Rozmiary, tablespace i miejsce na dysku

Poziom: podstawy **Dotyczy: PostgreSQL 9.6–17** **Czas: ~20 min**

Miejsce na dysku kończy się zawsze w najgorszym możliwym momencie — ta strona pokazuje, jak rozłożyć dane na kilka wolumenów przez tablespace, jak mierzyć, co realnie zajmuje przestrzeń, i dlaczego samo `DELETE` tego problemu nie rozwiązuje.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
GB	Gigabyte: miliard bajtów, jednostka pojemności pamięci i plików, tu najczęściej rozmiar modelu albo pamięci karty graficznej
NVMe	Non-Volatile Memory Express: protokół komunikacji z dyskami SSD podłączonymi przez złącze PCIe, znacznie szybszy niż starsze SATA i SAS, typowy dla szybkich dysków systemowych
SET	Switch Embedded Teaming: zespół kart sieciowych zbudowany w przełączniku wirtualnym Hyper-V; łączy dwie karty w jedno łącze bez ustawień po stronie przełącznika fizycznego, więc bez LACP
WAL	Write-Ahead Log: dziennik PostgreSQL, zapis zmian przed zapisem danych

Fiszka

ZMIENIA SYSTEM

Utworzenie nowego tablespace

```
CREATE TABLESPACE NazwaTablespace LOCATION '/postgres/dane_archiwalne';
```

Wskazujemy istniejący, pusty katalog na dysku, do którego trafią pliki obiektów przypisanych do tego tablespace. Katalog musi być wcześniej utworzony i zapisywalny dla użytkownika systemowego, na którym działa instancja — samo `CREATE TABLESPACE` katalogu nie tworzy.

Przykład:

```
CREATE TABLESPACE
```

◫◫ ZMIENIA SYSTEM

Przeniesienie istniejącej tabeli do innego tablespace

```
ALTER TABLE NazwaTabeli SET TABLESPACE NazwaTablespace;
```

Przenosimy plik tabeli na inny wolumen — operacja przepisuje dane od zera w nowej lokalizacji i trzyma blokadę `ACCESS EXCLUSIVE` przez cały czas trwania, dokładnie tak samo kosztowną jak `VACUUM FULL` opisane na stronie VACUUM, autovacuum i bloat.

Przykład:

```
ALTER TABLE
```

◫◫ **SET** = Switch Embedded Teaming: zespół kart sieciowych zbudowany w przełączniku wirtualnym Hyper-V; łączy dwie karty w jedno łącze bez ustawień po stronie przełącznika fizycznego, więc bez LACP

◫◫ ZMIENIA SYSTEM

Przeniesienie indeksu do innego tablespace

```
ALTER INDEX NazwaIndeksu SET TABLESPACE NazwaTablespace;
```

Indeksy mają własny tablespace niezależny od tabeli, którą obsługują — typowy wybór to trzymanie najbardziej obciążonych indeksów na najszybszym dostępnym wolumenie, niezależnie od tego, gdzie leżą same dane.

Przykład:

```
ALTER INDEX
```

◫◫ **SET** = Switch Embedded Teaming: zespół kart sieciowych zbudowany w przełączniku wirtualnym Hyper-V; łączy dwie karty w jedno łącze bez ustawień po stronie przełącznika fizycznego, więc bez LACP

□□ ZMIENIA SYSTEM

Przeniesienie całej bazy do innego tablespace

```
ALTER DATABASE NazwaBazy SET TABLESPACE NazwaTablespace;
```

Przenosimy wszystkie obiekty bazy naraz — polecenie wymaga, żeby nikt inny nie był w tej chwili podłączony do przenoszonej bazy, inaczej kończy się błędem o zajętych obiektach.

