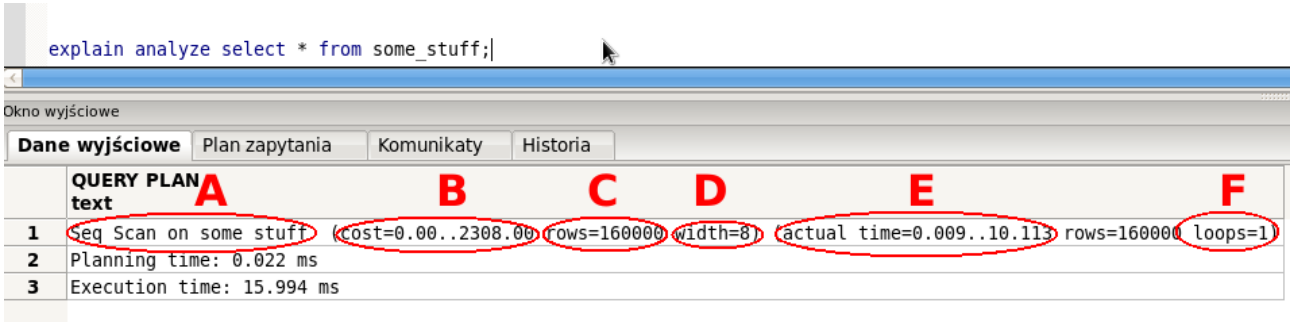
Analiza węzłów

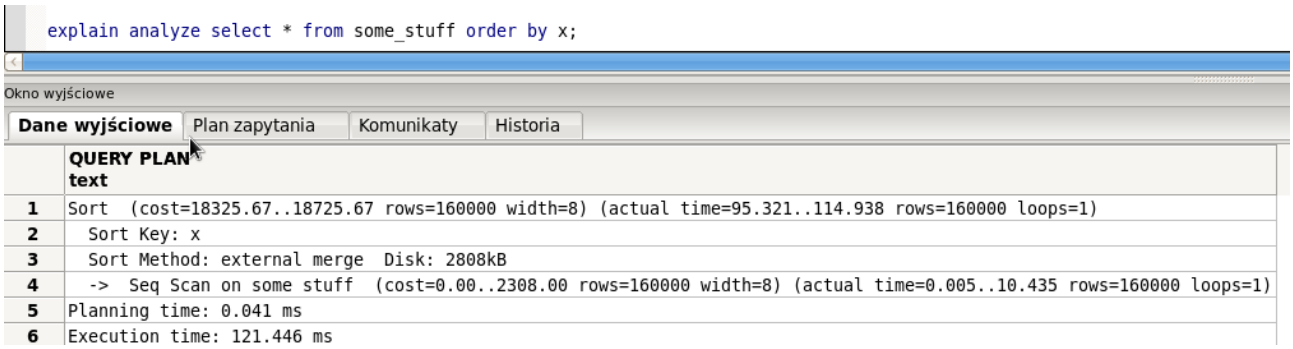
Parametry węzłów

Wiemy już w jaki sposób możemy wyświetlić plan wykonania zapytania, zajmiemy się więc teraz ich analizą. Na poniższej ilustracji oznaczyłem najważniejsze elementy literami:

A : Ta sekcja określa rodzaj skanu oraz obiekt na którym jest wykonywany. W tym przypadku jest to skan sekwencyjny na tabeli some_stuff.

B : W tej części znajdziesz dwie wartości kosztu wykonania zapytania. Pierwsza to koszt początkowy określający koszt pobrania pierwszego wiersza. Zaskakująca może być wartość 0. Skan sekwencyjny zaczyna pobierać wiersze od razu, nie potrzebuje żadnych przygotować. Stąd taka wartość. Dodałem do zapytania sortowanie. Plan tego zapytania jest wyświetlony na kolejnej ilustracji. W tym przypadku pierwszym węzłem jest sortowanie, które musi zostać wykonane zanim wiersze zaczną być przekazywane do aplikacji klienckiej. Zauważ że w tym przypadku koszt skanu sekwencyjnego pozostał taki sam, jednak jego rozpoczęcie wymaga wcześniejszego posortowania danych – tutaj koszt początkowy jest znacznie wyższy. Drugą wartością w podawanym koszcie jest kosztem pełnego wykonania węzła, a więc w tym przypadku odczytania całej tabeli.





C : To oszacowana liczba wierszy do wyświetlenia. Jeśli ta wartość bardzo się różni od rzeczywistej liczby wierszy w tabeli – powinien to być dla nas znak, że statystyki są nieaktualne i należy je odświeżyć.

D : Oszacowana liczba bajtów jaką średnio zajmuje jeden rekord. Oszacujmy więc wielkość tabeli na podstawie tych danych. Mamy (160000 wierszy * 8 bajtów każdy)/1024/1024 = 1,22 MB.

E : Tutaj mamy rzeczywiste dane obrazujące koszt wykonania i ilość wierszy jaka faktycznie została przetworzona w wyniku wykonania zapytania. Actual time – dwie wartości kosztu - tym razem rzeczywistego. Rows – faktyczna liczba przetworzonych wierszy.

F : Jeśli wartość parametru loops jest większa niż 1, oznacza to że dany węzeł był wykonywany więcej niż raz. Może tak się zdarzyć np. przy operacji łączenia tabel. Należy pamiętać, że wartość parametrów actual time i rows odnosi się do pojedynczego wykonania pętli. Jeśli ilość wykonań jest większa niż 1, należy te wartości pomnożyć przez ilość wykonań aby uzyskać faktyczny koszt i ilość wierszy przetworzonych w ramach danego węzła.

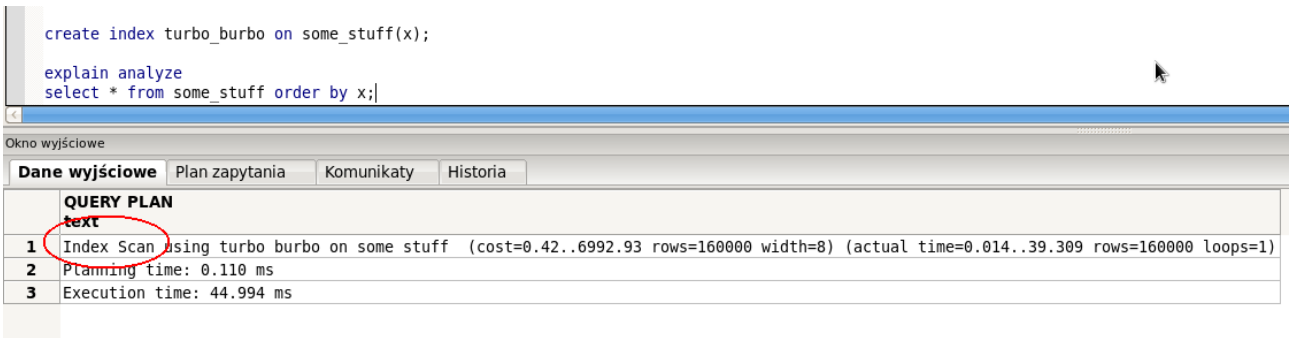
Skan po indeksie

Jeśli istnieje indeks który można byłoby wykorzystać w realizacji zapytania, najprawdopodobniej zostanie on wykorzystany zamiast skanu sekwencyjnego po tabeli. Na potrzeby przykładu stworzyłem indeks „turbo_burbo” którego dane są posortowane w sposób zgodny z wymogami zapytania.

create index turbo_burbo on some_stuff(x);

explain analyze

select * from some_stuff order by x;



Sortowanie

Na poniższej ilustracji przedstawiony jest plan wykonania zapytania wymagający posortowania danych. W tym jednak przypadku nie mam indeksu którego można by użyć i dane trzeba będzie posortować „na żywo”. Zaznaczony fragment jest bardzo ważny z punktu widzenia wydajności. Operacja sortowania może być wykonana z użyciem pamięci operacyjnej lub dysku. To w jaki sposób zostanie wykonane sortowanie zależy od ustawienia parametru work_mem. Jeśli całe sortowanie nie jest w stanie zostać wykonane w pamięci określonej przez ten parametr, zostanie wykonane z użyciem dysku tak jak to widzimy na poniższym przykładzie. External sort to właśnie sortowanie z użyciem dysku. Wielkość WORK_MEM mam ustawioną domyślnie na 4MB, a jak widzimy na potrzeby sortowania zostało wykorzystane mniej niż 3MB na dysku. Dlaczego więc nie zostało to posortowane w pamięci? Sortowanie z użyciem dysku potrzebuje mniej miejsca ponieważ jest wykonywane innym algorytmem. Na drugiej z poniższych ilustracji widzimy sortowanie po zwiększeniu parametru WORK_MEM do 64 MB.

explain analyze

select * from some_stuff order by y;

```
explain analyze
select * from some_stuff order by y;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

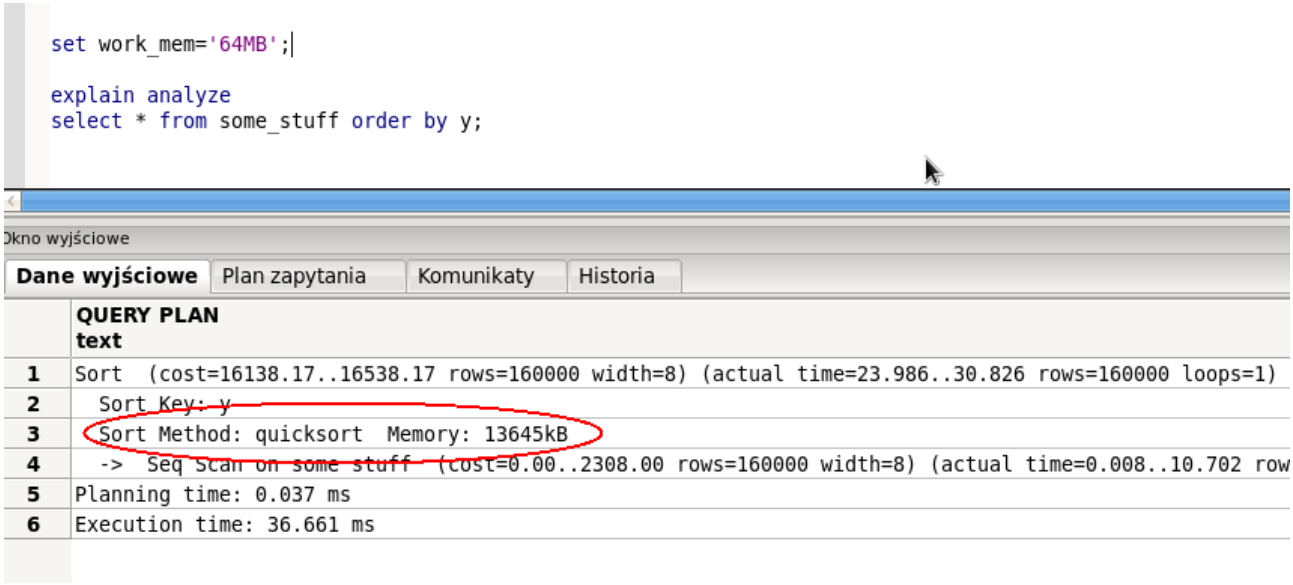
	QUERY PLAN text
1	Sort (cost=18325.67..18725.67 rows=160000 width=8) (actual time=33.726..46.540 rows=160000 loops=1)
2	Sort Key: y
3	Sort Method: external sort Disk: 2816kB
4	-> Seq Scan on some_stuff (cost=0.00..2308.00 rows=160000 width=8) (actual time=0.009..10.573 row
5	Planning time: 0.040 ms
6	Execution time: 52.847 ms

Po zmianie wielkości pamięci używanej m.in. do sortowania, zmienił się też plan wykonania zapytania. Tym razem sortowanie jest wykonywane z użyciem algorytmu QUICKSORT – czyli na poziomie pamięci operacyjnej. Widzimy teraz, że na potrzeby sortowania w pamięci zostało wykorzystane nie 3 a prawie 14MB. Dlatego właśnie wcześniej planer zapytania zadecydował o wykorzystaniu zewnętrznego sortowania z użyciem dysku – sortowanie w pamięci wymagało jak widać znacznie więcej przestrzeni niż mieliśmy dostępne w work_mem.

set work_mem='64MB';

explain analyze

select * from some_stuff order by y;



Indeksy

Proste indeksy B-Tree

Aby mieć na czym pracować stworzymy sobie prostą tabelę i wypełnimy ją danymi:

create table duza (

id serial primary key,

wartosc integer not null

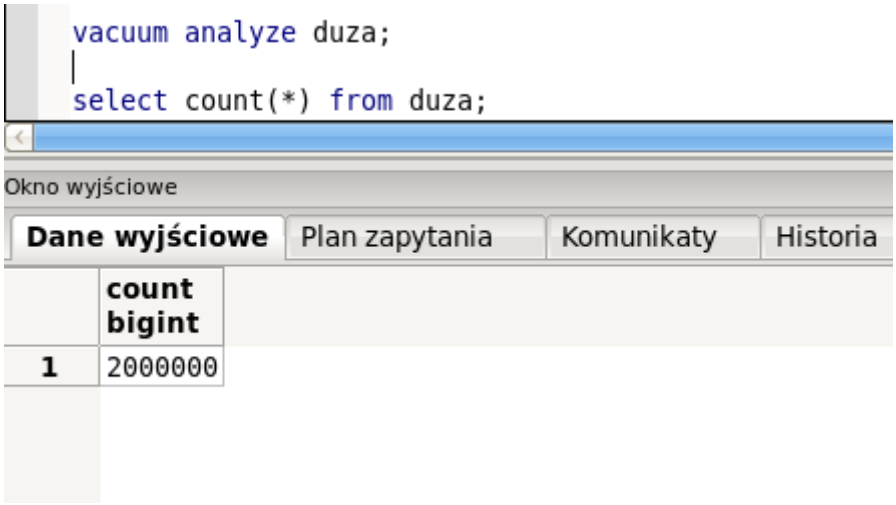
);

insert into duza (wartosc) values (generate_series(1,2000000));

Odświeżymy też statystyki dla niej:

vacuum analyze duza;

Jak widać poniżej, tabela ma dwa miliony wierszy.



Jej zawartość również nie jest specjalnie złożona, pokazuję ją byś widział mniej więcej co możemy w niej znaleźć i jaki jest układ danych.

```
select * from duza limit 100;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | H

	id integer	wartosc integer
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7

Sprawdźmy więc szacunkowy koszt odczytania 100 wierszy filtrując po drugiej kolumnie (nie ma na niej indeksu).

explain select * from duza where wartosc between 1000000 and 1000100;

```
explain select * from duza where wartosc between 1000000 and 1000100;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	QUERY PLAN text
1	Seq Scan on duza (cost=0.00..38850.00 rows=111 width=8)
2	Filter: ((wartosc >= 1000000) AND (wartosc <= 1000100))

Na potrzeby porównania sprawdźmy też czas pobrania samej kolumny wartość, ale posortowanej:

```
explain analyze select wartosc from duza order by wartosc;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	QUERY PLAN text
1	Sort (cost=265511.19..270511.19 rows=2000000 width=4) (actual time=952.285..1107.387 rows=2000000
2	Sort Key: wartosc
3	Sort Method: external sort Disk: 27376kB
4	-> Seq Scan on duza (cost=0.00..28850.00 rows=2000000 width=4) (actual time=0.010..184.974 row
5	Planning time: 0.137 ms
6	Execution time: 1179.872 ms

Przyjrzyj się zaznaczonym fragmentom. Pierwszy zaznaczony fragment to koszt sortowania. Zauważ że koszt samego sortowania jest niemal dziesięciokrotnie wyższy od kosztu odczytu całej tabeli (linia 4)! Wynika to z działania które jest zakreślone w linii 3. Planer oszacował, że dane w tej tabeli nie zmieszczą się w pamięci podczas sortowania, zadecydował więc o sortowaniu z użyciem dysku. Jak powszechnie wiadomo, wszelkie operacje I/O na dysku są wrogiem wydajności, ponieważ odczyt z dysku jest znacznie wolniejszy od odczytu z pamięci.Czas wykonania to 1,2s.

Założymy teraz zwykły indeks na kolumnie wartość. Taki indeks będzie posortowany wg wartości pierwszej kolumny:

create index przykladowy on duza(wartosc);

I porównujemy koszt odnalezienia danych:

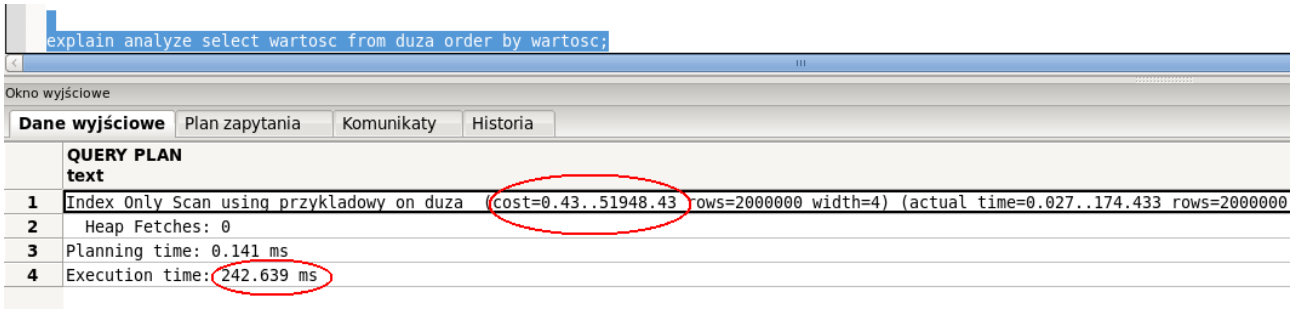
```
create index przykladowy on duza(wartosc);

explain select * from duza where wartosc between 1000000 and 1000100;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

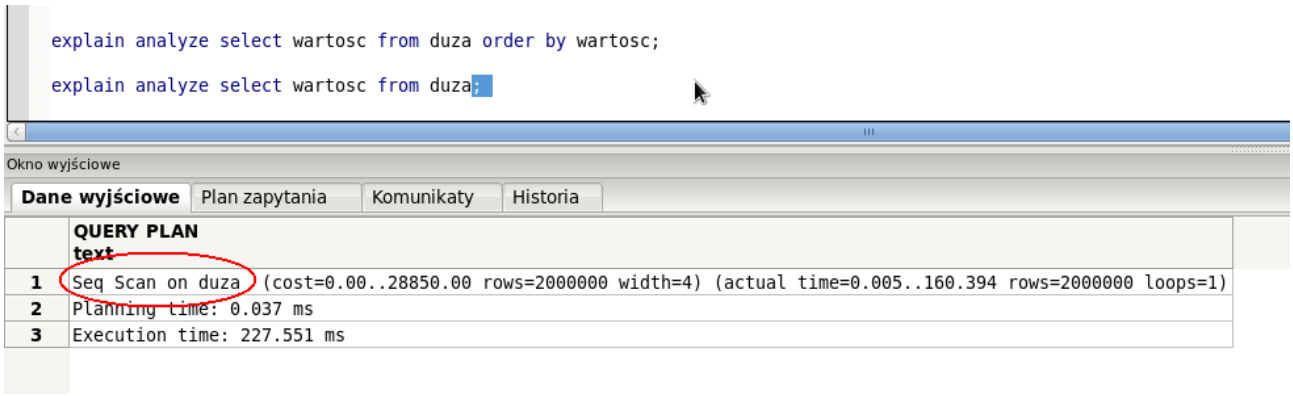
	QUERY PLAN text
1	Index Scan using przykladowy on duza (cost=0.43..10.65 rows=111 width=8)
2	Index Cond: ((wartosc >= 1000000) AND (wartosc <= 1000100))

Sprawdźmy więc różnicę w odczycie danych posortowanych:



Zatrzymajmy się na chwilę w linii 1. Zauważ że w miejsce sortowania mamy zwykły odczyt indeksu. Nie ma potrzeby sortowania, ponieważ indeks już jest posortowany wg kolumny na którą został założony. Odeszły nam więc również kosztowne operacje I/O. Czas wykonania – 0,24 s – czyli prawie 5cio krotnie szybciej. Weź pod uwagę że wraz ze wzrostem wielkości tabeli, będzie ten dysonans wzrastał.

Jeszcze ciekawostka. Z zapytania pozbyłem się sortowania, a planer zadecydował o użyciu skanu sekwencyjnego po tabeli zamiast indeksu. Wynika to z większego kosztu odczytu samych obiektów – nie uwzględniając sortowania.



Aspekt na który powinni zwrócić uwagę szczególnie użytkownicy baz danych Oracle – w PostgreSQL trzeba zapomnieć o wielu przyzwyczajeniach.... Zadajemy zapytanie kształtu:

select count(*) from duza;

Wiemy że mamy jeden indeks na kolumnie wartość a dodatkowo jest ona oznaczona jako NOT NULL, a ponadto mamy jeszcze indeks stworzony automagicznie w związku z założeniem klucza głównego. Czego można by się spodziewać w Oracle? Zapewnie zliczenia wartości na podstawie któregoś z tych indeksów. PostgreSQL natomiast zrobi pełny skan po tabeli:

```
explain analyze select count(*) from duza;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	QUERY PLAN text
1	Aggregate (cost=33850.00..33850.01 rows=1 width=0) (actual tim
2	-> Seq Scan on duza (cost=0.00..28850.00 rows=2000000 width
3	Planning time: 0.039 ms
4	Execution time: 223.127 ms

Dopiero gdy dodamy jakiś warunek odnoszący się do którejś z zaindeksowanych kolumn użyje skanu po indeksie:

