Sprawozdanie laboratoria Bazy Danych

Wydanie 1.0

Piotr Kotuła

04 lip 2025

Contents

CHAPTER 1

Wprowadzenie

Prowadzący:dr inż. Piotr Czaja

Kurs: Bazy Danych 1

Autor: Piotr Kotuła

Prezentowany projekt opiera się na materiałach z sieci - artykułach, oficjalnej dokumentacji i praktycznych poradnikach. W jego ramach zebraliśmy kluczowe informacje o modelach i technologiach bazodanowych oraz przedstawiliśmy je w zwięzłej formie.

CHAPTER 2

Przegląd literatury

2.1 Konfiguracja baz danych

2.1.1 Sprawozdanie: Konfiguracja i Zarządzanie Bazą Danych

Authors

- Piotr Domagała
- Piotr Kotuła
- Dawid Pasikowski

1. Konfiguracja bazy danych

Wprowadzenie do tematu konfiguracji bazy danych obejmuje podstawowe informacje na temat zarządzania i dostosowywania ustawień baz danych w systemach informatycznych. Konfiguracja ta jest kluczowa dla zapewnienia bezpieczeństwa, wydajności oraz stabilności działania aplikacji korzystających z bazy danych. Obejmuje m.in. określenie parametrów połączenia, zarządzanie użytkownikami, uprawnieniami oraz optymalizację działania systemu bazodanowego.

2. Lokalizacja i struktura katalogów

Każda baza danych przechowuje swoje pliki w określonych lokalizacjach systemowych, zależnie od używanego silnika. Przykładowe lokalizacje:

- **PostgreSQL**: `/var/lib/pgsql/data`
- **MySQL**: `/var/lib/mysql`
- **SQL Server**: `C:\Program Files\Microsoft SQL Server`

Struktura katalogów obejmuje katalog główny bazy danych oraz podkatalogi na pliki danych, logi, kopie zapasowe i pliki konfiguracyjne.

Przykład: W dużych środowiskach produkcyjnych często stosuje się osobne dyski do przechowywania plików danych i logów transakcyjnych. Takie rozwiązanie pozwala na zwiększenie wydajności operacji zapisu oraz minimalizowanie ryzyka utraty danych.

Dobra praktyka: Zaleca się, aby katalogi z danymi i logami były regularnie monitorowane pod kątem dostępnego miejsca na dysku. Przepełnienie któregoś z nich może doprowadzić do zatrzymania pracy bazy danych.

3. Katalog danych

Jest to miejsce, gdzie fizycznie przechowywane są wszystkie pliki związane z bazą danych, takie jak:

- Pliki tabel i indeksów
- Dzienniki transakcji
- Pliki tymczasowe

Przykładowo: W PostgreSQL katalog danych to `/var/lib/pgsql/data`, gdzie znajdują się zarówno pliki z danymi, jak i główny plik konfiguracyjny `postgresql.conf`.

Wskazówka: Dostęp do katalogu danych powinien być ograniczony tylko do uprawnionych użytkowników systemu, co zwiększa bezpieczeństwo i zapobiega przypadkowym lub celowym modyfikacjom plików bazy.

4. Podział konfiguracji na podpliki

Konfiguracja systemu bazodanowego może być rozbita na kilka mniejszych, wyspecjalizowanych plików, np.:

- `postgresql.conf` – główne ustawienia serwera
- `pg_hba.conf` – reguły autoryzacji i dostępu
- `pg_ident.conf` – mapowanie użytkowników systemowych na użytkowników PostgreSQL

Przykład: Jeśli administrator chce zmienić jedynie sposób autoryzacji użytkowników, edytuje tylko plik `pg_hba.conf`, bez ryzyka wprowadzenia niezamierzonych zmian w innych częściach konfiguracji.

Dobra praktyka: Rozdzielenie konfiguracji na podpliki ułatwia zarządzanie, pozwala szybciej lokalizować błędy i minimalizuje ryzyko konfliktów podczas aktualizacji lub wdrażania zmian.

5. Katalog Konfiguracyjny

To miejsce przechowywania wszystkich plików konfiguracyjnych bazy danych, takich jak główny plik konfiguracyjny, pliki z ustawieniami użytkowników, uprawnień czy harmonogramów zadań.

Typowe lokalizacje to:

- `/etc` (np. `my.cnf` dla MySQL)
- Katalog danych bazy (np. `/var/lib/pgsql/data` dla PostgreSQL)

Przykład: W przypadku awarii systemu administrator może szybko przywrócić działanie bazy, kopiując wcześniej zapisane pliki konfiguracyjne z katalogu konfiguracyjnego.

Wskazówka: Regularne wykonywanie kopii zapasowych katalogu konfiguracyjnego jest kluczowe – utrata tych plików może uniemożliwić uruchomienie bazy danych lub spowodować utratę ważnych ustawień systemowych.

6. Katalog logów i struktura katalogów w PostgreSQL

Katalog logów PostgreSQL zapisuje logi w różnych lokalizacjach, zależnie od systemu operacyjnego:

- Na Debianie/Ubuntu: `/var/log/postgresql`
- Na Red Hat/CentOS: `/var/lib/pgsql/<wersja>/data/pg_log`

> Uwaga: Aby zapisywać logi do pliku, należy upewnić się, że opcja `logging_collector` jest włączona w pliku `postgresql.conf`.

Struktura katalogów PostgreSQL:

```
base/          # dane użytkownika - jedna podkatalog dla każdej bazy danych
global/        # dane wspólne dla wszystkich baz (np. użytkownicy)
pg_wal/        # pliki WAL (Write-Ahead Logging)
pg_stat/       # statystyki działania serwera
pg_log/        # logi (jeśli skonfigurowane)
pg_tblspc/     # dowiązania do tablespace'ów
pg_twophase/   # dane dla transakcji dwufazowych
postgresql.conf  # główny plik konfiguracyjny
pg_hba.conf      # kontrola dostępu
pg_ident.conf    # mapowanie użytkowników systemowych na bazodanowych
```

7. Przechowywanie i lokalizacja plików konfiguracyjnych

Główne pliki konfiguracyjne:

- `postgresql.conf` – konfiguracja instancji PostgreSQL (parametry wydajności, logowania, lokalizacji itd.)
- `pg_hba.conf` – kontrola dostępu (adresy IP, użytkownicy, metody autoryzacji)
- `pg_ident.conf` – mapowanie użytkowników systemowych na użytkowników bazodanowych

8. Podstawowe parametry konfiguracyjne

Słuchanie połączeń:

```
listen_addresses = 'localhost'
port = 5432
```

Pamięć i wydajność:

```
shared_buffers = 512MB          # pamięć współdzielona
work_mem = 4MB                  # pamięć na operacje sortowania/złączeń
maintenance_work_mem = 64MB     # dla operacji VACUUM, CREATE INDEX
```

Autovacuum:

```
autovacuum = on
autovacuum_naptime = 1min
```

Konfiguracja pliku `pg_hba.conf`:

```
# TYPE  DATABASE  USER  ADDRESS         METHOD
local   all       all   md5
host    all       all   192.168.0.0/24  md5
```

Konfiguracja pliku `pg_ident.conf`:

```
# MAPNAME      SYSTEM-USERNAME   PG-USERNAME
local_users  ubuntu            postgres
local_users  jan_kowalski      janek_db
```

Można użyć tej mapy w pliku `pg_hba.conf`:

```
local   all     all     peer map=local_users
```

9. Wstęp teoretyczny

Systemy zarządzania bazą danych (DBMS – *Database Management System*) umożliwiają tworzenie, modyfikowanie i zarządzanie danymi. Ułatwiają organizację danych, zapewniają integralność, bezpieczeństwo oraz możliwość jednoczesnego dostępu wielu użytkowników.

9.1 Klasyfikacja systemów zarządzania bazą danych

Systemy DBMS można klasyfikować według:

- **Architektura działania:** - *Klient-serwer* – system działa jako niezależna usługa (np. PostgreSQL). - *Osadzony (embedded)* – baza danych jest integralną częścią aplikacji (np. SQLite).
- **Rodzaj danych i funkcjonalność:** - *Relacyjne (RDBMS)* – oparte na tabelach, kluczach i SQL. - *Nierelacyjne (NoSQL)* – oparte na dokumentach, modelu klucz-wartość lub grafach.

Oba systemy – **SQLite** oraz **PostgreSQL** – należą do relacyjnych baz danych, lecz różnią się architekturą, wydajnością, konfiguracją i przeznaczeniem.

9.2 SQLite

SQLite to lekka, bezserwerowa baza danych typu embedded, gdzie cała baza znajduje się w jednym pliku. Dzięki temu jest bardzo wygodna przy tworzeniu aplikacji lokalnych, mobilnych oraz projektów prototypowych.

Cechy SQLite:

- Brak osobnego procesu serwera – baza działa w kontekście aplikacji.
- Niskie wymagania systemowe – brak potrzeby instalacji i konfiguracji.
- Baza przechowywana jako pojedynczy plik (*.sqlite* lub *.db*).
- Pełna obsługa SQL (z pewnymi ograniczeniami) – wspiera standard SQL-92.
- Ograniczona skalowalność przy wielu użytkownikach.

Zastosowanie:

- Aplikacje desktopowe (np. Firefox, VS Code).
- Aplikacje mobilne (Android, iOS).
- Małe i średnie systemy bazodanowe.

9.3 PostgreSQL

PostgreSQL to zaawansowany system relacyjnej bazy danych typu klient-serwer, rozwijany jako projekt open-source. Zapewnia pełne wsparcie dla SQL oraz liczne rozszerzenia (np. typy przestrzenne, JSON).

Cechy PostgreSQL:

- Architektura klient-serwer – działa jako oddzielny proces.
- Wysoka skalowalność i niezawodność – obsługuje wielu użytkowników, złożone zapytania, replikację.
- Obsługa transakcji, MVCC, indeksowania oraz zarządzania uprawnieniami.
- Rozszerzalność – możliwość definiowania własnych typów danych, funkcji i procedur.

Konfiguracja: Plikami konfiguracyjnymi są:

- `postgresql.conf` – ustawienia ogólne (port, ścieżki, pamięć, logi).
- `pg_hba.conf` – reguły autoryzacji.
- `pg_ident.conf` – mapowanie użytkowników systemowych na bazodanowych.

Zastosowanie:

- Systemy biznesowe, bankowe, analityczne.
- Aplikacje webowe i serwery aplikacyjne.
- Środowiska o wysokich wymaganiach bezpieczeństwa i kontroli dostępu.

9.4 Cel użycia obu systemów

W ramach zajęć wykorzystano zarówno **SQLite** (dla szybkiego startu i analizy zapytań bez instalacji serwera), jak i **PostgreSQL** (dla nauki konfiguracji, zarządzania użytkownikami, uprawnieniami oraz obsługi złożonych operacji).

10. Zarządzanie konfiguracją w PostgreSQL

PostgreSQL oferuje rozbudowany i elastyczny mechanizm konfiguracji, umożliwiający precyzyjne dostosowanie działania bazy danych do potrzeb użytkownika oraz środowiska (lokalnego, deweloperskiego, testowego czy produkcyjnego).

10.1 Pliki konfiguracyjne

Główne pliki konfiguracyjne PostgreSQL:

- **postgresql.conf** – ustawienia dotyczące pamięci, sieci, logowania, autovacuum, planowania zapytań.
- **pg_hba.conf** – definiuje metody uwierzytelniania i dostęp z określonych adresów.
- **pg_ident.conf** – mapowanie nazw użytkowników systemowych na użytkowników PostgreSQL.

Pliki te zazwyczaj znajdują się w katalogu danych (np. `/var/lib/postgresql/15/main/` lub `/etc/postgresql/15/main/`).

10.2 Przykładowe kluczowe parametry `postgresql.conf`

Parametr	Opis
shared_buffers	Ilość pamięci RAM przeznaczona na bufor danych (rekomendacja: 25–40% RAM).
work_mem	Pamięć dla pojedynczej operacji zapytania (np. sortowania).
maintenance_work_mem	Pamięć dla operacji administracyjnych (np. VACUUM, CREATE INDEX).
effective_cache_size	Szacunkowa ilość pamięci dostępnej na cache systemu operacyjnego.
max_connections	Maksymalna liczba jednoczesnych połączeń z bazą danych.
log_directory	Katalog, w którym zapisywane są logi PostgreSQL.
autovacuum	Włącza lub wyłącza automatyczne odświeżanie nieużywanych wierszy.

10.3 Sposoby zmiany konfiguracji

1. **Edycja pliku** `postgresql.conf`

   Zmiany są trwałe, ale wymagają restartu serwera (w niektórych przypadkach wystarczy reload).

   Przykład:

```
shared_buffers = 512MB
work_mem = 64MB
```

2. **Dynamiczna zmiana poprzez SQL**

   Przykład:

```
ALTER SYSTEM SET work_mem = '64MB';
SELECT pg_reload_conf();  # ładowanie zmian bez restartu
```

3. **Tymczasowa zmiana dla jednej sesji**

   Przykład:

```
SET work_mem = '128MB';
```

10.4 Sprawdzanie konfiguracji

- Aby sprawdzić aktualną wartość parametru:

```
SHOW work_mem;
```

- Pobranie szczegółowych informacji:

```
SELECT name, setting, unit, context, source
FROM pg_settings
WHERE name = 'work_mem';
```

- Wylistowanie parametrów wymagających restartu serwera:

```
SELECT name FROM pg_settings WHERE context = 'postmaster';
```

10.5 Narzędzia pomocnicze

- **pg_ctl** – narzędzie do zarządzania serwerem (start/stop/reload).
- **psql** – klient terminalowy PostgreSQL do wykonywania zapytań oraz operacji administracyjnych.
- **pgAdmin** – graficzne narzędzie do zarządzania bazą PostgreSQL (umożliwia edycję konfiguracji przez GUI).

10.6 Kontrola dostępu i mechanizmy uwierzytelniania

Konfiguracja umożliwia określenie, z jakich adresów i w jaki sposób można łączyć się z bazą:

- **Dostęp lokalny (localhost)** – połączenia z tej samej maszyny.
- **Dostęp z podsieci** – administrator może wskazać konkretne podsieci IP (np. `192.168.0.0/24`).
- **Mechanizmy uwierzytelniania** – np. `md5`, `scram-sha-256`, `peer` (weryfikacja użytkownika systemowego) czy `trust`.

Ważne, aby mechanizm `peer` był odpowiednio skonfigurowany, gdyż umożliwia automatyczną autoryzację, jeśli nazwa użytkownika systemowego i bazy zgadza się.

11. Planowanie

Planowanie w kontekście PostgreSQL oznacza optymalizację wykonania zapytań oraz efektywne zarządzanie zasobami.