Przykład:

```
ALTER DATABASE
```

□□ **SET** = Switch Embedded Teaming: zespół kart sieciowych zbudowany w przełączniku wirtualnym Hyper-V; łączy dwie karty w jedno łącze bez ustawień po stronie przełącznika fizycznego, więc bez LACP

□□ ZMIENIA SYSTEM

Ustawienie domyślnego tablespace dla nowych obiektów: default_tablespace

```
SET default_tablespace = NazwaTablespace;
```

Każdy `CREATE TABLE` i `CREATE INDEX` wykonany w tej sesji, bez jawnie podanego `TABLESPACE`, trafi do wskazanego miejsca zamiast do domyślnego `pg_default`. Ustawienie na stałe dla roli albo bazy działa tak samo jak `statement_timeout` opisany na stronie Sesje, długie zapytania, kasowanie kwerendy: przez `ALTER ROLE NazwaUsera SET default_tablespace = NazwaTablespace`.

□□ **SET** = Switch Embedded Teaming: zespół kart sieciowych zbudowany w przełączniku wirtualnym Hyper-V; łączy dwie karty w jedno łącze bez ustawień po stronie przełącznika fizycznego, więc bez LACP

□□ PLIK KONFIGURACYJNY

Wskazanie tablespace dla plików tymczasowych: temp_tablespaces

```
temp_tablespaces = 'NazwaTablespace'
```

Pliki tymczasowe powstające przy przekroczeniu `work_mem`, opisane na stronie Logi serwera: konfiguracja i czytanie, domyślnie lądują w tym samym tablespace co reszta danych — wskazanie osobnego, szybkiego wolumenu odciąża dysk z danymi produkcyjnymi od dodatkowego ruchu sortowań i haszowań.

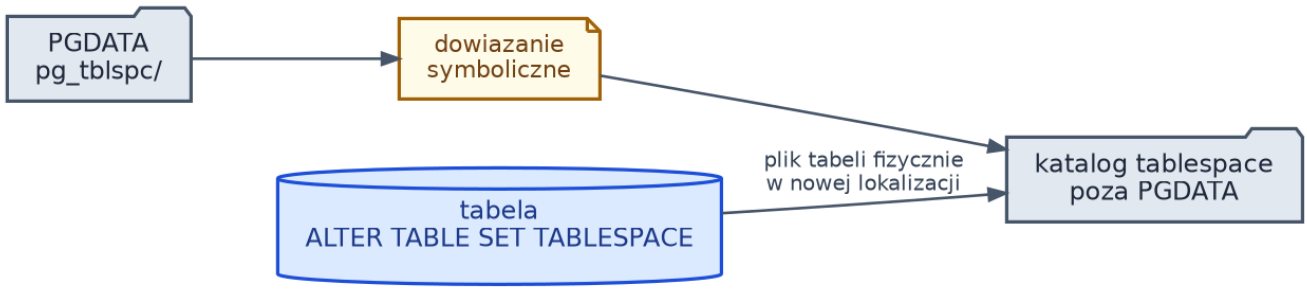
Z życia wzięte

Z ŻYCIA WZIĘTE

Dysk z danymi bazy cennik na pg01 zapełnia się szybciej niż reszta wolumenu, bo tabela z historią cen rośnie dwa razy szybciej, niż zakładano przy projektowaniu. Zakładamy nowy tablespace na wolniejszym, ale większym wolumenie i przenosimy tam tylko tę jedną tabelę poleceniem `ALTER TABLE SET TABLESPACE`. Operacja trzyma blokadę przez kilkanaście minut, więc wykonujemy ją wieczorem, kiedy nikt z działu handlowego nie pracuje na cenniku.

Czym jest tablespace

Pojęcie: tablespace. Katalog na dysku, leżący poza `PGDATA`, do którego PostgreSQL zapisuje pliki wybranych tabel, indeksów albo całej bazy — w samym `PGDATA` zostaje jedynie dowiązanie symboliczne w podkatalogu `pg_tblspc/`, wskazujące na rzeczywistą lokalizację. Mechanizm istnieje po to, żeby rozłożyć dane instancji na kilka fizycznych wolumenów: dane produkcyjne na szybkim dysku NVMe, archiwalne tabele na wolniejszym i tańszym dysku, bez przenoszenia całej instancji.