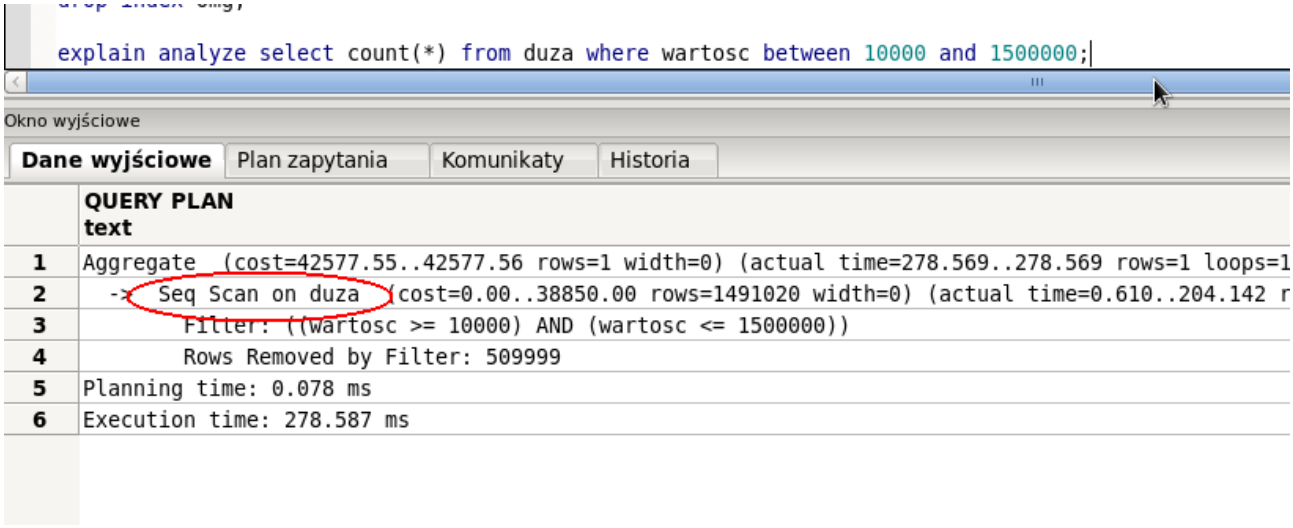
```
explain analyze select count(*) from duza where wartosc between 1000000 and 1000100;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	QUERY PLAN text
1	Aggregate (cost=6.75..6.76 rows=1 width=0) (actual time=0.031..0.031 rows=1 loops=1)
2	-> Index Only Scan using przykladowy on duza (cost=0.43..6.49 rows=103 width=0) (actual
3	Index Cond: ((wartosc >= 1000000) AND (wartosc <= 1000100))
4	Heap Fetches: 0
5	Planning time: 0.055 ms
6	Execution time: 0.050 ms

W tym przypadku jeszcze jest decyzja o odczycie z użyciem indeksu, ale nie zawsze tak będzie nawet jeśli indeks będzie istniał. Zależy to przede wszystkim od tak zwanej selektywności, czyli stosunku liczby wybieranych wierszy do całości. Tutaj pobieramy 0,00005 część całości danych, selektywność jest jednak bardzo niska. Gdybyśmy wybrali więcej danych, może się okazać że wydajniejszy będzie odczyt sekwencyjny całej tabeli:



Dokumentacja niestety milczy (stan na początek 2016 roku) na temat tego jaki jest próg przejścia ze skanu po indeksie na skan po tabeli, z moich obserwacji wynika jednak że duży wpływ na ten próg ma stosunek parametrów random_page_cost do seq_page_cost.

Indeksy wielokolumnowe

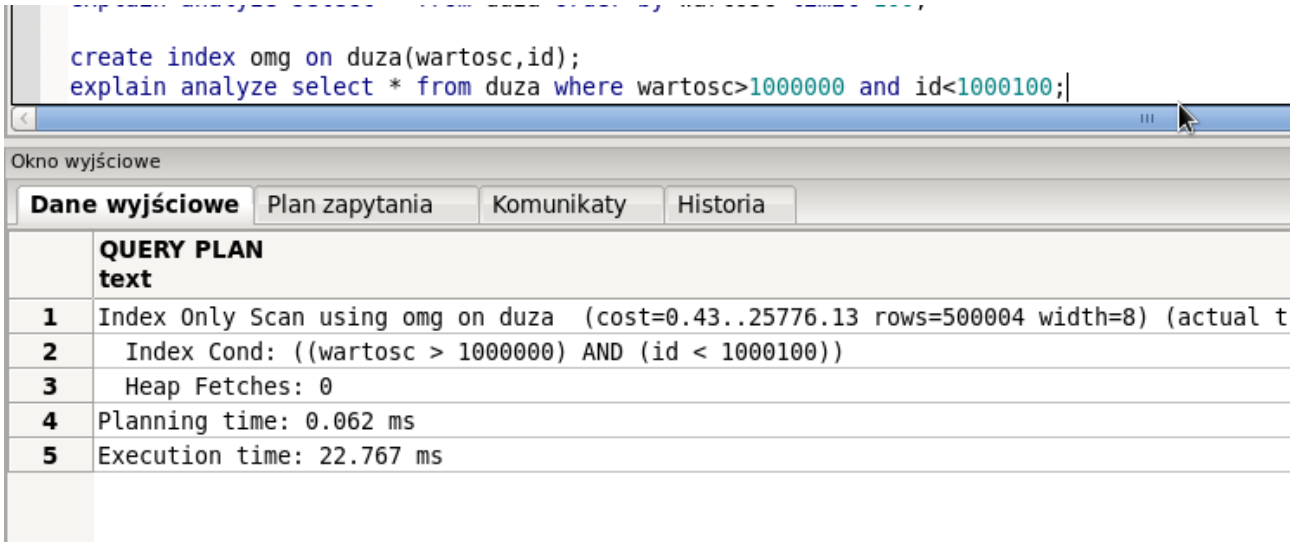
Oczywiście możemy też założyć indeksy wielokolumnowe, np. w przypadku filtrowania po kilku kolumnach:

explain analyze select * from duza where wartosc>1000000 and id<1000100;

Nic nie stoi na przeszkodzie by połączyć dwa indeksy, jednak będzie to na pewno droższe niż odczyt jednego podwójnego, ale za to tańsze niż odczyt sekwencyjny. Tutaj przykładowo zakładam indeks na obu kolumnach tabeli:

create index omg on duza(wartosc,id);

A plan wykonania wygląda teraz tak:



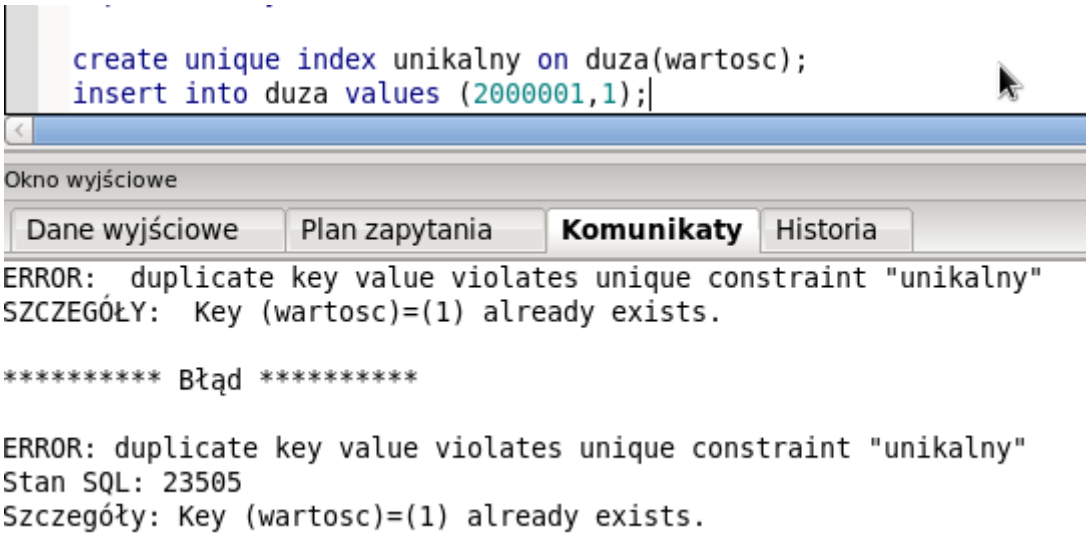
Indeksy B-tree mogą mieć maksymalnie 32 kolumny.

Indeksy unikalne

Indeksy mogą również wymuszać unikalność w kolumnie lub kombinacji wartości z kilku kolumn:

create unique index unikalny on duza(wartosc);

insert into duza values (2000001,1);



Możemy również stworzyć indeks unikalny wielokolumnowy:

create unique index mc_double on duza(id,wartosc);

W takim przypadku będzie musiała być zapewniona unikalność kombinacji wartości. Czyli np. może być 1000 Janów i 1000 Kowalskich, ale tylko 1 Jan Kowalski.

Mała uwaga dla Oracle'owców: Czy w Oracle mogliśmy założyć unikalny indeks na kolumnie na której jest już założony indeks nieunikalny? Nie mogliśmy ;) Przyjrzyj się ostatnim operacjom – w PostgreSQL się to udaje.

Indeksy częściowe

Wyobraźmy sobie sytuację w której mamy jakąś ogromną tabelę, ale najczęściej wyciągamy te same wiersze z tego samego zakresu. Moglibyśmy założyć indeks na całą długość kolumny, ale byłoby to marnowanie miejsca. W PostgreSQL istnieją indeksy częściowe, obejmujące tylko wskazaną część kolumny. Są one oparte o warunek WHERE. Poniżej przykład na naszej tabeli „duza”. Tworzę indeks obejmujący tylko wiersze o wartosci w kolumnie „Wartosc” w zakresie 1000000-1000500. Chwilę później uruchamiam zapytanie z którego warunku WHERE wynika, że wiersze o które pytam mają w kolumnie „wartosc” wartosc mieszcząca się w zakresie objętym przez indeks. Zauważ że zakres podany w zapytaniu jest różny od tego w indeksie, jednak się w nim zawiera. Jak widzisz planer zadecydował o wykorzystaniu naszego nowego indeksu.

create index sam_srodek on duza(wartosc) where wartosc between 1000000 and 1000500;

explain analyze select count(*) from duza where wartosc between 1000100 and 1000400;

```
create index sam_srodek on duza(wartosc) where wartosc between 1000000 and 1000500;
explain analyze select count(*) from duza where wartosc between 1000100 and 1000400;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	QUERY PLAN text
1	Aggregate (cost=18.80..18.81 rows=1 width=0) (actual time=0.387..0.387 rows=1 loops=1)
2	-> Index Only Scan using sam srodek on duza (cost=0.27..18.07 rows=290 width=0) (actua
3	Index Cond: ((wartosc >= 1000100) AND (wartosc <= 1000400))
4	Heap Fetches: 0
5	Planning time: 0.389 ms
6	Execution time: 0.462 ms

Indeksy a NULLe

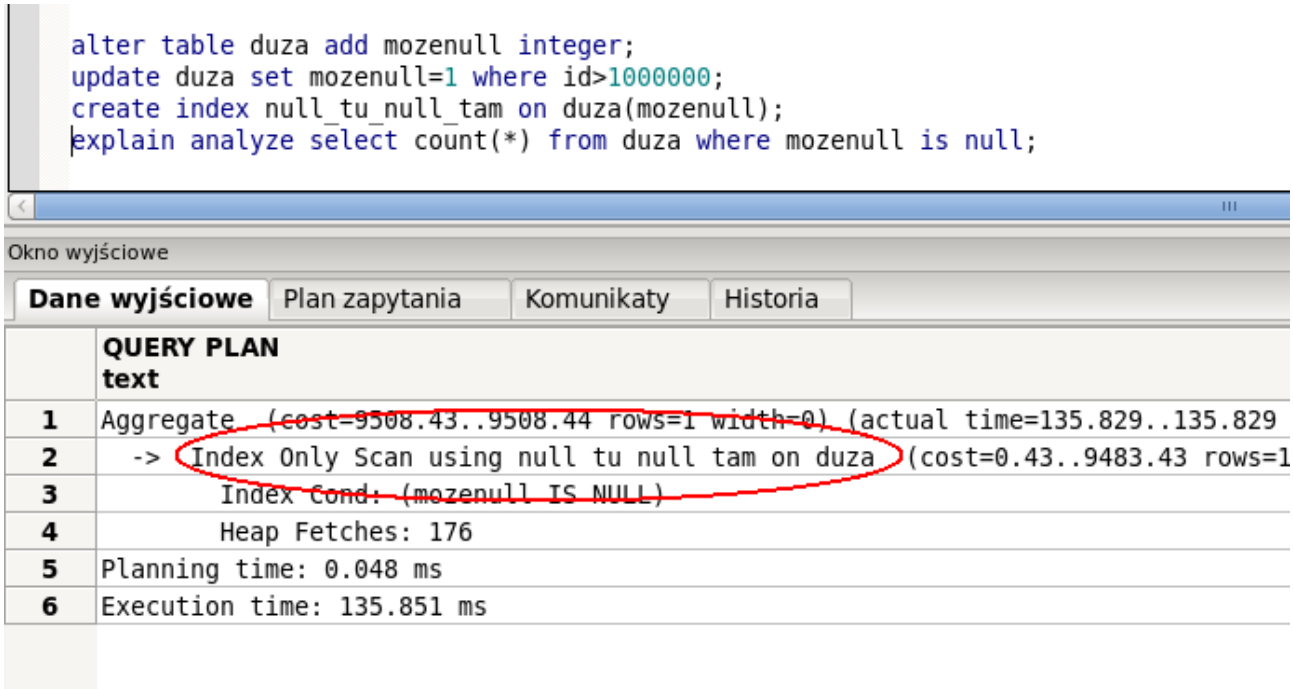
A tutaj może być pewne zaskoczenie dla osób dotychczas korzystających z baz danych Oracle. Od wersji 8.3 indeksy B-Tree mogą przechowywać Nulle i mogą być użyte do ich wyszukania lub ominięcia. Do naszej dużej tabelki dodaję jeszcze jedną kolumnę. Może ona przyjmować nulle. Zakładam na nową kolumnę indeks. Dla części wierszy ustawiam w niej jakąś wartość, a następnie sprawdzam plan wykonania zapytania zwracającego liczbę wierszy z nullem w tej kolumnie. Został użyty nowo utworzony indeks.

alter table duza add mozenull integer;

update duza set mozenull=1 where id>1000000;

create index null_tu_null_tam on duza(mozenull);

explain analyze select count(*) from duza where mozenull is null;



Indeksy funkcyjne

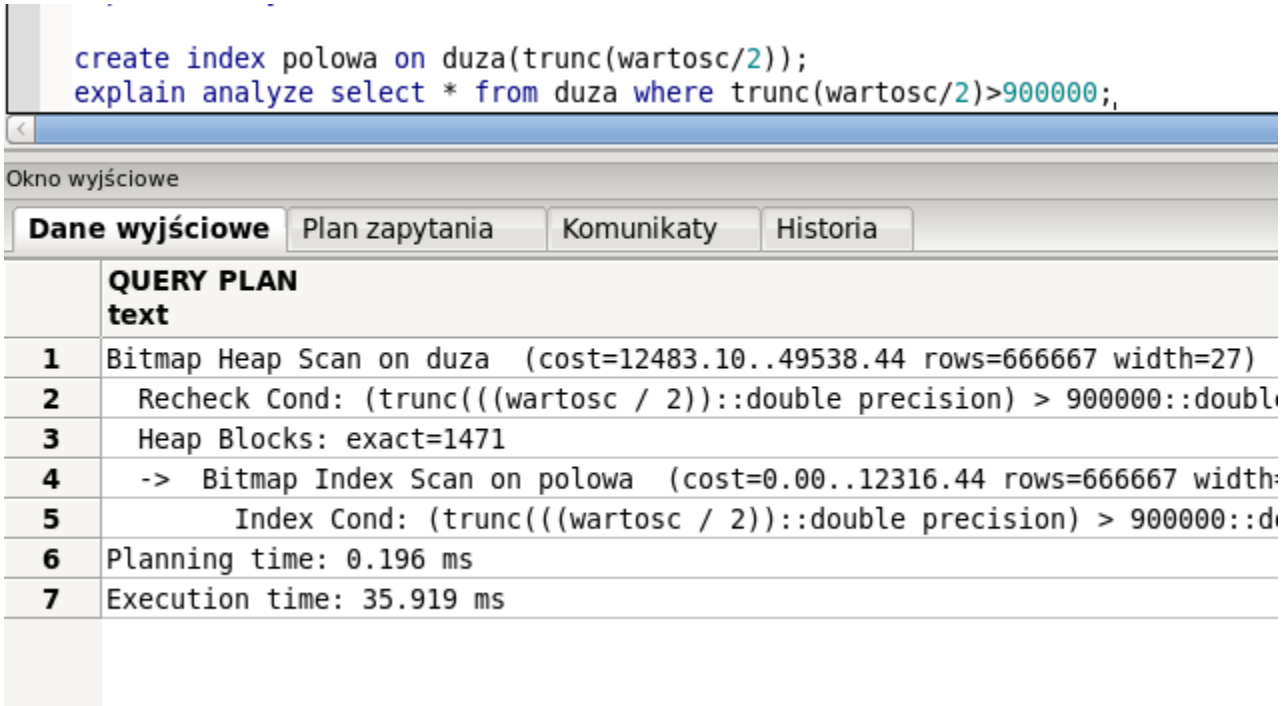
Jeśli na jakiejś kolumnie często w warunku WHERE dodatkowo stosujemy jakąś funkcję lub przeliczenie np.:

explain analyze select * from duza where trunc(wartosc/2)>900000;

możemy założyć indeks który zawierał będzie nie wartość bezpośrednią z kolumny a takie samo przeliczenie np.:

create index polowa on duza(trunc(wartosc/2));

Dzięki takiemu zabiegowi nie będzie potrzeby wykonywania tej operacji arytmetycznej „w locie" podczas wykonywania zapytania, a wartość od razu przeliczona zostanie odczytana bezpośrednio z indeksu:



Problemy wynikające z użycia indeksów

Konieczność aktualizacji

Z indeksami wcale nie jest tak różowo jak mogłoby się wydawać. Z jednej strony mogą nam pomóc w przyspieszeniu odczytu danych, ale weź też pod uwagę że takie indeksy trzeba będzie aktualizować przy wykonywaniu operacji UPDATE, DELETE i INSERT. To powoduje wydłużenie tych operacji. Nie zakładaj więc więcej indeksów niż jest niezbędne i nie stosuj ich tam gdzie korzyść z ich zastosowania jest znikoma.

Zajęte miejsce

Indeksy nie wiszą w powietrzu. Muszą być przechowywane na dysku podobnie jak tabele. Jeśli więc np. utworzysz na jakiejś tabeli indeksy na każdej kolumnie, musisz liczyć się z tym, że ilość zajmowanego miejsca na potrzeby danej tabeli oraz jej indeksów przynajmniej się podwoi.

Blokady podczas tworzenia i odbudowywania

Podczas budowania lub odbudowywania indeksu nakładana jest blokada na wszystkie wiersze których dotyczy. Wydawać by się mogło że to nic poważnego, a tymczasem wyobraź sobie tabelę wielkości kilkuset milionów wierszy. Zakładanie indeksu na takiej tabeli może trwać bardzo długo. Wszystkie transakcje które będą w tym czasie próbowały założyć swoją blokadę będą czekały (nie zostanie zgłoszony błąd) na zakończenie budowania indeksu. Weź też pod uwagę, że takie transakcje mogły wcześniej zablokować inne zasoby. Aby ten problem „obejść” możesz zastosować współbieżne tworzenie indeksu:

create index concurrently magic_trick on duza(wartosc) ;

Tworzenie takiego indeksu trwa jednak nieco dłużej od tworzenia zwykłego. Inny jest też sposób działania. PostgreSQL najpierw skanuje tabelę na której ma zostać założony indeks, odczytuje potrzebne mu wartości, następnie tworzy indeks i ponownie skanuje tabelę w poszukiwaniu wierszy które zostały dodane, skasowane lub zmienione w międzyczasie.

Widoki zmaterializowane

Widoki zmaterializowane to taki rodzaj widoku, który przechowuje wynik zapytania. Bliżej im w zasadzie do tabel budowanych na zasadzie CREATE TABLE XYZ AS SELECT … niż do zwykłych widoków. Różnice jednak są takie, że na widokach zmaterializowanych nie można wykonywać operacji UPDATE, INSERT ani DELETE, ale za to mogą być odświeżane. Widoki zmaterializowane zostały wprowadzone w wersji 9.3 PostgreSQL.

Stosuje się je najczęściej przy wyliczaniu agregatów tam gdzie wyliczenie wyniku zajmuje dużo czasu a dane źródłowe nie zmieniają się zbyt często. Taki przykład z życia wzięty. Pracujemy ostatnio nad portalem ze zleceniami online. Zlecenia są przypisane do kategorii i podkategorii. Teraz mamy zapytanie tego typu:

```
select nazwa_kategorii,count(*) liczba_zlecen
 from kategorie k join zlecenia z  on (k.id_kategorii=z.id_kategorii)
group by nazwa_kategorii;
```

Czyli po prostu lista kategorii zleceń i liczba zleceń przypisana do owych kategorii. Przy dużej liczbie zleceń takie zapytanie będzie mieliło dosyć długo, a tymczasem wynik tego zapytania musi być szybko wyświetlony na stronie. Ilość zleceń w poszczególnych kategoriach się zbyt często nie zmienia. W zasadzie tylko w przypadku dodania nowego zlecenia. Możemy więc stworzyć widok zmaterializowany oparty o takie zapytanie, a następnie odwoływać się do niego. Odczyt będzie znacznie szybszy, ponieważ nie będą mielone dane źródłowe a zostanie po prostu odczytany gotowy wynik:

```
create materialized view liczba_zlecen_kategorie as
select nazwa_kategorii,count(*) liczba_zlecen from kategorie k join zlecenia z  on (k.id_kategorii=z.id_kategorii)
group by nazwa_kategorii;
```

Odwołanie się do takiego widoku odbywa się identycznie jak do tabeli:

```
select *  from liczba_zlecen_kategorie;
```

Ponowne przeładowanie zawartości widoku zmaterializowanego odbywa się za pomocą wywołania klauzuli REFRESH:

refresh materialized view liczba_zlecen_kategorie ;

Musisz wiedzieć że refresh powoduje exclusive lock na obiektach źródłowych na czas odświeżania. W tym przypadku będzie to dotyczyło tabel kategorie i zlecenia. Istnieje też możliwość dosyć nietypowego odświeżenia – bo z wyczyszczeniem zawartości – zachowana zostanie jedynie struktura:

refresh materialized view liczba_zlecen_kategorie with no data;

Możesz też od razu stworzyć widok zmaterializowany bez danych:

create materialized view lzk as select * from liczba_zlecen_kategorie with no data;

Informacje o istniejących widokach zmaterializowanych i zapytaniach na bazie których powstały znajdziesz w słowniku pg_matviews:

select * from pg_matviews;

```
select * from pg_matviews;
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	schemaname name	matviewname name	matviewowner name	tablespace name	hasindexes boolean	ispopulated boolean	definition text
1	public	liczba_zlecen_kategorie	postgres		f	f	SELECT k.nazwa_kategorii,
2	public	lzk	postgres		f	f	SELECT liczba_zlecen_kategorie.nazwa_kategorii,

Partycjonowanie tabel

Jeśli tabela jest duża, to koszt jej skanowania również nie należy do niskich. Wyobraźmy sobie taką sytuację, że mamy tabelę z olbrzymią ilością faktur. Faktury dotyczą poszczególnych okresów – miesięcy bądź kwartałów. Są gromadzone od 2 dekad. Zazwyczaj na potrzeby analityczne będziemy przetwarzali dokumenty z aktualnego lub poprzedniego okresu, no dajmy na to najdalej roku. Jeśli będzie to zwykła niepartycjonowana tabela to przeszukiwana będzie cała długość tabeli bądź indeksu jeśli taki na niej utworzymy. To marnotrawstwo czasu i zasobów, bo przeszukiwanie wcześniejszych 19 lat jest całkiem bezcelowe. Co możemy zrobić? Możemy wydzielić w ramach tabeli partycje. Każda partycja będzie zawierała dane tylko z jednego okresu. Tabelę będzie można przeszukiwać jako całość, ale jeśli z warunków zapytania będzie wynikać że faktury których poszukujemy mogą znaleźć się tylko w określonych partycjach i nigdzie indziej, PostgreSQL przeszuka tylko te partycje. Taka konfiguracja jest zalecana przez dokumentację jeśli wielkość tabeli będzie większa od dostępnej pamięci lub wielkość tabeli przekroczy 100 milionów (!) wierszy. Jak to zrobić?

Podział na partycje

Zaczynamy od stworzenia tabeli :

```
create table some_stuff(
x integer primary key,
y integer
);
```

Jako partycję będą służyły osobne tabele które będą dziedziczyły konstrukcję po naszej właśnie stworzonej tabeli. W poniższym kodzie zwróć uwagę na parametr inherits i check. Inherits wskazuje po której tabeli dziedziczy tworzona tabela, a więc na której ma wzorować konstrukcję – dotyczy to ilości, nazw i typów kolumn. Klucze główne, obce ani indeksy nie są kopiowane. Parametr check określa warunek wskazujący warunki dla wierszy które się w tej partycji mają znaleźć. To na podstawie tego właśnie warunku PostgreSQL będzie określał które partycje należy przeszukiwać. Warto zadbać o to, by stworzyć partycje obejmujące cały zakres możliwych wartości.

Weź pod uwagę, że jeśli w tabeli bazowej (w tym przypadku some_stuff) już wcześniej znajdowały się jakieś dane, nie zostaną one rozdzielone na podległe tabele!

Na potrzeby niniejszego przykładu przyjąłem, że wstawiane wiersze mogą mieć wartość w kolumnie X w zakresie 1-900. Tworzę trzy podległe tabele (partycje jak kto woli) dzielące zakres na 3 części.