11.1 Co to jest planowanie zapytań?

Proces planowania zapytań obejmuje:

- Analizę składni i struktury zapytania SQL.
- Przegląd dostępnych statystyk dotyczących tabel, indeksów i danych.
- Dobór sposobu dostępu do danych (pełny skan, indeks, join, sortowanie).
- Tworzenie planu wykonania, czyli sekwencji operacji potrzebnych do uzyskania wyniku.

Administrator może również kontrolować częstotliwość aktualizacji statystyk (np. `default_statistics_target`, `autovacuum`).

11.2 Mechanizm planowania w PostgreSQL

PostgreSQL wykorzystuje kosztowy optymalizator; przy użyciu statystyk (liczby wierszy, rozkładu danych) szacuje „koszt" różnych metod wykonania zapytania, wybierając tę, która jest najtańsza pod względem czasu i zasobów.

11.3 Statystyki i ich aktualizacja

- Statystyki są tworzone przy pomocy polecenia `ANALYZE` – zbiera dane o rozkładzie wartości kolumn.
- Mechanizm autovacuum odświeża statystyki automatycznie.

Przykład:

```
ANALYZE [nazwa_tabeli];
```

W systemach o dużym obciążeniu planowanie uwzględnia również równoległość (parallel query).

11.4 Typy planów wykonania

Przykładowe typy planów wykonania:

- **Seq Scan** – pełny skan tabeli (gdy indeksy są niedostępne lub nieefektywne).
- **Index Scan** – wykorzystanie indeksu.
- **Bitmap Index Scan** – łączenie efektywności indeksów ze skanem sekwencyjnym.
- **Nested Loop Join** – efektywny join dla małych zbiorów.
- **Hash Join** – buduje tablicę hash dla dużych zbiorów.
- **Merge Join** – stosowany, gdy dane są posortowane.

11.5 Jak sprawdzić plan zapytania?

Aby zobaczyć plan wybrany przez PostgreSQL, można użyć:

```
EXPLAIN ANALYZE SELECT * FROM tabela WHERE kolumna = 'wartość';
```

- `EXPLAIN` – wyświetla plan bez wykonania zapytania.
- `ANALYZE` – wykonuje zapytanie i podaje rzeczywiste czasy wykonania.

Przykładowy wynik:

```
Index Scan using idx_kolumna on tabela (cost=0.29..8.56 rows=3 width=244)
Index Cond: (kolumna = 'wartość'::text)
```

11.6 Parametry planowania i optymalizacji

W pliku `postgresql.conf` można konfigurować m.in.:

- `random_page_cost` – koszt odczytu strony z dysku SSD/HDD.
- `cpu_tuple_cost` – koszt przetwarzania pojedynczego wiersza.
- `enable_seqscan`, `enable_indexscan`, `enable_bitmapscan` – włączanie/wyłączanie konkretnych typów skanów.

Dostosowanie tych parametrów pozwala zoptymalizować planowanie zgodnie ze specyfiką sprzętu i obciążenia.

12. Tabele – rozmiar, planowanie i monitorowanie

12.1 Rozmiar tabeli

Rozmiar tabeli w PostgreSQL obejmuje dane (wiersze), strukturę, indeksy, dane TOAST oraz pliki statystyk. Do monitorowania rozmiaru stosuje się funkcje:

- `pg_relation_size()` – rozmiar tabeli lub pojedynczego indeksu.
- `pg_total_relation_size()` – całkowity rozmiar tabeli wraz z indeksami i TOAST.

12.2 Planowanie rozmiaru i jego kontrola

Podczas projektowania bazy danych należy oszacować potencjalny rozmiar tabel, biorąc pod uwagę liczbę wierszy i rozmiar pojedynczego rekordu. PostgreSQL nie posiada sztywnego limitu (poza ograniczeniami systemu plików i 32-bitowym limitem liczby stron). Parametr `fillfactor` może być stosowany do optymalizacji częstotliwości operacji UPDATE i VACUUM.

12.3 Monitorowanie rozmiaru tabel

Przykład zapytania:

```
SELECT pg_size_pretty(pg_total_relation_size('nazwa_tabeli'));
```

Inne funkcje:

- `pg_relation_size` – rozmiar samej tabeli.
- `pg_indexes_size` – rozmiar indeksów.
- `pg_table_size` – zwraca łączny rozmiar tabeli wraz z TOAST.

12.4 Planowanie na poziomie tabel

Administrator może wpływać na fizyczne rozmieszczenie danych poprzez:

- **Tablespaces** – przenoszenie tabel lub indeksów na inne dyski/partycje.
- **Podział tabel (partitioning)** – rozbijanie dużych tabel na mniejsze części.

12.5 Monitorowanie stanu tabel

Monitorowanie obejmuje:

- Śledzenie fragmentacji danych.
- Kontrolę wzrostu tabel i indeksów.
- Statystyki dotyczące operacji odczytów i zapisów.

Narzędzia i widoki systemowe:

- `pg_stat_all_tables`
- `pg_stat_user_tables`
- `pg_stat_activity`

12.6 Konserwacja i optymalizacja tabel

Regularne uruchamianie poleceń:

- **VACUUM** – usuwa martwe wiersze, zapobiegając nadmiernej fragmentacji.
- **ANALYZE** – aktualizuje statystyki, ułatwiając optymalizację zapytań.

Dla bardzo dużych tabel można stosować `VACUUM FULL` lub reorganizację danych, aby odzyskać przestrzeń.

13. Rozmiar pojedynczych tabel, rozmiar wszystkich tabel, indeksów tabeli

Efektywne zarządzanie rozmiarem tabel oraz ich indeksów ma kluczowe znaczenie dla wydajności systemu.

13.1 Rozmiar pojedynczej tabeli

Do pozyskania informacji o rozmiarze konkretnej tabeli służą funkcje:

- `pg_relation_size('nazwa_tabeli')` – rozmiar danych tabeli (w bajtach).
- `pg_table_size('nazwa_tabeli')` – rozmiar danych tabeli wraz z danymi TOAST.
- `pg_total_relation_size('nazwa_tabeli')` – całkowity rozmiar tabeli wraz z indeksami i TOAST.

Przykład zapytania:

```
SELECT
  pg_size_pretty(pg_relation_size('nazwa_tabeli')) AS data_size,
  pg_size_pretty(pg_indexes_size('nazwa_tabeli')) AS indexes_size,
  pg_size_pretty(pg_total_relation_size('nazwa_tabeli')) AS total_size;
```

13.2 Rozmiar wszystkich tabel w bazie

Zapytanie pozwalające wylistować wszystkie tabele i ich rozmiary:

```
SELECT
  schemaname,
  relname AS table_name,
  pg_size_pretty(pg_total_relation_size(relid)) AS total_size
FROM
  pg_catalog.pg_statio_user_tables
ORDER BY
  pg_total_relation_size(relid) DESC;
```

13.3 Rozmiar indeksów tabeli

Funkcja:

```
pg_indexes_size('nazwa_tabeli')
```

Pozwala sprawdzić rozmiar wszystkich indeksów przypisanych do danej tabeli. Monitorowanie indeksów pomaga w podejmowaniu decyzji o ich przebudowie lub usunięciu.

13.4 Znaczenie rozmiarów

Duże tabele i indeksy mogą powodować:

- Wolniejsze operacje zapisu i odczytu.
- Wydłużony czas tworzenia kopii zapasowych.
- Większe wymagania przestrzeni dyskowej.

Regularne monitorowanie rozmiaru umożliwia planowanie działań optymalizacyjnych i konserwacyjnych.

14. Rozmiar

Pojęcie „rozmiar" odnosi się do przestrzeni dyskowej zajmowanej przez elementy bazy danych – tabele, indeksy, pliki TOAST, a także całe bazy danych lub schematy.

14.1 Rodzaje rozmiarów w PostgreSQL

- **Rozmiar pojedynczego obiektu** (tabeli, indeksu): Funkcje takie jak `pg_relation_size()`, `pg_table_size()`, `pg_indexes_size()` oraz `pg_total_relation_size()`.
- **Rozmiar schematu lub bazy danych**: Funkcje `pg_namespace_size('nazwa_schematu')` oraz `pg_database_size('nazwa_bazy')`.
- **Rozmiar plików TOAST**: Duże wartości (np. teksty, obrazy) są przenoszone do struktur TOAST, których rozmiar wliczany jest do rozmiaru tabeli, choć można go analizować osobno.

14.2 Monitorowanie i kontrola rozmiaru

Administratorzy baz danych powinni regularnie monitorować rozmiar baz danych i jej obiektów, aby:

- Zapobiegać przekroczeniu limitów przestrzeni dyskowej.
- Wcześniej wykrywać problemy z fragmentacją.
- Planować archiwizację lub czyszczenie danych.

Do monitoringu można wykorzystać zapytania SQL lub narzędzia zewnętrzne (np. pgAdmin, pgBadger).

14.3 Optymalizacja rozmiaru

Działania optymalizacyjne obejmują:

- **Reorganizację i VACUUM**: odzyskiwanie przestrzeni po usuniętych lub zaktualizowanych rekordach oraz poprawa statystyk.
- **Partycjonowanie tabel**: dzielenie dużych tabel na mniejsze, co ułatwia zarządzanie.
- **Ograniczenia i typy danych**: odpowiedni dobór typów danych (np. `varchar(n)` zamiast `text`) oraz stosowanie ograniczeń (np. CHECK) zmniejsza rozmiar danych.

14.4 Znaczenie zarządzania rozmiarem

Niewłaściwe zarządzanie przestrzenią dyskową może prowadzić do:

- Spowolnienia działania bazy.
- Problemów z backupem i odtwarzaniem.
- Wzrostu kosztów utrzymania infrastruktury.

Podsumowanie

Zarządzanie konfiguracją bazy danych PostgreSQL, optymalizacja zapytań oraz monitorowanie i konserwacja tabel stanowią fundament skutecznego zarządzania systemem bazodanowym. Prawidłowe podejście do tych elementów zapewnia wysoką wydajność, niezawodność i skalowalność systemu.

—

2.2 Sprzet_dla_bazy_danych documentation

Add your content using `reStructuredText` syntax. See the reStructuredText documentation for details.

2.2.1 Sprzęt dla baz danych

Wstęp

Systemy zarządzania bazami danych (DBMS) są fundamentem współczesnych aplikacji i usług – od rozbudowanych systemów transakcyjnych, przez aplikacje internetowe, aż po urządzenia mobilne czy systemy wbudowane. W zależności od zastosowania i skali projektu, wybór odpowiedniego silnika bazodanowego oraz towarzyszącej mu infrastruktury sprzętowej ma kluczowe znaczenie dla zapewnienia wydajności, stabilności i niezawodności systemu

Sprzęt dla bazy danych PostgreSQL

PostgreSQL to potężny system RDBMS, ceniony za swoją skalowalność, wsparcie dla zaawansowanych zapytań i dużą elastyczność. Jego efektywne działanie zależy w dużej mierze od odpowiednio dobranej infrastruktury sprzętowej.

Procesor

PostgreSQL obsługuje wiele wątków, jednak pojedyncze zapytania zazwyczaj są wykonywane jednordzeniowo. Z tego względu optymalny procesor powinien cechować się zarówno wysokim taktowaniem jak i odpowiednią liczbą rdzeni do równoczesnej obsługi wielu zapytań. W środowiskach produkcyjnych najczęściej wykorzystuje się procesory serwerowe takie jak Intel Xeon czy AMD EPYC, które oferują zarówno wydajność, jak i niezawodność.

Pamięć operacyjna

RAM odgrywa istotną rolę w przetwarzaniu danych, co znacząco wpływa na wydajność operacji. PostgreSQL efektywnie wykorzystuje dostępne zasoby pamięci do cache'owania, dlatego im więcej pamięci RAM tym lepiej. W praktyce, minimalne pojemności dla mniejszych baz to około 16–32 GB, natomiast w środowiskach produkcyjnych i analitycznych często stosuje się od 64 GB do nawet kilkuset.

Przestrzeń dyskowa

Dyski twarde to krytyczny element wpływający na szybkość działania bazy. Zdecydowanie zaleca się korzystanie z dysków SSD (najlepiej NVMe), które zapewniają wysoką przepustowość i niskie opóźnienia. Warto zastosować konfigurację RAID 10, która łączy szybkość z redundancją.

Sieć internetowa

W przypadku PostgreSQL działającego w klastrach, środowiskach chmurowych lub przy replikacji danych, wydajne połączenie sieciowe ma kluczowe znaczenie. Standardem są interfejsy 1 Gb/s, lecz w dużych bazach danych stosuje się nawet 10 Gb/s i więcej. Liczy się nie tylko przepustowość, ale też niskie opóźnienia i niezawodność.

Zasilanie

Niezawodność zasilania to jeden z filarów bezpieczeństwa danych. Zaleca się stosowanie zasilaczy redundantnych oraz zasilania awaryjnego UPS, które umożliwia bezpieczne wyłączenie systemu w przypadku awarii. Można użyć własnych generatorów prądu.

Chłodzenie

Intensywna praca serwera PostgreSQL generuje duże ilości ciepła. Wydajne chłodzenie powietrzne, a często nawet cieczowe jest potrzebne by utrzymać stabilność systemu i przedłużyć żywotność komponentów. W profesjonalnych serwerowniach stosuje się zaawansowane systemy klimatyzacji i kontroli termicznej.

Sprzęt dla bazy danych SQLite

SQLite to lekki, samodzielny silnik bazodanowy, nie wymagający uruchamiania oddzielnego serwera. Znajduje zastosowanie m.in. w aplikacjach mobilnych, przeglądarkach internetowych, systemach IoT czy oprogramowaniu wbudowanym.

Procesor

SQLite działa lokalnie na urządzeniu użytkownika. Dla prostych operacji wystarczy procesor z jednym, albo dwoma rdzeniami. W bardziej wymagających zastosowaniach (np. filtrowanie dużych zbiorów danych) przyda się szybszy CPU. Wielowątkowość nie daje istotnych korzyści.

Pamięć operacyjna

SQLite potrzebuje niewielkiej ilości pamięci RAM w wielu przypadkach wystarcza 256MB do 1GB. Jednak dla komfortowej pracy z większymi zbiorami danych warto zapewnić nieco więcej pamięci, czyli 2 GB lub więcej, szczególnie w aplikacjach desktopowych lub mobilnych.

Przestrzeń dyskowa

Dane w SQLite zapisywane są w jednym pliku. Wydajność operacji zapisu/odczytu zależy od nośnika. Dyski SSD lub szybkie karty pamięci są preferowane. W przypadku urządzeń wbudowanych, kluczowe znaczenie ma trwałość nośnika, zwłaszcza przy częstym zapisie danych.

Sieć internetowa

SQLite nie wymaga połączeń sieciowych – działa lokalnie. W sytuacjach, gdzie dane są synchronizowane z serwerem lub przenoszone przez sieć (np. w aplikacjach mobilnych), znaczenie ma jakość połączenia (Wi-Fi, LTE), choć wpływa to bardziej na komfort użytkowania aplikacji niż na samą bazę.

