Monitoring instancji PostgreSQL

Poziom: podstawy **Dotyczy: PostgreSQL 9.6–17** **Czas: ~25 min**

Kilkanaście widoków katalogowych, jedno rozszerzenie i jedna rola wbudowana — to cały zestaw, z którego składa się monitoring instancji PostgreSQL, niezależnie od tego, czy odczytuje go Zabbix co minutę, czy administrator ręcznie raz na tydzień.

Skróty użyte na tej stronie (1)

Skrót	Co znaczy
WAL	Write-Ahead Log: dziennik PostgreSQL, zapis zmian przed zapisem danych

Fiszka

TYLKO ODCZYT

Sprawdzenie skuteczności cache'a bazy: pg_stat_database

```
SELECT datname,
       blks_hit, blks_read,
       round(blks_hit::numeric / NULLIF(blks_hit + blks_read, 0) * 100, 2) AS
trafienia_procent
FROM pg_stat_database
WHERE datname NOT IN ('template0', 'template1');
```

Sprawdzamy, jaki procent żądań o stronę danych trafił w pamięć podręczną `shared_buffers`, a jaki wymagał odczytu z dysku — to najprostszy, jednoliczbowy wskaźnik ogólnej kondycji instancji pod kątem pamięci.

Przykład:

```
   datname   | blks_hit  | blks_read | trafienia_procent
-------------+-----------+-----------+--------------------
 produkcja   | 128401932 |    241053 |              99.81
 postgres    |     84021 |       912 |              98.91
(2 rows)
```

Uważaj: liczniki są skumulowane od ostatniego restartu instancji albo od ręcznego `pg_stat_reset()` — tuż po restarcie wskaźnik wygląda gorzej, niż jest w rzeczywistości, dopóki cache się nie rozgrzeje.

TYLKO ODCZYT

Sprawdzenie deadlocków i rollbacków: pg_stat_database

```
SELECT datname, deadlocks, xact_commit, xact_rollback
FROM pg_stat_database
WHERE datname NOT IN ('template0', 'template1');
```

Sprawdzamy, ile transakcji zakończyło się wycofaniem zamiast zatwierdzeniem i ile razy silnik musiał rozstrzygnąć zakleszczenie — zjawisko opisane ogólnie na stronie Zakleszczenie (deadlock): jak powstaje i jak je czytać z logu.

Przykład:

```
   datname   | deadlocks | xact_commit | xact_rollback
-------------+-----------+-------------+---------------
 produkcja   |         2 |     4821093 |          12481
(1 row)
```

Pojedyncze `deadlocks` w skali miliardów transakcji jest normalne — rosnąca liczba w krótkim czasie, a nie skumulowana latami, jest sygnałem do przejrzenia logu i kolejności blokowania w aplikacji.

TYLKO ODCZYT

Sprawdzenie zajętości limitu połączeń: pg_stat_activity wobec max_connections

```
SELECT count(*) AS polaczenia,
       current_setting('max_connections')::int AS limit_polaczen,
       round(count(*)::numeric / current_setting('max_connections')::int * 100, 1) AS
procent_limitu
FROM pg_stat_activity;
```

Samo zliczenie połączeń pokazuje strona Rozpoznanie bazy: rozmiary, obiekty, aktywność — tutaj dokładamy procent limitu, bo to on decyduje, czy sytuacja jest alarmowa, nie sama liczba bezwzględna.

Przykład:

```
 polaczenia | limit_polaczen | procent_limitu
------------+----------------+----------------
         87 |            100 |           87.0
(1 row)
```

TYLKO ODCZYT

Sprawdzenie najdłużej otwartej transakcji: pg_stat_activity

```
SELECT pid, usename, state, now() - xact_start AS czas_transakcji
FROM pg_stat_activity
WHERE xact_start IS NOT NULL
ORDER BY czas_transakcji DESC
LIMIT 5;
```

W odróżnieniu od zapytania z Sesje, długie zapytania, kasowanie kwerendy, które sortuje po czasie trwania samego zapytania (`query_start`), tutaj sortujemy po czasie trwania całej transakcji (`xact_start`) — transakcja może stać otwarta bardzo długo, mimo że jej ostatnie zapytanie skończyło się w ułamku sekundy, a właśnie to jest problemem dla autovacuum.

Przykład:

```
  pid  |   usename   |        state        | czas_transakcji
-------+-------------+---------------------+-----------------
 24501 | nazwa_usera | idle in transaction | 00:47:12
(1 row)
```

🔒 TYLKO ODCZYT

Sprawdzenie liczby sesji bezczynnych w otwartej transakcji: pg_stat_activity

```
SELECT count(*) FROM pg_stat_activity WHERE state = 'idle in transaction';
```

Liczymy sesje w stanie `idle in transaction`, opisanym na stronie Sesje, długie zapytania, kasowanie kwerendy — nawet pojedyncza taka sesja trzymana godzinami blokuje autovacuum na tabelach, których dotyczy jej transakcja.

Przykład:

```
 count
-------
     1
(1 row)
```

🔒 TYLKO ODCZYT

Sprawdzenie czasu ostatniego autovacuum na tabelach: pg_stat_user_tables

```
SELECT relname, last_autovacuum, last_autoanalyze
FROM pg_stat_user_tables
ORDER BY last_autovacuum NULLS FIRST
LIMIT 10;
```

Sprawdzamy, kiedy autovacuum ostatnio dotknął każdą tabelę — wartość `NULL` oznacza, że od utworzenia tabeli albo od ostatniego restartu statystyk autovacuum jeszcze na niej w ogóle nie zadziałał. Procent martwych krotek, do którego to zapytanie naturalnie się dokłada, rozwija strona VACUUM, autovacuum i bloat.

Przykład:

```
    relname    |       last_autovacuum        | last_autoanalyze
---------------+------------------------------+------------------------------
 log_zdarzen   |                              |
 zamowienia    | 2026-09-24 03:12:08.201+02   | 2026-09-24 03:14:51.009+02
```

```
(2 rows)
```

TYLKO ODCZYT

Sprawdzenie aktywności checkpointera: pg_stat_bgwriter

```
SELECT checkpoints_timed, checkpoints_req, buffers_checkpoint, buffers_clean,
buffers_backend
FROM pg_stat_bgwriter;
```

> **Pojęcie: pg_stat_bgwriter.** Widok zliczający, ile checkpointów wystąpiło z naturalnego harmonogramu (`checkpoints_timed`) i ile zostało wymuszonych wcześniej, bo `pg_wal` urósł ponad `max_wal_size`, opisany na stronie Logi WAL: włączenie i archiwizacja, zanim minął `checkpoint_timeout` (`checkpoints_req`). Dokłada też liczbę stron zapisanych przez sam checkpointer (`buffers_checkpoint`), przez proces `bgwriter` w tle (`buffers_clean`) i przez zwykłe procesy backendowe zmuszone zapisać stronę same, bo żaden z dwóch pierwszych mechanizmów nie zdążył (`buffers_backend`).

Przykład:

```
 checkpoints_timed | checkpoints_req | buffers_checkpoint | buffers_clean |
buffers_backend
-------------------+------------------+--------------------+---------------
+------------------
                842 |              412 |            1204821 |         98211
|            401203
(1 row)
```

Uważaj: `checkpoints_req` zbliżający się do `checkpoints_timed` albo go przewyższający oznacza, że `max_wal_size` jest zbyt mały wobec realnego ruchu transakcyjnego — checkpointy wymuszone kosztują więcej I/O niż te wykonane zgodnie z harmonogramem. Wysoki `buffers_backend` względem `buffers_checkpoint` i `buffers_clean` oznacza, że to zwykłe zapytania zapisujące dane, nie procesy w tle, wykonują pracę, która powinna być rozłożona wcześniej.

WAL = Write-Ahead Log: dziennik PostgreSQL, zapis zmian przed zapisem danych

TYLKO ODCZYT

Sprawdzenie wieku transakcji względem progu wraparound: pg_database

```
SELECT datname, age(datfrozenxid) AS wiek,
       round(age(datfrozenxid)::numeric / 200000000 * 100, 1) AS procent_limitu
FROM pg_database
ORDER BY wiek DESC;
```

Mechanizm wraparound i samo pojęcie wieku transakcji opisuje w pełni strona VACUUM, autovacuum i bloat — tutaj dokładamy procent domyślnego progu `autovacuum_freeze_max_age` (200 milionów), żeby liczba bezwzględna od razu przekładała się na to, jak blisko alarmu jesteśmy.

Przykład:

```
   datname   |   wiek    | procent_limitu
-------------+-----------+-----------------
 produkcja   |  42891203 |             21.4
(1 row)
```

Alarm ustawiamy zwykle na 80% limitu, nie na 100% — zostawia to czas na ręczną interwencję, zanim silnik sam zacznie odmawiać nowych transakcji.

TYLKO ODCZYT

Sprawdzenie rozmiaru wszystkich baz w klastrze: pg_database_size

```
SELECT datname, pg_size_pretty(pg_database_size(datname)) AS rozmiar
FROM pg_database
ORDER BY pg_database_size(datname) DESC;
```

Pojedynczą bazę pokazuje strona Rozpoznanie bazy: rozmiary, obiekty, aktywność — tutaj interesuje nas zestawienie wszystkich baz naraz, jako punkt odniesienia do śledzenia przyrostu w czasie. Sam przyrost mierzymy przez cykliczny zapis tej wartości (np. do tabeli monitoringu albo historii Zabbixa) i porównanie kolejnych odczytów — silnik nie trzyma własnej historii rozmiaru.

Przykład:

```
   datname   | rozmiar
-------------+---------
 produkcja   | 4823 MB
 postgres    | 8123 kB
(2 rows)
```

TYLKO ODCZYT

Sprawdzenie opóźnienia repliki: pg_stat_replication

```
SELECT client_addr, state, pg_wal_lsn_diff(sent_lsn, replay_lsn) AS bajtow_opoznienia
FROM pg_stat_replication;
```

Pełny obraz konfiguracji, weryfikacji i konfliktów replikacji fizycznej rozwija strona Replikacja strumieniowa — tutaj tylko liczba, którą warto trzymać pod stałą obserwacją: różnica w bajtach między pozycją wysłaną a odtworzoną na replice.

Przykład:

```
 client_addr |   state   | bajtow_opoznienia
-------------+-----------+------------------
 10.20.10.40 | streaming |             81920
(1 row)
```

TYLKO ODCZYT

Sprawdzenie rozmiaru katalogu pg_wal

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

Sprawdzamy, ile miejsca zajmują bieżące segmenty WAL — rozmiar znacząco przekraczający skonfigurowany `max_wal_size` przez dłuższy czas jest pierwszym objawem, że coś przetrzymuje stare segmenty, zamiast pozwolić silnikowi je odzyskać. Pełne omówienie tego jako części planowania pojemności dysku znajduje się na stronie Rozmiary, tablespace i miejsce na dysku.

Przykład:

```
14G     /var/lib/pgsql/15/data/pg_wal
```

Czternaście gigabajtów przy `max_wal_size = 1GB` to nie jest naturalne wahanie — najczęstsze przyczyny rozwija strona Typowe awarie i co robimy.

WAL = Write-Ahead Log: dziennik PostgreSQL, zapis zmian przed zapisem danych · **du** = disk usage: ile miejsca zajmuje katalog

TYLKO ODCZYT

Odnalezienie zapytań wolnych przeciętnie: pg_stat_statements po mean_exec_time

```
SELECT query, calls, round(mean_exec_time::numeric, 2) AS sredni_czas_ms
FROM pg_stat_statements
ORDER BY mean_exec_time DESC
LIMIT 5;
```

Instalację rozszerzenia i ranking po `total_exec_time` pokazuje strona Indeksy i plany wykonania w PostgreSQL — sortowanie po czasie łącznym premiuje zapytania wykonywane bardzo często, nawet jeśli każde pojedyncze wykonanie jest szybkie. Sortowanie po `mean_exec_time` wyłapuje przeciwny przypadek: raport uruchamiany dwa razy dziennie, trwający czterdzieści sekund za każdym razem, który w rankingu po czasie łącznym ginie za tysiącami krótkich zapytań aplikacyjnych.

Przykład:

```
                          query                           | calls | sredni_czas_ms
----------------------------------------------------------+-------+----------------
 SELECT * FROM zamowienia z JOIN klienci k ON ... GROUP BY ... |     2 |        41208.90
(1 row)
```

TYLKO ODCZYT

Reset statystyk pg_stat_statements

```
SELECT pg_stat_statements_reset();
```

Zerujemy zgromadzone liczniki, zwykle po dużej zmianie w aplikacji albo w schemacie, żeby kolejny ranking najkosztowniejszych zapytań pokazywał obraz od tej chwili, a nie uśredniony z tygodniami sprzed zmiany.

Przykład:

```
 pg_stat_statements_reset
--------------------------


(1 row)
```

Z życia wzięte

Z ŻYCIA WZIĘTE

Zabbix na mon01 zaczyna alarmować o rosnącym procencie limitu połączeń na pg01 tuż przed zamknięciem miesiąca. Sprawdzamy `pg_stat_activity` i widzimy, że to panel Oli otwiera nowe połączenia zamiast korzystać z puli po restarcie aplikacji. Ola dodaje pulę połączeń po stronie aplikacji, zanim limit zostanie faktycznie przekroczony.

Co obserwujemy — lista i progi

Zanim przejdziemy do pojedynczych zapytań, warto mieć w głowie, czego szukamy i przy jakiej wartości sygnał staje się alarmem, a nie ciekawostką:

- **skuteczność cache'a** (`pg_stat_database`) — poniżej 95–99% na bazie, która działa dłużej niż kilka godzin, sygnalizuje za mały `shared_buffers` albo zapytania czytające dużo danych sekwencyjnie
- **deadlocki i rollbacki** (`pg_stat_database`) — rosnąca liczba `deadlocks` w krótkim czasie wskazuje na konflikt w kolejności blokowania w aplikacji, nie na przypadek
- **liczba połączeń wobec** `max_connections` (`pg_stat_activity`) — powyżej 80% limitu to sygnał do reakcji, zanim kolejne połączenie skończy się `FATAL: sorry, too many clients already`
- **najdłuższa otwarta transakcja i sesje** `idle in transaction` (`pg_stat_activity`) — transakcja otwarta dłużej niż kilka minut blokuje autovacuum i trzyma blokady
- **martwe krotki i czas ostatniego autovacuum** (`pg_stat_user_tables`) — rozwinięte na stronie VACUUM, autovacuum i bloat

- **aktywność bufora i checkpointów** (`pg_stat_bgwriter`) — wysoki udział `checkpoints_req` względem `checkpoints_timed` oznacza checkpointy wymuszane zbyt często
- **wiek transakcji i wraparound** (`age(datfrozenxid)`) — próg alarmowy liczymy jako procent `autovacuum_freeze_max_age`, nie samą liczbę bezwzględną
- **rozmiar baz i tempo przyrostu** — potrzebne do prognozowania miejsca na dysku, rozwinięte na stronie Rozmiary, tablespace i miejsce na dysku
- **opóźnienie repliki** — rozwinięte na stronie Replikacja strumieniowa
- **rozmiar katalogu pg_wal** — rosnący bez odpowiadającego wzrostu ruchu transakcyjnego to pierwszy objaw porzuconego slotu replikacyjnego albo zepsutego `archive_command`
- **najkosztowniejsze zapytania** (`pg_stat_statements`) — zarówno łącznie, jak i przeciętnie na wywołanie

Konto monitorujące z rolą pg_monitor

Rolę wbudowaną `pg_monitor` i pełną procedurę założenia dedykowanego konta monitorującego, na przykładzie Zabbixa, pokazuje w całości strona Użytkownicy, role i uprawnienia — dzięki niej wszystkie zapytania z tej strony można uruchamiać z osobnego, nieuprzywilejowanego konta, bez nadawania mu dostępu do samych danych w tabelach ani statusu superusera.

Ogólne omówienie tego, co i po co monitorujemy w bazie danych niezależnie od silnika, rozwija strona Co monitorujemy w bazie danych.

Częste błędy

Wnioskowanie o awarii z jednego wskaźnika w oderwaniu od reszty. Sam spadek trafień w cache przy jednoczesnym, jednorazowym zapytaniu analitycznym czytającym całą tabelę jest oczekiwany — dopiero utrzymujący się spadek przy stabilnym obciążeniu jest sygnałem.

Resetowanie pg_stat_statements przed zapisaniem raportu. `pg_stat_statements_reset()` kasuje historię bezpowrotnie — jeśli comiesięczny przegląd najkosztowniejszych zapytań ma sens, dane trzeba wyeksportować przed resetem, nie po nim.

Powiązane: VACUUM, autovacuum i bloat · Sesje, długie zapytania, kasowanie kwerendy · Indeksy i plany wykonania w PostgreSQL · Użytkownicy, role i uprawnienia · Co monitorujemy w bazie danych

Źródła: dokumentacja PostgreSQL, rozdział „The Cumulative Statistics System" i „The Statistics Collector" · dokumentacja rozszerzenia `pg_stat_statements`

Revision #40
Created 2026-09-25 00:33:22 UTC by Tomasz Konieczny
Updated 2026-10-26 14:12:43 UTC by Tomasz Konieczny