```
create table x300 (
        check (x<=300 and x>0)
)
inherits(some_stuff);

create table x600 (
        check (x<=600 and x>300)
)
inherits(some_stuff);

create table x900 (
        check (x<=900 and x>600)
)
inherits(some_stuff);
```

Jeśli tabela bazowa miała klucz główny lub klucze obce, a chcemy te własności zachować i dla partycji, musimy niestety ręcznie to skonfigurować dla każdej podległej tabeli:

```
alter table x300  add constraint x300pk primary key(x);
alter table x600  add constraint x600pk primary key(x);
alter table x900  add constraint x900pk primary key(x);
```

Automatyczne rozdzielanie wstawianych wierszy

PostgreSQL domyślnie nie będzie rozdzielał wstawianych wierszy na partycje i sami musimy zadbać o odpowiedni mechanizm. Najwydajniejszym rozwiązaniem będzie zastosowanie reguł. Reguły służą do zastępowania czynności na obiektach innymi czynnościami. Można tu by było również użyć triggerów (wyzwalaczy), ale takie rozwiązanie charakteryzuje się mniejszą wydajnością. Kod z poniższego przykładu spowoduje że w zależności od zakresu wartości w kolumnie X wstawianego wiersza, wiersze będą lądowały w różnych partycjach. Niestety musimy stworzyć taką regułę dla każdej partycji:

```
create rule ss_insert_300 as
on insert to some_stuff where (x>0 and x<=300)
do instead
insert into x300 values (new.*);


create rule ss_insert_600 as
on insert to some_stuff where (x>300 and x<=600)
do instead
insert into x600 values (new.*);

create rule ss_insert_900 as
on insert to some_stuff where (x>600 and x<=900)
do instead
insert into x900 values (new.*);
```

Teraz w ramach doświadczenia wstawmy trzy wiersze o takim zakresie wartości w X, by każdy wiersz wylądował we właściwej „podtabeli”. Zwróć uwagę że inserty są wykonywane na naszej „głównej” tabeli a nie partycji. O odpowiednie rozdzielenie wierszy zadbają stworzone przed momentem reguły:

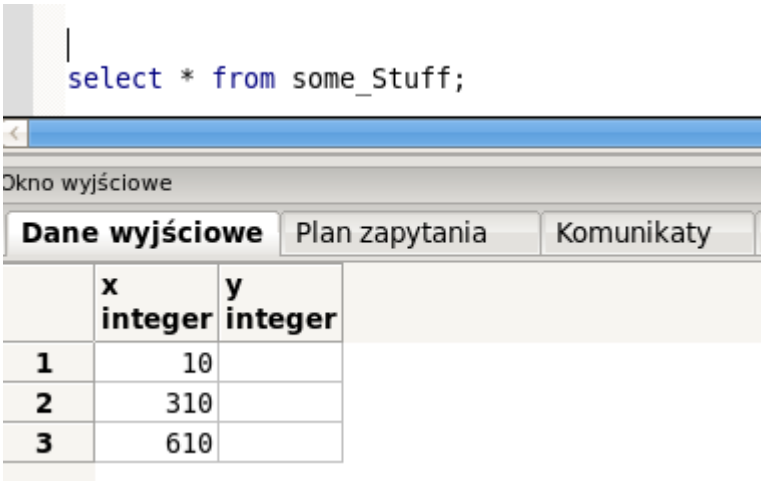
insert into some_Stuff values (10,null);

insert into some_Stuff values (310,null);

insert into some_Stuff values (610,null);

Zajrzyjmy teraz do naszej „głównej” tabeli:

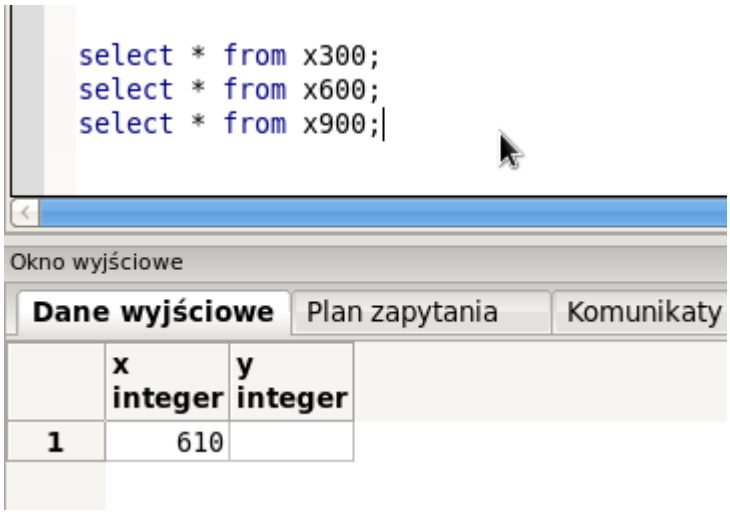
select * from some_Stuff;



Widzimy wszystkie wstawiane dane pomimo że znajdują się one tak naprawdę w osobnych tabelach.

Zajrzyjmy teraz do którejś partycji osobno:

select * from x900;

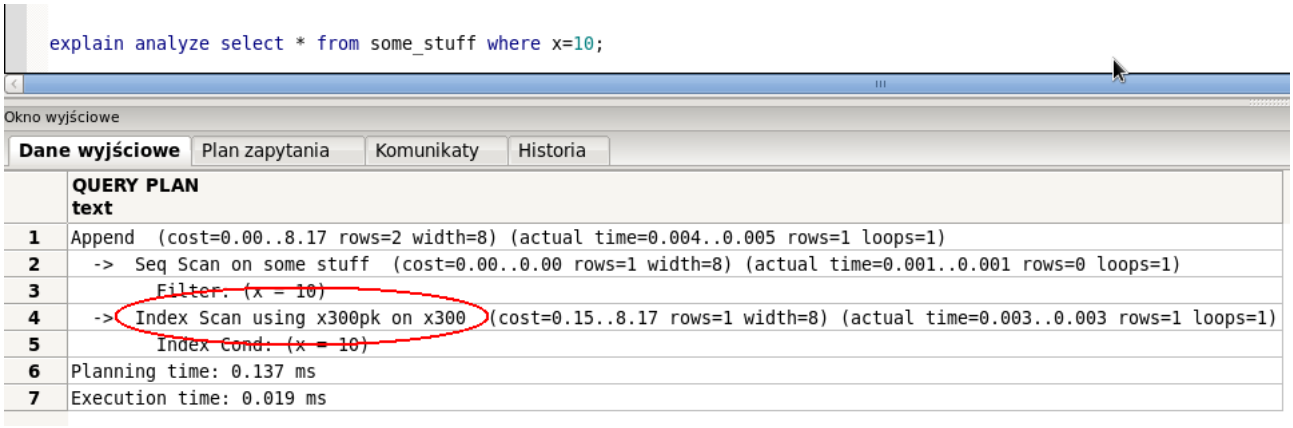


Czyli dane zostały rozdzielone na właściwe „podtabele”.

Automatyczne przeszukiwanie tylko właściwych partycji

Zobaczmy teraz co się stanie jeśli odpytam naszą tabelę nadrzędną, a z warunków WHERE będzie wynikać że dane których szukam mogą być wyłącznie w jednej partycji:

explain analyze select * from some_stuff where x=10;



Jak widzimy nastąpił skan z użyciem indeksu założonego w związku z utworzeniem klucza głównego na jednej tylko partycji. Czyli PostgreSQL zajrzał tak naprawdę tylko do jednej partycji.

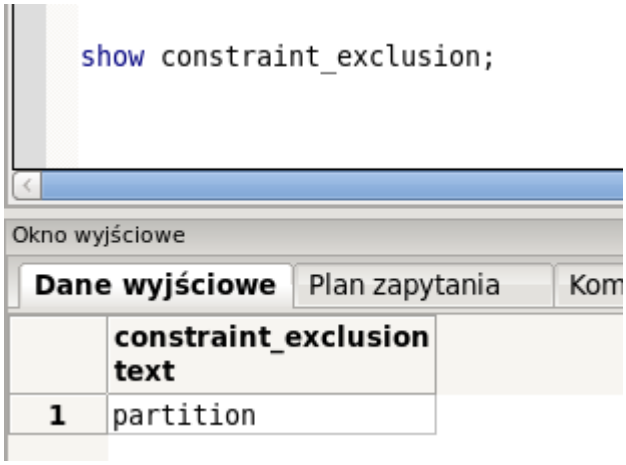
Uwagi do partycjonowania

Parametr constraint_exclusion

W PostgreSQL jest parametr constraint_exclusion od którego włączenia zależy czy mechanizm przeszukiwania wybranych partycji działa. Od wersji 8.4 jest on włączony domyślnie. Jeśli masz wcześniejszą wersję PostgreSQL i nie jesteś pewien czy u Ciebie jest to włączone, sprawdź to z użyciem komendy:

show constraint_exclusion;

Powinieneś dostać taki rezultat:



Automatyczne tworzenie nowych partycji

Niestety na ten moment (wersja 9.4) nie istnieje metoda na automatyczne tworzenie kolejnych partycji przez PostgreSQL. Robi się to po prostu przez automatyczne uruchamianie skryptów przez CRONa, jeśli np. mamy nowy okres rozliczeniowy a nie została jeszcze utworzona dla niego partycja.

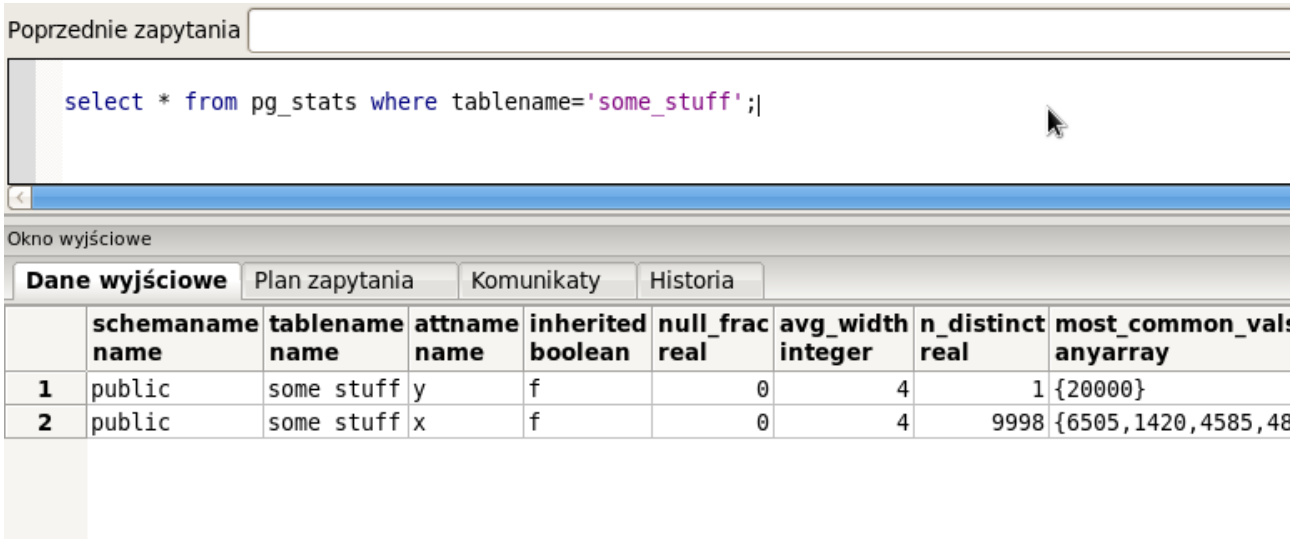
Statystyki obiektów

Informacje podstawowe

Statystyki służą lepszemu planowaniu wykonania zapytań. Odnoszą się np. do wielkości tabel, zróżnicowania danych w kolumnach etc. Im lepiej oddają one rzeczywisty stan danych, tym plany wykonania będą wydajniejsze. Mogą one być zbierane ręcznie z użyciem polecenia ANALYZE, lub przez demona autovacuum.

Dane statystyczne możemy przejrzeć np. z użyciem widoku pg_stats:

select * from pg_stats where tablename='some_stuff';



	schemaname name	tablename name	attname name	inherited boolean	null_frac real	avg_width integer	n_distinct real	most_common_vals anyarray
1	public	some stuff	y	f	0	4	1	{20000}
2	public	some stuff	x	f	0	4	9998	{6505,1420,4585,48

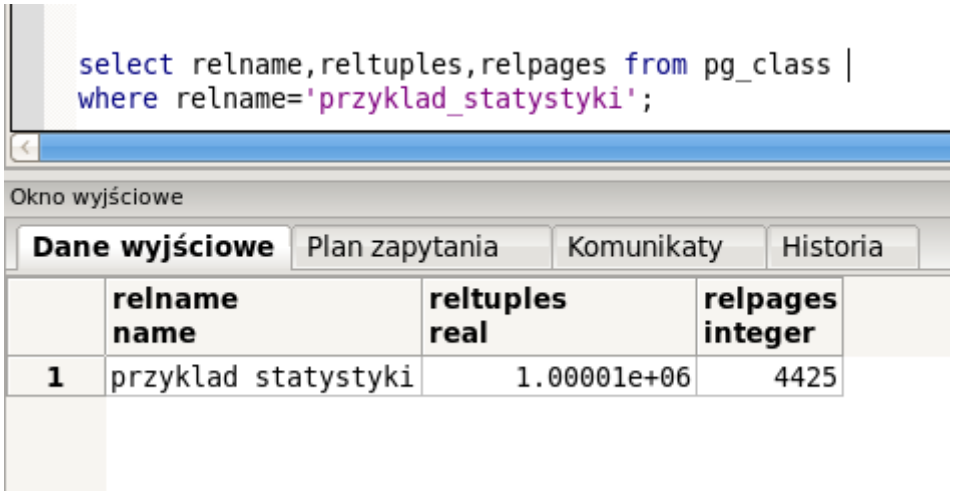
Jeśli uruchamiasz powyższy przykład jako inny niż superużytkownik user bazodanowy, pamiętaj że zobaczysz w tym słowniku informacje o obiekcie tylko jeśli masz uprawnienia dostępu do tego obiektu.

Możemy tam znaleźć na przykład informacje o zróżnicowaniu danych w poszczególnych

kolumnach, czy ilości wartości unikalnych. Ciekawe informacje znajdziemy również w słowniku pg_class:

select relname,reltuples,relpages from pg_class

where relname='przyklad_statystyki';



Z tego słownika dowiemy się ile jest wierszy w tabeli (reltuples), oraz jaką zajmują przestrzeń (relpages) wyrażoną w ilości bloków.

Odświeżanie statystyk

Statystyki możemy odświeżać ręcznie lub pozostawić to zadanie demonowi autovacuum. Aby odświeżyć statystyki ręcznie dla obiektu używamy komendy analyze:

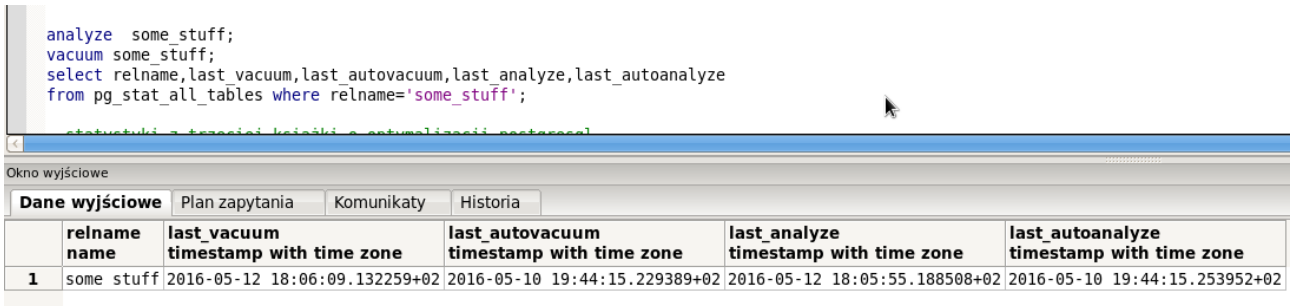
analyze some_stuff;

Jeśli zechcemy sprawdzić kiedy dla jakiegoś obiektu zostały ostatnio odświeżone statystyki, bądź odnaleźć obiekty dla których statystyki dawno nie były odświeżane możemy zajrzeć do słownika pg_stat_all_tables. Poniżej przykładowy kod z użyciem którego po odświeżeniu statystyk przeprowadziłem jeszcze czyszczenie tabeli i sprawdzam kiedy ostatnio było przeprowadzone czyszczenie i analiza zarówno ręcznie jak i automatycznie:

vacuum some_stuff;

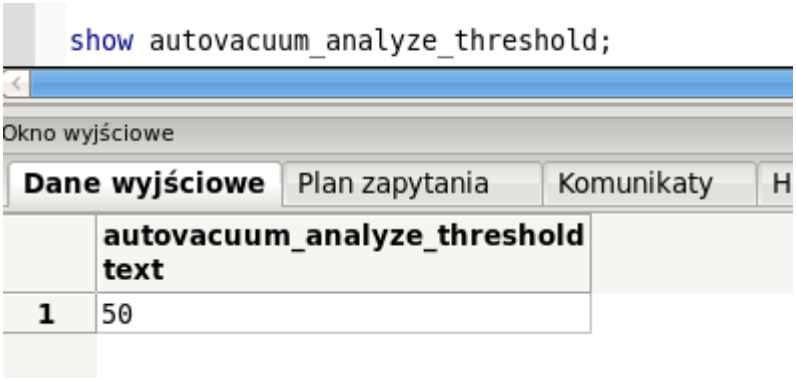
select relname,last_vacuum,last_autovacuum,last_analyze,last_autoanalyze

from pg_stat_all_tables where relname='some_stuff';



Proces autovacuum odświeża statystyki obiektów automatycznie. Robi to dla tych tabel, dla których ilość zmienionych wierszy przekroczy wartość określoną w parametrze autovacuum_analyze_threshold:

show autovacuum_analyze_threshold;



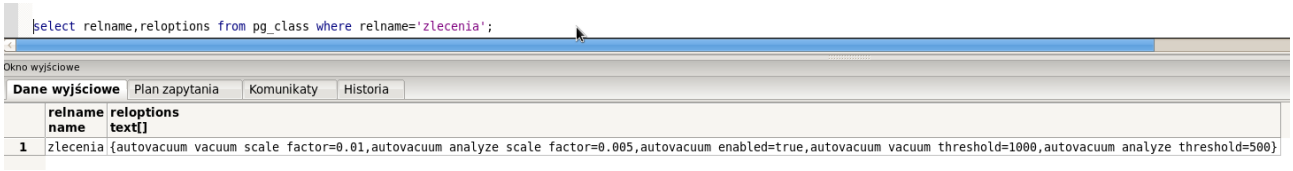
który jest domyślnie ustawiony na 50 wierszy. Parametr ten jest ustawiony dla całej bazy, jednak możemy zmienić jego wartość dla wybranych tabel indywidualnie:

ALTER TABLE zlecenia

SET (autovacuum_analyze_threshold=500);

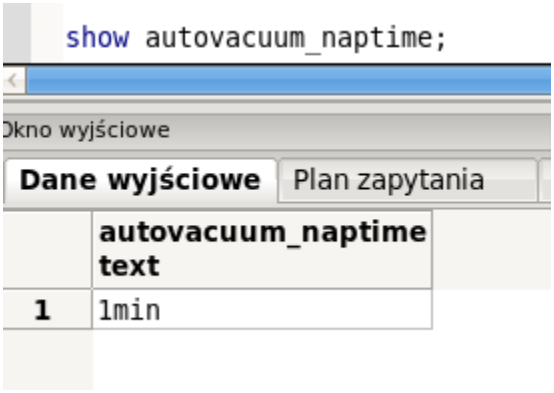
A jakbyśmy chcieli sprawdzić indywidualne ustawienia dla tabeli, wystarczy zajrzeć do słownika pg_class:

select relname,reloptions from pg_class where relname='zlecenia';



Proces autovacuum uruchamia się sam co czas określony w autovacuum_naptime domyślnie ustawionym na minutę:

show autovacuum_naptime;



Czyli w skrócie – autovacuum uruchomi się automatycznie co czas określony w parametrze **autovacuum_naptime** i odświeży statystyki dla tych tabel w których zostanie zmienione więcej wierszy niż określone w parametrze **autovacuum_analyze_threshold** dla bazy lub indywidualnie dla obiektu.

Default_statistics_target i histogram_bounds

Im więcej mamy danych statystycznych i im bardziej są one precyzyjne, tym lepiej planowane są algorytmy wykonania zapytań. Z drugiej strony zbieranie takich danych to zużycie zasobów systemowych i czas potrzebny na ich przetworzenie. To jak dużo jest zbieranych danych statystycznych zależy od parametru default_statistics_target. Myślę że najlepiej będzie to przeanalizować na przykładzie. Stwórzmy przykładową tabelkę i załadujmy do niej dane:

create table przyklad_statystyki (

x integer

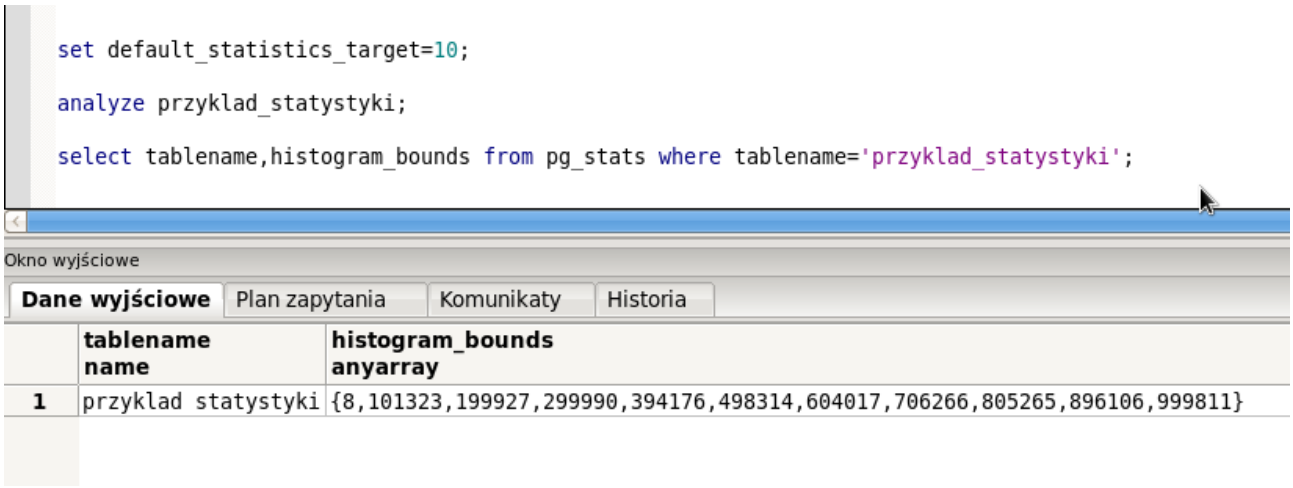
);

insert into przyklad_statystyki values (generate_series (1,1000000));

Zmienimy teraz parametr default_statistics_target na czas trwania sejsi na wartość 10. Jej domyślna wartość to 100. Następnie przeanalizujmy tabelkę . Przyjrzyj się kolumnie histogram_bounds w tym i następnym przykładzie.

set default_statistics_target=10;

analyze przyklad_statystyki;



Teraz zmienimy wartość parametru na 3 i ponownie przenalizujemy tabelę:

set default_statistics_target=3;

analyze przyklad_statystyki;

select tablename,histogram_bounds from pg_stats where tablename='przyklad_statystyki';