Zasilanie

W systemach mobilnych i IoT efektywne zarządzanie energią jest kluczowe. Aplikacje powinny ograniczać zbędne operacje odczytu i zapisu, by niepotrzebnie nie obciążać procesora i nie zużywać baterii. W zastosowaniach stacjonarnych problem ten zazwyczaj nie występuje.

Chłodzenie

SQLite nie generuje dużego obciążenia cieplnego. W większości przypadków wystarczy pasywne chłodzenie w zamkniętych obudowach, lecz warto zadbać o minimalny przepływ powietrza.

Podsumowanie

Zarówno PostgreSQL, jak i SQLite pełnią istotne role w ekosystemie baz danych, lecz ich wymagania sprzętowe są diametralnie różne. PostgreSQL, jako system serwerowy, wymaga zaawansowanego i wydajnego sprzętu: mocnych procesorów, dużej ilości RAM, szybkich dysków, niezawodnej sieci, zasilania i chłodzenia. Z kolei SQLite działa doskonale na skromniejszych zasobach, stawiając na lekkość i prostotę implementacyjną. Dostosowanie sprzętu do konkretnego silnika DBMS i charakterystyki aplikacji pozwala nie tylko na osiągnięcie optymalnej wydajności, ale też gwarantuje stabilność i bezpieczeństwo działania całego systemu.

2.3 Indices and tables

- genindex
- modindex
- search

2.4 Wydajność, skalowanie i replikacja

Autorzy

- Mateusz Brokos
- Szymon Blatkowski
- Maciej Gołębiowski

2.4.1 Wstęp

Celem niniejszej pracy jest omówienie kluczowych zagadnień związanych z wydajnością, skalowaniem oraz replikacją baz danych. Współczesne systemy informatyczne wymagają wysokiej dostępności i szybkiego przetwarzania danych, dlatego odpowiednie mechanizmy replikacji i optymalizacji wydajności odgrywają istotną rolę w zapewnieniu niezawodnego działania aplikacji. Praca przedstawia różne podejścia do replikacji danych, sposoby testowania wydajności sprzętu oraz techniki zarządzania zasobami i kontrolowania dostępu użytkowników. Omówiono również praktyczne rozwiązania stosowane w popularnych systemach baz danych, takich jak MySQL i PostgreSQL.

2.4.2 Buforowanie oraz zarządzanie połączeniami

Buforowanie i zarządzanie połączeniami to kluczowe mechanizmy zwiększające wydajność i stabilność systemu.

Buforowanie połączeń:

- Unieważnienie (Inwalidacja) bufora: Proces usuwania nieaktualnych danych z pamięci podręcznej, aby aplikacja zawsze korzystała ze świeżych informacji. Może być wykonywana automatycznie (np. przez wygasanie danych) lub ręcznie przez aplikację.
- Buforowanie wyników: Polega na przechowywaniu rezultatów złożonych zapytań w pamięci podręcznej, co pozwala uniknąć ich wielokrotnego wykonywania i poprawia wydajność systemu, zwłaszcza przy operacjach na wielu tabelach.
- Zapisywanie wyników zapytań: Wyniki często wykonywanych zapytań są przechowywane w cache, dzięki czemu aplikacja może je szybko odczytać, co zmniejsza obciążenie bazy danych i przyspiesza odpowiedź.

Zarządzanie połączeniami:

- Monitorowanie parametrów połączeń: Śledzenie wskaźników takich jak czas reakcji, błędy łączenia i ilość przesyłanych danych. Regularne monitorowanie pozwala szybko wykrywać i usuwać problemy, zwiększając stabilność i wydajność systemu.
- Zarządzanie grupami połączeń: Utrzymywanie zestawu aktywnych połączeń, które mogą być wielokrotnie wykorzystywane. Ogranicza to konieczność tworzenia nowych połączeń, co poprawia wydajność i oszczędza zasoby.
- Obsługa transakcji: Kontrola przebiegu transakcji w bazie danych w celu zapewnienia spójności i integralności danych. Wszystkie operacje w transakcji są realizowane jako jedna niepodzielna jednostka, co zapobiega konfliktom.

2.4.3 Wydajność

Wydajność bazy danych to kluczowy czynnik wpływający na skuteczne zarządzanie danymi i funkcjonowanie organizacji. W dobie cyfrowej transformacji optymalizacja działania baz stanowi istotny element strategii IT. W tym rozdziale omówiono sześć głównych wskaźników wydajności: czas odpowiedzi, przepustowość, współbieżność, wykorzystanie zasobów, problem zapytań N+1 oraz błędy w bazie danych. Regularne monitorowanie tych parametrów i odpowiednie reagowanie zapewnia stabilność systemu i wysoką efektywność pracy. Zaniedbanie ich kontroli grozi spadkiem wydajności, ryzykiem utraty danych i poważnymi awariami.

Klastry oraz indeksy

- Klaster w bazie danych to metoda organizacji, w której powiązane tabele są przechowywane na tym samym obszarze dysku. Dzięki relacjom za pomocą kluczy obcych dane znajdują się blisko siebie, co skraca czas dostępu i zwiększa wydajność wyszukiwania.
- Indeks w bazie danych to struktura przypominająca spis treści, która pozwala szybko lokalizować dane w tabeli bez konieczności jej pełnego przeszukiwania. Tworzenie indeksów na kolumnach znacząco przyspiesza operacje wyszukiwania i dostępu.

1. Współbieżność w bazach danych

Współbieżność w bazach danych oznacza zdolność systemu do jednoczesnego przetwarzania wielu operacji, co ma kluczowe znaczenie tam, gdzie wielu użytkowników korzysta z bazy w tym samym czasie. Poziom współbieżności mierzy się m.in. liczbą transakcji na sekundę (TPS) i zapytań na sekundę (QPS).

Na wysoką współbieżność wpływają:

- Poziomy izolacji transakcji, które równoważą spójność danych i możliwość równoległej pracy – wyższe poziomy izolacji zwiększają dokładność, ale mogą ograniczać współbieżność przez blokady.
- Mechanizmy blokad, które minimalizują konflikty między transakcjami i zapewniają płynne działanie systemu.
- Architektura systemu, zwłaszcza rozproszona, umożliwiająca rozłożenie obciążenia na wiele węzłów i poprawę skalowalności.

Do głównych wyzwań należą:

- Hotspoty danych, czyli miejsca często jednocześnie odczytywane lub modyfikowane, tworzące wąskie gardła.
- Zakleszczenia, gdy transakcje wzajemnie się blokują, uniemożliwiając zakończenie pracy.

- Głód zasobów, kiedy niektóre operacje monopolizują zasoby, ograniczając dostęp innym procesom i obniżając wydajność.

2. Przepustowość bazy danych

Przepustowość bazy danych to miara zdolności systemu do efektywnego przetwarzania określonej liczby operacji w jednostce czasu. Im wyższa, tym więcej zapytań lub transakcji baza obsłuży szybko i sprawnie.

Na przepustowość wpływają:

- Współbieżność: Skuteczne zarządzanie transakcjami i blokadami pozwala na równoczesne operacje bez konfliktów, co jest ważne przy dużym obciążeniu (np. w sklepach internetowych).
- Bazy NoSQL: Często stosują model ewentualnej spójności, umożliwiając szybsze zapisy bez oczekiwania na pełną synchronizację replik.
- Dystrybuowanie danych: Techniki takie jak sharding (NoSQL) czy partycjonowanie (SQL) rozkładają dane na różne serwery, zwiększając zdolność przetwarzania wielu operacji jednocześnie.

Podsumowując, odpowiednie zarządzanie współbieżnością, wybór architektury i rozproszenie danych to klucz do wysokiej przepustowości bazy danych.

3. Responsywność bazy danych

Czasy odpowiedzi bazy danych są kluczowe w środowiskach wymagających szybkich decyzji, np. w finansach czy sytuacjach kryzysowych.

Na czas reakcji bazy wpływają:

- Architektura bazy: dobrze zaprojektowane partycjonowanie, indeksowanie oraz bazy działające w pamięci operacyjnej znacząco przyspieszają dostęp do danych.
- Topologia oraz stan sieci: opóźnienia, przepustowość i stabilność sieci w systemach rozproszonych wpływają na szybkość przesyłu danych; optymalizacja i kompresja zmniejszają te opóźnienia.
- Balansowanie obciążeń oraz dostęp równoczesny: pooling połączeń, replikacja i równoważenie obciążenia pomagają utrzymać krótkie czasy odpowiedzi przy dużym ruchu.

Szybkie odpowiedzi podnoszą efektywność, satysfakcję użytkowników i konkurencyjność systemu bazodanowego.

4. Zapytania N+1

Problem zapytań typu N+1 to częsta nieefektywność w aplikacjach korzystających z ORM, polegająca na wykonywaniu wielu zapytań – jednego głównego i osobnego dla każdego powiązanego rekordu. Na przykład, pobranie 10 użytkowników i osobne zapytanie o profil dla każdego daje łącznie 11 zapytań.

Przyczyny to:

- Błędna konfiguracja ORM, szczególnie „leniwe ładowanie”, powodujące nadmiar zapytań.
- Nieoptymalne wzorce dostępu do danych, np. pobieranie danych w pętlach.
- Niewykorzystanie złączeń SQL (JOIN), które pozwalają na pobranie danych w jednym zapytaniu.

5. Błędy w bazach danych

Błędy wpływające na wydajność bazy danych to istotny wskaźnik kondycji systemu.

Najczęstsze typy błędów to:

- Błędy składni zapytań – wynikają z niepoprawnej składni SQL, powodując odrzucenie zapytania.
- Błędy połączenia – problemy z nawiązaniem połączenia, często przez awarie sieci, błędne konfiguracje lub awarie serwera.
- Błędy limitów zasobów – gdy system przekracza dostępne zasoby (dysk, CPU, pamięć), co może spowalniać lub zatrzymywać działanie.
- Naruszenia ograniczeń – próby wstawienia danych łamiących zasady integralności (np. duplikaty tam, gdzie wymagana jest unikalność).
- Błędy uprawnień i zabezpieczeń – brak odpowiednich praw dostępu skutkuje odmową operacji na danych.

Skuteczna identyfikacja i usuwanie tych błędów jest kluczowa dla stabilności i wydajności bazy danych.

6. Zużycie dostępnych zasobów

Zużycie zasobów w bazach danych to kluczowy czynnik wpływający na ich wydajność.

Najważniejsze zasoby to:

- CPU: Odpowiada za przetwarzanie zapytań i zarządzanie transakcjami. Nadmierne obciążenie może wskazywać na przeciążenie lub nieoptymalne zapytania.
- Operacje I/O na dysku: Odczyt i zapis danych. Wysoka liczba operacji może oznaczać słabe buforowanie; efektywne cache'owanie zmniejsza potrzebę częstego dostępu do dysku i eliminuje wąskie gardła.
- Pamięć RAM: służy do przechowywania często używanych danych i buforów. Jej niedobór lub złe zarządzanie powoduje korzystanie z wolniejszej pamięci dyskowej, co obniża wydajność.

Dobre zarządzanie CPU, pamięcią i operacjami dyskowymi jest niezbędne dla utrzymania wysokiej wydajności i stabilności systemu bazodanowego.

Prostota rozbudowy:

Bazy danych SQL typu scale-out umożliwiają liniową skalowalność przez dodawanie nowych węzłów do klastra bez przestojów i zmian w aplikacji czy sprzęcie. Każdy węzeł aktywnie przetwarza transakcje, a logika bazy jest przenoszona do tych węzłów, co ogranicza transfer danych w sieci i redukuje ruch. Tylko jeden węzeł obsługuje zapisy dla danego fragmentu danych, eliminując rywalizację o zasoby, co poprawia wydajność w porównaniu do tradycyjnych baz, gdzie blokady danych spowalniają system przy wielu operacjach jednocześnie.

Analityka czasu rzeczywistego:

Analityka czasu rzeczywistego w Big Data umożliwia natychmiastową analizę danych, dając firmom przewagę konkurencyjną. Skalowalne bazy SQL pozwalają na szybkie przetwarzanie danych operacyjnych dzięki technikom działającym w pamięci operacyjnej i wykorzystującym szybkie dyski SSD, bez potrzeby stosowania skomplikowanych rozwiązań. Przykłady Google (baza F1 SQL w Adwords) i Facebooka pokazują, że relacyjne bazy danych są efektywne zarówno w OLTP, jak i OLAP, a integracja SQL z ekosystemem Hadoop zwiększa możliwości analityczne przy jednoczesnym ograniczeniu zapotrzebowania na specjalistów.

Dostępność w chmurze:

Organizacje wymagają nieprzerwanej pracy aplikacji produkcyjnych, co zapewnia ciągłość procesów biznesowych. W przypadku awarii chmury szybkie przywrócenie bazy danych bez utraty danych jest kluczowe. Skalowalne bazy SQL realizują to poprzez mechanizmy wysokiej dostępności, które opierają się na replikacji wielu kopii danych, minimalizując ryzyko ich utraty.

Unikanie wąskich gardeł:

W skalowalnych bazach danych SQL rozwiązano problem logu transakcyjnego, który w tradycyjnych systemach często stanowił wąskie gardło wydajności. W klasycznych rozwiązaniach wszystkie rekordy muszą być najpierw zapisane w logu transakcyjnym przed zakończeniem zapytania. Niewłaściwa konfiguracja lub awarie mogą powodować nadmierne rozrosty logu, czasem przekraczające rozmiar samej bazy, co skutkuje znacznym spowolnieniem operacji zapisu, nawet przy użyciu szybkich dysków SSD.

2.4.4 Skalowanie

Bazy danych SQL nie są tak kosztowne w rozbudowie, jak się często uważa, ponieważ oferują możliwość skalowania poziomego. Ta cecha jest szczególnie cenna w analizie danych biznesowych, gdzie rośnie potrzeba przetwarzania danych klientów z wielu źródeł w czasie rzeczywistym. Obok tradycyjnych rozwiązań dostępne są również bazy NoSQL, NewSQL oraz platformy oparte na Hadoop, które odpowiadają na różne wyzwania związane z przetwarzaniem dużych ilości danych. Skalowanie poziome z optymalnym balansem pomiędzy pamięcią RAM a pamięcią flash pozwala osiągnąć wysoką wydajność. Przykłady nowoczesnych skalowalnych baz SQL, takich jak InfiniSQL, ClustrixDB czy F1, potwierdzają, że tradycyjne bazy SQL mogą efektywnie skalować się wszerz.

