Replikacja strumieniowa

Typ: procedura

Dotyczy: PostgreSQL 12–17 (starsze wersje przez recovery.conf zamiast standby.signal)

Czas: ~40 min

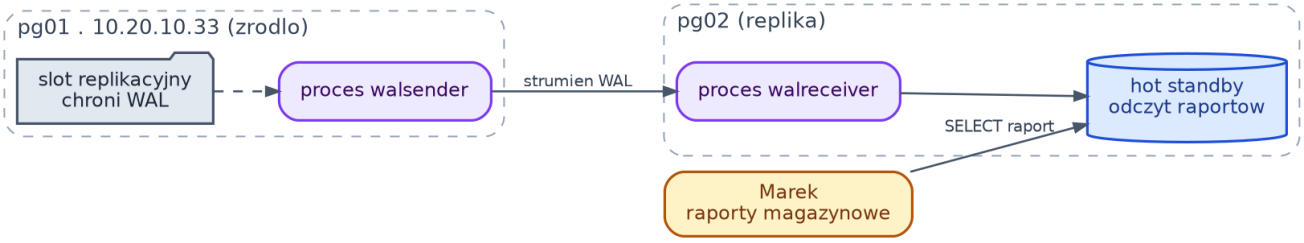
Efekt: druga instancja, `pg02`, na bieżąco odtwarzająca strumień WAL ze źródła `pg01`, gotowa do promocji na wypadek awarii tego drugiego.

Skróty użyte na tej stronie (2)

Skrót	Co znaczy
SQL	Structured Query Language: język zapytań do relacyjnych baz danych
WAL	Write-Ahead Log: dziennik PostgreSQL, zapis zmian przed zapisem danych

Czym jest replikacja fizyczna

> **Pojęcie: replikacja fizyczna.** Mechanizm, w którym replika odtwarza dokładnie te same segmenty WAL, opisane na stronie Logi WAL: włączenie i archiwizacja, które powstały na instancji źródłowej — nie wykonuje ponownie tych samych poleceń SQL, tylko odgrywa zapisane zmiany na poziomie stron danych, bajt po bajcie. Efektem jest kopia całego klastra: wszystkich baz naraz, identyczna aż do numeru wersji silnika. Ogólne umiejscowienie replikacji fizycznej wśród innych strategii opisuje strona Replikacja: synchroniczna, asynchroniczna, logiczna, fizyczna; replikacja obejmująca wybrane tabele zamiast całego klastra jest tematem osobnej strony, Replikacja logiczna.



Slot replikacyjny chroni WAL na pg01, walsender wysyła strumień do walreceivera na pg02, skąd Marek czyta raporty magazynowe (kliknięcie otwiera pełny rozmiar)

Zanim zaczniemy

> **Z ŻYCIA WZIĘTE**
>
> Stawiamy pg02 jako replikę bazy Zabbixa z pg01, żeby Marek mógł odpalać ciężkie raporty magazynowe bez obciążania instancji produkcyjnej. Kilka dni później replika traci łączność na godzinę przez awarię przełącznika w serwerowni, ale dzięki utworzonemu wcześniej slotowi replikacyjnemu żaden potrzebny segment WAL nie znika. Po powrocie sieci pg02 sam dogania opóźnienie bez konieczności ponownego `pg_basebackup`.

Wymaganie	Jak sprawdzić
`wal_level` co najmniej `replica` na źródle	`SHOW wal_level;`
Sieć między `pg01` a `pg02` otwarta na porcie instancji	`psql -h pg01 -p 5432` z hosta repliki
Wolne miejsce na `pg02` co najmniej równe rozmiarowi bazy źródłowej	Rozmiary, tablespace i miejsce na dysku
Ta sama wersja główna PostgreSQL na obu hostach	`psql -V` na obu maszynach

Uważaj: `pg_basebackup`, którym kopiujemy dane na replikę, wymaga, żeby katalog docelowy na `pg02` był pusty — nadpisanie działającej instancji tym poleceniem nie jest wspierane i kończy się błędem albo, gorzej, uszkodzonym katalogiem danych.

Przebieg

1. Konfiguracja instancji źródłowej

```
wal_level = replica
max_wal_senders = 10
```

`max_wal_senders` ogranicza liczbę jednoczesnych procesów wysyłających strumień WAL do replik i do narzędzi backupu takich jak `pg_basebackup` — każda replika i każdy trwający `pg_basebackup` zajmuje jeden slot z tej puli. Oba parametry wymagają restartu instancji źródłowej.

```
sudo systemctl restart postgresql-15.service
```

2. Utworzenie konta replikacyjnego i reguły w pg_hba.conf

Konto z atrybutem `REPLICATION` zakładamy dokładnie tak, jak pokazuje fiszka Dodanie usera backupu z prawem replikacji na stronie Użytkownicy, role i uprawnienia — ten sam typ konta obsługuje zarówno `pg_basebackup` do backupu jednorazowego, jak i ciągły strumień replikacji.