```
set default_statistics_target=3;

analyze przyklad_statystyki;

select tablename,histogram_bounds from pg_stats where tablename='przyklad_statystyki';
```

Okno wyjściowe

Dane wyjściowe | Plan zapytania | Komunikaty | Historia

	tablename name	histogram_bounds anyarray
1	przyklad statystyki	{169,347591,636846,999769}

W kolumnie histogram_bounds znajdziemy informacje na temat dystrybucji danych w kolumnie. W tabeli przyklad_statystyki mamy tylko jedną kolumnę, o nazwie X tak więc nie wyświetlałem jej. Pamiętaj że histogram_bounds nie odnosi się do tabeli a do kolumn! Porównaj ilość wpisów w tym i poprzednim przykładzie. W kolumnie X mamy wartości w zakresie 1-1000000. Informacje z histogram_bounds oddają mniej więcej rozłożenie tych danych. Im więcej danych mamy w histogram_bounds, tym lepiej może być szacowana np. selektywność przy wybieraniu wierszy z użyciem warunków w WHERE.

Parametr default_statistics_target przyjmuje wartości w zakresie 1-1000. Jak widzisz w powyższych przykładach wartość ta odnosi się do ilości histogramów dla poszczególnych kolumn. Parametr ten możesz zmieniać dla sesji, albo dla całej bazy danych. Domyślne ustawienie (100) jest zwykle wystarczające, choć zależy głównie od wielkości tabeli i rozłożenia wystąpień wartości. Wyobraź sobie że masz tabelę w które przechowujesz informację o obywatelach Polski i kodzie pocztowym do jakiego są przypisani. Gdyby do każdego kodu pocztowego była przypisana taka sama liczba obywateli, mało precyzyjny histogram (mający mało wartości) byłby wystarczający, ponieważ selektywność dla każdego kodu pocztowego byłaby zbliżona. Wystarczyłaby nam zasadniczo nawet średnia selektywność. Ponieważ rozłożenie jest całkiem inne w rzeczywistości, może dojść do sytuacji takiej – dla 90% kodów pocztowych średnio w jednym jest np. 200 obywateli (wartość wyjęta z kapelusza – zupełnie nie mam pojęcia czy jest właściwa, przyjąłem taką na potrzeby przykładu) a 10% po 30000 (np. w centrach dużych miast). Jeśli mielibyśmy histogram mało precyzyjny – np. ustawilibyśmy wartość 5 dla default_statistics_target mogłoby się

okazać że histogram_bounds wygląda tak: 203,240,189,196,233,210. Na takim histogramie bazowałoby obliczenie selektywności przy wyborze wierszy. Lepiej byłoby, żeby PostgreSQL wiedział o istnieniu takich anomaliów w całym zakresie jak kody pocztowe pod którymi mieszka 30000 obywateli, ponieważ wtedy plan wykonania mógłby wyglądać całkiem inaczej.

Podsumowując – jeśli mamy duże zróżnicowanie zagęszczenia wartości w kolumnie, a jednocześnie dana kolumna często występuje w warunkach where – warto rozważyć wyższą precyzję statystyk dla niej. Możemy ją ustawić indywidualnie dla każdej z kolumn – niezależnie od odgórnego ustawienia default_statistics_target. Aby ustawić pożądany poziom statystyk dla wybranej kolumny tabeli wydajemy polecenie:

ALTER TABLE nazwa_tabeli ALTER COLUMN nazwa_kolumny SET STATISTICS x;

Aby przywrócić pierwotną domyślną wartość wydajemy polecenie:

ALTER TABLE nazwa_tabeli ALTER COLUMN nazwa_kolumny SET STATISTICS -1;

Wartość -1 ustawia domyślną wartość określaną przez parametr default_statistics_target dla jego aktualnego ustawienia. Oczywiście nie możemy zapomnieć o ponownym uruchomieniu polecenia ANALYZE!

Klastrowanie tabel

Operacja klastrowania służy takiemu rozłożeniu danych w tabeli, by wiersze mające te same wartości w wybranej zaindeksowanej kolumnie leżały obok siebie. Dzięki temu że dane są bliżej siebie, również odczytywanie ich będzie się odbywało szybciej. Wieść gminna niesie, że odczyty zakresowe i po wartości potrafią przyspieszyć nawet 20 krotnie! Twierdzenie to opieram na badaniach Hansa Hasselberga. Klastrowanie jest operacją jednorazową. Klastrujemy tabelę według wskazanego indeksu. Musisz pamiętać, że nowe dane dochodzące do tabeli po klastrowaniu nie będą uporządkowane w ten sam sposób. To wymusza okresowe odtwarzanie klastrowania gdybyś zechciał uporządkować nowe wiersze. Ważna uwaga – podczas klastrowania na tabeli zakładana jest blokada wyłączna co skutkuje całkowitym zablokowaniem dostępu do niej – z odczytem włącznie na czas operacji.

Jeśli zazwyczaj odczytujesz wiersze z tabeli w sposób sekwencyjny, klastrowanie Ci nie pomoże. Odczujesz korzyść z klastrowania wyłącznie jeśli wybierasz je według jakiejś wartości w kolumnie na której założony jest indeks o który oparte jest klastrowanie. Kolejna ważna uwaga. Dokumentacja głosi, że na czas skanowania indeksu o który oparty jest klaster tworzona jest tymczasowa tabela zawierająca dane z tabeli uporządkowane wg sortowania w indeksie. Tworzone są również kopie innych indeksów leżących na tej tabeli. To oznacza, że musimy dysponować wolną przestrzenią o wielkości co najmniej takiej jak tabela i indeksy na niej razem.

Najlepiej będzie przetestować efektywność tego rozwiązania na przykładzie. Ponownie skorzystamy z naszej przykładowej tabeli „zlecenia”. Sprawdzam ilość wartości występujących w kolumnie budżet. Mamy 100 różnych wartości na ponad 2 miliony wierszy. Zróżnicowanie nie jest więc zbyt duże.

```
select count(distinct budzet) from zlecenia;
```

Okno wyjściowe

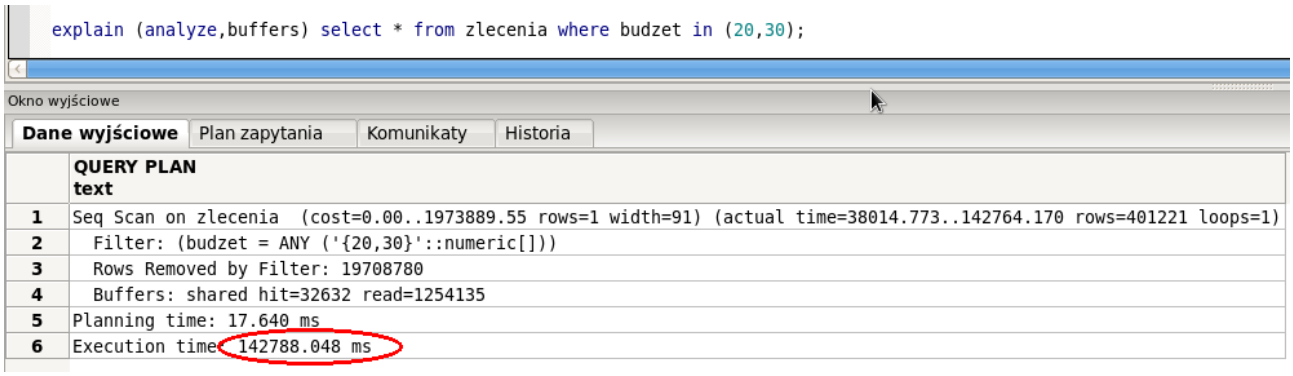
Dane wyjściowe | Plan zapytania | Komunikaty | Histori

	count bigint
1	100

Sprawdźmy więc jak przedstawia się czas wybrania zleceń o budżetach 20 i 30 z tabeli. Na razie nie

mam żadnych indeksów, ani tabela nie jest klastrowana:

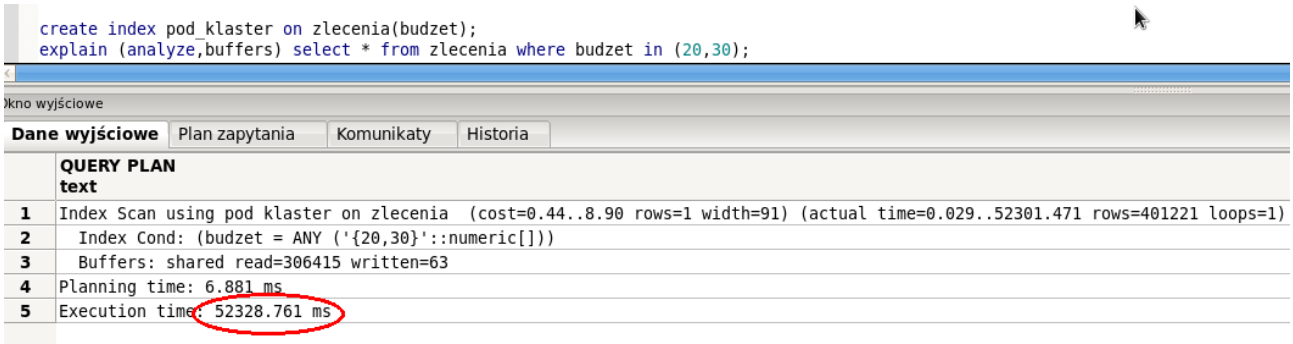
explain (analyze,buffers) select * from zlecenia where budzet in (20,30);



Zakładam indeks na kolumnę budżet. Zobaczmy o ile spadnie koszt zapytania:

create index pod_klaster on zlecenia(budzet);

explain (analyze,buffers) select * from zlecenia where budzet in (20,30);

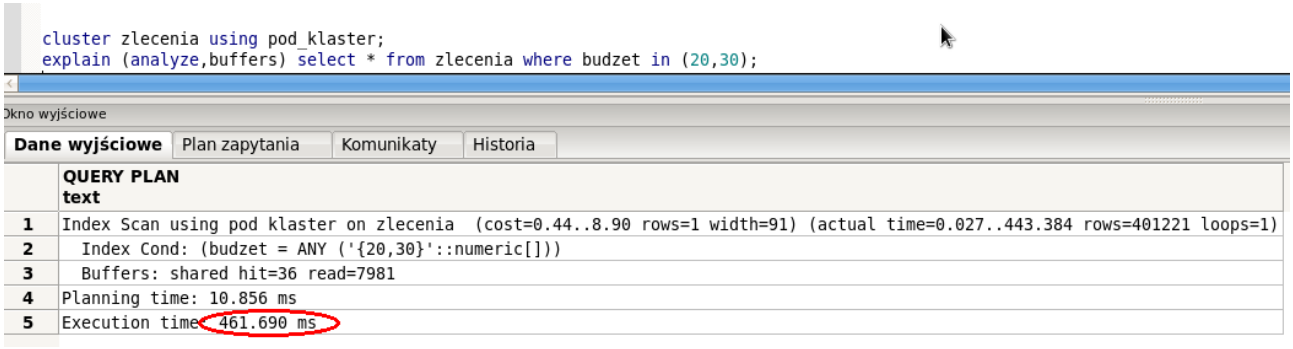


Koszt spadł niemal trzykrotnie. Selektywność jest na poziomie 2%. Gdyby wiersze z taką samą wartością w kolumnie budżet leżały obok siebie w tabeli, na pewno ich odczytanie byłoby znacznie szybsze.

Klastruję tabelę wg nowo stworzonego indeksu:

cluster zlecenia using pod_klaster;

explain (analyze,buffers) select * from zlecenia where budzet in (20,30);



Trochę to potrwało, ale i opłacało się. Czas wykonania w sumie spadł z ok 150 tys do 461...

Tabela może być sklastowana tylko wg jednego kryterium. Nie możemy założyć kilku indeksów i sklastrować tabeli wg każdego z nich. Wybieramy więc taką kolumnę, po której filtrowanie koszuje najwięcej.

Pamiętaj że dane zostały dobrze poukładane na ten moment, nowo wstawiane zlecenia będą już leżały w innym miejscu tabeli. Dobrze jest więc raz na jakiś czas odświeżyć klastrowanie. Jeśli już raz przeprowadziłeś tę operację dla wybranej tabeli możesz wywołać taką komendę:

cluster zlecenia;

Lub ponowić operację dla wszystkich klastrowanych tabel w bazie:

cluster;

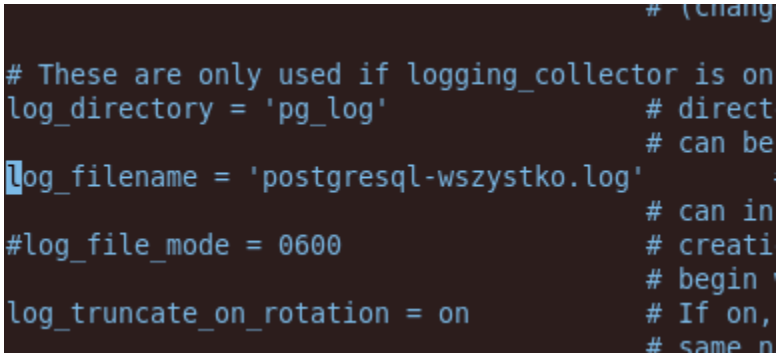
Logowanie wolnych zapytań

Możemy znać dziesiątki technik optymalizacyjnych, ale co nam po tym jeśli nie wiemy wobec jakich zapytań je stosować? W ramach tego artykułu ustawimy taką konfigurację, aby wszystkie zapytania których czas wykonania przekroczy wskazaną wartość pojawiały nam się w logu. Ponadto zadbamy o to aby widzieć które zapytania tworzą pliki tymczasowe na dysku (np. podczas sortowania) i jakiej wielkości są te pliki, które zapytania czekają z powodu blokad.

Ustawienie logowania do jednego pliku

W domyślnej konfiguracji logi są zrzucane do 7 plików w katalogu $PGDATA/pg_log, po jednym dla każdego dnia tygodnia które są rotacyjnie nadpisywane. Jeśli chciałbyś mieć log swojego serwera cały czas na podglądzie to nie jest to najbardziej wygodne rozwiązanie. Zaczniemy więc od takiego ustawienia, które sprawi że wszystkie logi będą lądowały w jednym pliku.

Wyedytuj plik postgresql.conf i odnajdź parametr log_filename. Ten parametr określa nazwę pliku logów. Domyślnie w tej nazwie występuje zmienna %D która to właśnie sprawia że dla każdego dnia tygodnia powstaje nowy plik logów. Usuń tę zmienną, bądź w ogóle zmień nazwę pliku na inną:



Aby opcja ta zaczełą działać musimy przeładować konfigurację:

psql -U postgres -d zlecenia -c "select pg_reload_conf()";

```
[root@vps-1077604-8206 data]# psql -U postgres -d zlecenia -c "select pg_reload_conf()";
 pg_reload_conf
----------------
 t
(1 wiersz)

[root@vps-1077604-8206 data]#
```

Oczywiście zmień nazwę bazy danych na swoją :) Po tym zabiegu powinien nam powstać nowy plik logów w katalogu pg_log:

```
[root@vps-1077604-8206 data]# ls pg_log
postgresql-Fri.log  postgresql-Sat.log  postgresql-Thu.log  postgresql-Wed.log
postgresql-Mon.log  postgresql-Sun.log  postgresql-Tue.log  postgresql-wszystko.log
[root@vps-1077604-8206 data]#
```

aby uruchomić jego podgląd wystarczy wykorzystać narzędzie tail:

tail -f /var/lib/pgsql/9.4/data/pg_log/postgresql-wszystko.log

```
[root@vps-1077604-8206 ~]# tail -f /var/lib/pgsql/9.4/data/pg_log/postgresql-wsz
ystko.log
< 2016-05-19 15:59:18 CEST [nieznany] zlecenia zlecenia 31.179.131.210(60692)  >
DZIENNIK:  plik tymczasowy: ścieżka "base/pgsql_tmp/pgsql_tmp19372.2", rozmiar 5
496832
< 2016-05-19 15:59:22 CEST [nieznany] zlecenia zlecenia 31.179.131.210(60692)  >
DZIENNIK:  czas trwania: 3172.689 ms  wykonanie <unnamed>: select * from zleceni
```

Ustawienia logowania

Wszystkie parametry dotyczące logowania znajdują się w pliku postgresql.conf, tak więc otwórz go ulubionym edytorem. Będziemy zmieniać 4 parametry:

log_min_duration_statement

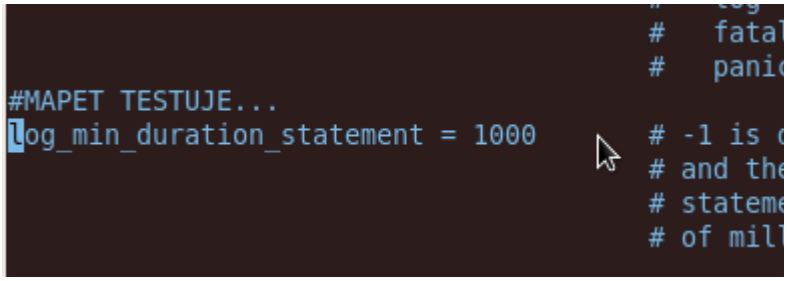
log_line_prefix

log_lock_waits

log_temp_files

LOG_MIN_DURATION_STATEMENT

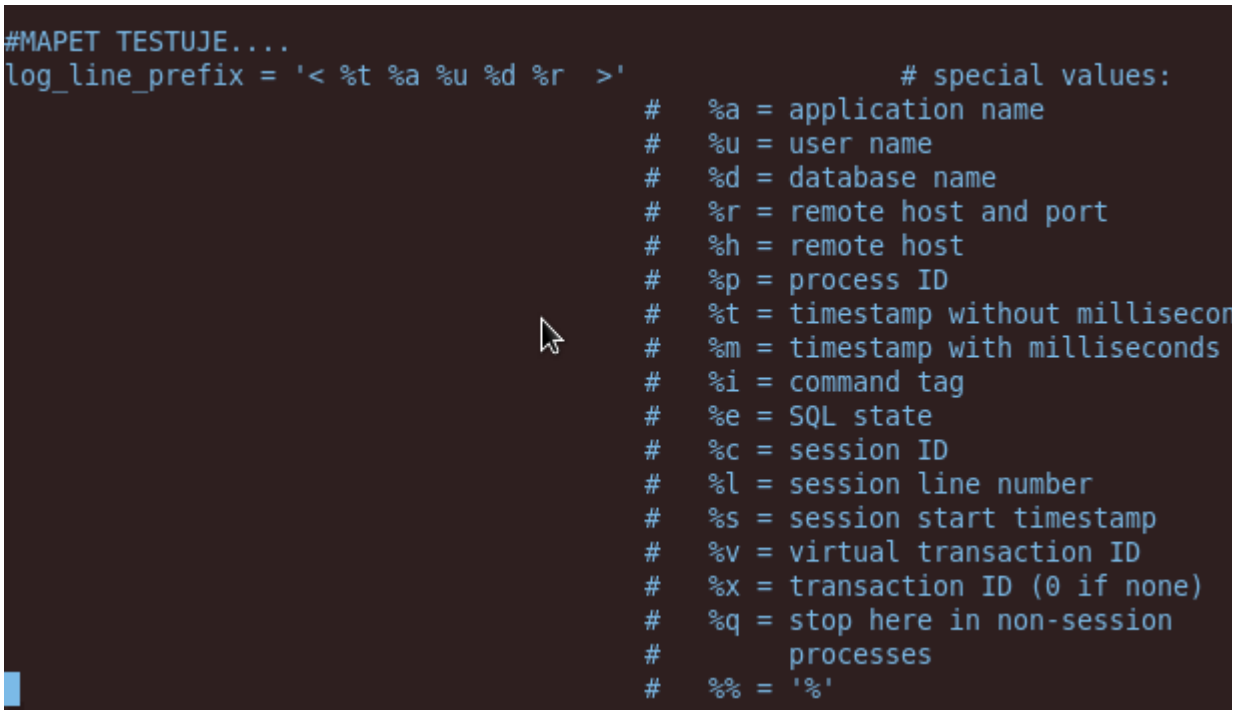
Jak opisałem to na początku – będziemy logować wystąpienia zapytań trwających dłużej niż wskazany czas. Należałoby więc ten czas wskazać i robimy to właśnie z użyciem tego parametru. Jego wartość jest wyrażana w milisekundach. Ja ustawiłem na sekundę – czyli 1000 milisekund.



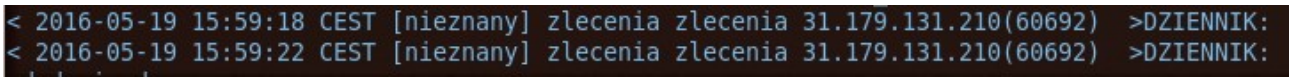
LOG_LINE_PREFIX

Wpisy w logach będą domyślnie zawierały tylko czas wykonania zapytania, a my je sobie wzbogacimy o dodatkowe informacje. Ten parametr określa właśnie jakie mają to być informacje.

Ja ustawiam parametry %t, %a, %u, %d i %r. Oznaczają one logowanie kolejno: moment uruchomienia zapytania, aplikacja która je uruchamia, użytkownika bazodanowego, nazwę bazy na której jest wykonane zapytanie, host kliencki z którego zapytanie zostało uruchomione.



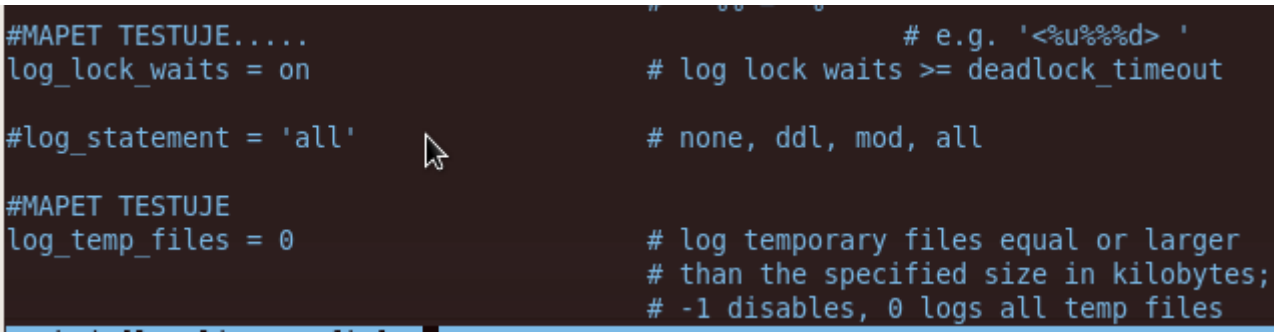
Przykładowy fragment prefiksu wygenerowany w wyniku w.w ustawień:



LOG_LOCK_WAITS i LOG_TEMP_FILES

LOG_LOCK_WAITS to parametr którego włączenie spowoduje logowanie zapytań które musiały oczekiwać na wykonanie w związku z założoną blokadą na obiektach, niezależnie od czasu zapytania i ustawień parametru log_min_duration_statement.

Ustawienie parametru log_temp_files spowoduje logowanie zapytań które w związku z wykonaniem zapytania utworzyły pliki tymczasowe na dysku. Ustawienie go na 0 spowoduje logowanie każdego zapytania które utworzy taki plik, na wartość >0 tylko tych zapytań które utworzą plik o wielkości większej niż podana wartość (w kilobajtach). Ustawienie na -1 wyłącza logowanie takich zapytań. Warto jest mieć tę opcję włączoną, ponieważ takie pliki są wytwarzane w związku np. z sortowaniem i ich pojawienie się jest sygnałem dla administratora że trzeba zwiększyć wartość parametru work_mem, ponieważ sortowanie z użyciem dysku drastycznie spowalnia zapytania.

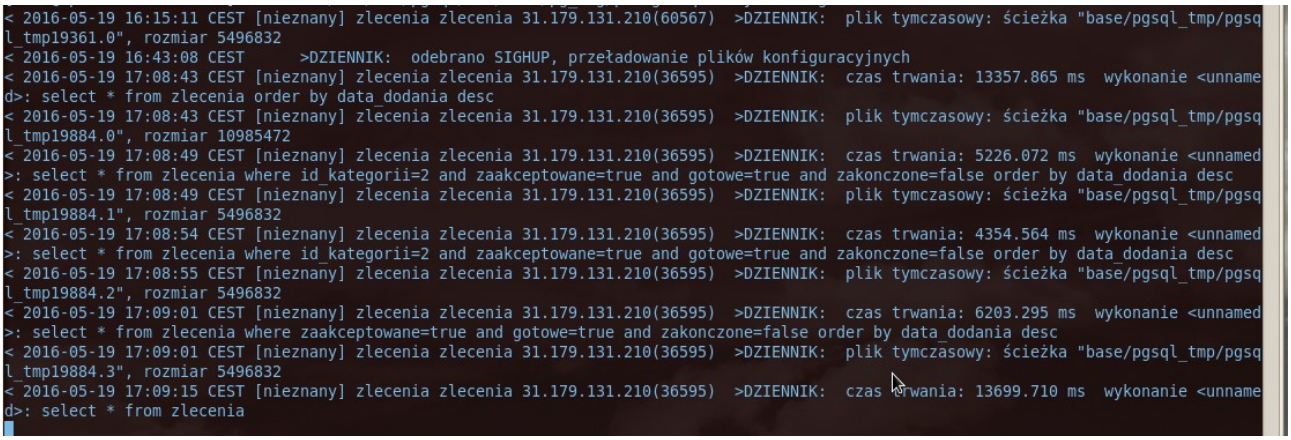


Aby zmiany parametrów obowiązywały należy jeszcze przeładować konfigurację:

psql -U postgres -d zlecenia -c "select pg_reload_conf()";

Przeglądanie logów

Poniżej przykład wpisów w logu jakie pojawiają się w wyniku ustawienia wymienionych wcześniej ustawień:



Jest jeszcze parametr debug_print_plan który umożliwia logowanie również planów wykonywanych zapytań, ale format wyświetlania jest tak nieczytelny że znacznie wygodniej jest zweryfikować plany wykonań „ręcznie".

PgBench – testy wydajnościowe bazy danych

Pgbench jest narzędziem typu benchmark który możemy wykorzystać do testowania wydajności baz danych PostgreSQL.

Przygotowanie środowiska

Narzędzie to na potrzeby testów wymaga utworzenia kilku tabel, na których następnie będzie wykonywał testy wydajnościowe mierząc wydajność operacji SELECT,UPDATE,DELETE,INSERT. Z tego powodu na potrzeby pgbench warto przygotować sobie osobną bazę danych.

create database pgbench;

Wielkość tabel tworzonych przez PgBench może być różna i zależy od wartości którą podamy przy inicjalizacji. Na potrzeby testów tworzone są tabele zawierające informacje o oddziałach fikcyjnego banku, inkasentach i rachunkach. Wielkość tabel określamy przy użyciu skali wprowadzanej przy użyciu przełącznika -s. Przy użyciu tego parametru podajemy skalę będącą ilością oddziałów. I tak – na każdy jeden oddział przypada 10 kasjerów i 100 000 rachunków. Oznacza to, że uruchomienie np. skali 20 spowoduje dodanie 20 oddziałów, 200 kasjerów i 2 000 000 rachunków. Wielkość bazy testowej silnie wpływa na wykorzystanie buforów i ilość operacji I/O podczas testów. Aby testy były względnie miarodajne, powinniśmy utworzyć bazę wielkości zbliżonej do naszej bazy produkcyjnej. Jeśli pgbench nie jest zainstalowany w Twojej bazie danych zainstaluj go:

yum install postgresql-contrib
yum install postgresql94-contrib

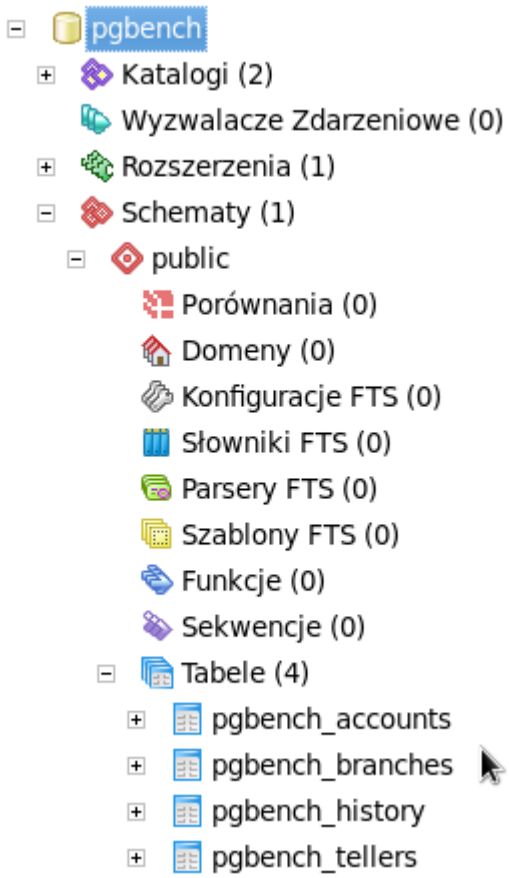
Przechodzimy teraz do inicjalizacji bazy testowej. Przejdź do katalogu z binariami PostgreSQL i uruchom instrukcję podobną do poniższej. Przełącznik -i oznacza inicjalizację, -s oznacza skalę – w tym przypadku stworzone zostanie 20 oddziałów. Na końcu podajemy bazę danych na której mają być przeprowadzane testy – tj. w której mają zostać utworzone tabele na potrzeby testów. Jeśli nie podamy skali, zostanie przyjęta domyślna wartość 1.

./pgbench -i -s 20 pgbench

Po uruchomieniu zostaniemy poproszeni o hasło do bazy (ściślej dla użytkownika postgres).