2.4.5 Replikacja

Replikacja danych to proces kopiowania informacji między różnymi serwerami baz danych, który przynosi wiele korzyści: - Zwiększenie skalowalności – obciążenie systemu jest rozdzielane między wiele serwerów; zapisy i aktualizacje odbywają się na jednym serwerze, natomiast odczyty i wyszukiwania na innych, co poprawia wydajność. - Poprawa bezpieczeństwa – tworzenie kopii bazy produkcyjnej pozwala chronić dane przed awariami sprzętu, choć nie zabezpiecza przed błędnymi operacjami wykonywanymi na bazie (np. DROP TABLE). - Zapewnienie separacji środowisk – kopia bazy może być udostępniona zespołom programistycznym i testerskim, umożliwiając pracę na izolowanym środowisku bez ryzyka wpływu na bazę produkcyjną. - Ułatwienie analizy danych – obciążające analizy i obliczenia mogą być wykonywane na oddzielnym serwerze, dzięki czemu nie obciążają głównej bazy danych i nie wpływają na jej wydajność.

Mechanizmy replikacji

Replikacja w bazach danych polega na kopiowaniu i synchronizowaniu danych oraz obiektów z serwera głównego (master) na serwer zapasowy (slave), aby zapewnić spójność i wysoką dostępność danych.

Mechanizm replikacji MySQL działa w następujący sposób: - Serwer główny zapisuje wszystkie zmiany w plikach binarnych (bin-logach), które zawierają instrukcje wykonane na masterze. - Specjalny wątek na masterze przesyła bin-logi do serwerów slave. - Wątek SQL, który odczytuje relay-logi i wykonuje zapisane w nich zapytania, aby odtworzyć zmiany w lokalnej bazie. - Wątek I/O, który odbiera bin-logi i zapisuje je do relay-logów (tymczasowych plików na slave). Podsumowując, replikacja w MySQL polega na automatycznym przesyłaniu i odtwarzaniu zmian, dzięki czemu baza na serwerze zapasowym jest na bieżąco synchronizowana z bazą główną.

Rodzaje mechanizmów replikacji

- Replikacja oparta na zapisie (Write-Ahead Logging): Ten typ replikacji jest często wykorzystywany w systemach takich jak PostgreSQL. Polega na tym, że zmiany w transakcjach są najpierw zapisywane w dzienniku zapisu, a następnie jego zawartość jest kopiowana na serwery repliki.
- Replikacja oparta na zrzutach (Snapshot-Based Replication): W niektórych systemach stosuje się okresowe tworzenie pełnych zrzutów bazy danych, które są przesyłane do serwerów repliki.
- Replikacja oparta na transakcjach (Transaction-Based Replication): W tym modelu każda transakcja jest przekazywana i odtwarzana na serwerach repliki, co sprawdza się w systemach wymagających silnej spójności.
- Replikacja asynchroniczna i synchroniczna: W replikacji asynchronicznej dane najpierw trafiają do głównej bazy, a potem na repliki. W replikacji synchronicznej zapisy są wykonywane jednocześnie na serwerze głównym i replikach.
- Replikacja dwukierunkowa (Bi-Directional Replication): Pozwala na wprowadzanie zmian na dowolnym z serwerów repliki, które są synchronizowane z pozostałymi, co jest szczególnie użyteczne w systemach o wysokiej dostępności.

PostgreSQL oferuje różne metody replikacji, w tym opartą na zapisie (WAL), asynchroniczną, synchroniczną oraz replikację logiczną. Mechanizm WAL zapewnia bezpieczeństwo danych przez zapisywanie wszystkich zmian w dzienniku przed ich zastosowaniem i przesyłanie go na repliki. W trybie asynchronicznym dane trafiają najpierw na serwer główny, a potem na repliki, natomiast w trybie synchronicznym zapisy są realizowane jednocześnie. Dodatkowo, replikacja logiczna umożliwia kopiowanie wybranych tabel lub baz, co jest przydatne w przypadku bardzo dużych zbiorów danych.

Zalety i Wady replikacji

Zalety:

- Zwiększenie wydajności i dostępności: Replikacja pozwala rozłożyć obciążenie zapytań na wiele serwerów, co poprawia wydajność systemu. Użytkownicy mogą kierować zapytania do najbliższych serwerów repliki, skracając czas odpowiedzi. W przypadku awarii jednego serwera pozostałe repliki kontynuują obsługę zapytań, zapewniając wysoką dostępność.
- Ochrona danych: Replikacja wspiera tworzenie kopii zapasowych i odzyskiwanie danych. W razie awarii głównej bazy replika może służyć jako źródło do odtworzenia informacji.
- Rozproszenie danych geograficzne: Umożliwia przenoszenie danych do różnych lokalizacji. Międzynarodowa firma może replikować dane między oddziałami, co pozwala lokalnym użytkownikom na szybki dostęp.
- Wsparcie analizy i raportowania: Dane z replik mogą być wykorzystywane do analiz i raportów, co odciąża główną bazę danych i utrzymuje jej wysoką wydajność.

Wady:

- Replikacja nie gwarantuje, że po wykonaniu operacji dane na serwerze głównym zostaną w pełni odzwierciedlone na serwerze zapasowym.
- Mechanizm nie chroni przed skutkami działań, takich jak przypadkowe usunięcie tabeli (DROP TABLE).

2.4.6 Kontrola dostępu i limity systemowe

Limity systemowe w zarządzaniu bazami danych określają maksymalną ilość zasobów, które system jest w stanie obsłużyć. Są one ustalane przez system zarządzania bazą danych (DBMS) i zależą od zasobów sprzętowych oraz konfiguracji. Na przykład w Azure SQL Database limity zasobów różnią się w zależności od wybranego poziomu cenowego. W MySQL maksymalny rozmiar tabeli jest zwykle ograniczony przez parametry systemu operacyjnego dotyczące wielkości plików.

Kontrola dostępu użytkowników w DBMS to mechanizm umożliwiający lub blokujący dostęp do danych. Składa się z dwóch elementów: uwierzytelniania, czyli potwierdzania tożsamości użytkownika, oraz autoryzacji, czyli ustalania jego uprawnień. Wyróżnia się modele takie jak Kontrola Dostępu Uzależniona (DAC), Obowiązkowa (MAC), oparta na Rolach (RBAC) czy na Atrybutach (ABAC).

PostgreSQL oferuje narzędzia do zarządzania limitami systemowymi i kontrolą dostępu. Administratorzy mogą ustawiać parametry takie jak maksymalna liczba połączeń, limity pamięci, maksymalny rozmiar pliku danych czy wielkość tabeli. W zakresie kontroli dostępu PostgreSQL zapewnia mechanizmy uwierzytelniania i autoryzacji. Administratorzy mogą tworzyć role i nadawać uprawnienia dotyczące baz danych, schematów, tabel i kolumn. PostgreSQL obsługuje uwierzytelnianie oparte na hasłach i certyfikatach SSL, umożliwiając skuteczne zarządzanie bezpieczeństwem i poufnością danych.

2.4.7 Testowanie wydajności sprzętu na poziomie OS

Testy wydajności kluczowych komponentów sprzętowych na poziomie systemu operacyjnego są niezbędne do optymalizacji działania baz danych. Obejmują oceny pamięci RAM, procesora (CPU) oraz dysków twardych (HDD) i SSD — elementów mających największy wpływ na szybkość i efektywność systemu. Analiza wyników pomaga wskazać elementy wymagające modernizacji lub optymalizacji, co pozwala podnieść ogólną wydajność systemu bazodanowego, niezależnie od używanego oprogramowania.

Testy pamięci RAM pozwalają zmierzyć jej szybkość i stabilność, co przekłada się na wydajność bazy danych. W tym celu często stosuje się narzędzia takie jak MemTest86.

Testy procesora oceniają jego moc obliczeniową i zdolność do przetwarzania zapytań. Popularnym programem jest Cinebench R23.

Testy dysków sprawdzają szybkość operacji odczytu i zapisu, co jest kluczowe, ponieważ baza danych przechowuje dane na nośnikach dyskowych. Do pomiarów wykorzystuje się narzędzia takie jak CrystalDiskMark 8 czy Acronis Drive Monitor.

2.4.8 Podsumowanie

W pracy przedstawiono kluczowe zagadnienia związane z zarządzaniem bazami danych, w tym rodzaje replikacji, metody kontroli dostępu użytkowników, limity systemowe oraz znaczenie testów wydajności komponentów sprzętowych. Omówiono zalety i wady replikacji, takie jak zwiększenie dostępności czy ryzyko niespójności danych. Scharakteryzowano mechanizmy uwierzytelniania i autoryzacji, które zapewniają bezpieczeństwo informacji, oraz wskazano, jak limity zasobów wpływają na działanie systemu. Zwrócono także uwagę na rolę testów pamięci RAM, procesora i dysków w optymalizacji wydajności środowiska bazodanowego. Całość podkreśla znaczenie świadomego projektowania i utrzymywania infrastruktury baz danych w celu zapewnienia jej niezawodności, bezpieczeństwa i wysokiej efektywności pracy.

2.4.9 Bibliografia

[1] PostgreSQL Documentation – Performance Tips https://www.postgresql.org/docs/current/performance-tips.html

[2] SQLite Documentation – Query Optimizer Overview https://sqlite.org/optoverview.html

[3] F. Hecht, Scaling Database Systems https://www.cockroachlabs.com/docs/stable/scaling-your-database.html

[4] DigitalOcean, How To Optimize Queries and Tables in PostgreSQL https://www.digitalocean.com/community/tutorials/how-to-optimize-queries-and-tables-in-postgresql

[5] PostgreSQL Documentation – High Availability, Load Balancing, and Replication https://www.postgresql.org/docs/current/different-replication-solutions.html

[6] SQLite Documentation – How Indexes Work https://www.sqlite.org/queryplanner.html

[7] Redgate, The Importance of Database Performance Testing https://www.red-gate.com/simple-talk/sql/performance/the-importance-of-database-performance-testing/

[8] Materiały kursowe przedmiotu „Bazy Danych”, Politechnika Wrocławska, Piotr Czaja.

2.5 Bezpieczeństwo

Autorzy

- Katarzyna Tarasek
- Błażej Uliasz

2.5.1 1. pg_hba.conf — opis pliku konfiguracyjnego PostgreSQL

Plik `pg_hba.conf` (skrót od *PostgreSQL Host-Based Authentication*) kontroluje, kto może się połączyć z bazą danych PostgreSQL, skąd, i w jaki sposób ma zostać uwierzytelniony.

Format pliku

Każdy wiersz odpowiada jednej regule dostępu:

```
<typ>  <baza danych>  <użytkownik>  <adres>  <metoda>  [opcje]
```

Opis elementów:

Znaczenie Elementów

- `<typ>` — Typ połączenia – np. `local`, `host`, `hostssl`, `hostnossl`
- `<baza>` — Nazwa bazy danych, do której ma być dostęp – konkretna lub `all`
- `<użytkownik>` — Nazwa użytkownika PostgreSQL lub `all`
- `<adres>` — Adres IP lub zakres CIDR klienta (np. `192.168.1.0/24`); pomijany dla `local`
- `<metoda>` — Metoda uwierzytelnienia – np. `md5`, `trust`, `scram-sha-256`
- `[opcje]` — Opcjonalne dodatkowe parametry (np. `clientcert=1`)

Typy połączeń

- `local` — Umożliwia połączenia **lokalne przez Unix socket** (pliki specjalne w systemie plików, np. `/var/run/postgresql/.s.PGSQL.5432`). Ten tryb jest dostępny **tylko na systemach Unix/Linux** i ignoruje pole `<adres>`.
- `host` — Oznacza połączenia **przez TCP/IP**, niezależnie od tego, czy klient znajduje się na tym samym hoście, czy w sieci. Wymaga podania adresu IP lub zakresu IP (w polu `<adres>`).
- `hostssl` — Jak `host`, ale **wymusza użycie SSL/TLS**. Połączenia bez szyfrowania będą odrzucone. Wymaga, aby serwer PostgreSQL był poprawnie skonfigurowany do obsługi SSL (np. pliki `server.crt`, `server.key`).
- `hostnossl` — Jak `host`, ale **odrzuca połączenia przez SSL/TLS**. Działa tylko dla połączeń nieszyfrowanych. Może być używane do rozróżnienia reguł dla klientów z/do SSL i bez SSL.

Metody uwierzytelniania

- `trust` — brak uwierzytelnienia (niezalecane!)
- `md5` — Klient musi podać hasło, które jest przesyłane jako skrót MD5. To popularna metoda w starszych wersjach PostgreSQL, ale obecnie uznawana za przestarzałą (choć nadal obsługiwana).
- `scram-sha-256` — Nowoczesna, bezpieczna metoda uwierzytelniania oparta na protokole SCRAM i algorytmie SHA-256. Zalecana w produkcji od PostgreSQL 10 wzwyż. Wymaga, aby hasła w systemie były zapisane jako SCRAM, a nie MD5.
- `peer` — Tylko dla połączeń `local`. Sprawdza, czy nazwa użytkownika systemowego (OS) pasuje do użytkownika PostgreSQL. Stosowane w systemach Unix/Linux.
- `ident` — Tylko dla połączeń TCP/IP. Wymaga usługi ident (lub pliku mapowania ident), aby ustalić, kto próbuje się połączyć. Bardziej złożona i rzadziej używana niż `peer`.
- `reject` — Zawsze odrzuca połączenie. Może być użyte do celowego blokowania określonych adresów lub użytkowników.

Przykładowy wpis

```
# 1. Lokalny dostęp bez hasła
local   all             postgres                                peer
```

Zmiany i przeładowanie

Po zmianach w pliku należy przeładować konfigurację PostgreSQL:

```
pg_ctl reload
-- lub:
SELECT pg_reload_conf();
```

2.5.2 2. Uprawnienia użytkownika

PostgreSQL pozwala na bardzo precyzyjne zarządzanie uprawnieniami użytkowników lub roli poprzez wiele poziomów dostępu — od globalnych uprawnień systemowych, przez bazy danych, aż po pojedyncze kolumny w tabelach.

Poziom systemowy

To najwyższy poziom uprawnień, nadawany roli jako atrybut. Dotyczy całego klastra PostgreSQL:

- *SUPERUSER* — Pełna kontrola nad serwerem, obejmuje wszystkie uprawnienia
- *CREATEDB* — Możliwość tworzenia nowych baz danych
- *CREATEROLE* — Tworzenie i zarządzanie rolami/użytkownikami
- *REPLICATION* — Umożliwia replikację danych (logiczna/strumieniowa)
- *BYPASSRLS* — Omija polityki RLS (Row-Level Security)

Poziom bazy danych

Uprawnienia do konkretnej bazy danych:

- *CONNECT* — Pozwala na połączenie z bazą danych
- *CREATE* — Pozwala na tworzenie schematów w tej bazie
- *TEMP* — Możliwość tworzenia tymczasowych tabel

Poziom schematu

Schemat (np. *public*) to kontener na tabele, funkcje, typy. Uprawnienia:

- *USAGE* — Umożliwia dostęp do schematu (bez tego SELECT/INSERT nie zadziała)
- *CREATE* — Pozwala tworzyć obiekty (np. tabele) w schemacie

Poziom tabeli

Uprawnienia do całej tabeli :

- *SELECT* — Odczyt danych
- *INSERT* — Wstawianie danych
- *UPDATE* — Modyfikacja danych
- *DELETE* — Usuwanie danych

Przykład

```
GRANT SELECT, UPDATE ON employees TO hr_team;
REVOKE DELETE ON employees FROM kontraktorzy;
```

2.5.3 3. Zarządzanie użytkownikami a dane wprowadzone

Zarządzanie użytkownikami w PostgreSQL dotyczy tworzenia, usuwania i modyfikowania użytkowników. Sytuacja na którą trzeba tutaj zwrócić uwagę jest usuwanie użytkonika ale pozostawienie danych, które wprowadził.