PGDATA trzyma w pg_tblspc dowiązanie symboliczne do katalogu tablespace poza nim, tam trafia plik tabeli po ALTER TABLE SET TABLESPACE (kliknięcie otwiera pełny rozmiar)

Rozmiary — fiszki

Sprawdzenie zajętości konkretnego tablespace

```
SELECT pg_size_pretty(pg_tablespace_size('NazwaTablespace'));
```

Sprawdzamy, ile faktycznie zajmuje jeden konkretny tablespace, niezależnie od tego, ile baz i tabel jest do niego przypisanych.

Przykład:

```
 pg_size_pretty
-----------------
 812 GB
(1 row)
```

Rozmiar pojedynczej bazy i wszystkich baz naraz pokazują strony Rozpoznanie bazy: rozmiary, obiekty, aktywność i Monitoring instancji.

Sprawdzenie największych tabel z podziałem na dane i indeksy

```
SELECT relname AS tabela,
       pg_size_pretty(pg_table_size(oid)) AS rozmiar_danych,
       pg_size_pretty(pg_indexes_size(oid)) AS rozmiar_indeksow,
       pg_size_pretty(pg_total_relation_size(oid)) AS razem
FROM pg_class
WHERE relkind = 'r'
ORDER BY pg_total_relation_size(oid) DESC
LIMIT 10;
```

> **Pojęcie: pg_table_size kontra pg_indexes_size kontra pg_total_relation_size.**
> `pg_table_size` liczy samą tabelę wraz z jej danymi TOAST, ale bez indeksów. `pg_indexes_size` liczy sumę wszystkich indeksów założonych na tabeli, bez samych danych. `pg_total_relation_size` to suma obu — dopiero ona pokazuje, ile miejsca na dysku tabela realnie kosztuje w całości. Rozbicie na te dwie składowe pokazuje, czy dana tabela puchnie od danych, czy od nadmiaru indeksów.

Przykład:

```
   tabela    | rozmiar_danych | rozmiar_indeksow |  razem
-------------+----------------+------------------+----------
 zamowienia  | 2145 MB        | 890 MB           | 3035 MB
 klienci     | 98 MB          | 41 MB            | 139 MB
(2 rows)
```

Sprawdzenie rozmiaru całego schematu

```
SELECT n.nspname AS schemat,
       pg_size_pretty(sum(pg_total_relation_size(c.oid))) AS rozmiar
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind = 'r'
GROUP BY n.nspname
ORDER BY sum(pg_total_relation_size(c.oid)) DESC;
```

Sumujemy rozmiar wszystkich tabel danego schematu wraz z ich indeksami — przydatne, gdy baza trzyma kilka schematów o różnym przeznaczeniu i chcemy wiedzieć, który z nich faktycznie zajmuje miejsce.

Przykład:

```
  schemat  | rozmiar
-----------+---------
 sprzedaz  | 3174 MB
 public    | 245 MB
(2 rows)
```

Sprawdzenie rozmiaru katalogu pg_wal

```
du -sh /var/lib/pgsql/15/data/pg_wal
```

Katalog WAL, opisany na stronie Logi WAL: włączenie i archiwizacja, rośnie i maleje niezależnie od rozmiaru samych danych — przy planowaniu pojemności wolumenu, na którym leży `PGDATA`, trzeba doliczyć zapas na `pg_wal` osobno, zwykle w okolicach dwu-trzykrotności skonfigurowanego `max_wal_size`, żeby wytrzymać chwilowe skoki ruchu bez zapełnienia dysku.

Przykład:

```
1.2G    /var/lib/pgsql/15/data/pg_wal
```

Odczyt tej samej wartości pod kątem alarmu progowego, nie jednorazowego sprawdzenia, opisuje strona Monitoring instancji.

Dlaczego DELETE nie zwalnia miejsca

Mechanizm martwych krotek i to, dlaczego `DELETE` oznacza wiersz jako usunięty zamiast fizycznie zwolnić miejsce, opisuje w pełni strona VACUUM, autovacuum i bloat. Z punktu widzenia planowania pojemności dysku konsekwencja jest prosta: duża operacja usuwania danych nie oddaje miejsca systemowi plików sama z siebie — dopiero `VACUUM FULL` albo `CLUSTER` przepisuje tabelę do nowego, mniejszego pliku, zwalniając resztę z powrotem do systemu operacyjnego.

Miejsce potrzebne na VACUUM FULL i na REINDEX

Uważaj: obie operacje potrzebują dodatkowego miejsca na dysku w trakcie działania, nie po jego zakończeniu — brak tego zapasu kończy operację błędem w połowie, zostawiając tabelę zablokowaną bez korzyści z przepisania.

`VACUUM FULL`, opisane na stronie VACUUM, autovacuum i bloat, buduje nowy plik tabeli obok starego i dopiero po zakończeniu podmienia je miejscami — przez cały czas trwania oba pliki, stary i nowy, istnieją na dysku równolegle. Bezpieczny zapas to rozmiar tabeli sprzed operacji, w najgorszym przypadku tyle samo, ile tabela zajmuje w tej chwili.

`REINDEX` bez `CONCURRENTLY` potrzebuje dodatkowego miejsca w wysokości rozmiaru przebudowywanego indeksu — stary indeks jest zwalniany dopiero po zbudowaniu nowego.

Wariant `REINDEX CONCURRENTLY`, opisany na stronie Indeksy i plany wykonania w PostgreSQL, potrzebuje tego zapasu tym bardziej: buduje nowy indeks bez blokowania zapisów, co oznacza, że stary indeks pozostaje w pełni aktywny i zajęty przez cały, zwykle dłuższy czas budowy.

Prognozowanie przyrostu

Silnik nie trzyma własnej historii rozmiaru bazy — prognozowanie wymaga cyklicznego zapisywania wyniku `pg_database_size` albo `pg_tablespace_size` w osobnym miejscu (tabeli monitoringu, historii Zabbixa) i policzenia różnicy między kolejnymi odczytami.

Przykład prostego zestawienia z trzech kolejnych dni, z którego liczymy dzienny przyrost i szacujemy datę zapełnienia wolumenu:

```
     data     | rozmiar_gb
--------------+-------------
 2026-09-23   |      812.4
 2026-09-24   |      814.1
 2026-09-25   |      815.9
```

Przyrost rzędu 1,7 GB dziennie przy wolumenie o pojemności 1 TB i obecnym zapełnieniu 815 GB daje około 108 dni zapasu — liczba warta przeliczenia częściej niż raz, bo tempo przyrostu potrafi się zmienić skokowo po wdrożeniu nowej funkcji aplikacji, nie tylko rosnąć liniowo.

Częste błędy

Zakładanie, że miejsce zwalnia się od razu po dużym DELETE. Bez `VACUUM FULL` plik tabeli zostaje tego samego rozmiaru — wolne miejsce w środku jest tylko gotowe do ponownego wykorzystania przez przyszłe zapisy, nie oddane systemowi operacyjnemu.

Uruchomienie VACUUM FULL bez sprawdzenia wolnego miejsca na wolumenie. Operacja potrzebuje chwilowo dodatkowego miejsca w rozmiarze zbliżonym do samej tabeli — na wolumenie zapełnionym w 95% kończy się to błędem `No space left on device` w połowie przepisywania, temat rozwinięty na stronie Typowe awarie i co robimy.

Pomijanie temp_tablespaces przy planowaniu pojemności osobnego wolumenu. Wolumen wydzielony pod dane bazy nie musi mieć zapasu na pliki tymczasowe, jeśli `temp_tablespaces` wskazuje gdzie indziej — ale ten inny wolumen wtedy potrzebuje własnego zapasu, o którym łatwo zapomnieć.

Powiązane: VACUUM, autovacuum i bloat · Rozpoznanie bazy: rozmiary, obiekty, aktywność · Monitoring instancji · Logi WAL: włączenie i archiwizacja · Architektura PostgreSQL: klaster, baza, schemat, procesy

Źródła: dokumentacja PostgreSQL, rozdział „Tablespaces" i „System Administration Functions"

Revision #40
Created 2026-09-25 00:33:28 UTC by Tomasz Konieczny
Updated 2026-10-26 14:12:47 UTC by Tomasz Konieczny