```
bash-4.1$ ./pgbench -i -s 20 pgbench
Password:
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
creating tables...
100000 of 2000000 tuples (5%) done (elapsed 0.21 s, remaining 4.04 s).
200000 of 2000000 tuples (10%) done (elapsed 0.64 s, remaining 5.75 s).
300000 of 2000000 tuples (15%) done (elapsed 1.14 s, remaining 6.45 s).
400000 of 2000000 tuples (20%) done (elapsed 1.39 s, remaining 5.56 s).
500000 of 2000000 tuples (25%) done (elapsed 1.72 s, remaining 5.17 s).
600000 of 2000000 tuples (30%) done (elapsed 2.29 s, remaining 5.34 s).
700000 of 2000000 tuples (35%) done (elapsed 3.90 s, remaining 7.25 s).
800000 of 2000000 tuples (40%) done (elapsed 4.36 s, remaining 6.54 s).
900000 of 2000000 tuples (45%) done (elapsed 4.60 s, remaining 5.62 s).
1000000 of 2000000 tuples (50%) done (elapsed 4.93 s, remaining 4.93 s).
1100000 of 2000000 tuples (55%) done (elapsed 5.96 s, remaining 4.88 s).
1200000 of 2000000 tuples (60%) done (elapsed 6.86 s, remaining 4.57 s).
1300000 of 2000000 tuples (65%) done (elapsed 7.20 s, remaining 3.88 s).
1400000 of 2000000 tuples (70%) done (elapsed 7.97 s, remaining 3.42 s).
1500000 of 2000000 tuples (75%) done (elapsed 10.30 s, remaining 3.43 s).
```

Po inicjalizacji możesz zajrzeć do wskazanej testowej bazy danych, aby sprawdzić czy tabele na pewno zostały utworzone.

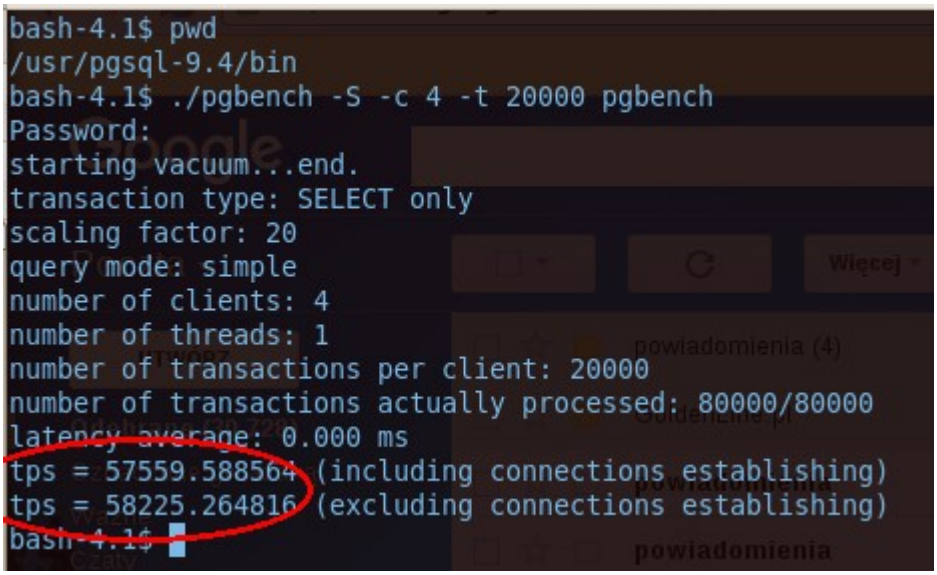


Pierwszy test

Zaczniemy od prostego testu wydajności opartego tylko na operacjach odczytu – SELECT.

./pgbench -S -c 4 -t 20000 pgbench

Parametr -S oznacza testy tylko z użyciem operacji SELECT, parametr -c oznacza ilość klientów bazy danych poprzez które mają być przeprowadzane testy, parametr -t oznacza ilość transakcji do wykonania przez każdego klienta. W tym przypadku zostanie więc wykonane 80000 zapytań SELECT z użyciem 4 klientów.



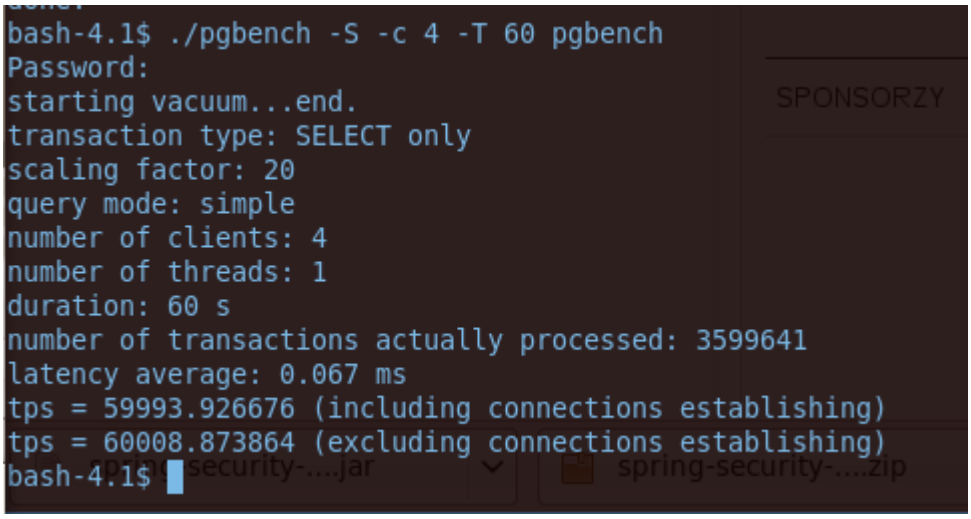
Poza informacjami konfiguracyjnymi które sami wprowadziliśmy, na końcu zobaczymy najważniejszą dla nas średnią ilość transakcji którą udało się przeprowadzić w ciągu sekundy (tps – transaction per second).

Rodzaje testów i przełączniki

W poprzednim przykładzie uruchamiając testy podałem ilość transakcji do wykonania. Możemy również wykorzystać przełącznik -T aby zamiast ilości transakcji podać czas przez jaki mają być przeprowadzane testy. Wartość którą podajemy to ilość sekund. Zauważ że wielkość liter w przypadku przełączników PgBench ma znaczenie. Małe t oznacza ilość transakcji, duże T oznacza czas wyrażony w sekundach.

Czas wykonywania testów

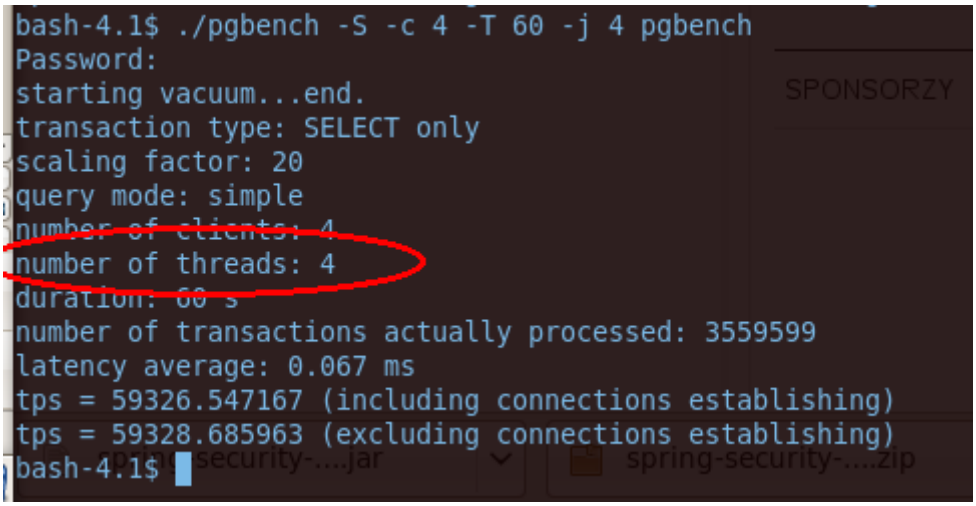
./pgbench -S -c 4 -T 60 pgbench



Ilość wątków

W pierwszym przykładowym teście został zastosowany jedynie jeden wątek. Możemy zmusić go do zastosowania większej liczby wątków z użyciem przełącznika -j:

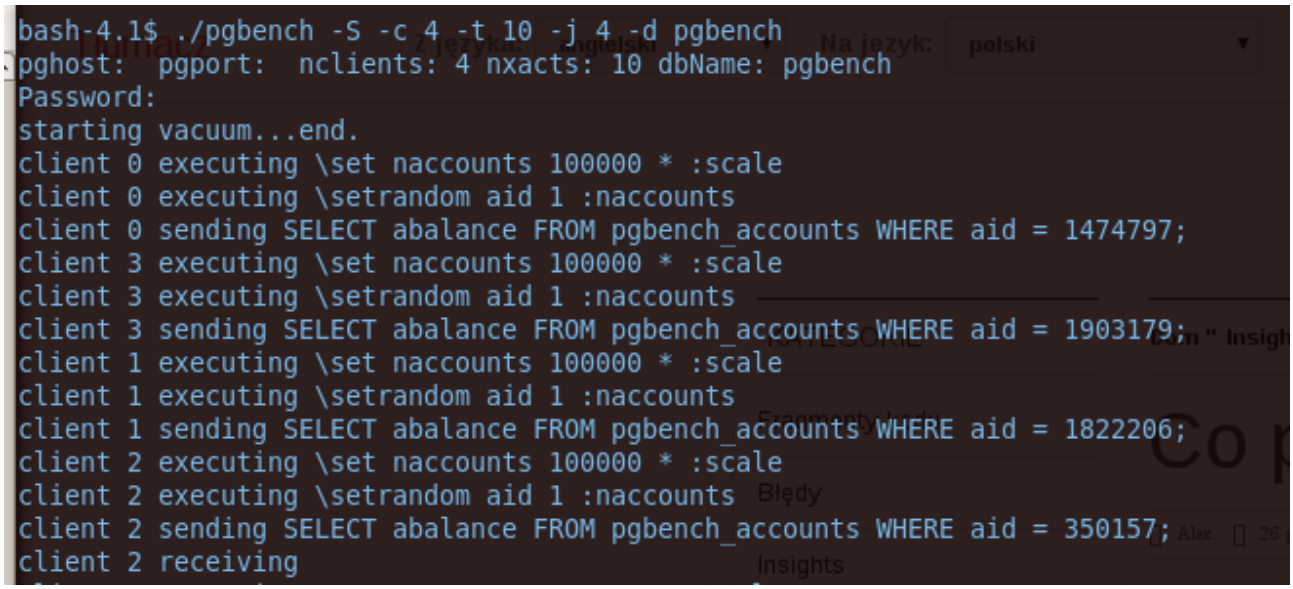
./pgbench -S -c 4 -T 60 -j 4 pgbench



Tryb debug

Jeśli ciekawi co tam się dzieje „w środku” możesz posłużyć się przełącznikiem -d który uruchomi tryb debug i pokaże zapytania wykonywane w ramach testów:

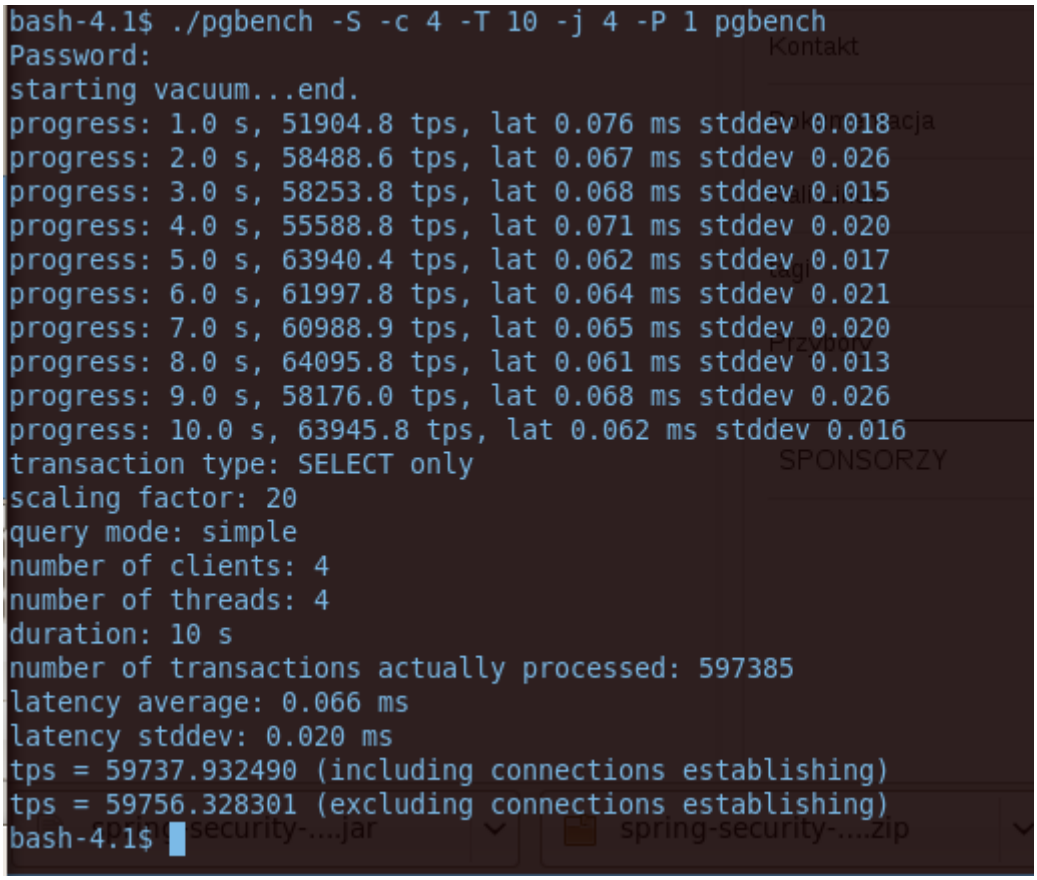
./pgbench -S -c 4 -t 10 -j 4 -d pgbench



Obserwacja postępów procesu testowania

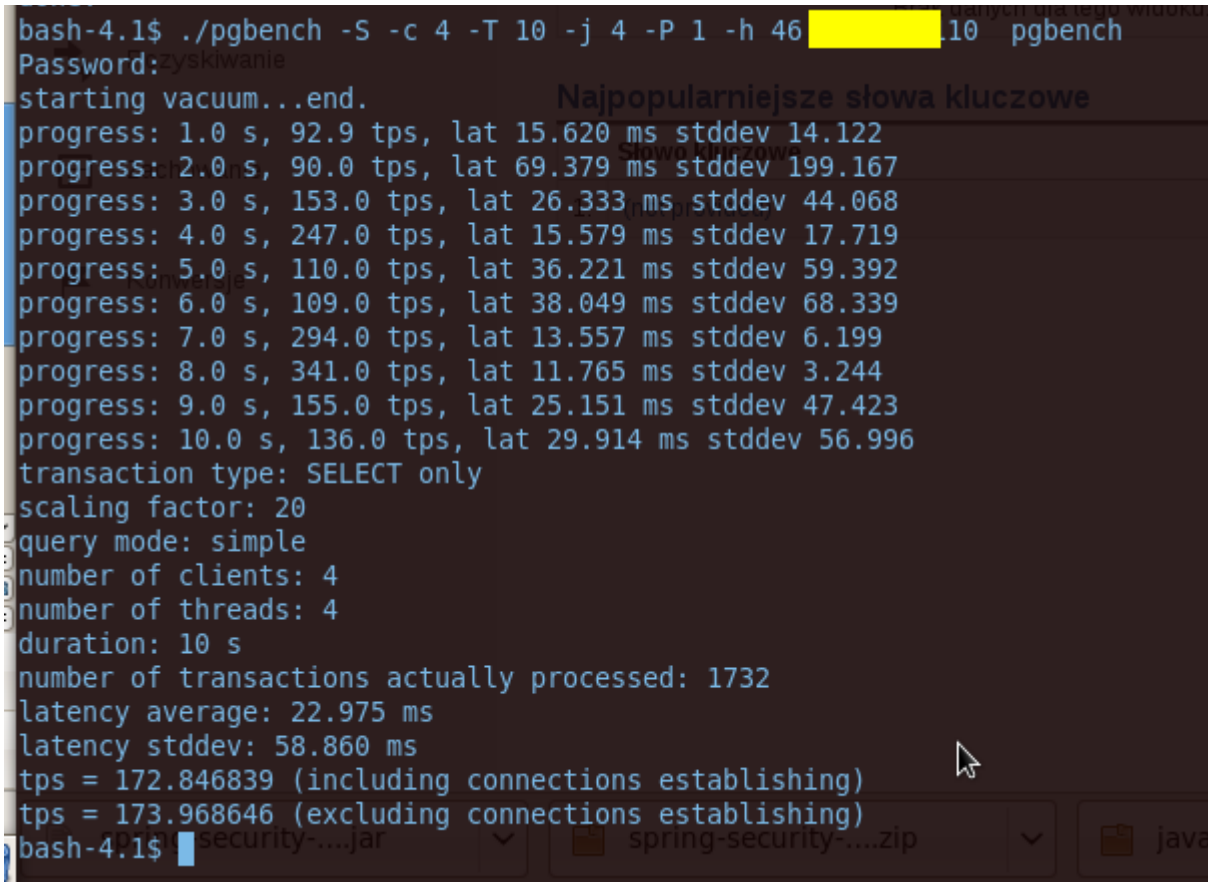
Inny użyteczny przełącznik -P pokaże nam postęp przetwarzania testów co X sekund podane jako parametr tego przełącznika:

./pgbench -S -c 4 -T 10 -j 4 -P 1 pgbench



Testy na zdalnym hoście

Nic nie stoi na przeszkodzie byśmy przeprowadzali testy na zdalnym hoście. Można oczywiście zawsze podpiąć się przez SSH, jednak nie zawsze jest taka możliwość (np. gdy jest to serwer postawiony na Windows (fuu)). Jak się nietrudno domyślić, baza przykładowa musi być też zainicjalizowana ;) Pamiętaj że taki test przeprowadzany zdalnie przez sieć będzie zdecydowanie wolniejszy niż gdybyś podłączył się poprzez SSH i wykonał testy lokalnie.



Uwagi

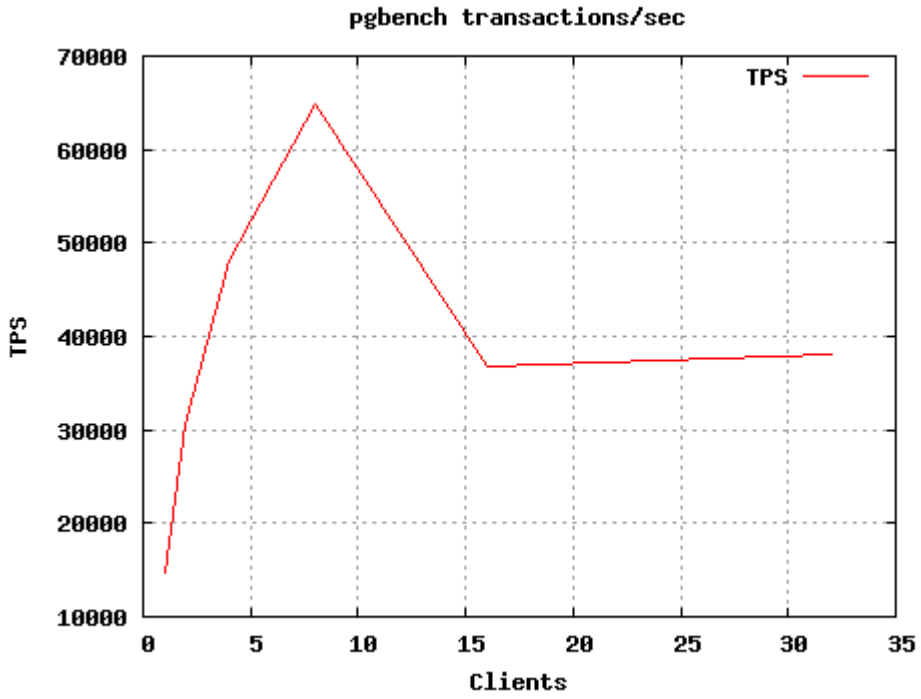
We wszystkich przykładach powyżej używałem testów trwających góra minutę. W realnym środowisku aby testy były bardziej miarodajne lepiej jest używać dłuższych czasów – np. 20-30 minut. Aby się o tym przekonać zrób kilkukrotnie testy trwające 10 sekund a następnie analogicznie 10 minut. Porównaj zróżnicowanie wyników w zależności od czasu trwania testów. Te prowadzone w krótszym czasie będą znacznie bardziej „rozstrzelone”.

Wszędzie powyżej używałem też przełącznika -S który sprawiał, że testy były oparte wyłącznie na instrukcjach SELECT. Jeśli go nie zastosujesz będą również używane UPDATE,INSERT i DELETE. Nastaw się jednak na to, że powstanie spora liczba zarchiwizowanych plików WAL jeśli Twój serwer działa w trybie archiwizacji ciągłej.

PgBench-tools – automatyczne narzędzie testujące

Wdrożenie

Wykorzystując samego PgBench można robić ręcznie wiele testów, ale gdybyśmy zechcieli zrobić kompleksowe testy np. sprawdzające kiedy ilość transakcji na sekundę spada poniżej określonego limitu, byłoby to bardzo pracochłonne. Trzeba byłoby zrobić zestawienie wyników dla różnej wielkości bazy, różnej ilości wątków i klientów i wielokrotnie całość powtarzać. Na szczęście jest narzędzie które zrobi to za nas, a nawet przedstawi wyniki w formie graficznych wykresów. Przykładowy zrzut z wyników:



Jeśli zechcemy skorzystać z tego narzędzia, musimy oczywiście posiadać pakiet pgbench i musi on być zainstalowany w bazie. Następnie tworzymy bazę danych w której mają zostać utworzone tabele na potrzeby testów (najlepiej by baza ta była pusta), oraz bazę która będzie przechowywała wyniki testów:

create database pgbench;

create database results;

Teraz musimy zainstalować narzędzie GIT które posłuży nam do pobrania pgbench-tools:

yum install git

Przechodzimy do katalogu w który będziemy chcieli mieć zainstalowane narzędzia dla pgbench. Zadbajmy też o odpowiednie uprawnienia. Wydajemy komendę:

git clone git://git.postgresql.org/git/pgbench-tools.git

PgBench-tools wykorzystuje pakiet „gnuplot” służący do generowania wykresów, a bez którego nie zobaczymy wyników (jakoś producentom zapomniało się dodać o tym informacji w dokumentacji i w pliki readme). Zainstalujmy go więc:

yum install gnuplot

Przechodzimy teraz do katalogu rozpakowanego pakietu:

cd pgbench-tools/

Bazę przechowującą wyniki musimy zainicjalizować:

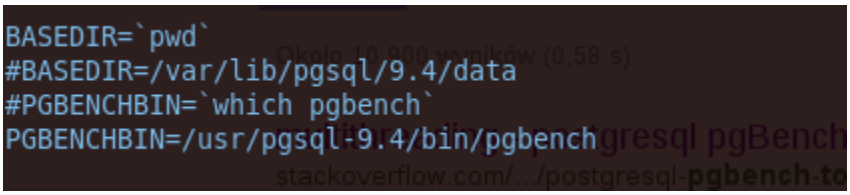
psql -f init/resultdb.sql -d results

Konfiguracja i uruchamianie testów

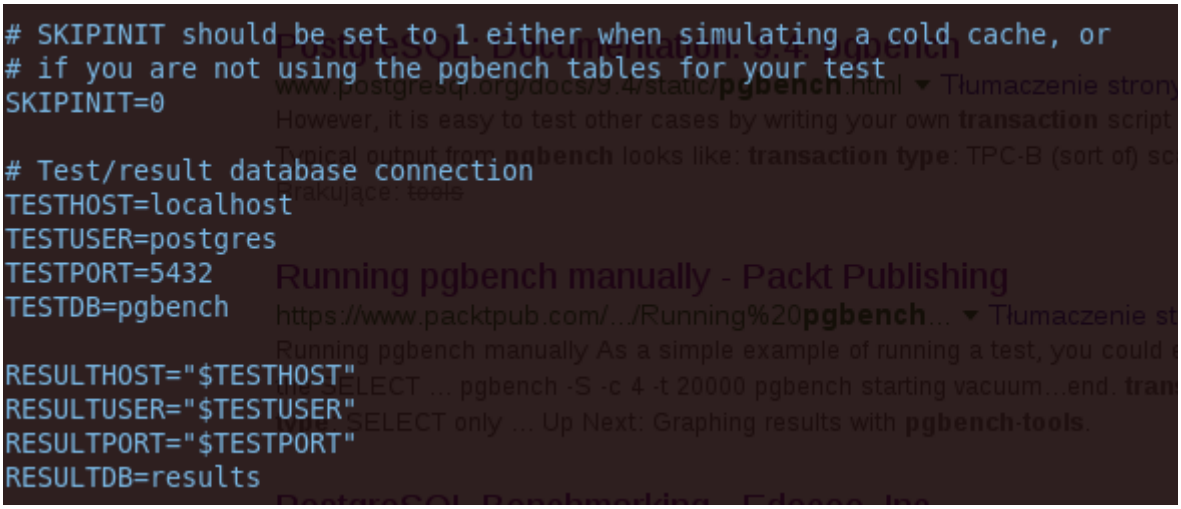
Od uruchomienia testów dzieli nas już tylko konfiguracja w pliku config znajdującym się w katalogu pgbench-tools. Przechodzimy do edycji:

nano config

Kilka rzeczy trzeba ustawić aby to narzędzie w ogóle działało i od tego zaczniemy. W pliku config znajdziemy parametry BASEDIR I PGBENCHBIN. Pierwszy odnosi się do katalogu w którym mają zostać wygenerowane wyniki, drugi powinien wskazywać ścieżkę do narzędzia pgbench (powinien być tam gdzie wszystkie binaria PostgreSQL).



Teraz ustawienia opcjonalne które znajdziemy w dalszej części pliku:



Możemy tutaj zmienić hosta na którym znajduje się baza testowa i baza na wyniki, a także ustawić port na którym nasłuchuje PostgreSQL. Od razu uprzedzam – bardzo nie polecam puszczania testów przez sieć. Odbywają się one wtedy okropnie wolno.

Kolejna część parametrów:

```
SCRIPT="select.sql"
#SCALES="1 10 100 1000"
SCALES="1"
#SETCLIENTS="1 2 4 8 16 32"
SETCLIENTS="1"
SETTIMES=3
```

Przyjrzyjmy się parametrom SCRIPT,SCALES,SETCLIENTS i SETTIMES. Pierwszy parametr SCRIPT określa skrypt używany do testów. Jest ich kilka i wykonują różne operacje. Możesz nawet stworzyć własne. Wszystkie one znajdują się w podkatalogu tests. Poniżej wyświetlam ich listę oraz zawartość tego ustawionego domyślnie.

```
bash-4.1$ ls tests
insert-size.sql  insert.sql  nobranch.sql  select.sql  tpc-b.sql  update.sql
bash-4.1$ cat tests/select.sql
\set naccounts 100000 * :scale
\setrandom aid 1 :naccounts
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

Drugi parametr SCALES określa skalę – dla jakiej wielkości bazy mają odbywać się testy (to determinuje wielkość tabel tworzonych przez pgbencha – zależności są tu takie same jak w przełączniku -s samego pgbench). Parametr SETCLIENTS określa ilość klientów z których mają być przeprowadzane testy. W domyślnym ustawieniu najpierw testy przebiegają z użyciem jednego klienta, następnie dwóch, czterech itd. Parametr SETTIMES określa ile razy mają być przeprowadzane testy.

Na potrzeby testów do tej publikacji ustawiam skalę na 1 i 10, 1 2 i 4 klientów oraz 3-krotne uruchomienie testów.

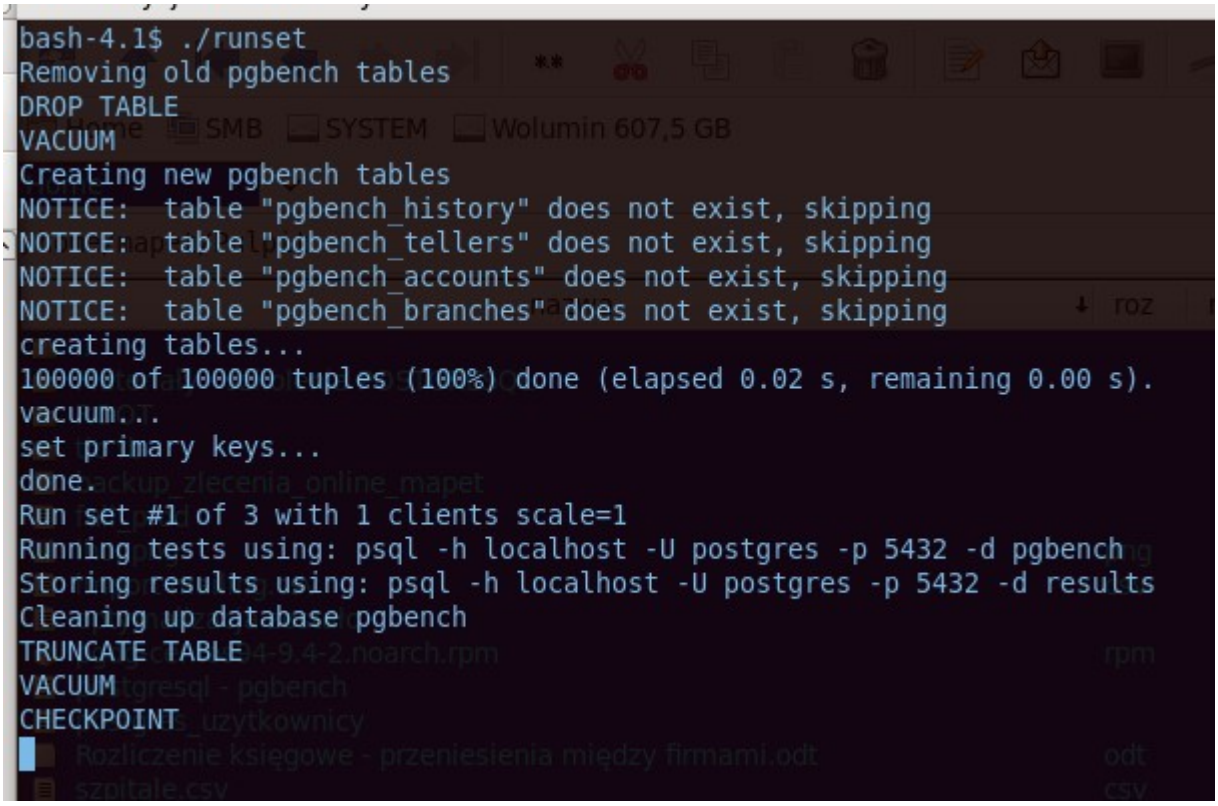
Zanim uruchomimy testy, upewnij się że masz wystarczającą ilość miejsca na dysku na ewentualnie powstające zarchiwizowane pliki WAL! Jeśli tego miejsca zabraknie, testy zostaną przerwane (sprawdzone na własnej skórze po kilku godzinach mielenia testów).

Aby rozpocząć testy uruchamiamy skrypt runset:

./runset

Po uruchomieniu skrypt zacznie tworzyć potrzebne mu tabele. Ewentualnymi ostrzeżeniami o braku jakiejś tabeli się po prostu nie przejmujemy. Przyjrzyj się linijce zaczynającej się od „Run set”. Informuje nas ona który jest to test. W związku z moimi ustawieniami przeprowadzone będą TRZYKROTNIE następujące testy:

Skala	**Ilość klientów**
1	1
1	2
1	4
10	1
10	2
10	4



W sumie przeprowadzone będzie 18 testów. W linii „Run set” widzimy które to uruchomienie, z iloma klientami i dla jakiej skali. Same testy mogą trochę potrwać, a im więcej dałeś skal i klientów tym dłużej. Podczas testów wywoływane są checkpointy, więc będą nam się archiwizowały pliki WAL. Monitoruj przestrzeń w której są składowane, aby nie została zapełniona w 100%.

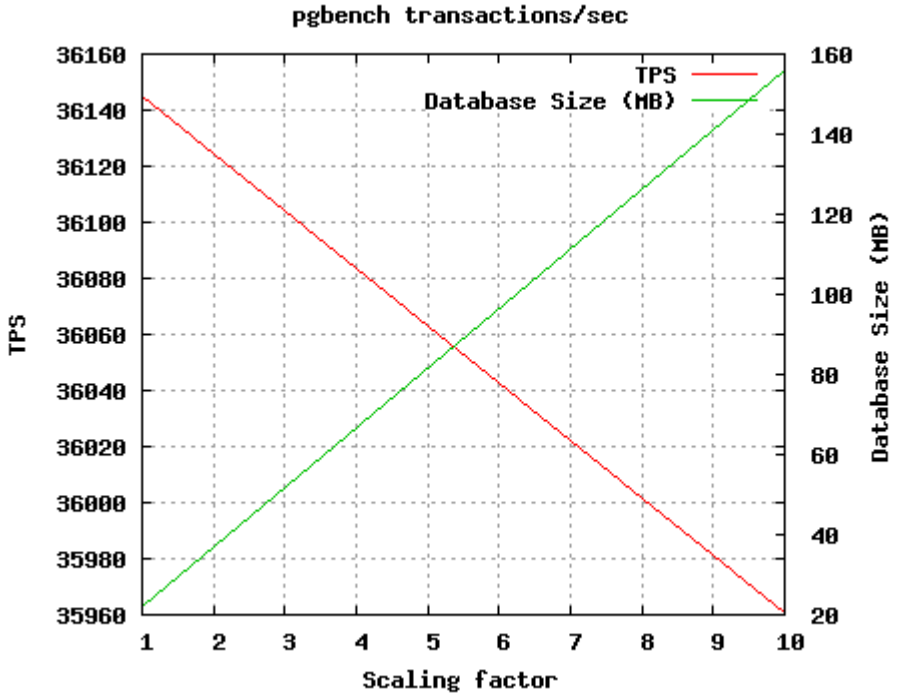
Takie testy są baaaardzo obciążające dla bazy. Nie wykonuj ich więc na środowisku na której aktualnie pracują użytkownicy!

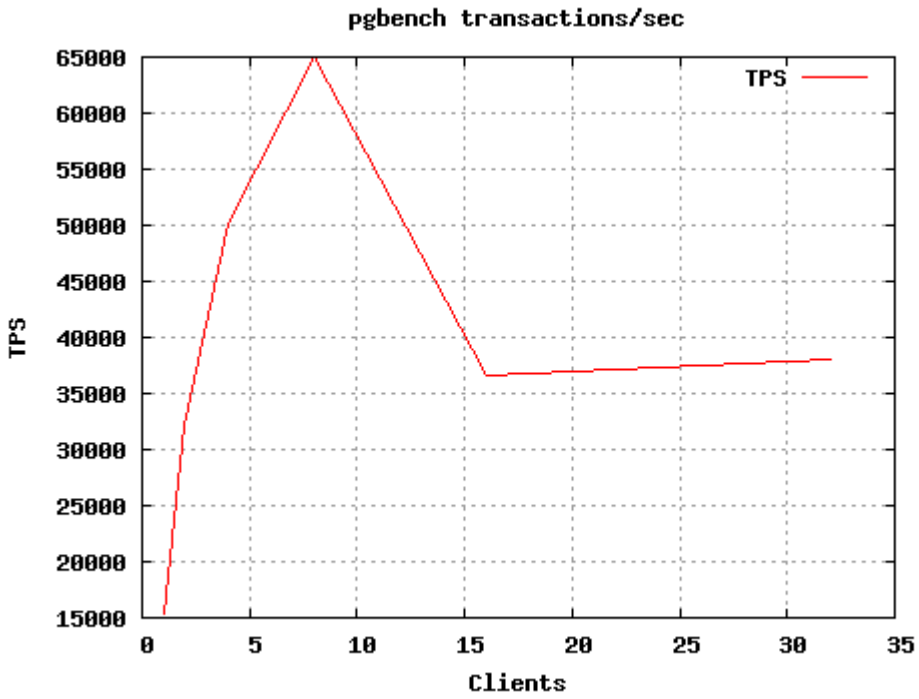
Przeglądanie wyników testów i ich analiza

Kiedy testy dobiegną końca, wchodzimy w podkatalog results i uruchamiamy znajdujący się w nim plik index.htm Powinniśmy tam zobaczyć wyniki testów. Jeśli nie zainstalowałeś wspomnianego wcześniej pakietu gnuplot, nie zobaczysz takich eleganckich wykresów jak poniżej. Pierwszy wykres obrazuje stosunek wielkości bazy danych do ilości przeprowadzanych transakcji na sekundę. Zasadniczo wynik był do przewidzenia, im większa baza tym mniej traksakcji jest się w stanie odbyć w ciągu sekundy. Jeśli przypomnimy sobie zawartość testowego pliku select.sql, zauważymy że są to zapytania wyciągające po jednym wierszu. Do przewidzenia było więc że wydajność będzie spadała liniowo w stosunku do ilości wierszy w tabelach. Za chwilę zobaczymy jak to wygląda dla operacji DML, tutaj mogą być nieco bardziej zaskakujące wyniki, a także stworzymy własny plik z instrukcjami dla testów.

```
bash-4.1$ ls tests
insert-size.sql  insert.sql  nobranch.sql  select.sql  tpc-b.sql  update.sql
bash-4.1$ cat tests/select.sql
\set naccounts 100000 * :scale
\setrandom aid 1 :naccounts
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

Przyjrzyj się za to drugiemu wykresowi. Mamy tutaj przedstawioną wydajność wyrażoną w liczbie transakcji na sekundę w stosunku do ilości klientów. Przyjrzyj się najwyższemu punktowi wykresu. Przy jakiej ilości klientów osiągnięto najwyższą wydajność? Przy 8! Tak się składa że na komputerze na którym uruchamiałem te testy jest akurat 8 rdzeni :) To jest moment maksymalnego wykorzystania ich wydajności. Potem gdy klientów jest więcej wydajność zaczyna spadać a wynika to z konieczności przełączania się procesora między wątkami.


pgbench transactions/sec
TPS
Database Size (MB)
TPS
Database Size (MB)
Scaling factor
36160
36140
36120
36100
36080
36060
36040
36020
36000
35980
35960
160
140
120
100
80
60
40
20
1
2
3
4
5
6
7
8
9
10


pgbench transactions/sec
TPS
TPS
Clients
65000
60000
55000
50000
45000
40000
35000
30000
25000
20000
15000
0
5
10
15
20
25
30
35

Nieco niżej znajdziemy tabelkę z zestawionymi wynikami rozbitymi na poszczególne testy.

Detail for test set 1:

set	test	scale	clients	tps	max_latency	chkpts	buf_check	buf_clean	buf_backend	buf_alloc	max_clean	backend_sync
1	1	1	1	15119	0.731	0	0	0	0	256	0	0
1	7	1	1	14843	4.145	0	0	0	0	0	0	0
1	13	1	1	14982	2.93	0	0	0	0	0	0	0
1	37	1	1	14788	4.075	0	0	0	0	255	0	0
1	38	1	1	14928	10.573	0	0	0	0	0	0	0
1	39	1	1	14738	4.828	0	0	0	0	0	0	0
1	40	1	1	15485	2.277	0	0	0	0	253	0	0
1	43	1	1	16154	2.181	0	0	0	0	1	0	0
1	46	1	1	15429	4.069	0	0	0	0	0	0	0
1	2	1	2	33558	146.669	0	0	0	0	0	0	0
1	8	1	2	31305	48.395	0	0	0	0	0	0	0
1	14	1	2	31596	194.101	0	0	0	0	0	0	0
1	41	1	2	33561	294.237	0	0	0	0	0	0	0
1	44	1	2	34205	122.765	0	0	0	0	0	0	0

Myślę że jego interpretacja nie powinna sprawić problemu. Widzimy tutaj skale i ilość klientów których sami konfigurowaliśmy w pliku config. Obok liczba transakcji na sekundę przy danym teście, maksymalne opóźnienie etc. Możemy kliknąć na numer testu by wejść w jego szczegóły:

Indeks – /tmp/pgbench-tools/results/1/

Nazwa	Rozmiar	Data modyfikacji
[katalog główny]		
index.html	142 B	04.05.2016 16:18:37
pg_settings.txt	1,5 kB	04.05.2016 16:18:37
results.txt	313 B	04.05.2016 16:18:21

Pod linkiem pg_settings.txt zobaczymy aktualną konfigurację jaka obowiązywała podczas przeprowadzania danego testu.