Tworzenie i modyfikacja użytkowników

Do tworzenia nowych użytkowników używamy polecenia `CREATE USER`. Do modyfikowania użytkowników, którzy już istnieją, używamy polecenia `ALETER USER`:

```
CREATE USER username WITH PASSWORD 'password';
ALTER USER username WITH PASSWORD 'new_password';
```

Usuwanie użytkowników

Do usuwania użytkowników, używamy polecenia ``DROP USER`:

```
DROP USER username;
```

Dane wprowadzone przez uśytkownika np. za pomocą polecenia `INSERT` pozostają, nawet jeśli jego konto zostało usunięte.

Usunięcie użytkownika, a dane które posiadał

Po usunięciu używtkonika dane, które posiadał nie są automatycznie usuwane. Dane te pozostają w bazie danych ale stają się „niedostępne” dla tego użytkownika. Aby się ich pozbyć, musi to zrobić użytkownik który ma do nich uprawnienia, korzystając z plecenia `DROP`.

Usunięcie użytkowników, a obietky

Usuniecie użytkownika, który jest właścicielem obiektów, wygląda inaczej niż przy wcześniejszych danych. Jeżeli użytkownik jest właścicielem jakiegoś obiektu, to jego usunięcie skutkuje błędem:

```
ERROR: role "username" cannot be droped becouse some objects depend on it
```

Aby zapobiec takim błędom stosujemy poniższe rozwiazanie:

```
REASSIGN OWNED BY username TO nowa_rola;
DROP OWNER BY username;
DROP ROLE username;
```

2.5.4 4. Zabezpieczenie połączenia przez SSL/TLS

TLS (Transport Layer Security) i jego poprzednik SSL (Secure Sockets Layer) to kryptograficzne protokoły służące do zabezpieczania połączeń sieciowych. W PostgreSQL służą one do szyfrowania transmisji danych pomiędzy klientem a serwerem, uniemożliwiając podsłuch, modyfikację lub podszywanie się pod jedną ze stron.

Konfiguracja SSL/TLS w PostgreSQL

Konfiguracja serwera: musimy edytować dwa pliki i zrestartować serwer PostgreSQL. Plik `postgresql.conf`:

```
ssl = on
ssl_cert_file = 'server.crt'
ssl_key_file = 'server.key'
ssl_ca_file = 'root.crt'
ssl_min_protocol_version = 'TLSv1.3'
```

oraz «»pg_hba.conf»»:

```
hostssl all all 0.0.0.0/0 cert
```

Generowanie certyfikatów: jeśli nie używamy komercyjnego CA, możemy sami go wygerenować, a pomocą poniższych komend:

```
openssl genrsa -out server.key 2048
openssl req -new -key server.key -out server.csr
openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

Konfiguracja klienta: parametry SSL, których możemy użyć.

- `sslmode` - kontroluje wymuszanie i weryfikację SSL (`require`, `verify-ca`, `verify-full`)
- `sslcert` - ścieżka do certyfikatu klienta (jeśli wymagane uwierzytelnienie certyfikatem)
- `sslkey` - klucz prywatny klienta
- `sslrootcert` - certyfikat CA do weryfikacji certyfikatu serwera

Monitorowanie i testowanie SSL/TLS

Sprawdzenie czy połączenie jest szyfrowanie w PostgreSQL wystarczy użyć prostego polecenia `SELECT ssl_is_used();`. Jeśli jednak chcemy dostać więcej informacji, musimy wpisać poniższe polecenia:

```
SELECT datname, usename, ssl, client_addr, application_name, backend_type
FROM pg_stat_ssl
JOIN pg_stat_activity ON pg_stat_ssl.pid = pg_stat_activity.pid
ORDER BY ssl;
```

Testowanie z poziomu terminala pozwala podejrzeć szczegóły TLS takie jak certyfikaty, wesję protokołu czy użyty szyft. Wpisujemy poniższą komendę:

```
openssl s_client -starttls postgres -connect example.com:5432 -showcerts
```

2.5.5 5. Szyfrowanie danych

Szyfrowanie danych w PostgreSQL odgrywa kluczową rolę w zapewnianiu poufności, integralności i ochrony danych przed nieautoryzowanym dostępem. Można je realizować na różnych poziomach: transmisji (in-transit), przechowywania (at-rest) oraz aplikacyjnym.

Szyfrowanie transmisji

Korzystając z technologi SSL/TLS chroni dane przesyłane pomiędzy klientem, a serwerem przed podsłuchiwaniem lub modyfikacją. Wymaga konfiguracji serwera PostgreSQL do obsługi SSL oraz klienci muszą łączyć się przez SSL.

Szyfrowanie całego dysku

Dane są szyfrowane na poziomie systemu operacyjnego lub warstwy przechowywania. Stosowanymi roziazaniami jest LUKS, BitLocker, szyfrowanie oferowane przez chmury. Zaletami tego szyfrowania jest transparentność dla PostgrSQL i łatwość w implementacji. Wadami za to jest brak selektywnego szyfrowania oraz fakt, że jeśli system jest aktywny to dane są odszyfrowane i dostępne.

Szyfrowanie na poziomie kolumn z użyciem pgcrypto

Pozwala na szyfrowanie konkretnych kolumn danych. Rozszerzenie to `pgcrypto`. Funkcje takiego szyfrowania to:

- symetryczne szyfrowanie

```
SELECT pgp_sym_encrypt('tajne dane', 'haslo');
SELECT pgp_sym_decrypt(kolumna::bytea, 'haslo');
```

- asymetryczne szyfrowanie (z uśyciem kluczy publicznych/prywatnych)
- haszowanie

```
SELECT digest('haslo', 'sha256');
```

Zaletami tego szyfrowania jest duża elastyczność i selektywne szyfrowanie. Wadami zaś wydajność i konieczność zarządzania kluczami w aplikacji.

Szyfrowanie na poziomie aplikacji

Dane są szyfrowane przed zapisaniem do bazy danych i odszyfrowywane po odczycie. Używane biblioteki:

- Python – cryptography, pycryptodome,
- Java – javax.crypto, Bouncy Castle,
- JavaScript – crypto, sjcl.

Zaletami jest pełna kontrola nad szyfrowaniem oraz fakt, że dane są chronione nawet w razie włamania do bazy. Wadami zaś trudniejsze wyszukiwanie i indeksowanie, konieczność przeniesienia odpowiedzialności za bezpieczeństwo do aplikacji oraz problemy ze zgodnością przy migracjach danych.

Zarządzanie kluczami szyfrującymi

Niezależnie od rodzaju szyfrowania, bezpieczne zarządzanie kluczami jest kluczowe dla ochrony danych. Klucze powinny być generowane, przechowywane, dystrybuowane i niszczone w sposób bezpieczny. Potrzebne są do tego odpowiednie narzędzia. Rekomendowanymi narzędziami do bezpiecznego zarządzania kluczami są:

- Sprzętowe moduły bezpieczeństwa (HSM) - Urządzenia te oferują bezpieczne środowisko do generowania, przechowywania i zarządzania kluczami. HSM-y są odporne na fizyczne ataki i zapewniają wysoki poziom bezpieczeństwa.
- Systemy zarządzania kluczami (KMS) - KMS to oprogramowanie, które centralizuje zarządzanie kluczami, umożliwiając ich bezpieczne przechowywanie, rotację i dystrybucję.
- Narzędzia do bezpiecznej komunikacji - Narzędzia takie jak Signal czy WhatsApp oferują szyfrowanie end-to-end, które chroni komunikację przed nieautoryzowanym dostępem.
- Narzędzia do szyfrowania dysków - Takie jak BitLocker czy FileVault, które pozwalają na zaszyfrowanie całego dysku twardego lub jego partycji.

2.6 Monitorowanie i diagnostyka

Autorzy

- Dominika Półchłopek
- Kacper Rasztar
- Grzegorz Szczepanek

2.6.1 Wstęp

Monitorowanie i diagnostyka baz danych PostgreSQL stanowią fundamentalne elementy zapewniające wydajność, bezpieczeństwo oraz stabilność środowiska produkcyjnego. Nowoczesne rozwiązania monitorowania umożliwiają administratorom proaktywne wykrywanie problemów, optymalizację wydajności oraz zapewnienie zgodności z przepisami bezpieczeństwa. Efektywne monitorowanie PostgreSQL obejmuje szeroki zakres metryk - od aktywności sesji użytkowników, przez analizę operacji na danych, po szczegółowe śledzenie logów systemowych i zasobów na poziomie systemu operacyjnego.

2.6.2 Monitorowanie Sesji i Użytkowników

Analiza Aktywności Użytkowników

Systematyczne obserwowanie działań wykonywanych przez użytkowników bazy danych stanowi podstawę skutecznego monitorowania PostgreSQL. Kluczowym narzędziem w tym obszarze jest widok systemowy pg_stat_activity, który umożliwia śledzenie bieżących zapytań, czasu ich trwania oraz identyfikowanie użytkowników i aplikacji korzystających z bazy. Widok pg_stat_activity przedstawia informacje o aktywnych procesach serwera wraz ze szczegółami dotyczącymi powiązanych sesji użytkowników i zapytań. Każdy wiersz w tym widoku reprezentuje proces serwera z danymi o bieżącym stanie połączenia bazy danych.

Praktyczne zastosowanie pg_stat_activity obejmuje monitorowanie aktywności w czasie rzeczywistym oraz generowanie powiadomień w przypadku wykrycia nieprawidłowości. Narzędzia takie jak pgAdmin, Zabbix czy Prometheus wykorzystują te dane do wizualizacji i automatyzacji procesów monitorowania. Administratorzy mogą wykorzystywać proste zapytania SQL do analizy aktywności, na przykład: „SELECT * FROM pg_stat_activity;” pozwala na wyświetlenie wszystkich aktywnych sesji.

Zarządzanie Zasobami i Limity

Efektywne zarządzanie zasobami w PostgreSQL opiera się na odpowiedniej konfiguracji parametrów systemowych takich jak max_connections czy work_mem, które kontrolują liczbę jednoczesnych połączeń i wykorzystanie pamięci. Monitorowanie wykorzystania zasobów realizowane jest poprzez narzędzia systemowe jak top, htop, vmstat czy iostat w środowisku Linux, a także dedykowane rozwiązania do monitorowania baz danych.

Kluczowe metryki obejmują współczynnik trafień cache'u, który powinien utrzymywać się powyżej 99% dla systemów produkcyjnych. Zapytanie sprawdzające ten wskaźnik: „SELECT sum(heap_blks_hit) / (sum(heap_blks_hit) + sum(heap_blks_read)) as ratio FROM pg_statio_user_tables;" pozwala na ocenę efektywności wykorzystania pamięci.

Wykrywanie Problemów z Blokadami

Identyfikacja i analiza blokad stanowi istotny element zapewniający płynność działania aplikacji. PostgreSQL udostępnia widok pg_locks umożliwiający śledzenie blokad i konfliktów między transakcjami. Specjalistyczne narzędzia jak pganalyze oferują zaawansowane funkcje monitorowania blokad z automatycznym wykrywaniem sytuacji deadlock oraz powiadomieniami o potencjalnych zagrożeniach .

Podstawowe zapytanie do identyfikacji nieprzyznanych blokad: „SELECT relation::regclass, * FROM pg_locks WHERE NOT granted;" pozwala na szybkie wykrycie problemów. Bardziej zaawansowane analizy wymagają łączenia informacji z widoków pg_locks i pg_stat_activity w celu identyfikacji procesów blokujących i blokowanych.

2.6.3 Monitorowanie Dostępu do Tabel i Operacji na Danych

Analiza Użycia Danych

Administratorzy baz danych wykorzystują narzędzia monitorujące takie jak pg_stat_user_tables w PostgreSQL do zrozumienia wzorców wykorzystania tabel oraz identyfikacji najczęściej wykonywanych operacji. Analiza tych danych pozwala zidentyfikować najbardziej obciążone tabele, ocenić rozkład ruchu oraz przewidzieć przyszłe potrzeby związane z rozbudową infrastruktury.

Narzędzia do wizualizacji jak Grafana czy Prometheus umożliwiają prezentację trendów w użyciu tabel i pomagają w planowaniu optymalizacji. Kompleksowe monitorowanie obejmuje śledzenie operacji SELECT, INSERT, UPDATE, DELETE oraz analizę wzorców dostępu do danych w różnych okresach czasowych.

Wykrywanie Nieprawidłowych Zapytań

Do wykrywania zapytań o długim czasie wykonania lub wysokim zużyciu zasobów wykorzystuje się rozszerzenie pg_stat_statements, które pozwala monitorować wydajność zapytań, analizować plany wykonania i identyfikować operacje wymagające optymalizacji. Moduł pg_stat_statements zapewnia śledzenie statystyk planowania i wykonania wszystkich instrukcji SQL wykonywanych przez serwer.

Konfiguracja pg_stat_statements wymaga dodania modułu do shared_preload_libraries w postgresql.conf oraz restartu serwera. Widok pg_stat_statements zawiera po jednym wierszu dla każdej unikalnej kombinacji identyfikatora bazy danych, użytkownika i zapytania, do maksymalnej liczby różnych instrukcji, które moduł może śledzić.

Bezpieczeństwo i Zgodność

Śledzenie dostępu do tabel jest kluczowe z punktu widzenia bezpieczeństwa oraz zgodności z przepisami takimi jak RODO czy PCI DSS. W PostgreSQL do audytu operacji na danych służy rozszerzenie pgaudit, które pozwala rejestrować szczegółowe informacje o działaniach na poziomie zapytań i transakcji. PGAudit zapewnia narzędzia potrzebne do tworzenia logów audytowych wymaganych do przejścia określonych audytów rządowych, finansowych lub certyfikacji ISO.

Systemy takie jak ELK Stack czy Splunk umożliwiają centralizację i analizę logów oraz konfigurację alertów na podejrzane działania, co wzmacnia bezpieczeństwo środowiska bazodanowego. Automatyczne powiadomienia można skonfigurować dla zdarzeń takich jak próby logowania poza godzinami pracy lub masowe operacje na wrażliwych tabelach.

2.6.4 Monitorowanie Logów i Raportowanie Błędów

Analiza Logów Systemowych

PostgreSQL generuje szczegółowe logi systemowe i dzienniki błędów stanowiące podstawowe źródło informacji o stanie bazy danych. Dzienniki rejestrują wszelkie błędy, ostrzeżenia, nietypowe zdarzenia oraz informacje o operacjach wykonywanych przez użytkowników i aplikacje, obejmując kody błędów, czas wystąpienia problemu, tekst zapytania SQL oraz szczegóły środowiska wykonania.

Regularna analiza logów pozwala administratorom na szybkie wykrywanie i rozwiązywanie problemów przed ich wpływem na użytkowników końcowych. Do analizy wykorzystuje się narzędzia takie jak ELK Stack (Elasticsearch, Logstash, Kibana), Splunk, pgBadger czy wbudowane funkcje PostgreSQL. pgBadger stanowi szczególnie efektywne narzędzie - jest to szybki analizator logów PostgreSQL napisany w Perl, który przetwarza dane wyjściowe logów na raporty HTML z szczegółowymi informacjami o wydajności.

Automatyczne Raportowanie i Alerty

Automatyzacja raportowania i alertowania stanowi kluczowy element szybkiego reagowania na incydenty. Narzędzia takie jak pgAdmin, Zabbix, Prometheus czy Grafana umożliwiają konfigurację reguł automatycznego generowania raportów oraz wysyłania powiadomień przy wykryciu określonych zdarzeń.

Skuteczne alertowanie wymaga ostrożnego ustawiania progów i właściwej priorytetyzacji. Alerty o wysokim priorytecie obejmują opóźnienia replikacji przekraczające 2 minuty, liczę połączeń przekraczającą 85% max_connections oraz współczynnik trafień cache'u spadający poniżej 98% dla systemów produkcyjnych. Powiadomienia mogą być wysyłane poprzez e-mail, SMS, Slack lub inne kanały komunikacji.

Konfiguracja Logowania dla pgBadger

Aby efektywnie wykorzystać pgBadger, logowanie w PostgreSQL powinno być skonfigurowane w sposób zapewniający maksimum informacji. Podstawowe ustawienia konfiguracyjne w postgresql.conf obejmują: log_checkpoints = on, log_connections = on, log_disconnections = on, log_lock_waits = on, log_temp_files = 0, log_autovacuum_min_duration = 0.

Szczególnie wartościowe są raporty wolnych zapytań generowane przez pgBadger, które polegają na ustawieniu log_min_duration_statement. pgBadger może przetwarzać logi PostgreSQL niezależnie od tego, czy są to syslog, stderr czy csvlog, o ile linie logów zawierają wystarczające informacje w prefiksie .

2.6.5 Monitorowanie na Poziomie Systemu Operacyjnego

Narzędzia Systemowe

Monitorowanie zasobów systemowych takich jak procesor, pamięć, dysk i sieć jest kluczowe dla zapewnienia stabilnej pracy PostgreSQL. W środowisku Linux administratorzy wykorzystują narzędzia takie jak top (wyświetlające listę procesów i zużycie zasobów w czasie rzeczywistym), htop (oferujące graficzne przedstawienie obciążenia), iostat (monitorujące statystyki wejścia/wyjścia) oraz vmstat (dostarczające informacji o pamięci i aktywności procesora).

W środowisku Windows popularne narzędzia obejmują Menedżer zadań umożliwiający monitorowanie użycia CPU, pamięci, dysku i sieci przez poszczególne procesy oraz Monitor systemu (Performance Monitor) - zaawansowane narzędzie do śledzenia wielu wskaźników wydajności. Te narzędzia umożliwiają szybkie wykrywanie i diagnozowanie problemów z wydajnością zarówno na poziomie systemu operacyjnego, jak i samej bazy danych.

Efektywne monitorowanie systemu wymaga śledzenia kluczowych metryk: wykorzystania CPU (wysokie użycie może ograniczać przetwarzanie zapytań), CPU steal time (szczególnie w środowiskach zwirtualizowanych), wykorzystania pamięci przez PostgreSQL oraz ogólnego obciążenia pamięci systemu. Krytyczne jest unikanie wykorzystania swap przez PostgreSQL, ponieważ drastycznie pogarsza to wydajność.

Integracja z Narzędziami Zewnętrznymi

PostgreSQL doskonale integruje się z zaawansowanymi narzędziami monitorowania infrastruktury IT, umożliwiającymi centralizację nadzoru oraz automatyzację reakcji na incydenty. Nagios, popularny system monitorowania infrastruktury, pozwala na monitorowanie stanu serwerów, usług, zasobów sprzętowych oraz sieci z konfiguracją alertów powiadamiających o przekroczeniu progów wydajności.

Prometheus stanowi narzędzie do zbierania i przechowywania metryk współpracujące z wieloma eksporterami, w tym dedykowanymi dla PostgreSQL. OpenTelemetry Collector oferuje nowoczesne podejście, działając jako agent pobierający dane telemetryczne z systemów i eksportujący je do backendu OpenTelemetry. Grafana zapewnia zaawansowaną wizualizację danych, umożliwiając tworzenie interaktywnych dashboardów prezentujących kluczowe wskaźniki wydajności PostgreSQL.

2.6.6 Narzędzia Monitorowania PostgreSQL

Narzędzia Open Source

Ekosystem narzędzi open source dla PostgreSQL jest bogaty i różnorodny. pgAdmin oferuje graficzny interfejs do administrowania bazami danych z funkcjami monitorowania aktywności serwera, wydajności zapytań oraz obiektów bazy danych. Dashboard serwera w pgAdmin dostarcza przegląd ważnych metryk, w tym wykorzystania CPU, pamięci, miejsca na dysku i aktywnych połączeń.

pgBadger stanowi jedną z najpopularniejszych opcji - to szybki analizator logów PostgreSQL zbudowany dla wydajności, który tworzy szczegółowe raporty w formacie HTML5 z dynamicznymi wykresami. Najnowsza wersja pgBadger 13.0 wprowadza nowe funkcje, w tym konfigurowalne histogramy czasów zapytań i sesji. Narzędzie jest idealne do zrozumienia zachowania serwerów PostgreSQL i identyfikacji zapytań wymagających optymalizacji.

PGWatch reprezentuje kolejne zaawansowane rozwiązanie - to elastyczne, samodzielne narzędzie do monitorowania metryk PostgreSQL oferujące instalację w jedną minutę przy użyciu Dockera. PGWatch charakteryzuje się nieinwazyjną konfiguracją, intuicyjną prezentacją metryk przy użyciu Grafany oraz łatwą rozszerzalnością poprzez definiowanie metryk w czystym SQL.

Rozwiązania Komercyjne

DataDog APM zapewnia komercyjną platformę monitorowania i analizy ze specjalistyczną integracją PostgreSQL. Platforma oferuje łatwą w użyciu integrację PostgreSQL umożliwiającą zbieranie i monitorowanie metryk wydajności bez ręcznej instrumentacji. Agent DataDog automatycznie pobiera metryki PostgreSQL udostępniane przez serwer, obejmując połączenia z bazą danych, wydajność zapytań, statystyki puli buforów oraz status replikacji.

Sematext Monitoring skupia się na logach, infrastrukturze, śledzeniu i monitorowaniu wydajności nie tylko dla PostgreSQL, ale także dla wielu innych baz danych. Rozwiązanie oferuje łatwy w konfiguracji agent PostgreSQL oraz wbudowaną integrację logów PostgreSQL pozwalającą identyfikować wolne zapytania, błędy i ostrzeżenia.

pganalyze stanowi wyspecjalizowane narzędzie monitorowania PostgreSQL umożliwiające optymalizację i analizę zapytań, łatwe monitorowanie bieżących zapytań w czasie rzeczywistym oraz zbieranie planów zapytań. Dzięki kompleksowym danym o wydajności zapytań pganalyze pozwala szybko identyfikować przyczyny problemów i sprawdzać skuteczność wdrożonych rozwiązań.

Zabbix dla PostgreSQL

Zabbix stanowi open-source'owe rozwiązanie monitorowania obsługujące PostgreSQL poprzez wbudowane szablony i niestandardowe skrypty. System opiera się na agentach instalowanych na systemach docelowych - w przypadku PostgreSQL wymaga konfiguracji agenta Zabbix na serwerze PostgreSQL.

Implementacja Zabbix dla PostgreSQL wymaga stworzenia użytkownika monitorowania z odpowiednimi prawami dostępu. Dla PostgreSQL w wersji 10 i wyższej: „CREATE USER zbx_monitor WITH PASSWORD «<PASSWORD>» INHERIT; GRANT pg_monitor TO zbx_monitor;". Po zaimportowaniu szablonu PostgreSQL Zabbix automatycznie zbiera metryki takie jak liczba połączeń, wskaźniki transakcji, wydajność zapytań i inne.

2.6.7 Najlepsze Praktyki Monitorowania

Ustanawianie Baselines Wydajności

Tworzenie baselines wydajności stanowi fundament skutecznego wykrywania anomalii. Bez zrozumienia normalnych wzorców zachowania identyfikacja problematycznych odchyleń staje się zgadywaniem zamiast analizy opartej na danych. Kompleksowe ustalanie baselines wymaga zbierania metryk w różnych ramach czasowych i wzorcach obciążenia, obejmując dzienne wzorce (szczczyty w godzinach biznesowych i nocne przetwarzanie), tygodniowe różnice oraz miesięczne i sezonowe wariacje.

Dla każdego wzorca należy dokumentować wskaźniki przepustowości zapytań, poziomy wykorzystania zasobów, zakresy liczby połączeń, wskaźniki transakcji oraz rozkłady zdarzeń oczekiwania. Zaleca się zbieranie co najmniej trzech cykli każdego typu wzorca przed ustaleniem wartości progowych.

Korelacja Metryk Międzysystemowych

Problemy wydajności PostgreSQL rzadko występują w izolacji. Najbardziej wartościowe implementacje monitorowania korelują metryki z różnych podsystemów w celu ujawnienia związków przyczynowo-skutkowych. Efektywne strategie korelacji obejmują łączenie metryk wykonania zapytań z metrykami zasobów systemowych, korelację zdarzeń wdrożeniowych aplikacji z metrykami wydajności bazy danych oraz analizę metryk przy użyciu spójnych okien czasowych.

Implementacja zwykle wymaga ujednoliconego oznaczania czasowego w systemach monitorowania, spójnego tagowania metadanych dla usług i komponentów oraz scentralizowanego logowania zdarzeń systemowych. Narzędzia wizualizacji powinny obsługiwać nakładanie różnych typów metryk w celu efektywnej analizy.

Konfiguracja Efektywnych Alertów

Strategie alertowania wymagają starannego ustawiania progów i właściwej priorytetyzacji. Alerty o wysokim priorytecie wymagające natychmiastowej akcji obejmują opóźnienia replikacji przekraczające 2 minuty, liczę połączeń przekraczającą 85% max_connections, wskaźniki wycofywania transakcji powyżej 10% utrzymujące się przez 5+ minut oraz przestrzeń dyskową poniżej 15% na wolumenach bazy danych.

Alerty o średnim priorytecie wymagające badania obejmują czasy zapytań przekraczające 200% historycznych baselines, nietypowy wzrost użycia plików tymczasowych, rozdęcie tabel przekraczające 30% rozmiaru tabeli oraz brak działania autovacuum przez 24+ godziny. Implementacja wielopoziomowego alertowania z progami ostrzeżeń na poziomie 70-80% wartości krytycznych zapewnia wczesne powiadomienie o rozwijających się problemach.

2.6.8 Monitorowanie Wysokiej Dostępności

Monitorowanie Statusu Replikacji

Monitorowanie klastrów PostgreSQL o wysokiej dostępności wymaga dodatkowych wymiarów poza monitorowaniem pojedynczej instancji. Kluczowe obszary obejmują śledzenie opóźnienia replikacji w jednostkach bajtów i czasu, monitorowanie wskaźnika generowania WAL na głównej instancji w porównaniu do wskaźnika odtwarzania na replikach oraz sprawdzanie akumulacji slotów replikacji, które mogą powodować zapełnienie dysku.

Zapytanie monitorujące opóźnienie replikacji: „SELECT application_name, pg_wal_lsn_diff(pg_current_wal_lsn(), replay_lsn) AS lag_bytes FROM pg_stat_replication;” pozwala na wykrywanie rosnącego opóźnienia wskazującego, że repliki nie nadążają za instancją główną. Regularne testowanie możliwości promocji repliki oraz monitorowanie mechanizmów automatycznego failover jest kluczowe dla gotowości na awarię.

Weryfikacja Spójności

Implementacja niezależnego monitorowania każdego węzła klastra z osobną instancją monitorowania poza klastrem bazy danych zapewnia widoczność podczas problemów z całym klastrem. Okresowe sprawdzenie spójności danych między instancją główną a replikami, monitorowanie konfliktów replikacji w konfiguracjach replikacji logicznej oraz śledzenie sum kontrolnych tabel są kluczowe dla utrzymania integralności danych.

Monitorowanie rozkładu połączeń obejmuje śledzenie liczby połączeń na głównej instancji i replikach odczytu, monitorowanie konfiguracji load balancera oraz weryfikację możliwości failover w connection stringach aplikacji. Sprawdzanie nieodpowiednich zapisów kierowanych do replik pomaga uniknąć błędów aplikacyjnych podczas przełączeń.

2.6.9 Wniosek

Monitorowanie i diagnostyka PostgreSQL stanowią kompleksowy proces wymagający holistycznego podejścia obejmującego multiple warstwy systemu. Skuteczna implementacja łączy monitorowanie na poziomie bazy danych, systemu operacyjnego oraz aplikacji, wykorzystując zarówno narzędzia wbudowane w PostgreSQL, jak i zewnętrzne rozwiązania specjalistyczne. Kluczem do sukcesu jest ustanowienie solidnych baseline'ów wydajności, implementacja inteligentnego systemu alertów oraz regularna analiza trendów umożliwiająca proaktywne zarządzanie zasobami i optymalizację wydajności przed wystąpieniem problemów krytycznych.

2.6.10 Bibliografia:

1. https://betterstack.com/community/comparisons/postgresql-monitoring-tools/
2. https://uptrace.dev/tools/postgresql-monitoring-tools
3. https://documentation.red-gate.com/pgnow
4. https://last9.io/blog/monitoring-postgres/
5. https://stackoverflow.com/questions/17654033/how-to-use-pg-stat-activity
6. https://pganalyze.com/blog/postgres-lock-monitoring
7. https://www.pgaudit.org
8. https://www.postgresql.org/docs/current/pgstatstatements.html
9. https://github.com/darold/pgbadger
10. https://hevodata.com/learn/elasticsearch-to-postgresql/
11. https://www.zabbix.com/integrations/postgresql
12. https://sematext.com/blog/postgresql-monitoring/
13. https://www.alibabacloud.com/help/en/analyticdb/analyticdb-for-postgresql/use-cases/use-pg-stat-activity-to-analyze-and-diagnose-active-sql-queries
14. https://wiki.postgresql.org/wiki/Lock_Monitoring
15. https://severalnines.com/blog/postgresql-log-analysis-pgbadger/
16. https://pgwatch.com
17. https://www.depesz.com/2022/07/05/understanding-pg_stat_activity/
18. https://www.postgresql.org/about/news/pgbadger-v124-released-2772/
19. https://docs.yugabyte.com/preview/explore/observability/pg-stat-activity/
20. https://www.postgresql.org/about/news/pgbadger-130-released-2975/
21. https://techdocs.broadcom.com/us/en/vmware-tanzu/data-solutions/tanzu-greenplum/6/greenplum-database/ref_guide-system_catalogs-pg_stat_activity.html
22. https://www.postgresql.org/docs/current/monitoring.html
23. https://www.reddit.com/r/PostgreSQL/comments/1auy79s/suggestions_for_postgresql_monitoring_tool/
24. https://wiki.postgresql.org/wiki/Monitoring
25. https://www.site24x7.com/learn/postgres-monitoring-guide.html
26. https://www.softwareandbooz.com/introducing-pgnow/
27. https://www.postgresql.org/docs/current/monitoring-stats.html
28. https://docs.dhis2.org/fr/topics/tutorials/analysing-postgresql-logs-using-pgbadger.html
29. https://dev.to/full_stack_adi/pgbadger-postgresql-log-analysis-made-easy-54ki
30. https://support.nagios.com/kb/article/xi-5-10-0-and-newer-postgress-to-mysql-conversion-560.html
31. https://github.com/melli0505/Docker-ELK-PostgreSQL

CHAPTER 3

Sprawozdanie:Projektowanie bazy danych - modele

author
Piotr Kotuła

Prowadzący: dr inż. Piotr Czaja

3.1 Wprowadzenie

Celem tego dokumentu jest zaprezentowanie struktury bazy danych odzwierciedlającej funkcjonowanie małej organizacji aukcyjnej. Na bazie PostgreSQL zbudowano modele encji i relacji, po czym oceniono wydajność kluczowych zapytań i wskazano sposoby ich optymalizacji.

3.2 Model Konceptualny