Dopisujemy w `pg_hba.conf` na `pg01` regułę dopuszczającą połączenie replikacyjne z adresu repliki, metodą `scram-sha-256` opisaną na stronie Uwierzytelnianie i bezpieczeństwo połączeń:

```
host    replication     replikator      10.20.10.40/32  scram-sha-256
```

```
SELECT pg_reload_conf();
```

3. Utworzenie slotu replikacyjnego na źródle

```
SELECT pg_create_physical_replication_slot('NazwaSlotu');
```

> **Pojęcie: slot replikacyjny.** Rezerwacja na instancji źródłowej, gwarantująca, że segment WAL potrzebny konkretnej replice nie zostanie usunięty ani zarchiwizowany-i-skasowany, dopóki ta replika go nie odczyta — bez slotu chwilowa przerwa w łączności między `pg01` a `pg02` (restart sieci, krótka awaria repliki) może skończyć się tym, że potrzebny segment zniknie, zanim replika zdąży nadrobić zaległość, i replikację trzeba zestawiać od nowa przez pełny `pg_basebackup`.

Przykład:

```
 slot_name  | lsn
------------+-----------
 srv_db02   | 0/5000060
(1 row)
```

Uważaj: slot chroni WAL kosztem ryzyka odwrotnego — replika trwale odłączona (repliki już nie ma, a slotu nikt nie usunął) sprawia, że źródło przechowuje segmenty WAL bezterminowo, aż zapełni dysk. Dokładnie ten scenariusz rozwija strona Typowe awarie i co robimy w przypadku rosnącego katalogu `pg_wal`.

4. Wykonanie pg_basebackup na repliku z opcją -R

```
pg_basebackup -h pg01 -U replikator -D /var/lib/pgsql/15/data -P -R -S srv_db02
```

Kopiujemy cały katalog danych ze źródła na hosta repliki. Opcja `-R` zapisuje po zakończeniu kopiowania plik `standby.signal` (obecność tego pliku każe silnikowi wystartować w trybie repliki, nie głównej instancji) oraz `primary_conninfo` w `postgresql.auto.conf`, ze wskazaniem hosta źródłowego i użytego konta — bez `-R` obie te rzeczy trzeba by przygotować ręcznie. Opcja `-S` wiąże kopię z utworzonym wcześniej slotem, żeby replika od razu korzystała z ochrony, którą on daje.

Przykład:

```
23945/23945 kB (100%), 1/1 tablespace
```

Uważaj: na PostgreSQL starszym niż 12 nie istnieje `standby.signal` — ten sam efekt dawał plik `recovery.conf` z dyrektywą `standby_mode = 'on'` w katalogu danych, usunięty jako osobny mechanizm w wersji 12 na rzecz obecnego podziału na `standby.signal` i zwykłe parametry `postgresql.conf`.

5. Uruchomienie usługi na replice

```
sudo systemctl start postgresql-15.service
```

Instancja startuje, widzi `standby.signal`, i zamiast otwierać się do zapisu, zaczyna nawiązywać połączenie replikacyjne do `pg01` wskazanego w `primary_conninfo`.

Weryfikacja

Sprawdzenie stanu repliki po stronie źródła

```
SELECT client_addr, state, sync_state, sent_lsn, replay_lsn
FROM pg_stat_replication;
```

Przykład:

```
 client_addr |   state   | sync_state | sent_lsn  | replay_lsn
-------------+-----------+------------+-----------+------------
 10.20.10.40 | streaming | async      | 0/5000210 | 0/5000210
(1 row)
```

`state = streaming` oznacza, że replika jest podłączona i na bieżąco odbiera strumień — `sent_lsn` równy `replay_lsn` oznacza, że nie ma żadnego opóźnienia w tej chwili.

Sprawdzenie, że instancja rzeczywiście działa jako replika

```
SELECT pg_is_in_recovery();
```

Wykonane na `pg02`:

```
 pg_is_in_recovery
-------------------
 t
(1 row)
```

Sprawdzenie opóźnienia po stronie repliki

```
SELECT now() - pg_last_xact_replay_timestamp() AS opoznienie;
```

Przykład:

```
 opoznienie
-------------
 00:00:00.412
(1 row)
```

Replika gorąca i konflikt z odtwarzaniem

> **Pojęcie: hot_standby.** Tryb, w którym replika, mimo że nieustannie odtwarza WAL, dopuszcza jednocześnie zapytania odczytu — bez niego replika służyłaby wyłącznie jako gorąca rezerwa, bez możliwości odciążenia instancji źródłowej raportami. Od PostgreSQL 10 `hot_standby = on` jest wartością domyślną.

> **Pojęcie: konflikt zapytania z odtwarzaniem.** Sytuacja, w której odtwarzanie WAL na replice chce usunąć wersję wiersza, którą wciąż widzi trwające na tej replice zapytanie odczytu — replika musi wybrać: albo wstrzymać odtwarzanie i poczekać na koniec zapytania, albo przerwać zapytanie i iść dalej z odtwarzaniem. Dwa parametry decydują, które z tych zachowań przeważy:

Parametr	Co robi
`max_standby_streaming_delay`	maksymalny czas, o jaki odtwarzanie może się opóźnić, czekając na zakończenie kolidującego zapytania, zanim to zapytanie zostanie przerwane
`hot_standby_feedback`	włączony, wysyła do źródła informację o najstarszej transakcji trwającej na replice, dzięki czemu autovacuum na źródle nie usuwa wersji wierszy, których replika jeszcze potrzebuje

Uważaj: `hot_standby_feedback = on` rozwiązuje konflikty kosztem przeniesienia problemu na źródło — długo trwające zapytanie na replice trzyma w ten sposób martwe krotki na źródle tak samo, jak trzymałaby je długa transakcja uruchomiona bezpośrednio tam, temat rozwinięty na stronie VACUUM, autovacuum i bloat.

Promocja repliki

```
pg_ctl promote -D /var/lib/pgsql/15/data
```

Kończymy tryb odtwarzania i otwieramy replikę do zapisu jako pełnoprawną, samodzielną instancję — plik `standby.signal` zostaje usunięty automatycznie, a kolejny restart nie wraca już do trybu repliki.

Przykład:

```
waiting for server to promote.... done
server promoted
```

> **Pojęcie: failover kontra switchover.** Failover to promocja repliki wykonana w reakcji na nieplanowaną awarię instancji źródłowej — źródło jest niedostępne, więc promocja odbywa się bez jego udziału i bez gwarancji, że każdy bajt WAL zdążył dotrzeć przed awarią. Switchover to ta sama operacja wykonana planowo, z obiema instancjami działającymi: źródło jest najpierw świadomie zatrzymywane albo przełączane w tryb tylko do odczytu, replika dogania ostatni bajt WAL, dopiero potem następuje promocja — zero utraconych transakcji, bo źródło współpracuje w całym procesie.

Wycofanie zmian

Zatrzymujemy replikę i usuwamy jej katalog danych, jeśli decydujemy się nie kontynuować konfiguracji:

```
sudo systemctl stop postgresql-15.service
```

Na źródle usuwamy slot, żeby przestał rezerwować miejsce w `pg_wal`, dopiero po upewnieniu się, że żadna inna replika ani proces backupu z niego nie korzysta:

```
SELECT pg_drop_replication_slot('NazwaSlotu');
```

Gdy coś nie zadziała

Objaw	Przyczyna	Naprawa

`FATAL: could not connect to the primary server` w logu repliki	sieć zamknięta albo brak reguły w `pg_hba.conf` na źródle	`telnet pg01 5432` z hosta repliki, sprawdzenie `pg_hba_file_rules` opisanego na stronie Uwierzytelnianie i bezpieczeństwo połączeń
`state` w `pg_stat_replication` pokazuje `catchup` bez końca	replika nie nadąża, sieć wolniejsza niż tempo generowania WAL na źródle	sprawdzenie przepustowości sieci, tymczasowe ograniczenie ruchu zapisu na źródle
`FATAL: requested WAL segment ... has already been removed`	slot nie istniał albo replika była odłączona dłużej niż pozwalał na to `wal_keep_size` bez slotu	ponowny `pg_basebackup` od zera, tym razem ze slotem powiązanym przez `-S`
Replika nie startuje, log pokazuje `database system identifier differs`	katalog danych repliki pochodzi z innej instancji niż ta wskazana w `primary_conninfo`	ponowny `pg_basebackup` z właściwego źródła, katalog docelowy musi być pusty

Powiązane: Logi WAL: włączenie i archiwizacja · Replikacja logiczna · Użytkownicy, role i uprawnienia · Uwierzytelnianie i bezpieczeństwo połączeń · Typowe awarie i co robimy

Źródła: dokumentacja PostgreSQL, rozdział „High Availability, Load Balancing, and Replication" · `man 1 pg_basebackup`

Revision #40
Created 2026-09-25 00:33:25 UTC by Tomasz Konieczny
Updated 2026-10-26 14:12:54 UTC by Tomasz Konieczny