```
Test results:
   script   | clients |  tps  | avg_latency | max_latency
------------+---------+-------+-------------+-------------
 select.sql |       1 | 15119 |           0 |           1
(1 wiersz)

Server mapet, client mapet

Server settings in postgresql.conf:
            name             |        current_setting
-----------------------------+------------------------------
 archive_command             | cp %p /home/mapet/wal_arch/%f
 archive_mode                | on
 default_text_search_config  | pg_catalog.simple
 dynamic_shared_memory_type  | posix
 log_destination             | stderr
 log_filename                | postgresql_all.log
 log_line_prefix             | < %m >
 log_timezone                | Poland
 logging_collector           | on
 max_connections             | 100
 shared_buffers              | 256MB
 TimeZone                    | Poland
 wal_level                   | archive
(13 wierszy)


benchmark client OS Configuration (may not be the same as the server)
Linux mapet 2.6.32-573.22.1.el6.x86_64 #1 SMP Wed Mar 23 03:35:39 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
/proc/sys/vm/dirty_background_bytes=0
/proc/sys/vm/dirty_background_ratio=10
/proc/sys/vm/dirty_bytes=0
/proc/sys/vm/dirty_expire_centisecs=3000
/proc/sys/vm/dirty_ratio=20
/proc/sys/vm/dirty_writeback_centisecs=500

/etc/lsb-release:
LSB_VERSION=base-4.0-amd64:base-4.0-noarch:core-4.0-amd64:core-4.0-noarch:graphics-4.0-amd64:graphics-4.0-

/etc/redhat-release:
CentOS release 6.7 (Final)
```

Pod linkiem result.txt zobaczymy szczegóły danego testu.

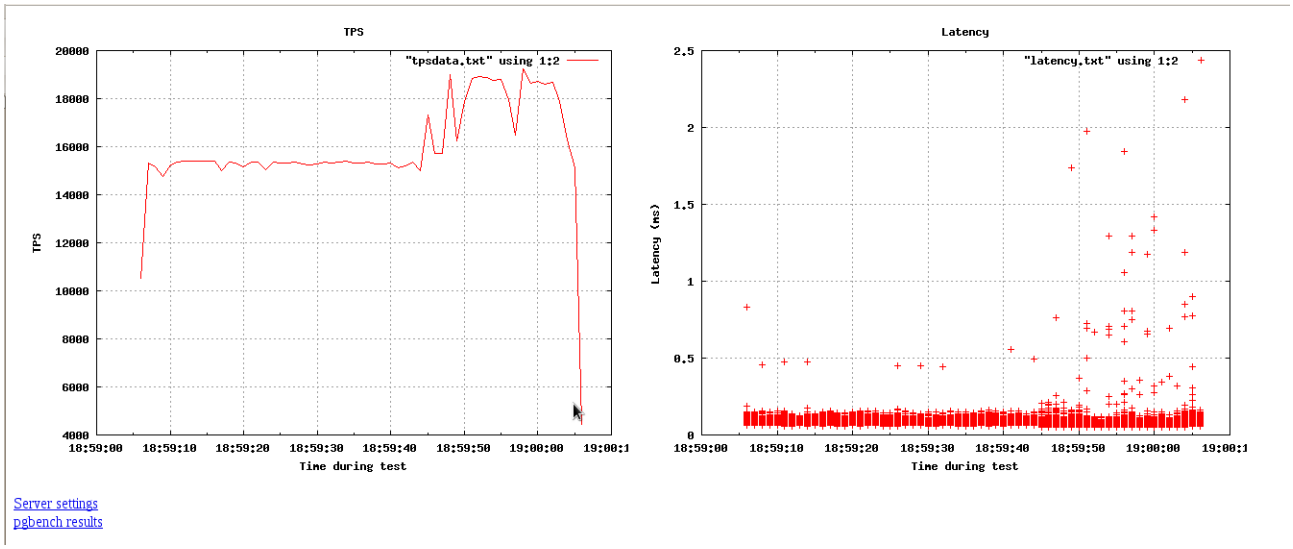
```
transaction type: Custom query
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
duration: 60 s
number of transactions actually processed: 907121
latency average: 0.066 ms
tps = 15118.644025 (including connections establishing)
tps = 15118.967318 (excluding connections establishing)
```

W niektórych przypadkach po kliknięciu na numer testu zobaczymy nieco więcej rzeczy. Są wygenerowane pliki graficzne. Nie we wszystkich testach mi się to wygenerowało, niby nie dostałem żadnego błędu, ale musiał zaistnieć jakiś problem dla którego nie utworzyły się te pliki. Powinny być wszędzie. Te pliki graficzne to obrazy dla danego testu.

Indeks – /home/mapet/Pulpit/pgbench-tools/results/47/

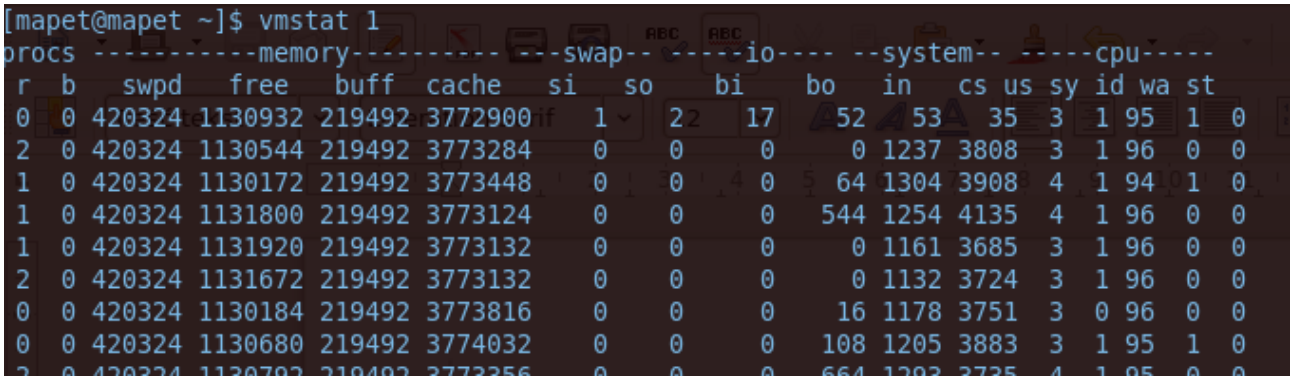
Nazwa	Rozmiar	Data modyfikacji
[katalog główny]		
index.html	142 B	04.05.2016 19:08:27
latency.png	4,2 kB	04.05.2016 19:08:23
pg_settings.txt	1,5 kB	04.05.2016 19:08:28
results.txt	314 B	04.05.2016 19:07:49
tps.png	5,3 kB	04.05.2016 19:08:18

Kiedy klikniemy na zawarty w tym samym katalogu plik index.html powinniśmy zobaczyć jeszcze owe wykresy:



Narzędzia systemu Linux

Vmstat



```
[mapet@mapet ~]$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0 420324 1130932 219492 3772900    1    2    17    52   53   35  3  1 95  1  0
 2  0 420324 1130544 219492 3773284    0    0     0     0 1237 3808  3  1 96  0  0
 1  0 420324 1130172 219492 3773448    0    0     0    64 1304 3908  4  1 94  1  0
 1  0 420324 1131800 219492 3773124    0    0     0   544 1254 4135  4  1 96  0  0
 1  0 420324 1131920 219492 3773132    0    0     0     0 1161 3685  3  1 96  0  0
 2  0 420324 1131672 219492 3773132    0    0     0     0 1132 3724  3  1 96  0  0
 0  0 420324 1130184 219492 3773816    0    0     0    16 1178 3751  3  0 96  0  0
 0  0 420324 1130680 219492 3774032    0    0     0   108 1205 3883  3  1 95  1  0
```

VmStat to narzędzie pozwalające podglądać co się w danym momencie dzieje w systemie. Zrzuca co wskazany w sekundach czas migawki zawierające informacje o aktualnym obciążeniu systemu. Narzędzie wywołujemy z parametrem określającym w sekundach co jaki czas ma być wykonywany zrzut statystyk.

Wyjaśnienie kolumn w wyniku:

r: liczba procesów w kolejce oczekujących na uruchomienie.

b: liczba procesów w stanie uśpienia

swpd: użyta pamięć wirtualna

free: wolna pamięć

buff: zużycie pamięci na potrzeby buforów

cache: pamięć użyta o buforowania

si: ilość bloków (1kB) pamięci odczytana z pamięci wirtualnej

so: ilość bloków (1kB) pamięci zapisana w pamięci wirtualnej

bi: ilość bloków (1kB) odczytanych z dysku

bo: ilość bloków (1kB) zapisanych na dysku

in: liczba przerwań procesora

cs: ilość przełączeń kontekstu

us: czas procka wykorzystany na programy inne niż elementy systemu operacyjnego.

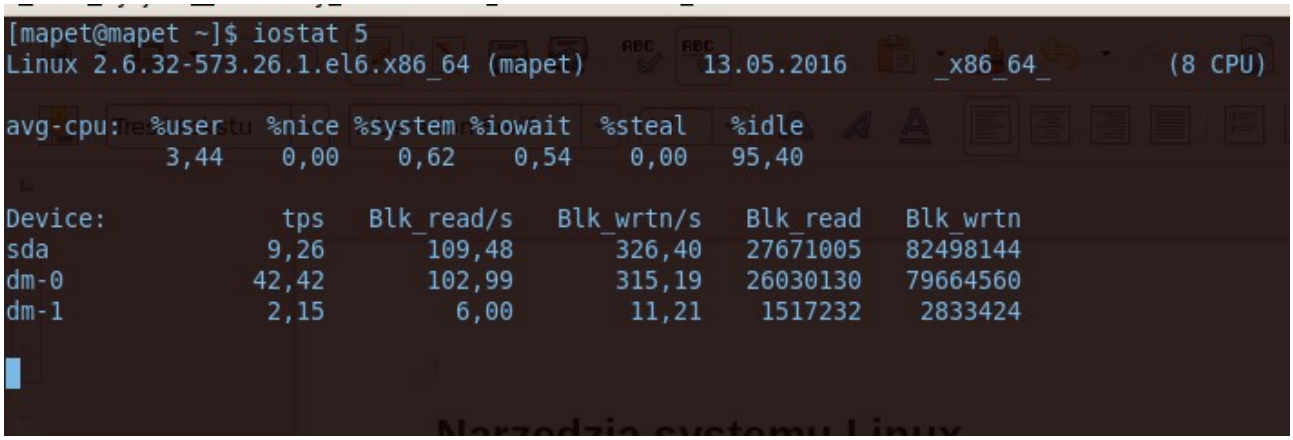
sy: czas procka wykorzystany na działalność związaną z systemem operacyjnym

id: Czas bezczynności procesora

wa: tak zwane „waits” czyli oczekiwanie na urządzenia typu dysk twardy

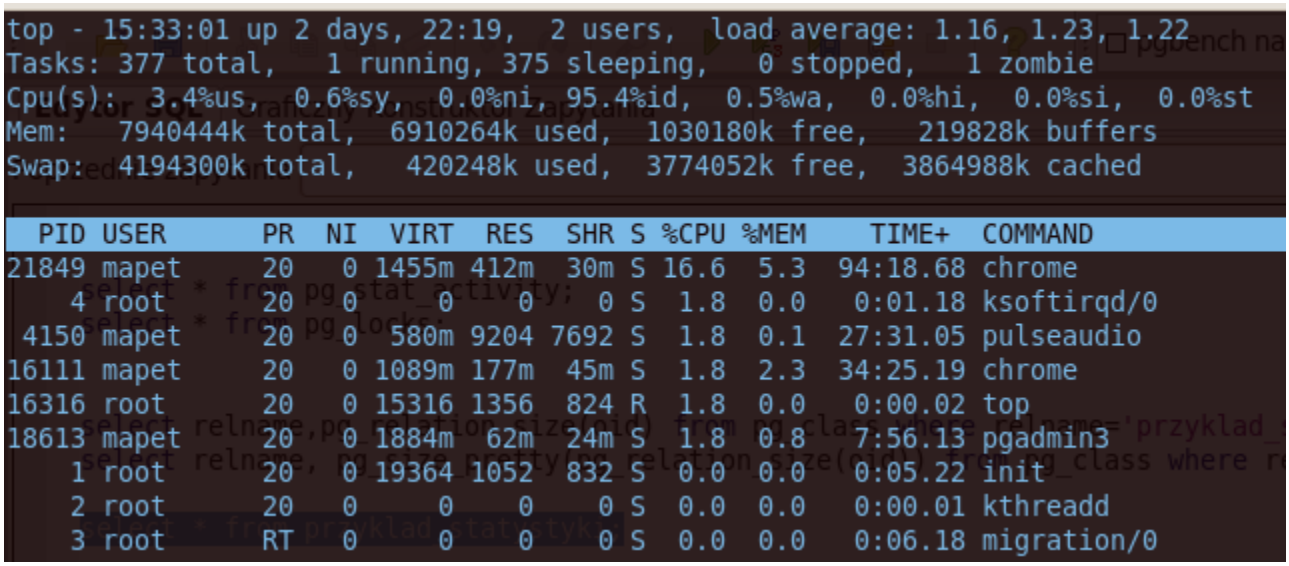
Iostat

Program umożliwia sprawdzenie statystyk użycia poszczególnych dysków. Wyniki które zobaczymy to dane od ostatniego restartu. Dzięki iostat możemy sprawdzić ile danych zostało odczytanych i zapisanych na poszczególnych dyskach, od uruchomienia systemu, a także jaka była średnia prędkość zapisu i odczytu na nich w tym czasie. Parametr 5 podany na końcu sprawia że wyniki będą się pojawiały co 5 sekund.

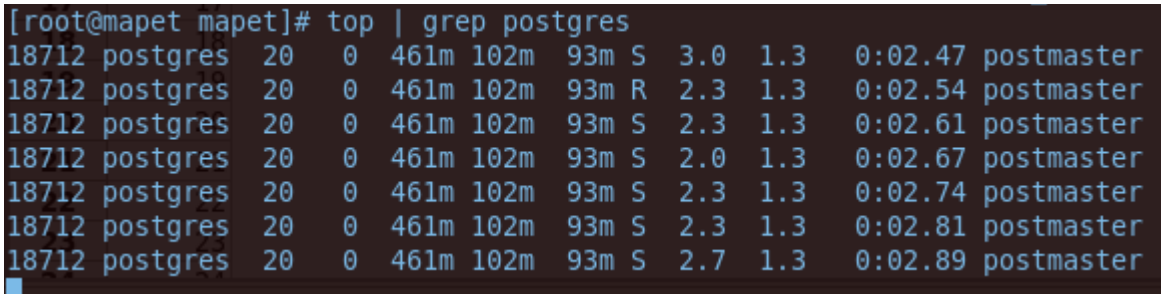


TOP

Top to narzędzie służące do sprawdzenia najbardziej obciążających system procesów. Możemy dzięki niemu podejrzeć co najbardziej zużywa nam pamięć operacyjną czy procesor.

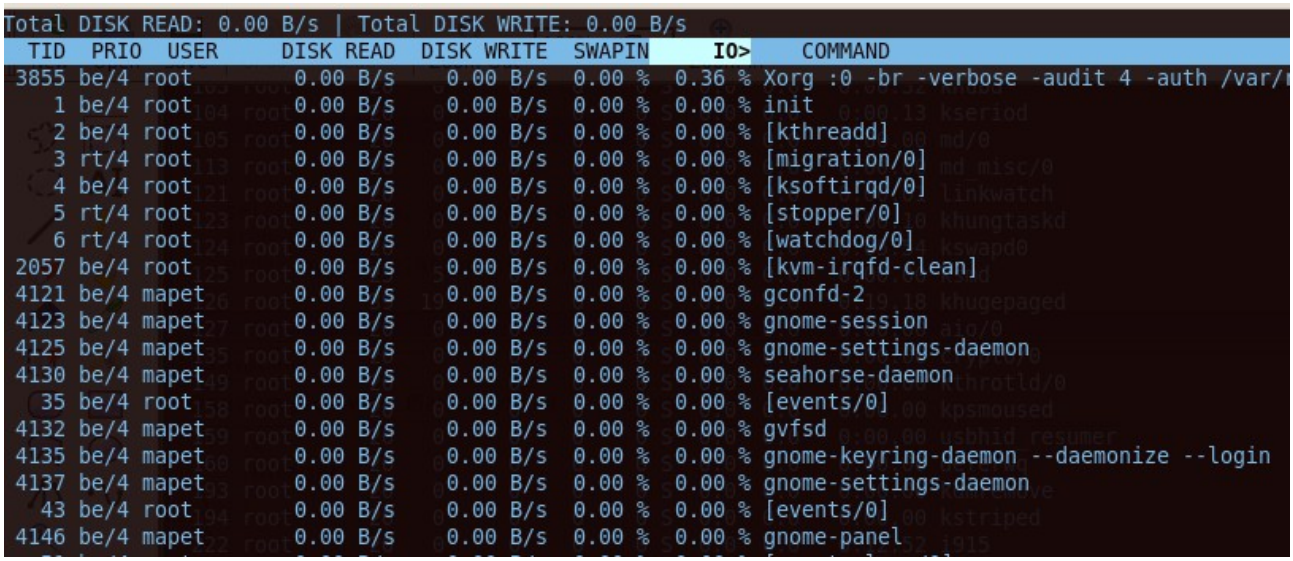


Zasadniczo aby go uruchomić wystarczy w konsoli wywołać „top”, ale jeśli interesują nas konkretnie procesy PostgreSQL możemy wywołać „ top | grep postgres ”:

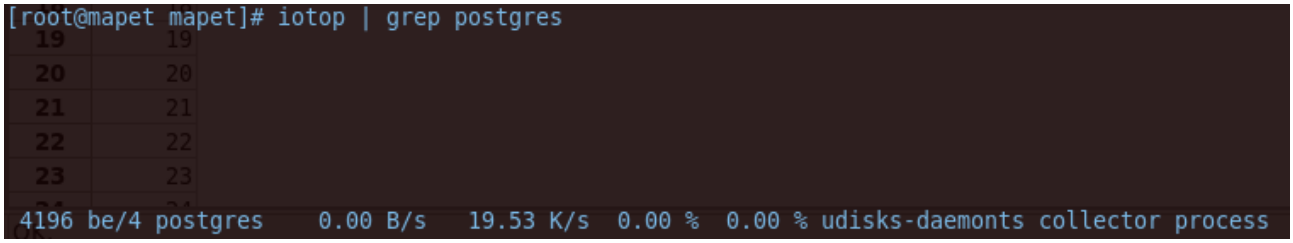


IOTOP

Narzędzie zbliżone do TOP, z tą różnicą że prezentowane obciążenia dotyczą procesów generujących I/O.

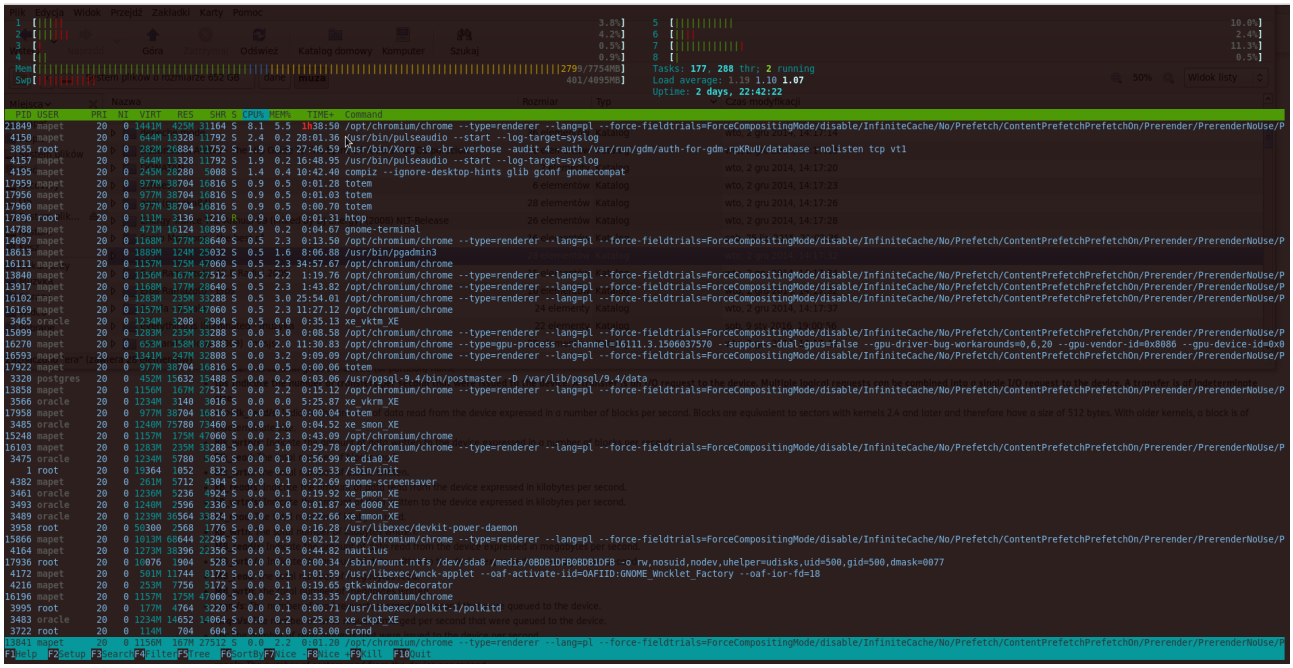


Tutaj również możemy użyć grep aby odfiltrować zdarzenia związane z PostgreSQL:



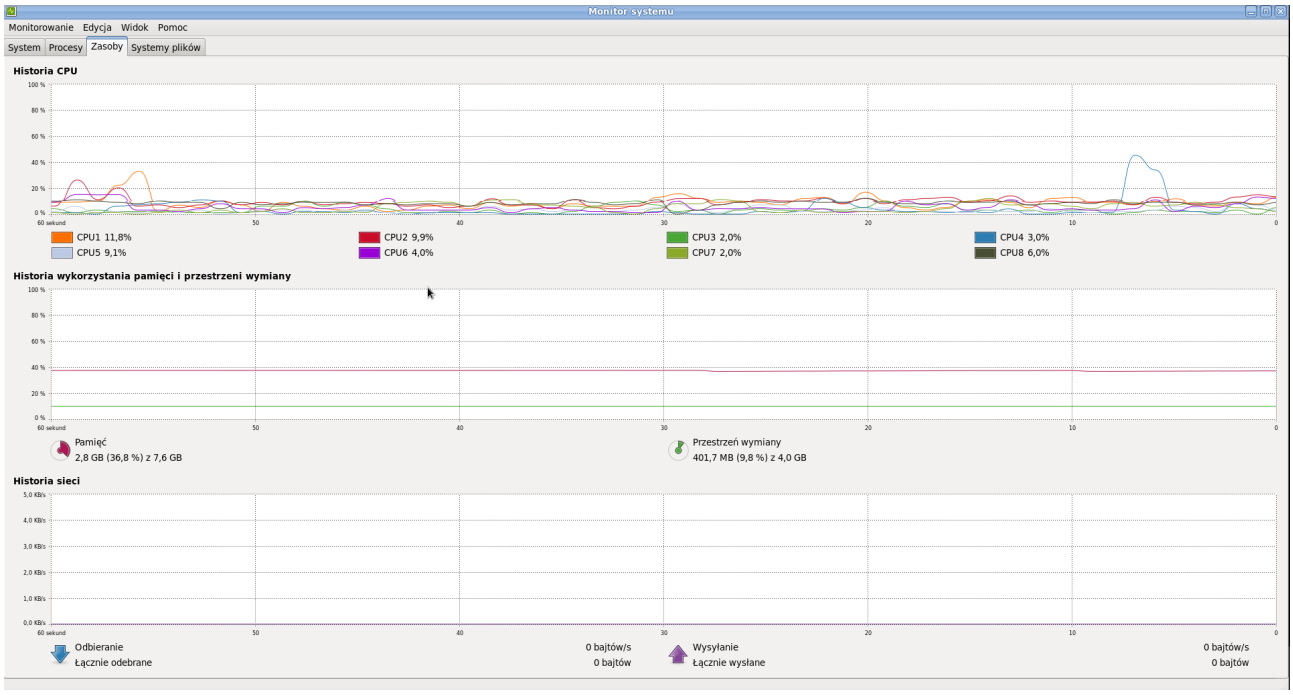
HTOP

Nieco przyjemniejsze dla oka niż TOP narzędzie do badania aktualnego obciążenia. Taka „lepsza” wersja TOPa. – uwzględnia np. zużycie poszczególnych procesorów.



GNOME SYSTEM MONITOR

Bardzo wygodne narzędzie przedstawiające graficznie aktualne wykorzystanie procesorów, przestrzeni wymiany i sieci.



SAR

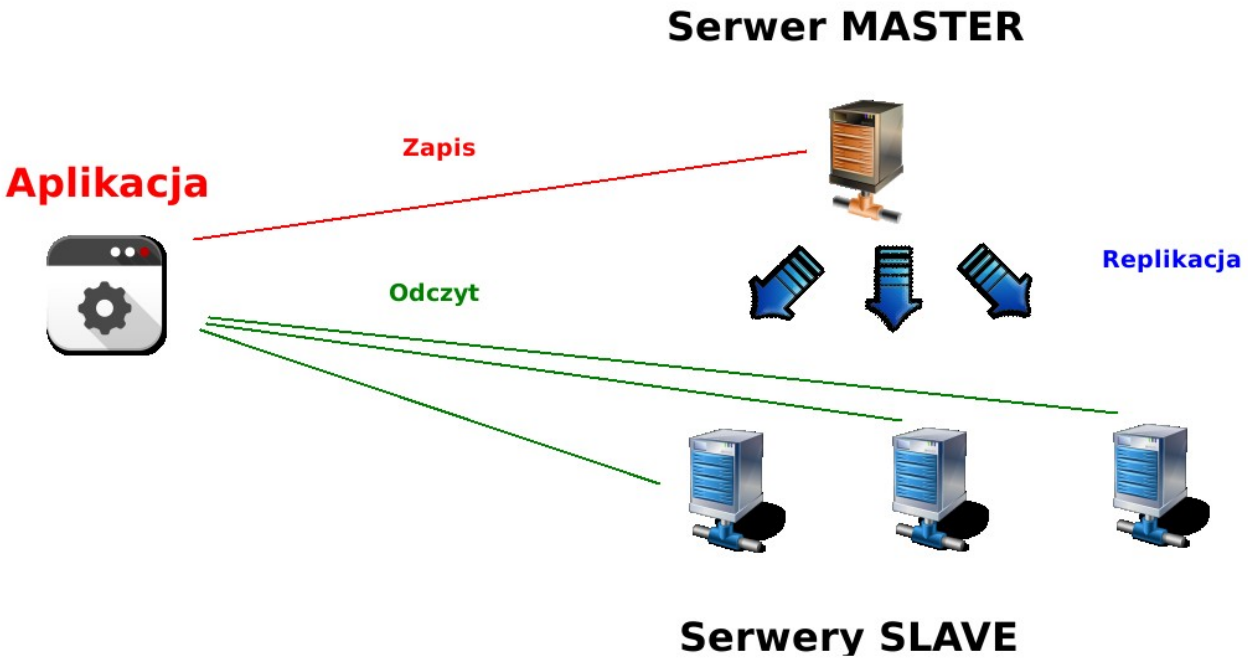
To jest narzędzie-nakładka dla sysstat. Należy więc zainstalować obydwa. Sysstat zbiera dane o aktualnym obciążeniu systemu, a sar je prezentuje. Jak widać na poniższej ilustracji, SAR prezentuje zrzuty obciążenia systemu co 10 minut. Dzięki niemu możemy podejrzeć kiedy w ciągu dnia system ma najmniej pracy, gdybyśmy zechcieli zaplanować jakieś obciążające operacje związane z utrzymaniem.