```
  +-----------+      1       *      +------------+
  |   Klient  |----------------------|   Aukcja   |
  +-----------+                      +------------+
  | PK: dowód |                      | PK: id_tx  |
  +-----------+                      +------------+
       ^  \
       |   \ nabywa (0..1)
wystawia   \
(1..*)      v
  +-----------+
  |   Aukcja  |
  +-----------+
  |    ...      |
  +-----------+
       ^
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
      |  prowadzony przez
      |       (1..*)
      v
  +-------------+
  | Pracownik   |
  +-------------+
  | PK: id_prac |
  +-------------+
```

Legenda:

- Klient – może wystawiać wiele aukcji (1..*)
- Aukcja – ma dokładnie jednego sprzedawcę i opcjonalnie jednego nabywcę (0..1)
- Pracownik – prowadzi wiele aukcji (1..*)

3.3 Model Logiczny

```
+-------------------+       +--------------------+       +-------------------+
|     Pracownik     |       |      Klient        |       |    LogAukcji      |
+-------------------+       +--------------------+       +-------------------+
| PK: numer_pracown |       | PK: numer_dowodu   |       | PK: id_transak    |
| imie              |       | imie               |       | sprzedawca (FK)   |
| nazwisko          |       | nazwisko           |       | nabywca   (FK)    |
+-------------------+       | ilosc_wyst_przedm  |       | data_transakcji   |
                            | wymaga_kaucji      |       | cena_sprzedazy    |
                            +--------------------+       | numer_pracow (FK) |
                                                         +-------------------+
```

Połączenia:

- LogAukcji.sprzedawca → Klient.numer_dowodu (1:N)
- LogAukcji.nabywca → Klient.numer_dowodu (0..1:N)
- LogAukcji.numer_pracownika → Pracownik.numer_pracownika (1:N)

3.4 Model Fizyczny

```
CREATE TABLE Pracownik (
    numer_pracownika SERIAL PRIMARY KEY,
    imie       VARCHAR(50) NOT NULL,
    nazwisko   VARCHAR(50) NOT NULL
);

CREATE TABLE Klient (
    numer_dowodu                  VARCHAR(20) PRIMARY KEY,
    imie                          VARCHAR(50) NOT NULL,
    nazwisko                      VARCHAR(50) NOT NULL,
    ilosc_wystawionych_przedmiotow INT DEFAULT 0,
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    wymaga_kaucji                BOOLEAN DEFAULT FALSE
);

CREATE TABLE LogAukcji (
    id_transakcji    SERIAL PRIMARY KEY,
    sprzedawca       VARCHAR(20) NOT NULL,
    nabywca          VARCHAR(20),
    data_transakcji  DATE NOT NULL,
    cena_sprzedazy   NUMERIC(10,2) NOT NULL DEFAULT 0,
    numer_pracownika INT NOT NULL,
    CONSTRAINT fk_sprzedawca FOREIGN KEY (sprzedawca) REFERENCES Klient(numer_dowodu),
    CONSTRAINT fk_nabywca FOREIGN KEY (nabywca) REFERENCES Klient(numer_dowodu),
    CONSTRAINT fk_pracownik FOREIGN KEY (numer_pracownika) REFERENCES Pracownik(numer_
↪pracownika)
);
```

3.5 Przykładowe rekordy

3.5.1 Tabela Pracownik

numer_pracownika	imię	nazwisko
1	Anna	Kowalska
2	Marek	Nowak
3	Ewa	Zielińska
…	…	…

3.5.2 Tabela Klient

numer_dowodu	imię	nazwisko	ilosc_wystawionych_przedmiotów	wymaga_kaucji
ID100001	Jan	Kowalski	3	1
ID100002	Anna	Nowak	5	0
…	…	…	…	…

3.5.3 Tabela LogAukcji

id_transakcji	sprzedawca	nabywca	data_transakcji	cena_sprzedazy	numer_pracownika
1	ID100001	ID100050	2025-06-01	150.75	1
2	ID100002	ID100049	2025-06-02	220.00	2
…	…	…	…	…	…

CHAPTER 4

Analiza Bazy danych i optymalizacja zapytań

4.1 Analiza normalizacji

Schemat jest zgodny z 3NF: - Każda tabela ma klucz główny. - Wszystkie atrybuty są bezpośrednio zależne od klucza. - Brak redundancji poza LogAukcji (wskazania na ID klientów), co sprzyja JOIN-om.

Ewentualna denormalizacja — np. dodanie imienia i nazwiska do LogAukcji — wymaga uzasadnienia korzyści wydajnościowych.

4.2 Potencjalne problemy wydajnościowe

- Brak indeksów poza kluczami głównymi
- JOIN-y na sprzedawca/nabywca oraz filtrowanie po data_transakcji mogą wymagać pełnych skanów
- LogAukcji rośnie — przy dużej liczbie rekordów wydajność spada
- Złożone zapytania agregujące (np. raporty miesięczne) będą kosztowne bez wsparcia

4.3 Strategie optymalizacji

Indeksy:

```
CREATE INDEX idx_log_sprzedawca ON LogAukcji(sprzedawca);
CREATE INDEX idx_log_nabywca ON LogAukcji(nabywca);
CREATE INDEX idx_log_data ON LogAukcji(data_transakcji);
CREATE INDEX idx_log_cena_data ON LogAukcji(data_transakcji, cena_sprzedazy);
```

Partycjonowanie: - Partycjonuj LogAukcji po data_transakcji (np. miesiąc lub rok) - Ułatwia archiwizację i przyspiesza zapytania zakresowe

Widoki materializowane:

```sql
CREATE MATERIALIZED VIEW mv_miesieczna_sprzedaz AS
SELECT date_trunc('month', data_transakcji) AS m,
       COUNT(*) AS cnt,
       SUM(cena_sprzedazy) AS suma
FROM LogAukcji
GROUP BY 1;
```

- Odświeżaj np. codziennie

Optymalizacja zapytań:

- Korzystaj z *EXPLAIN (ANALYZE, BUFFERS)* do analizy kosztów
- Unikaj *SELECT **
- Jeśli masz wiele schematów: używaj w pełni kwalifikowanych nazw

4.4 Przykład optymalizacji konkretnego zapytania

Przed optymalizacją:

```sql
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC;
```

Po optymalizacji: – indeks na (sprzedawca, data_transakcji) już utworzony .. code-block:: sql

    EXPLAIN ANALYZE SELECT k.imie, k.nazwisko, COUNT(*) AS cnt FROM LogAukcji l JOIN Klient k ON l.sprzedawca = k.numer_dowodu WHERE l.data_transakcji BETWEEN «2025-06-01» AND «2025-12-31» GROUP BY k.imie, k.nazwisko ORDER BY cnt DESC LIMIT 10; – jeśli potrzebujemy tylko top 10

4.5 Wykorzystanie EXPLAIN w PostgreSQL do analizy wydajności zapytań

EXPLAIN to podstawowe narzędzie w PostgreSQL do poznania, jak silnik bazodanowy realizuje zapytanie. W wersji z `ANALYZE` i `BUFFERS` pozwala zmierzyć realne czasy i zużycie I/O.

4.6 Formy EXPLAIN

- `EXPLAIN` <zapytanie> Pokaże szacowany plan (bez wykonania).
- `EXPLAIN ANALYZE` <zapytanie> Wykona zapytanie, zwracając rzeczywiste czasy (actual).
- Dodatek `BUFFERS` Pokazuje liczbę odczytanych/zapisanych bloków (shared hit/read/write, temp).
- Dodatki `VERBOSE`, `FORMAT JSON` lub `FORMAT YAML` Pełniejsze opisy node'ów i struktury planu (łatwiejsze parsowanie skryptami).

Przykład:

```
EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
SELECT ...;
```

4.7 Kluczowe metryki w EXPLAIN ANALYZE

Metryka	Co mówi	Punkt kontroli
startup cost	czas uruchomienia node'a	wczesne agregacje / wyszukiwanie
total cost	szacowany koszt wykonania	najdroższy etap w planie
actual time (first)	czas pierwszego wiersza	opóźnienia dostępu do danych
actual time (last)	czas przetwarzania wierszy	wolne sortowania / łączenia
rows vs loops	liczba wierszy vs liczba wywołań	niska selektywność filtrów/joinów
buffers read/hit	odczyty z dysku vs cache	niewystarczający cache
temp read/write	użycie plików tymczasowych	duże sorty / HashJoin bez pamięci
planning time	czas budowy planu	skomplikowane zapytania/statystyki
execution time	czas wykonania zapytania	kryterium SLA

4.8 Jak interpretować plan?

1. Przejdź drzewo node'ów od góry (root) do liści:
   a. Zidentyfikuj najwyższy `actual total cost` vs `estimated cost` – to może być wąskie gardło.
   b. Porównaj `rows` vs `actual rows` – duża rozbieżność może oznaczać nieaktualne statystyki.
2. Zwróć uwagę na typ node'a:
   - `Seq Scan` → możliwy dodatek indeksu
   - `Index Scan` → indeks działa
   - `Nested Loop`, `Hash Join`, `Merge Join` → dobierz strategię joinów do danych
3. Sprawdź zużycie buforów:
   - duża liczba `shared read` → praca na dysku
   - `temp read/write` → może wymagać większego `work_mem`

4.9 Systematyczny proces analizy planu

- **Wyznacz kluczowe zapytania** Raporty, dashboardy, API endpointy.
- **Przeanalizuj bez indeksów** `EXPLAIN (ANALYZE, BUFFERS)` i zapisz czasy oraz ilości odczytanych bloków.
- **Wdroż indeksy lub refaktoryzuj zapytania** Porównaj metryki „przed–po".
- **Dokumentuj** - Fragmenty `JSON/YAML` do analizy skryptami - Wybrane node'y z najwyższym kosztem
- **Ustal punkty alarmowe** Dla execution_time, temp space, zużycia I/O
- **Automatyzuj** - Włącz `auto_explain` w pliku `postgresql.conf` - Loguj plany przekraczające próg czasu - Analizuj dane historyczne z `pg_stat_statements` lub narzędzi jak `pgBadger`, `pganalyze`

4.10 Wnioski

- `EXPLAIN ANALYZE` to nie tylko plan, ale też **realne metryki**: czas wykonania, I/O, cache.
- Systematyczne porównywanie metryk **przed i po** wdrożeniu indeksów/zmian daje wymierne wyniki.
- Kluczowe punkty kontroli: - `execution_time` - `buffers` - rozbieżność między `rows` a `actual rows`
- Automatyzacja (np. `auto_explain` i `pg_stat_statements`) pozwala wykrywać problemy zanim pojawią się skargi od użytkowników.

4.11 Analiza i optymalizacja na danych SQLite (SQLite in-memory)

Przykładowe zapytanie:

```
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC;
```

Dzięki indeksom na `(data_transakcji, sprzedawca)` oraz `Klient(numer_dowodu)`, SQLite wykonuje index-seek, a następnie szybki sort/LIMIT zamiast pełnego skanu tabeli.

4.12 Indeksy w SQLite

Dodajemy indeksy:

```
CREATE INDEX IF NOT EXISTS idx_la_sprzedawca    ON LogAukcji(sprzedawca);
CREATE INDEX IF NOT EXISTS idx_la_nabywca       ON LogAukcji(nabywca);
CREATE INDEX IF NOT EXISTS idx_la_data          ON LogAukcji(data_transakcji);
CREATE INDEX IF NOT EXISTS idx_la_data_spr      ON LogAukcji(data_transakcji,␣
↪sprzedawca);
```

4.13 Przykłady pomiaru czasu wykonania

1. Pomiar przed optymalizacją

```
start_time = time.time()
cursor.execute("""
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC
""")
results = cursor.fetchall()
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
end_time = time.time()
print(f"Czas wykonania zapytania: {end_time - start_time:.4f} sekundy")
```

Wynik: `0.0019 sekundy`

2. Pomiar z optymalizacją LIMIT

```
start_time = time.time()
cursor.execute("""
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC
LIMIT 10;
""")
results = cursor.fetchall()
end_time = time.time()
print(f"Czas wykonania zapytania: {end_time - start_time:.4f} sekundy")
```

Wynik: `0.0013 sekundy`

3. Pomiar po dodaniu indeksów

```
cursor.execute("CREATE INDEX IF NOT EXISTS idx_log_sprzedawca ON LogAukcji(sprzedawca);")
cursor.execute("CREATE INDEX IF NOT EXISTS idx_log_nabywca ON LogAukcji(nabywca);")
cursor.execute("CREATE INDEX IF NOT EXISTS idx_log_data ON LogAukcji(data_transakcji);")
cursor.execute("CREATE INDEX IF NOT EXISTS idx_log_data_sprzed ON LogAukcji(data_
↪transakcji, sprzedawca);")

start_time = time.time()
cursor.execute("""
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC
LIMIT 10;
""")
results = cursor.fetchall()
end_time = time.time()
print(f"Czas wykonania zapytania: {end_time - start_time:.4f} sekundy")
```

Wynik: `0.0005 sekundy`

4.13.1 Wnioski:

Wprowadzenie indeksów oraz LIMIT istotnie zmniejsza czas wykonania zapytania w SQLite nawet na niewielkiej próbce. Przy większych zbiorach danych efekty te są jeszcze bardziej wyraźne.

4.14 Pomiar wydajności na przykładowym zapytaniu

Przykład zapytania:

```
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC;
```

4.14.1 ograniczenie ilości wyników

```
Limit 10;
```

4.14.2 Indeksowanie

Po załadowaniu danych, w celu przyspieszenia zapytań zawierających filtry i JOIN-y, należy dodać indeksy:

```
-- Przyspieszenie filtrów i JOIN-ów na LogAukcji
CREATE INDEX idx_log_sprzedawca  ON LogAukcji(sprzedawca);
CREATE INDEX idx_log_nabywca     ON LogAukcji(nabywca);
CREATE INDEX idx_log_data        ON LogAukcji(data_transakcji);

-- Kompozytowy indeks dla częstych filtrów po dacie i sprzedawcy
CREATE INDEX idx_log_data_sprzed ON LogAukcji(data_transakcji, sprzedawca);
```

4.14.3 PostgreSQL optymalizacja zapytań

Pomiar czasu bez LIMIT

```
explain_limit = """
EXPLAIN (ANALYZE, BUFFERS)
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC
"""
with engine.begin() as conn:
    result = conn.execute(text(explain_limit))
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    for row in result:
        print(row[0])
```

Wynik:

```
Planning:
  Buffers: shared hit=7
Planning Time: 0.099 ms
Execution Time: 0.266 ms
```

Pomiar z LIMIT 10

```
explain_limit = """
EXPLAIN (ANALYZE, BUFFERS)
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC
LIMIT 10;
"""
with engine.begin() as conn:
    result = conn.execute(text(explain_limit))
    for row in result:
        print(row[0])
```

Wynik

```
Planning:
  Buffers: shared hit=3
Planning Time: 0.169 ms
Execution Time: 0.478 ms
```

Pomiar z dodanymi indeksami

```
create_indexes = """
CREATE INDEX IF NOT EXISTS idx_log_sprzedawca  ON LogAukcji(sprzedawca);
CREATE INDEX IF NOT EXISTS idx_log_nabywca     ON LogAukcji(nabywca);
CREATE INDEX IF NOT EXISTS idx_log_data        ON LogAukcji(data_transakcji);
CREATE INDEX IF NOT EXISTS idx_log_data_sprzed ON LogAukcji(data_transakcji, sprzedawca);
"""
with engine.begin() as conn:
    conn.execute(text(create_indexes))

explain_limit = """
EXPLAIN (ANALYZE, BUFFERS)
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
LIMIT 10;
"""
with engine.begin() as conn:
    result = conn.execute(text(explain_limit))
    for row in result:
        print(row[0])
```

Wynik:

```
Planning:
  Buffers: shared hit=3
Planning Time: 0.099 ms
Execution Time: 0.828 ms
```

4.14.4 Wniosek:

Zastosowanie indeksów i *LIMIT* pozwala lepiej kontrolować koszt zapytania — choć Execution Time może wzrosnąć w przypadku dodatkowej pracy np. na sortowaniu wyników, plan wykonania staje się bardziej przewidywalny. W przypadku większych danych różnice w buforach i czasie wykonania stają się znacznie bardziej zauważalne.

CHAPTER 5

Spis repozytoriów

1. Sprawozdanie: https://github.com/Chaiolites/Sprawozdanie_BD/tree/main/Sprawozdanie-main
2. Przegląd literatury: https://github.com/Chaiolites/Konfiguracja_baz_danych/tree/main
3. Kod: https://github.com/Chaiolites/Kody-Python-Bazy

Pozostały przegląd literatury:

- https://github.com/Broksonn/Wydajnosc_Skalowanie_i_Replikacja/blob/main/Wydajnosc-Skalowanie-i-Replikacja.rst
- https://github.com/oszczeda/Sprzet-dla-bazy-danych/blob/main/source/SprzetDlaBazyDanych.rst
- https://github.com/GrzegorzSzczepanek/repo-wspolne/blob/0e261b0e74837c01b429c09b9188cefc5cef546d/index.rst
- https://github.com/BlazejUl/bezpieczenstwo/blob/main/index.rst

CHAPTER 6

Skrypty z baz danych

6.1 Kod bazy danych:

6.1.1 PostgreSQL

```
import simplejson
from sqlalchemy import create_engine, text

with open("/home/student06/Bazy/database_creds.json") as db_con_file:
    creds = simplejson.loads(db_con_file.read())

connection = 'postgresql+psycopg://' + \
                creds['user_name'] + ':' + creds['password'] + '@' + \
                creds['host_name'] + ':' + creds['port_number'] + '/' + \
                creds['db_name']

engine = create_engine(connection)

# POSTGRESQL
# Tabela przechowująca dane pracowników

create_table_Pracownik = """
CREATE TABLE Pracownik (
    numer_pracownika SERIAL PRIMARY KEY,
    imie VARCHAR(50) NOT NULL,
    nazwisko VARCHAR(50) NOT NULL
);"""

with engine.begin() as conn:
    conn.execute(text(create_table_Pracownik))
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
# Tabela przechowująca dane klientów (zarówno sprzedawców, jak i nabywców)
create_table_Klient = """
CREATE TABLE Klient (
    numer_dowodu VARCHAR(20) PRIMARY KEY,
    imie VARCHAR(50) NOT NULL,
    nazwisko VARCHAR(50) NOT NULL,
    ilosc_wystawionych_przedmiotow INT DEFAULT 0,
    wymaga_kaucji BOOLEAN DEFAULT FALSE
);"""

with engine.begin() as conn:
    conn.execute(text(create_table_Klient))

# Tabela rejestrująca transakcje aukcyjne
create_table_LogAukcji = """
CREATE TABLE LogAukcji (
    id_transakcji SERIAL PRIMARY KEY,
    sprzedawca VARCHAR(20) NOT NULL,
    nabywca VARCHAR(20),
    data_transakcji DATE NOT NULL,
    cena_sprzedazy NUMERIC(10,2) NOT NULL DEFAULT 0,
    numer_pracownika INT NOT NULL,
    CONSTRAINT fk_sprzedawca FOREIGN KEY (sprzedawca) REFERENCES Klient (numer_dowodu),
    CONSTRAINT fk_nabywca FOREIGN KEY (nabywca) REFERENCES Klient (numer_dowodu),
    CONSTRAINT fk_pracownik FOREIGN KEY (numer_pracownika) REFERENCES Pracownik (numer_
↪pracownika)
);"""

with engine.begin() as conn:
    conn.execute(text(create_table_LogAukcji))

insert_Pracownik = """
INSERT INTO Pracownik (imie, nazwisko) VALUES
('Anna', 'Kowalska'),
('Jan', 'Nowak'),
...
('Patryk', 'Majewski');
"""

with engine.begin() as conn:
    conn.execute(text(insert_Pracownik))

insert_Klient = """
INSERT INTO Klient (numer_dowodu, imie, nazwisko, ilosc_wystawionych_przedmiotow, wymaga_
↪kaucji) VALUES
('ID100001', 'Jan', 'Kowalski', 3, TRUE),
...
('ID100200', 'Katarzyna', 'Czarnecka', 2, FALSE);
"""

with engine.begin() as conn:
    conn.execute(text(insert_Klient))
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
insert_LogAukcji = """
INSERT INTO LogAukcji (sprzedawca, nabywca, data_transakcji, cena_sprzedazy, numer_
↪pracownika) VALUES
('ID100001', 'ID100050', '2025-06-01', 150.75, 1),
...
('ID100095', 'ID100071', '2025-09...
"""

with engine.begin() as conn:
    conn.execute(text(insert_LogAukcji))
# przykładowe zapytanie do bazy danych w celu zmierzenia czasu wykonania i pobrania␣
↪wyniku
explain_limit = """
EXPLAIN (ANALYZE, BUFFERS)
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC
"""

with engine.begin() as conn:
    conn.execute(text(explain_limit))

# wypisanie wyniku funkcji EXPLAIN, interesującym nas parametrem jest czas wykonania
with engine.begin() as conn:
    result = conn.execute(text(explain_limit))
    for row in result:
        print(row[0])

# przykładowe zapytanie do bazy danych z podstawową optymalizacją zapytań poprzez␣
↪dodanie LIMIT 10
explain_limit = """
EXPLAIN (ANALYZE, BUFFERS)
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC
LIMIT 10;
"""

with engine.begin() as conn:
    conn.execute(text(explain_limit))

with engine.begin() as conn:
    result = conn.execute(text(explain_limit))
    for row in result:
        print(row[0])  # plan wykonania jako pojedyncze linie tekstu
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
# dodanie indeksów do tabeli w celu zwiększenia prędkości wykonania zapytania
create_indexes = """
CREATE INDEX IF NOT EXISTS idx_log_sprzedawca  ON LogAukcji(sprzedawca);
CREATE INDEX IF NOT EXISTS idx_log_nabywca     ON LogAukcji(nabywca);
CREATE INDEX IF NOT EXISTS idx_log_data        ON LogAukcji(data_transakcji);
CREATE INDEX IF NOT EXISTS idx_log_data_sprzed ON LogAukcji(data_transakcji, sprzedawca);
"""
with engine.begin() as conn:
    conn.execute(text(create_indexes))

explain_limit = """
EXPLAIN (ANALYZE, BUFFERS)
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC
LIMIT 10;
"""

with engine.begin() as conn:
    conn.execute(text(explain_limit))

with engine.begin() as conn:
    result = conn.execute(text(explain_limit))
    for row in result:
        print(row[0])

# usunięcie tabel
Drop3 = """
DROP TABLE LogAukcji
"""
with engine.begin() as conn:
    conn.execute(text(Drop3))

Drop1 = """
DROP TABLE Pracownik
"""
with engine.begin() as conn:
    conn.execute(text(Drop1))

Drop2 = """
DROP TABLE Klient
"""
with engine.begin() as conn:
    conn.execute(text(Drop2))
```

6.2 Kod SQLight

```
import sqlite3
import time
con = sqlite3.connect(":memory:")
cursor = con.cursor()

#LightSQL

# Tabela przechowująca dane pracowników
cursor.execute("""
CREATE TABLE Pracownik (
    numer_pracownika INTEGER PRIMARY KEY AUTOINCREMENT,
    imie            VARCHAR(50) NOT NULL,
    nazwisko        VARCHAR(50) NOT NULL
);
""")

# Tabela przechowująca dane klientów (sprzedawcy oraz nabywcy)
cursor.execute("""
CREATE TABLE Klient (
    numer_dowodu                    VARCHAR(20) PRIMARY KEY,
    imie                            VARCHAR(50) NOT NULL,
    nazwisko                        VARCHAR(50) NOT NULL,
    ilosc_wystawionych_przedmiotow    INTEGER DEFAULT 0,
    wymaga_kaucji                    BOOLEAN DEFAULT FALSE
)
""")

# Tabela rejestrująca transakcje aukcyjne
cursor.execute("""
CREATE TABLE LogAukcji (
    id_transakcji    INTEGER PRIMARY KEY AUTOINCREMENT,
    sprzedawca       VARCHAR(20) NOT NULL,
    nabywca          VARCHAR(20),
    data_transakcji  DATE NOT NULL,
    cena_sprzedazy   NUMERIC(10,2) NOT NULL DEFAULT 0,
    numer_pracownika INTEGER NOT NULL,
    FOREIGN KEY (sprzedawca) REFERENCES Klient(numer_dowodu),
    FOREIGN KEY (nabywca) REFERENCES Klient(numer_dowodu),
    FOREIGN KEY (numer_pracownika) REFERENCES Pracownik(numer_pracownika)
);
""")

cursor.execute("""
INSERT INTO Pracownik (imie, nazwisko) VALUES
('Anna', 'Kowalska'),
('Marek', 'Nowak'),
('Ewa', 'Zielińska'),
('Tomasz', 'Wiśniewski'),
('Karolina', 'Mazur');
""")
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
# Klienci
cursor.execute("""
INSERT INTO Klient (numer_dowodu, imie, nazwisko, ilosc_wystawionych_przedmiotow, wymaga_
↪kaucji) VALUES
('ID100001', 'Jan', 'Kowalski', 3, 1),
('ID100002', 'Anna', 'Nowak', 5, 0),
...
""")
# zacznij liczyć czas
start_time = time.time()

# przykładowe zapytanie do bazy danych w celu zmierzenia czasu wykonania i pobrania␣
↪wyniku
cursor.execute("""
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC
""")
# pobranie wyniku
results = cursor.fetchall()
# skończ liczyć czas
end_time = time.time()
# policz różnicę czasów od początku do końca
elapsed = end_time - start_time
# wypisz czas wykonania
print(f"Czas wykonania zapytania: {elapsed:.4f} sekundy")

start_time = time.time()

# przykładowe zapytanie do bazy danych z podstawową optymalizacją zapytania poprzez␣
↪dodanie LIMIT 10
cursor.execute("""
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC
LIMIT 10;
""")
results = cursor.fetchall()

end_time = time.time()
elapsed = end_time - start_time

print(f"Czas wykonania zapytania: {elapsed:.4f} sekundy")

# dodanie indeksów do tabeli w celu zwiększenia prędkości wykonania zapytania
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
cursor.execute("CREATE INDEX IF NOT EXISTS idx_log_sprzedawca ON LogAukcji(sprzedawca);")
cursor.execute("CREATE INDEX IF NOT EXISTS idx_log_nabywca ON LogAukcji(nabywca);")
cursor.execute("CREATE INDEX IF NOT EXISTS idx_log_data ON LogAukcji(data_transakcji);")
cursor.execute("CREATE INDEX IF NOT EXISTS idx_log_data_sprzed ON LogAukcji(data_
↪transakcji, sprzedawca);")

start_time = time.time()

cursor.execute("""
SELECT k.imie, k.nazwisko, COUNT(*) AS cnt
FROM LogAukcji l
JOIN Klient k ON l.sprzedawca = k.numer_dowodu
WHERE l.data_transakcji BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY k.imie, k.nazwisko
ORDER BY cnt DESC
LIMIT 10;
""")
results = cursor.fetchall()

end_time = time.time()
elapsed = end_time - start_time

print(f"Czas wykonania zapytania: {elapsed:.4f} sekundy")

cursor.execute("""
DROP TABLE LogAukcji
""")

cursor.execute("""
DROP TABLE Klient
""")

cursor.execute("""
DROP TABLE Pracownik
""")

connection.commit()
cursor.close()
connection.close()
```