```
[root@mapet mapet]# sar
Linux 2.6.32-573.26.1.el6.x86_64 (mapet)        13.05.2016      _x86_64_        (8 CPU)

00:00:01        CPU     %user     %nice   %system   %iowait    %steal     %idle
00:10:01        all      0,45      0,00      0,09      0,31      0,00     99,15
00:20:01        all      0,43      0,00      0,09      0,35      0,00     99,13
00:30:01        all      0,43      0,00      0,09      0,34      0,00     99,14
00:40:01        all      0,42      0,00      0,09      0,35      0,00     99,15
00:50:01        all      0,47      0,00      0,10      0,34      0,00     99,10
01:00:01        all      0,42      0,00      0,09      0,34      0,00     99,14
01:10:01        all      0,44      0,00      0,10      0,34      0,00     99,13
01:20:01        all      0,49      0,00      0,10      0,36      0,00     99,05
01:30:01        all      0,44      0,00      0,10      0,35      0,00     99,12
01:40:01        all      0,46      0,00      0,10      0,35      0,00     99,09
01:50:01        all      0,46      0,00      0,10      0,33      0,00     99,11
02:00:01        all      0,40      0,00      0,09      0,35      0,00     99,15
02:10:01        all      0,41      0,00      0,09      0,33      0,00     99,16
02:20:01        all      0,50      0,00      0,11      0,48      0,00     98,92
02:30:01        all      0,43      0,00      0,09      0,34      0,00     99,14
02:40:01        all      0,43      0,00      0,09      0,34      0,00     99,13
```

Replikacja

Skalowanie z użyciem replikacji

Czy baza danych np. Google mogłaby być oparta na jednej bazie danych – jednym serwerze? Wątpliwe :) Jeśli zbiór danych musi być jeden a potrzebujemy mocy obliczeniowej wielu serwerów możemy do skalowania tego typu wykorzystać dostępną w PostgreSQL replikację. System który za chwilę skonfigurujemy będzie działał w taki sposób, że będzie jeden serwer Master z którego baza będzie replikowana na kilka innych serwerów. W zależności od sposobu konfiguracji replikacji (a może być synchroniczna lub asynchroniczna) możemy mieć dokładne kopie bazy na kilku serwerach. Serwer master będzie działał w trybie zapis-odczyt, a serwery slave w trybie tylko do odczytu. Aplikacja korzystająca z takie systemu wszelkie zmiany danych będzie przeprowadzać na serwerze master, ale odczyt danych będzie rozłożony na kilka serwerów (w końcu będą miały te same dane). Dzięki czemu rozłożone zostanie również obciążenie wynikające z odpytywania baz. Z takiej konfiguracji wynika jeszcze jedna dodatkowa korzyść. Mianowicie jeśli serwer Master ulegnie awarii, któryś z serwerów Slave może go zastąpić. Tryb Hot Standby a więc taki rodzaj konfiguracji jaki za chwilę przeprowadzimy jest dostępny od wersji 8.2. Możliwość działania bazy replikującej w trybie tylko do odczytu wprowadzono w wersji 9.0.



Konfiguracja serwera MASTER

Zaczniemy od konfiguracji serwera MASTER – a więc tego który ma być replikowany.

Z informacji wstępnych dla wyjaśnienia – serwer MASTER ma u mnie adres IP = 192.168.0.101, serwer SLAVE – 192.168.0.100. Replikacja została wykonana na serwerach z CentOS 6.7 i PostgreSQL w wersji 9.4.

Opisany tutaj sposób replikacji to replikacja strumieniowa asynchroniczna. Oznacza to, że powtórzenie zmiany na serwerze SLAVE nie następuje od razu (choć zazwyczaj dość szybko, zależy to od aktualnego obciążenia serwerów i sieci).

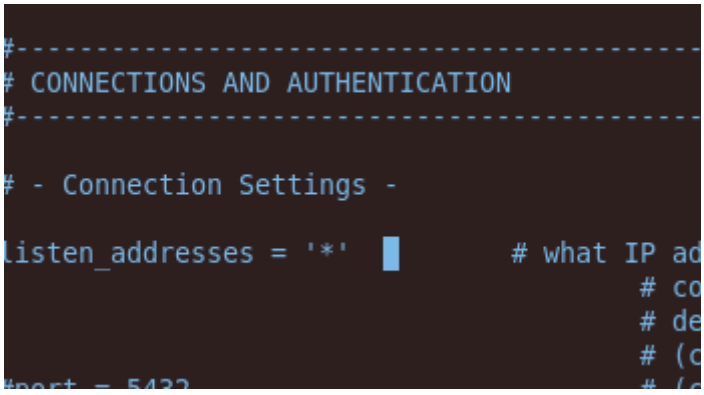
Przejdź do systemowego użytkownika „postgres” i z poziomu psql utwórz użytkownika z poziomu którego przeprowadzana będzie replikacja:

create user rep replication;

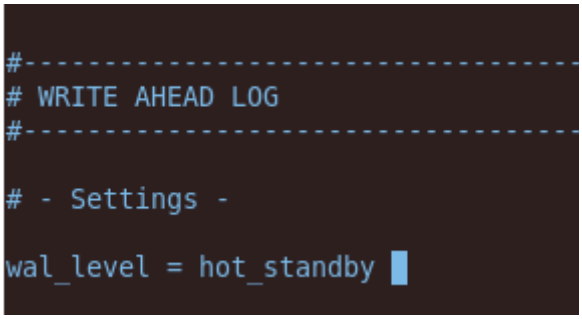
Następnie przejdź do edycji pliku postgresql.conf:

nano /var/lib/pgsql/9.4/data/postgresql.conf

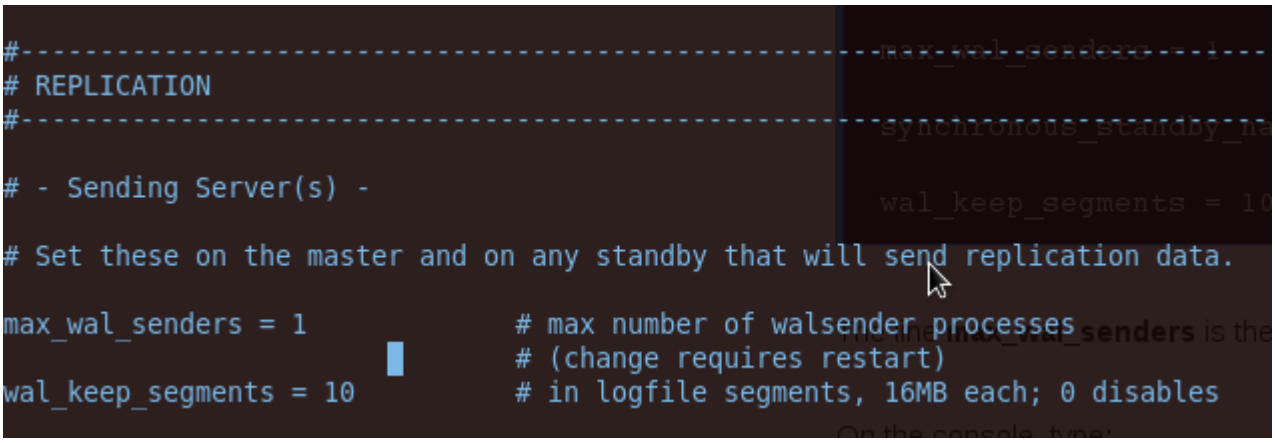
Musimy zmienić kilka parametrów. Zacznijmy od listen_address – aby serwer SLAVE mógł dokonywać replikacji będziemy potrzebować możliwości dobicia się do niego przez sieć, tak więc ustaw adresy które mają być akceptowane, bądź * . Pamiętaj by odremowywać wszystkie parametry!



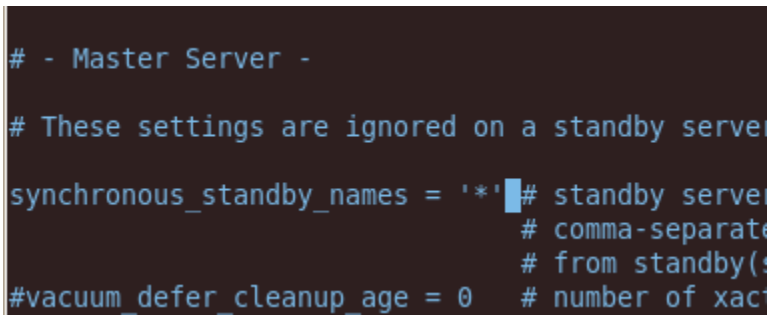
Kolejna rzecz to parametr wal_level którą ustawiamy na wartość hot_standby. Umożliwi to replikację wpisów z dzienników WAL:



Kolejne dwa bardzo ważne parametry to max_wal_senders i wal_keep_segments. max_wal_senders określa ile może być maksymalnie serwerów SLAVE – czyli w zasadzie na ilu serwerach może być prowadzona replikacja naszego serwera. Ja daję na potrzeby tego przykładu 1, ale oczywiście możesz dać też wyższą wartość. Parametr wal_keep_segments określa liczbę plików WAL która ma być przetrzymywana na wypadek gdyby przykładowo serwer SLAVE uległ awarii i nie można było na bieżąco kopiować wpisów. Pamiętaj że każdy plik WAL to 16MB, musisz więc dysponować przestrzenią wystarczającą do ewentualnego przechowania takiego zbioru plików.



Nieco niżej znajduje się parametr synchronous_standby_names. Wprowadzamy tam listę serwerów które mogą prowadzić replikację – na ten moment daję * czyli wszystkie, później w razie potrzeby to zmienię.



Poniżej lista parametrów i ich wartości jakie powinny być ustawione:

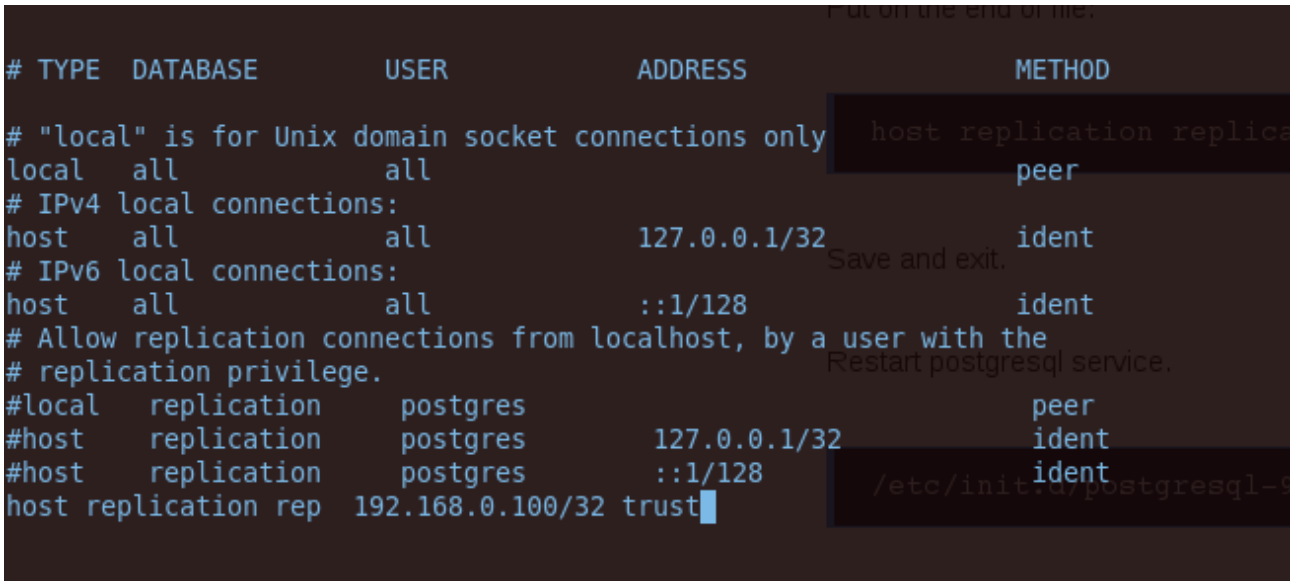
listen_address='*'
wal_level = hot_standby
max_wal_senders = 1
wal_keep_segments = 10
synchronous_standby_names = '*'

Przechodzimy teraz do pliku pg_hba.conf i przeprowadzamy drobną edycję aby umożliwić serwerowi SLAVE podpięcie się do serwera MASTER w celu replikacji:

nano /var/lib/pgsql/9.4/data/pg_hba.conf

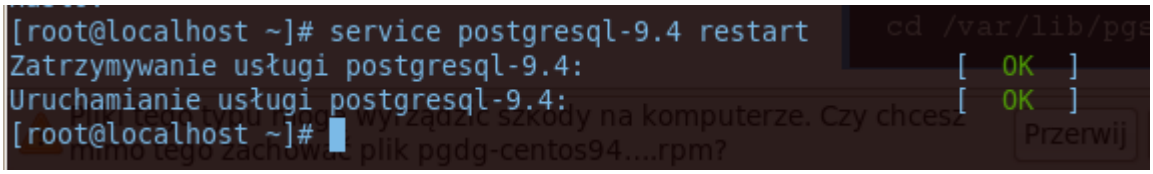
Na końcu dodajemy wpis podobny do poniższego. Pierwszy parametr oznacza połączenia sieciowe, drugi rodzaj połączenia (tutaj replikacja). Trzeci użytkownika któremu na to pozwalamy, czwarty to adres IP serwera SLAVE lub ewentualnie sieć w której będą serwery kopiujące. Ostatni to rodzaj autoryzacji - „trust" oznacza dostęp bez hasła – ostatecznie użytkownikowi REP nie ustawialiśmy żadnego. Nie przejmuj się za bardzo bezpieczeństwem w tym przypadku, ponieważ dostęp do serwera MASTER w celu replikacji będzie możliwy tylko ze wskazanych hostów.

host replication rep 192.168.0.100/32 trust



Aby przeładować konfigurację zrestartuj teraz serwer:

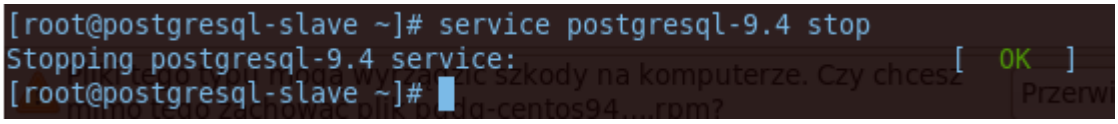
service postgresql-9.4 restart



Konfiguracja serwera SLAVE

Ponieważ będziemy usuwać zawartość katalogu z plikami danych, zatrzymaj wcześniej serwer PostgreSQL:

service postgresql-9.4 stop

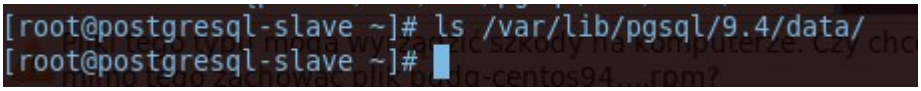


Usuń zawartość katalogu z plikami danych. Za chwilę zostaną tutaj skopiowane pliki z serwera MASTER.

rm -rf /var/lib/pgsql/9.4/data/*

Upewnij się jeszcze czy faktycznie katalog jest pusty:

ls /var/lib/pgsql/9.4/data/

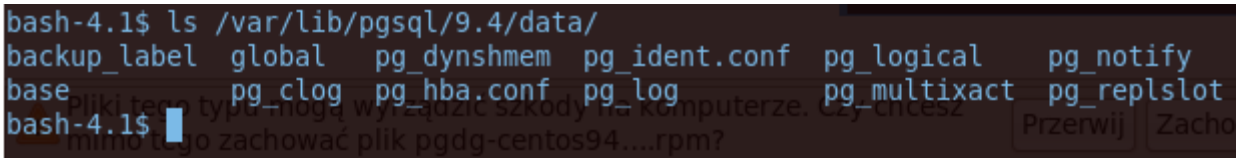


Ponieważ replikacja polega na powtarzaniu na serwerze SLAVE operacji które zostały wykonane na serwerze MASTER, musimy mieć jakiś punkt wyjścia. Tym punktem wyjścia będzie kopia zapasowa. Tutaj wykorzystujemy program pg_basebackup (powinien być w katalogu z binariami – tam gdzie m.in. psql). Jako parametr pierwszy wskazujemy katalog w którym mają znaleźć się pliki danych i pliki konfiguracyjne, drugi parametr to adres IP serwera MASTER, trzeci to użytkownik z poziomu którego chcemy tę kopię wykonać. W tym przypadku jest to przed momentem stworzony użytkownik REP.

pg_basebackup -D /var/lib/pgsql/9.4/data/ -h 192.168.0.101 -U rep

Operacja ta może potrwać, uzbrój się więc w cierpliwość. Kopiowana jest przecież cała baza, a jeśli Twoja jest duża to zajmie to chwilę. Gdy operacja się zakończy zweryfikuj czy pliki powstały:

ls /var/lib/pgsql/9.4/data/



Tworzymy teraz plik recovery.conf w katalogu /var/lib/pgsql/9.4/data/ . Taki plik zazwyczaj służy po prostu do odtwarzania bazy, w tym jednak przypadku wskazuje serwer z którego ma być przeprowadzana replikacja (parametr primary_conninfo), oraz plik którego pojawienie się spowoduje uruchomienie serwera SLAVE w normalnym trybie działania zapis/odczyt (np. w sytuacji gdy będzie on musiał awaryjnie zastąpić serwer MASTER). Takie rozwiązanie może być skuteczne w przypadku gdybyśmy musieli szybko przywrócić system oparty o tę bazę do działania, jednak powoduje przerwanie replikacji! Serwer SLAVE przechodzi w takim wypadku z trybu tylko odczyt do trybu zapis-odczyt.

cd /var/lib/pgsql/9.4/data/

nano recovery.conf

Do tego pliku wprowadzać informacje wg poniższego wzorca:

standby_mode=on
trigger_file='/tmp/odpalacz'
primary_conninfo='host=192.168.0.101 port=5432 user=rep application_name=mapetyzmy'

Przejdźmy teraz na moment do pliku postgresql.conf. Dokonamy zmiany która umożliwi dostęp do serwera SLAVE w trybie tylko do odczytu. Znajduje to zastosowanie jako swoisty „load balancing”. Serwer master służy w takiej konfiguracji do przeprowadzania zmian na danych, a kilka serwerów SLAVE do uruchamiania kwerend.

nano postgresql.conf

zmieniamy parametr hot_standby:

hot_standby = on

I restartujemy Postgresa:

service postgresql-9.4 restart

Testy działania

Oba serwery pierwotnie były kompletnie puste, dopiero po instalacji. Na obu serwerach sprawdzam tabele jakie są. Następnie na serwerze MASTER tworzę tabelę OMG, która po chwili bez żadnej ingerencji z mojej strony pojawia się również na serwerze SLAVE: