

---

# Sprawozdanie

*Release 1.0*

**Mateusz Brokos**

**Jul 06, 2025**

## CONTENTS

<b>1</b>	<b>Wprowadzenie</b>	<b>3</b>
<b>2</b>	<b>Badania literaturowe</b>	<b>4</b>
2.1	Wydajność, skalowanie i replikacja	4
2.1.1	Wstęp	4
2.1.2	Buforowanie oraz zarządzanie połączeniami	4
2.1.3	Wydajność	5
2.1.4	Skalowanie	8
2.1.5	Replikacja	8
2.1.6	Kontrola dostępu i limity systemowe	10
2.1.7	Testowanie wydajności sprzętu na poziomie OS	10
2.1.8	Podsumowanie	10
2.1.9	Bibliografia	11
2.2	Sprzęt dla baz danych	11
2.2.1	Wstęp	11
2.2.2	Sprzęt dla bazy danych PostgreSQL	11
2.2.3	Sprzęt dla bazy danych SQLite	12
2.2.4	Podsumowanie	13
2.3	Sprawozdanie: Konfiguracja i Zarządzanie Bazą Danych	13
2.3.1	1. Konfiguracja bazy danych	13
2.3.2	2. Lokalizacja i struktura katalogów	13
2.3.3	3. Katalog danych	14
2.3.4	4. Podział konfiguracji na podpliki	14
2.3.5	5. Katalog Konfiguracyjny	14
2.3.6	6. Katalog logów i struktura katalogów w PostgreSQL	15
2.3.7	7. Przechowywanie i lokalizacja plików konfiguracyjnych	15
2.3.8	8. Podstawowe parametry konfiguracyjne	15
2.3.9	9. Wstęp teoretyczny	16
2.3.10	10. Zarządzanie konfiguracją w PostgreSQL	17
2.3.11	11. Planowanie	19
2.3.12	12. Tabele – rozmiar, planowanie i monitorowanie	20
2.3.13	13. Rozmiar pojedynczych tabel, rozmiar wszystkich tabel, indeksów tabeli	22
2.3.14	14. Rozmiar	23
2.3.15	Podsumowanie	23
2.4	Bezpieczeństwo	24
2.4.1	1. pg_hba.conf — opis pliku konfiguracyjnego PostgreSQL	24
2.4.2	2. Uprawnienia użytkownika	25
2.4.3	3. Zarządzanie użytkownikami a dane wprowadzone	26

2.4.4	4. Zabezpieczenie połączenia przez SSL/TLS . . . . .	27
2.4.5	5. Szyfrowanie danych . . . . .	28
2.5	Monitorowanie i diagnostyka . . . . .	29
2.5.1	Wstęp . . . . .	29
2.5.2	Monitorowanie Sesji i Użytkowników . . . . .	30
2.5.3	Monitorowanie Dostępu do Tabel i Operacji na Danych . . . . .	30
2.5.4	Monitorowanie Logów i Raportowanie Błędów . . . . .	31
2.5.5	Monitorowanie na Poziomie Systemu Operacyjnego . . . . .	32
2.5.6	Narzędzia Monitorowania PostgreSQL . . . . .	32
2.5.7	Najlepsze Praktyki Monitorowania . . . . .	33
2.5.8	Monitorowanie Wysokiej Dostępności . . . . .	34
2.5.9	Wniosek . . . . .	35
2.5.10	Bibliografia: . . . . .	35
2.6	Kopie zapasowe i odzyskiwanie danych w PostgreSQL . . . . .	36
2.6.1	Wprowadzenie . . . . .	36
2.6.2	Mechanizmy wbudowane do tworzenia kopii zapasowych całego systemu PostgreSQL . . . . .	36
2.6.3	Mechanizmy wbudowane do tworzenia kopii zapasowych poszczególnych baz danych . . . . .	37
2.6.4	Odzyskiwanie usuniętych lub uszkodzonych danych . . . . .	38
2.6.5	Dedykowane oprogramowanie i skrypty zewnętrzne do automatyzacji . . . . .	40
2.6.6	Podsumowanie . . . . .	41
2.7	Kontrola i konserwacja baz danych . . . . .	42
2.7.1	Wprowadzenie . . . . .	42
2.7.2	Podział konserwacji baz danych . . . . .	42
2.7.3	Różnice konserwacyjne w zależności od rodzaju bazy danych . . . . .	43
2.7.4	Planowanie konserwacji bazy danych . . . . .	44
2.7.5	Uruchamianie, zatrzymywanie i restartowanie serwera bazy danych . . . . .	45
2.7.6	Zarządzanie połączeniami użytkowników . . . . .	46
2.7.7	Proces VACUUM . . . . .	47
2.7.8	Schemat bazy danych . . . . .	48
2.7.9	Transakcje . . . . .	49
2.7.10	Literatura . . . . .	50
2.8	Partycjonowanie danych w PostgreSQL – analiza, typy, zastosowania i dobre praktyki . . . . .	50
2.8.1	Streszczenie . . . . .	50
2.8.2	1. Wprowadzenie . . . . .	50
2.8.3	2. Definicja i cel partycjonowania . . . . .	50
2.8.4	3. Modele i typy partycjonowania w PostgreSQL . . . . .	51
2.8.5	4. Implementacja partycjonowania w praktyce . . . . .	52
2.8.6	5. Monitorowanie i administracja . . . . .	53
2.8.7	6. Typowe scenariusze zastosowań . . . . .	54
2.8.8	7. Dobre praktyki projektowania partycji . . . . .	54
2.8.9	8. Ograniczenia i potencjalne problemy . . . . .	54
2.8.10	9. Podsumowanie i wnioski . . . . .	54
2.8.11	10. Krótkie porównanie partycjonowania w PostgreSQL i innych systemach bazodanowych . . . . .	54
2.8.12	11. Przykład migracji niepartycjonowanej tabeli na partycjonowaną . . . . .	55
2.8.13	12. Bibliografia . . . . .	56
<b>3</b>	<b>Projektowanie bazy danych - modele</b>	<b>57</b>
3.1	Wprowadzenie . . . . .	57

3.2	Model Konceptualny . . . . .	57
3.3	Model Logiczny . . . . .	58
3.4	Model Fizyczny . . . . .	58
3.5	Przykładowe rekordy . . . . .	59
3.5.1	Tabela Specialization . . . . .	59
3.5.2	Tabela Doctor . . . . .	59
3.5.3	Tabela Patient . . . . .	59
3.5.4	Tabela Visit . . . . .	60
<b>4</b>	<b>Analiza Bazy danych i optymalizacja zapytań</b>	<b>61</b>
4.1	Analiza normalizacji . . . . .	61
4.1.1	Model logiczny jest w 3NF: . . . . .	61
4.2	Potencjalne problemy wydajnościowe . . . . .	61
4.3	Strategie optymalizacji . . . . .	61
4.4	Prezentacja skryptów wspomagających . . . . .	62
4.5	Wnioski . . . . .	63
<b>5</b>	<b>Podsumowanie</b>	<b>64</b>
5.1	Spis repozytoriów . . . . .	64

## WPROWADZENIE

Prowadzący: Piotr Czaja

Kurs: Bazy Danych 1

Student oddający pracę: Mateusz Brokos

Celem niniejszego raportu jest szczegółowe przedstawienie procesu projektowania, implementacji oraz analizy wydajności relacyjnej bazy danych wspierającej obsługę przychodni lekarskiej. Opracowana baza umożliwia rejestrację pacjentów, lekarzy i wizyt, zapewniając przejrzyste modele danych oraz możliwość generowania raportów i analiz statystycznych. W pracy uwzględniono aspekty normalizacji struktury tabel, optymalizacji zapytań oraz przykłady wykorzystania narzędzi wspomagających dokumentację kodu. Raport dokumentuje zarówno etap projektowy, jak i praktyczne aspekty realizacji rozwiązania.

Celem następnej sekcji jest pogłębienie wiedzy na temat baz danych przeprowadzając rozeznania literaturowe a następnie wyciągnięcie i utrwalenie najważniejszych informacji.

## **BADANIA LITERATUROWE**

Poniżej zamieszczono dokumentację poszczególnych przeglądów literatury:

### **2.1 Wydajność, skalowanie i replikacja**

#### **Autorzy**

- Mateusz Brokos
- Szymon Błatkowski
- Maciej Gołębiowski

#### **2.1.1 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.

#### **2.1.2 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.

### 2.1.3 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, umożliwiającą 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 obciążeń 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.

## **2.1.4 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 wszcz.

## **2.1.5 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).

### 2.1.6 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.

### 2.1.7 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.

### 2.1.8 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.

## 2.1.9 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.

## 2.2 Sprzęt dla baz danych

### 2.2.1 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

### 2.2.2 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.

## 2.2.3 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.

### 2.2.4 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 Sprawozdanie: Konfiguracja i Zarządzanie Bazą Danych

### Authors

- Piotr Domagała
- Piotr Kotuła
- Dawid Pasikowski

### 2.3.1 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.

### 2.3.2 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.

### 2.3.3 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.

### 2.3.4 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.

### 2.3.5 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.

### 2.3.6 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
```

### 2.3.7 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

### 2.3.8 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
-------	-----	-----	------	-----------------

### 2.3.9 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).

## 2.3.10 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.

## 2.3.11 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.

## 2.3.12 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.

### 2.3.13 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.

### 2.3.14 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.

### 2.3.15 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.

## 2.4 Bezpieczeństwo

### Autorzy

- Katarzyna Tarasek
- Błażej Uliasz

### 2.4.1 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 uwierzytniony.

#### 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 uwierzytnienia – 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 uwierzytniania

- `trust` — brak uwierzytnienia (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();
```

## 2.4.2 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;
```

## 2.4.3 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 użytkownika 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 *ALTER 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 użytkownika np. za pomocą polecenia *INSERT* pozostają, nawet jeśli jego konto zostało usunięte.

### Usunięcie użytkownika, a dane które posiadał

Po usunięciu użytkownika 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 polecenia *DROP*.

## Usunięcie użytkowników, a obietki

Usunięcie 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 dropped because some objects depend on it
```

Aby zapobiec takim błędom stosujemy poniższe rozwiązanie:

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

### 2.4.4 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 wygenerować, 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 szyfrowane w PostgreSQL wystarczy użyć prostego polecenia `SELECT ssl_is_used()`; . Jeśli jednak chcemy dostać więcej informacji, musimy wpisać poniższe polecenia:

```
SELECT datname, username, 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, wersję protokołu czy użyty szyft. Wpisujemy poniższą komendę:

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

### 2.4.5 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 technologii 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 rozwiązaniami jest LUKS, BitLocker, szyfrowanie oferowane przez chmury. Zaletami tego szyfrowania jest transparentność dla PostgreSQL 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 użyciem 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.

## 2.5 Monitorowanie i diagnostyka

### Autorzy

- Dominika Półchłopek
- Kacper Rasztar
- Grzegorz Szczepanek

### 2.5.1 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.

## 2.5.2 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.

## 2.5.3 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 podejrzone 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.

### 2.5.4 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 .

## 2.5.5 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.

## 2.5.6 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.

pganalyzer 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 pganalyzer 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.

## 2.5.7 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 (szczyty 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.

## **2.5.8 Monitorowanie Wysokiej Dostępności**

### **Monitorowanie Statusu Replikacji**

Monitorowanie klastrów PostgreSQL o wysokiej dostępności wymaga dodatkowych wymiarów poza monitorowaniem pojedynczej instancji. Kluczowe obszary obejmują śledzenie opóźnień 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.

### 2.5.9 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.

### 2.5.10 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](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/](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](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/](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](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>

## 2.6 Kopie zapasowe i odzyskiwanie danych w PostgreSQL

### Autorzy

Miłosz Śmieja Szymon Piskorz Mateusz Wasilewicz

### 2.6.1 Wprowadzenie

System zarządzania bazą danych PostgreSQL oferuje kompleksowy zestaw narzędzi i mechanizmów służących do tworzenia kopii zapasowych oraz odzyskiwania danych. Skuteczne zarządzanie kopiami zapasowymi stanowi fundament bezpieczeństwa danych i ciągłości działania systemów bazodanowych.

PostgreSQL dostarcza zarówno mechanizmy wbudowane, jak i możliwość integracji z zewnętrznymi narzędziami automatyzacji.

### 2.6.2 Mechanizmy wbudowane do tworzenia kopii zapasowych całego systemu PostgreSQL

PostgreSQL oferuje kilka mechanizmów tworzenia kopii zapasowych na poziomie całego systemu, które zapewniają kompleksową ochronę wszystkich baz danych w klastrze.

#### **pg\_basebackup**

**pg\_basebackup** stanowi podstawowe narzędzie do tworzenia fizycznych kopii zapasowych całego klastra PostgreSQL.

Kluczowe cechy:

- Działa w trybie online - możliwość wykonywania kopii zapasowych bez zatrzymywania działania serwera
- Tworzy dokładną kopię wszystkich plików danych
- Zawiera pliki konfiguracyjne, dzienniki transakcji oraz wszystkie bazy danych w klastrze

#### **Continuous Archiving (Point-in-Time Recovery)**

**Continuous Archiving** reprezentuje zaawansowany mechanizm tworzenia ciągłych kopii zapasowych poprzez archiwizację dzienników WAL (Write-Ahead Logging).

Zalety:

- Umożliwia odtworzenie stanu bazy danych w dowolnym momencie czasowym
- Szczególnie wartościowe w środowiskach produkcyjnych wymagających minimalnej utraty danych
- Zapewnia wysoką granularność odzyskiwania danych



## Streaming Replication

**Streaming Replication** może służyć jako mechanizm kopii zapasowych poprzez utrzymywanie synchronicznych lub asynchronicznych replik głównej bazy danych.

Funkcjonalności:

- Repliki funkcjonują jako kopie zapasowe w czasie rzeczywistym
- Oferuje możliwość szybkiego przełączenia w przypadku awarii systemu głównego
- Wspiera zarówno tryb synchroniczny, jak i asynchroniczny

## File System Level Backup

**File System Level Backup** polega na tworzeniu kopii zapasowych na poziomie systemu plików.

Wymagania:

- Zatrzymanie serwera PostgreSQL lub zapewnienie spójności
- Wykorzystanie mechanizmów snapshot systemu plików:
  - LVM snapshots
  - ZFS snapshots

### 2.6.3 Mechanizmy wbudowane do tworzenia kopii zapasowych poszczególnych baz danych

PostgreSQL dostarcza precyzyjne narzędzia umożliwiające tworzenie kopii zapasowych pojedynczych baz danych lub ich wybranych elementów.

#### pg\_dump

**pg\_dump** stanowi najczęściej wykorzystywane narzędzie do tworzenia logicznych kopii zapasowych pojedynczych baz danych.

Charakterystyka:

- Tworzy skrypt SQL zawierający wszystkie polecenia niezbędne do odtworzenia struktury bazy danych oraz jej danych
- Oferuje liczne opcje konfiguracji:
  - Możliwość wyboru formatu wyjściowego
  - Filtrowanie obiektów
  - Kontrola nad poziomem szczegółowości kopii zapasowej

#### pg\_dumpall

**pg\_dumpall** rozszerza funkcjonalność **pg\_dump** o możliwość tworzenia kopii zapasowych wszystkich baz danych w klastrze.

Dodatkowe funkcje:

- Backup obiektów globalnych:
  - Role użytkowników
  - Tablespaces

- Ustawienia konfiguracyjne na poziomie klastra

### **COPY command**

**COPY command** umożliwia eksport danych z poszczególnych tabel do plików w różnych formatach.

Obsługiwane formaty:

- CSV
- Text
- Binary

Zastosowania:

- Tworzenie selektywnych kopii zapasowych dużych tabel
- Migracje danych

### **pg\_dump z opcjami selektywnymi**

**pg\_dump z opcjami selektywnymi** pozwala na tworzenie kopii zapasowych wybranych obiektów bazy danych.

Możliwości filtrowania:

- Konkretne tabele
- Schematy
- Sekwencje

Funkcjonalność ta jest nieoceniona w scenariuszach wymagających granularnej kontroli nad procesem tworzenia kopii zapasowych.

## **2.6.4 Odzyskiwanie usuniętych lub uszkodzonych danych**

PostgreSQL oferuje różnorodne mechanizmy odzyskiwania danych w zależności od rodzaju i zakresu uszkodzeń.

### **Odzyskiwanie z kopii logicznych**

**Odzyskiwanie z kopii logicznych** wykonanych przy użyciu `pg_dump` realizowane jest poprzez `psql` lub `pg_restore`.

Proces odzyskiwania:

- Wykonanie skryptów SQL
- Przywrócenie plików dump w odpowiednim formacie

Zaawansowane opcje `pg_restore`:

- Selektywne przywracanie obiektów
- Równoległe przetwarzanie
- Kontrola nad kolejnością przywracania



## Point-in-Time Recovery (PITR)

**Point-in-Time Recovery (PITR)** umożliwia przywrócenie bazy danych do konkretnego momentu w czasie.

Wykorzystywane komponenty:

- Kombinacja kopii bazowej
- Archiwalne dzienniki WAL

Zastosowania:

- Cofnięcie zmian do momentu poprzedzającego wystąpienie błędu
- Odzyskiwanie po uszkodzeniu danych

### Note

PITR jest szczególnie wartościowy w przypadkach, gdy konieczne jest cofnięcie zmian do momentu poprzedzającego wystąpienie błędu lub uszkodzenia.

## Odzyskiwanie tabel z tablespaces

**Odzyskiwanie tabel z tablespaces** może wymagać specjalnych procedur w przypadku uszkodzenia przestrzeni tabel.

Możliwości PostgreSQL:

- Odtworzenie tablespaces
- Przeniesienie tabel między różnymi lokalizacjami
- Odzyskiwanie danych nawet w przypadku częściowego uszkodzenia systemu plików

## Transaction log replay

**Transaction log replay** wykorzystuje dzienniki WAL do odtworzenia zmian wprowadzonych po utworzeniu kopii zapasowej.

Charakterystyka:

- Automatycznie wykorzystywany podczas standardowych procedur odzyskiwania
- Możliwość ręcznej kontroli w szczególnych sytuacjach

## Odzyskiwanie na poziomie klastra

**Odzyskiwanie na poziomie klastra** przy wykorzystaniu `pg_basebackup` wymaga przywrócenia wszystkich plików klastra oraz odpowiedniej konfiguracji parametrów `recovery`.

Zakres procesu:

- Odtworzenie całego środowiska PostgreSQL
- Konfiguracja ról i uprawnień
- Przywrócenie ustawień systemowych

## 2.6.5 Dedykowane oprogramowanie i skrypty zewnętrzne do automatyzacji

Automatyzacja procesów tworzenia kopii zapasowych stanowi kluczowy element profesjonalnego zarządzania bazami danych PostgreSQL.

### pgBackRest

**pgBackRest** reprezentuje kompleksowe rozwiązanie do zarządzania kopiami zapasowymi PostgreSQL.

Zaawansowane funkcje:

- Incremental i differential backups
- Kompresja danych
- Szyfrowanie
- Weryfikacja integralności kopii
- Możliwość przechowywania kopii w chmurze
- Automatyzacja procesów zarządzania kopiami zapasowymi
- Uprozczone procedury odzyskiwania

#### Important

pgBackRest automatyzuje wiele procesów związanych z zarządzaniem kopiami zapasowymi i znacznie upraszcza procedury odzyskiwania.

### Barman (Backup and Recovery Manager)

**Barman** stanowi dedykowane narzędzie stworzone przez 2ndQuadrant do zarządzania kopiami zapasowymi PostgreSQL w środowiskach enterprise.

Kluczowe funkcjonalności:

- Centralne zarządzanie kopiami zapasowymi wielu serwerów PostgreSQL
- Monitoring procesów backup
- Automatyczne testowanie procedur recovery
- Integracja z narzędziami monitorowania

### WAL-E i WAL-G

**WAL-E i WAL-G** specjalizują się w archiwizacji dzienników WAL w środowiskach chmurowych.

Oferowane funkcje:

- Efektywna kompresja
- Szyfrowanie danych
- Przechowywanie kopii zapasowych w serwisach chmurowych:
  - Amazon S3
  - Google Cloud Storage
  - Azure Blob Storage

## Skrypty shell i cron jobs

**Skrypty shell i cron jobs** stanowią tradycyjne podejście do automatyzacji kopii zapasowych.

Możliwości automatyzacji:

- Wykonywanie `pg_dump` i `pg_basebackup`
- Zarządzanie cyklem życia kopii zapasowych
- Rotacja i czyszczenie starych kopii

### Tip

Właściwie napisane skrypty mogą automatyzować wykonywanie `pg_dump`, `pg_basebackup` oraz zarządzanie cyklem życia kopii zapasowych, w tym rotację i czyszczenie starych kopii.

## Narzędzia automatyzacji infrastruktury

**Ansible, Puppet, Chef** jako narzędzia automatyzacji infrastruktury mogą być wykorzystywane do zarządzania konfiguracją procesów backup na większą skalę.

Korzyści:

- Standaryzacja procedur backup w środowiskach wieloserwerowych
- Zapewnienie konsystencji konfiguracji
- Skalowalne zarządzanie infrastrukturą

## Monitoring i alertowanie

**Prometheus i Grafana** w połączeniu z `postgres_exporter` umożliwiają monitoring procesów backup oraz alertowanie w przypadku niepowodzeń.

Zakres monitorowania:

- Śledzenie czasu wykonywania kopii
- Monitorowanie rozmiaru kopii zapasowych
- Wskaźnik sukcesu procesów backup
- Alertowanie w czasie rzeczywistym

## 2.6.6 Podsumowanie

Skuteczne zarządzanie kopiami zapasowymi w PostgreSQL wymaga kombinacji mechanizmów wbudowanych oraz zewnętrznych narzędzi automatyzacji. Wybór odpowiedniej strategii backup zależy od specyficznych wymagań organizacji, w tym:

- **RTO (Recovery Time Objective)** - maksymalny akceptowalny czas odzyskiwania
- **RPO (Recovery Point Objective)** - maksymalna akceptowalna utrata danych
- Dostępne zasoby
- Złożoność środowiska

### Kluczowe wnioski

**Mechanizmy wbudowane** PostgreSQL, takie jak `pg_dump`, `pg_basebackup` czy PITR, oferują solidne podstawy dla większości scenariuszy backup i recovery.

**W środowiskach produkcyjnych** o wysokich wymaganiach dotyczących dostępności i niezawodności, integracja z dedykowanymi narzędziami takimi jak pgBackRest czy Barman staje się niezbędna.

### Najważniejsze zalecenia

#### Warning

Kluczowym elementem każdej strategii backup jest regularne testowanie procedur odzyskiwania danych. Kopie zapasowe mają wartość tylko wtedy, gdy można z nich skutecznie odzyskać dane w sytuacji kryzysowej.

**Kompleksowa strategia backup** powinna obejmować:

1. Tworzenie kopii zapasowych
2. Regularne testy restore
3. Dokumentację procedur
4. Szkolenie personelu odpowiedzialnego za zarządzanie bazami danych

## 2.7 Kontrola i konserwacja baz danych

### 2.7.1 Wprowadzenie

Autor: Bartłomiej Czyż

Systemy baz danych są niezwykle ważnym elementem infrastruktury informatycznej współczesnych organizacji. Umożliwiają przechowywanie, zarządzanie i analizę danych w sposób bezpieczny oraz wydajny. Aby zapewnić ich niezawodność, integralność i wysoką dostępność, konieczne jest prowadzenie regularnych działań z zakresu kontroli i konserwacji. Działania te można podzielić na część fizyczną oraz część programową, a sposób ich przeprowadzania różni się w zależności od rodzaju i architektury używanej bazy danych.

### 2.7.2 Podział konserwacji baz danych

Autor: Bartłomiej Czyż

#### Konserwacja fizyczna

Konserwacja fizyczna obejmuje wszystkie działania związane z infrastrukturą sprzętową i zasobami systemowymi, na których działa baza danych. Do najważniejszych elementów tej konserwacji należą:

- Monitorowanie stanu dysków twardych – pozostała przestrzeń na dyskach, zużycie dysków oraz fragmentacja danych,
- Zabezpieczenie fizyczne serwerów – kontrola dostępu, ochrona przeciwpożarowa, klimatyzacja,
- Zasilanie awaryjne (UPS) - zabezpieczenie bazy przed skutkami nagłego zaniku zasilania,
- Monitoring stanu sieci – wydajność i stabilność połączenia między bazą a klientami,

- Tworzenie kopii zapasowych na nośnikach fizycznych – np. dyskach zewnętrznych czy taśmach LTO.

## Konserwacja programowa

Konserwacja programowa odnosi się do czynności wykonywanych na poziomie oprogramowania i logiki działania systemu bazy danych. Obejmuje:

- Zarządzanie użytkownikami i ich uprawnieniami,
- Optymalizację zapytań SQL,
- Aktualizację oprogramowania bazodanowego (np. MySQL, PostgreSQL),
- Defragmentację indeksów,
- Weryfikację integralności danych i naprawę uszkodzonych rekordów,
- Automatyczne zadania konserwacyjne (cron, schedulery),
- Reduplikację i redundancję - konfiguracja serwerów zapasowych.

### 2.7.3 Różnice konserwacyjne w zależności od rodzaju bazy danych

Autor: Bartłomiej Czyż

#### PostgreSQL

PostgreSQL to zaawansowany system RDBMS, znany z silnego wsparcia dla różnych typów danych i transakcyjności.

##### 1. Fizyczna konserwacja:

- Złożona struktura katalogów danych (base, pg\_wal, pg\_tblspc) – wymaga regularnego monitoringu,
- Możliwość wykorzystania narzędzia pg\_basebackup do tworzenia pełnych kopii fizycznych.

##### 2. Programowa konserwacja:

- Automatyczne zadania VACUUM, ANALYZE – zapewniają odzyskiwanie przestrzeni po usunięciu rekordów,
- Możliwość używania pg\_repack do defragmentacji bez przestojów,
- Silne wsparcie dla replikacji strumieniowej i klastrów wysokiej dostępności (HA).

#### MySQL

MySQL jest obecnie jedną z najpopularniejszych relacyjnych baz danych, szeroko stosowana w aplikacjach webowych.

##### 1. Fizyczna konserwacja:

- Wymaga monitorowania plików .ibd (w przypadku silnika InnoDB), które mogą znacznie rosnąć,
- Backup danych realizowany poprzez mysqldump lub system replikacji binlogów.

##### 2. Programowa konserwacja:

- Regularne sprawdzanie indeksów (ANALYZE TABLE, OPTIMIZE TABLE),

- Używanie narzędzi typu mysqlcheck do weryfikacji i naprawy tabel,
- Konfiguracja pliku my.cnf w celu dostosowania do wymagań aplikacji.

### SQLite (np. LightSQL)

SQLite, używana w aplikacjach mobilnych i desktopowych, różni się znacznie od serwerowych baz danych.

1. Fizyczna konserwacja:
  - Brak klasycznego serwera – baza to pojedynczy plik .db,
  - Konieczność regularnego kopiowania pliku bazy danych jako backup.
2. Programowa konserwacja:
  - Użycie polecenia VACUUM do defragmentacji i zmniejszenia rozmiaru pliku,
  - Ograniczone możliwości równoczesnego dostępu – wymaga uwagi w aplikacjach wielowątkowych,
  - Nie wymaga osobnych usług do zarządzania – działa bezpośrednio w aplikacji.

### Microsoft SQL Server

System korporacyjny, szeroko wykorzystywany w dużych organizacjach.

1. Fizyczna konserwacja:
  - Obsługuje macierze RAID i pamięci masowe SAN,
  - Regularne kopie pełne, różnicowe i dzienniki transakcyjne.
2. Programowa konserwacja:
  - Zaawansowany SQL Server Agent – możliwość harmonogramowania zadań,
  - Narzędzia do monitorowania stanu instancji (SQL Profiler, Database Tuning Advisor),
  - Wsparcie dla Always On Availability Groups dla wysokiej dostępności.

## 2.7.4 Planowanie konserwacji bazy danych

Autor: Piotr Mikołajczyk

Konserwację bazy danych należy przeprowadzać regularnie, np. co tydzień lub co miesiąc. Nie powinna mieć miejsca w godzinach szczytu. Przeprowadzenie konserwacji może również okazać się konieczne po wykryciu błędu lub wystąpieniu awarii.

Konserwacja może obejmować m.in. zmianę parametrów konfiguracji bazy, przeprowadzenie procesu VACUUM, zmianę uprawnień użytkowników, aktualizacje systemowe i wykonanie backupów lub przywrócenie danych.

Działanie te muszą zostać przeprowadzone w czasie, gdy mamy pewność, że żaden klient nie będzie podłączony, nie będą przeprowadzane żadne transakcje. Użytkownicy powinni być uprzednio poinformowani o czasie przeprowadzenia konserwacji. Mimo to, należy wcześniej sprawdzić, czy nie ma aktywnych sesji.

## 2.7.5 Uruchamianie, zatrzymywanie i restartowanie serwera bazy danych

Autor: Piotr Mikołajczyk

Działania, takie jak aktualizacja oprogramowania, instalacja rozszerzeń, wprowadzenie pewnych zmian w plikach konfiguracyjnych, migracja danych, wykonanie backupów bazy, wymagają zrestartowania, zatrzymania bądź ponownego uruchomienia serwera bazy danych.

### Uruchamianie

Linux:

```
sudo systemctl start postgresql
```

Windows CMD:

```
net start postgresql-x64-15
```

Windows PowerShell

```
Start-Service -Name postgresql-x64-15
```

### Zatrzymywanie

Linux:

```
sudo systemctl stop postgresql
```

Windows CMD:

```
net stop postgresql-x64-15
```

Windows PowerShell

```
Stop-Service -Name postgresql-x64-15
```

### Restartowanie

Linux:

```
sudo systemctl restart postgresql
```

W CMD nie istnieje osobne polecenie restartowania. Należy zatrzymać serwer, a następnie uruchomić go ponownie.

Windows PowerShell

```
Restart-Service -Name postgresql-x64-15
```

Polecenia CMD mogą zostać również użyte w PowerShell.

## 2.7.6 Zarządzanie połączeniami użytkowników

Autor: Piotr Mikołajczyk

Oprócz sytuacji, gdy trzeba zamknąć dostęp do bazy danych na czas konserwacji, połączenia użytkowników należy ograniczyć także wtedy, gdy sesja użytkownika została zawieszona lub zbyt wiele połączeń skutkuje nadmiernym zużyciem pamięci i mocy obliczeniowej, uniemożliwiając nawiązywanie nowych połączeń i spowalniając działanie serwera.

### Ograniczanie użytkowników

Istnieje kilka sposobów ograniczenia dostępu użytkownika:

- Odebranie użytkownikowi prawa dostępu do bazy:

```
REVOKE CONNECT ON DATABASE baza FROM user;
```

- Limit liczby jednoczesnych połączeń:

```
ALTER ROLE user CONNECTION LIMIT 3;
```

### Ręczne rozłączanie użytkowników

Według nazwy danego użytkownika:

```
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE username = 'user';
```

Według PID (np. 12340):

```
SELECT pg_terminate_backend(12340);
```

### Automatyczne rozłączanie użytkowników

Sesja użytkownika lub jego zapytania mogą zostać rozłączone automatycznie, jeśli wprowadzimy pewne ograniczenia czasowe:

- Rozłączenie sesji po przekroczeniu limitu czasu bezczynności podczas zapytania:

- dla bieżącej sesji:

```
SET idle_in_transaction_session_timeout = '5min';
```

- dla danego użytkownika:

```
ALTER ROLE user SET idle_in_transaction_session_timeout =
→ '5min';
```

- Limit czasu zapytania:

```
ALTER ROLE user SET statement_timeout = '30s';
```



## Zapobieganie nowym połączeniom

Zablokowanie logowania konkretnego użytkownika:

```
ALTER ROLE user NOLOGIN;
```

Odblokowanie:

```
ALTER ROLE user LOGIN;
```

Blokowanie nowych połączeń do bazy danych:

```
REVOKE CONNECT ON DATABASE baza FROM PUBLIC;
```

PUBLIC oznacza wszystkich użytkowników. Nadal połączeni użytkownicy nie są rozłączani.

### 2.7.7 Proces VACUUM

Autor: Piotr Mikołajczyk

DELETE nie usuwa rekordów z tabeli, jedynie oznacza je jako martwe. Podobnie UPDATE pozostawia stare wersje zaktualizowanych krotek.

Proces VACUUM przeszukuje tabele i indeksy, szukając martwych wierszy, które można fizycznie usunąć lub oznaczyć do nadpisania.

Może zostać przeprowadzony na kilka sposobów:

```
VACUUM;
```

Usuwa martwe krotki, ale nie odzyskuje miejsca z dysku, a jedynie udostępnia je dla przyszłych danych,

```
VACUUM FULL;
```

Kompaktuje tabelę do nowego pliku, zwalnia miejsce w pamięci,

```
VACUUM ANALYZE
```

Usuwa martwe krotki i przeprowadza aktualizację statystyk, nie odzyskuje miejsca.

### Autovacuum

Autovacuum działa w tle, automatycznie wykonując VACUUM na odpowiednich tabelach. Dzięki niemu nie trzeba ręcznie uruchamiać VACUUM po każdej modyfikacji tabeli. Autovacuum posiada wiele parametrów, od których zależy kiedy wykonany zostanie proces, między innymi:

- autovacuum - parametr logiczny, decyduje, czy serwer będzie uruchamiał launcher procesu autovacuum,
- autovacuum\_max\_workers - liczba całkowita, określa maksymalną ilość procesów autovacuum mogących działać w tym samym czasie, domyślnie 3,
- autovacuum\_vacuum\_threshold - liczba całkowita, określa ile wierszy w jednej tabeli musi zostać usunięte lub zmienione, aby wywołano VACUUM, domyślnie 50,
- autovacuum\_vacuum\_scale\_factor - liczba zmiennoprzecinkowa, jaki procent tabeli musi zostać zmieniony aby wywołano VACUUM, domyślna wartość to 0.2 (20%).

Analogiczne parametry warunkują również wywołanie ANALYZE, na przykład `autovacuum_analyze_threshold`.

Próg uruchamiania VACUUM ustala się wzorem:

$$\text{autovacuum\_vacuum\_threshold} + \text{autovacuum\_vacuum\_scale\_factor} * \text{liczba\_wierszy}$$

Podobnie dla ANALYZE:

$$\text{autovacuum\_analyze\_threshold} + \text{autovacuum\_analyze\_scale\_factor} * \text{liczba\_wierszy}$$

### 2.7.8 Schemat bazy danych

Autor: Bartłomiej Czyż

#### Czym jest schemat bazy danych?

Schemat bazy danych to logiczna struktura opisująca organizację danych, typy danych, relacje między tabelami, ograniczenia integralności, procedury składowane, widoki i inne obiekty. Innymi słowy, schemat jest “szkieletem” bazy danych.

Przykładowe elementy schematu:

- Tabele (np. `users`, `orders`),
- Typy danych (np. `INT`, `VARCHAR`, `DATE`),
- Klucze główne i obce,
- Indeksy,
- Widoki (`VIEW`),
- Procedury i funkcje (`STORED PROCEDURES`),
- Ograniczenia (`CHECK`, `NOT NULL`, `UNIQUE`).

#### Rola schematu w konserwacji bazy danych

Schemat ma kluczowe znaczenie dla utrzymania spójności i integralności danych, dlatego jego kontrola i konserwacja obejmuje m.in.:

- Dokumentację schematu - niezbędna przy aktualizacjach i migracjach,
- Weryfikację integralności relacji - sprawdzenie czy klucze obce i reguły są respektowane,
- Normalizację - kontrola nad nadmiarem danych i poprawnością logiczną,
- Aktualizacje schematu - np. dodawanie nowych kolumn, zmiana typu danych,
- Kontrola zgodności - wersjonowanie schematu (np. za pomocą narzędzi typu Liquibase, Flyway),
- Zabezpieczenia schematów - nadawanie uprawnień tylko zaufanym użytkownikom.

Przykład konserwacji:

W PostgreSQL można analizować i optymalizować strukturę przy pomocy pgAdmin oraz narzędzi takich jak `pg_dump --schema-only`.

## Różnice w implementacji schematu w różnych systemach

- MySQL - obsługuje wiele schematów w jednej bazie; ograniczone typy kolumn w starszych wersjach,
- PostgreSQL - bardzo elastyczny system schematów - możliwość tworzenia przestrzeni nazw,
- SQLite - pojedynczy schemat, uproszczony system typów,
- SQL Server - schemat jako logiczna przestrzeń obiektów, np. dbo, hr, finance.

### 2.7.9 Transakcje

Autor: Bartłomiej Czyż

#### Czym jest transakcja?

Transakcja to zbiór operacji na bazie danych, które są traktowane jako jedna, nierozdzielna całość. Albo wykonują się wszystkie operacje, albo żadna - zasada atomiczności. Transakcje są podstawą do zachowania spójności danych, szczególnie w środowiskach wieloużytkownikowych.

#### Zasady ACID

Transakcje w bazach danych opierają się na czterech podstawowych zasadach, znanych jako ACID:

- A - Atomicity (Atomowość) - operacje wchodzące w skład transakcji są niepodzielne - wszystkie muszą się powieść, lub wszystkie są wycofywane,
- C - Consistency (Spójność) - transakcje przekształcają dane ze stanu spójnego w stan spójny,
- I - Isolation (Izolacja) - równoczesne transakcje nie wpływają na siebie nawzajem,
- D - Durability (Trwałość) - po zatwierdzeniu transakcji dane są trwale zapisane, nawet w przypadku awarii.

#### Rola transakcji w kontroli i konserwacji

Transakcje mają ogromne znaczenie dla bezpieczeństwa danych, dlatego są nieodłącznym elementem procesów konserwacyjnych. Ich zastosowanie obejmuje:

- Zabezpieczenie operacji aktualizacji - np. przy masowych zmianach danych,
- Replikacja i synchronizacja danych - transakcje zapewniają spójność między główną bazą, a replikami,
- Zarządzanie błędami - w przypadku błędu można wykonać ROLLBACK i przywrócić stan bazy,
- Tworzenie backupów spójnych z punktu w czasie - snapshoty danych często wymagają wsparcia transakcyjnego,
- Ochrona przed uszkodzeniami logicznymi - np. przez niekompletne aktualizacje.

#### Różnice w implementacji transakcji w różnych systemach

- MySQL - w pełni wspierane w silniku InnoDB; START TRANSACTION, COMMIT, ROLLBACK,
- PostgreSQL - silne wsparcie ACID, zaawansowana izolacja (REPEATABLE READ, SERIALIZABLE),
- SQLite - transakcje działają w trybie plikowym; BEGIN, COMMIT i ROLLBACK są wspierane,

- SQL Server - zaawansowany mechanizm transakcji z kontrolą poziomów izolacji, także eksplcytny SAVEPOINT.

### 2.7.10 Literatura

- [Oficjalna dokumentacja PostgreSQL](#)
- Riggs S., Krosing H., PostgreSQL. Receptury dla administratora, Helion 2011
- Matthew N., Stones R., Beginning Databases with PostgreSQL. From Novice to Professional, Apress 2006
- Juba S., Vannahme A., Volkov A., Learning PostgreSQL, Packt Publishing 2015

## 2.8 Partycjonowanie danych w PostgreSQL – analiza, typy, zastosowania i dobre praktyki

### Autor

Bartosz Potoczny

### Data

2025-06-12

### 2.8.1 Streszczenie

Celem niniejszego sprawozdania jest kompleksowa analiza zagadnienia partycjonowania danych w systemie zarządzania relacyjną bazą danych PostgreSQL. Praca omawia teoretyczne podstawy partycjonowania, szczegółowo wyjaśnia wszystkie dostępne mechanizmy oraz przedstawia metody realizacji partycjonowania w praktyce. Zaprezentowano również typowe scenariusze użycia, narzędzia monitorowania oraz najlepsze praktyki projektowe. Całość przeanalizowano pod kątem wydajności, utrzymania i bezpieczeństwa danych.

### 2.8.2 1. Wprowadzenie

Współczesne systemy informatyczne generują i przetwarzają coraz większe ilości danych, co wymusza stosowanie zaawansowanych mechanizmów optymalizacji przechowywania i dostępu do informacji. Partycjonowanie danych jest jedną z kluczowych technik pozwalających na poprawę wydajności, skalowalności i zarządzalności baz danych. PostgreSQL, jako zaawansowany system zarządzania relacyjną bazą danych (RDBMS), oferuje rozbudowane wsparcie dla partycjonowania, umożliwiając dostosowanie architektury bazy do indywidualnych potrzeb.

### 2.8.3 2. Definicja i cel partycjonowania

Partycjonowanie polega na logicznym podziale dużej tabeli na mniejsze, łatwiejsze w zarządzaniu fragmenty zwane partycjami. Mimo fizycznego rozdzielania, partycje są prezentowane użytkownikowi jako jedna wspólna tabela nadrzędna (ang. partitioned table, master table). Celem partycjonowania jest:

- Zwiększenie wydajności operacji SELECT, INSERT, UPDATE, DELETE poprzez ograniczenie zakresu danych do przeszukania (partition pruning).
- Ułatwienie zarządzania i archiwizacji danych (np. szybkie usuwanie lub przenoszenie całych partycji).
- Lepsze rozłożenie obciążenia (możliwość przechowywania partycji na różnych dyskach/tablespaces).

- Zmniejszenie ryzyka zablokowania całej tabeli podczas operacji konserwacyjnych (VACUUM, REINDEX itp.).

### 2.8.4 3. Modele i typy partycjonowania w PostgreSQL

PostgreSQL obsługuje trzy podstawowe typy partycjonowania:

#### ### 3.1 Partycjonowanie zakresowe (RANGE)

Dane są przypisywane do partycji na podstawie wartości mieszczącej się w określonym zakresie (np. daty, numery, id). Każda partycja odpowiada innemu przedziałowi.

**Przykład:**

```
CREATE TABLE events (
    event_id serial PRIMARY KEY,
    event_date date NOT NULL,
    description text
) PARTITION BY RANGE (event_date);

CREATE TABLE events_2023 PARTITION OF events
    FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');

CREATE TABLE events_2024 PARTITION OF events
    FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');
```

**Zastosowania:** logi systemowe, zamówienia, dane czasowe.

#### ### 3.2 Partycjonowanie listowe (LIST)

Dane są przypisywane do partycji na podstawie konkretnej wartości z listy (np. kraj, status, kategoria).

**Przykład:**

```
CREATE TABLE sales (
    sale_id serial PRIMARY KEY,
    country text,
    value numeric
) PARTITION BY LIST (country);

CREATE TABLE sales_pl PARTITION OF sales FOR VALUES IN ('Poland');
CREATE TABLE sales_de PARTITION OF sales FOR VALUES IN ('Germany');
CREATE TABLE sales_other PARTITION OF sales DEFAULT;
```

**Zastosowania:** dane geograficzne, statusowe, podział według typu klienta.

#### ### 3.3 Partycjonowanie haszowe (HASH)

Dane są rozdzielane pomiędzy partycje na podstawie funkcji haszującej zastosowanej do wybranej kolumny. Pozwala to równomiernie rozłożyć dane, gdy nie ma logicznego podziału zakresowego ani listowego.

**Przykład:**

```
CREATE TABLE logs (
    log_id serial PRIMARY KEY,
```

(continues on next page)

(continued from previous page)

```

    user_id int,
    log_time timestamp
) PARTITION BY HASH (user_id);

CREATE TABLE logs_p0 PARTITION OF logs FOR VALUES WITH (MODULUS 4, REMAINDER
↪0);
CREATE TABLE logs_p1 PARTITION OF logs FOR VALUES WITH (MODULUS 4, REMAINDER
↪1);
CREATE TABLE logs_p2 PARTITION OF logs FOR VALUES WITH (MODULUS 4, REMAINDER
↪2);
CREATE TABLE logs_p3 PARTITION OF logs FOR VALUES WITH (MODULUS 4, REMAINDER
↪3);

```

**Zastosowania:** przypadki wymagające równomiernego rozłożenia danych, np. duże systemy telemeryczne.

### ### 3.4 Partycjonowanie wielopoziomowe (Composite/Hierarchical Partitioning)

PostgreSQL umożliwia tworzenie partycji podrzędnych, czyli partycjonowanie już partycjonowanych tabel (tzw. subpartitioning).

#### Przykład:

```

CREATE TABLE measurements (
    id serial PRIMARY KEY,
    region text,
    measurement_date date,
    value numeric
) PARTITION BY LIST (region);

CREATE TABLE measurements_europe PARTITION OF measurements
    FOR VALUES IN ('Europe') PARTITION BY RANGE (measurement_date);

CREATE TABLE measurements_europe_2024 PARTITION OF measurements_europe
    FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');

```

**Zastosowania:** bardzo duże tabele, złożona struktura danych (np. po regionie i dacie).

## 2.8.5 4. Implementacja partycjonowania w praktyce

### ### 4.1 Tworzenie i zarządzanie partycjami

- **Tworzenie partycji:** Partycje tworzone są jako osobne tabele, ale zarządzane przez tabelę nadrzędną.
- **Dodawanie partycji:** Możliwe w dowolnym momencie przy użyciu CREATE TABLE ... PARTITION OF.
- **Usuwanie partycji:** ALTER TABLE ... DETACH PARTITION + DROP TABLE (po odłączeniu partycji).
- **Domyślna partycja:** Można zdefiniować partycję przechowującą dane niepasujące do żadnej innej (DEFAULT).

### ### 4.2 Wstawianie i odczyt danych

- Dane są automatycznie kierowane do właściwej partycji na podstawie klucza partycjonowania.
- W przypadku braku pasującej partycji (i braku DEFAULT) – błąd constraint violation.
- Zapytania ograniczone do klucza partycjonowania korzystają z partition pruning – przeszukują tylko wybrane partycje.

#### ### 4.3 Indeksowanie partycji

- Możliwe jest tworzenie indeksów na każdej partycji osobno lub dziedziczenie indeksów z tabeli nadrzędnej (od PostgreSQL 11 wzwyż).
- Indeksy globalne (na całą tabelę partycjonowaną) nie są jeszcze dostępne (stan na 2025).

#### ### 4.4 Ograniczenia partycjonowania

- Klucz partycjonowania musi być częścią klucza głównego (PRIMARY KEY).
- Niektóre operacje mogą wymagać wykonywania osobno na każdej partycji (np. VACUUM, REINDEX).
- Wersje PostgreSQL <10 obsługują partycjonowanie tylko przez dziedziczenie – obecnie uznawane za przestarzałe.

## 2.8.6 5. Monitorowanie i administracja

### ### 5.1 Sprawdzanie rozmieszczenia danych

```
SELECT tableoid::regclass AS partition, * FROM measurements;
```

### ### 5.2 Lista partycji

```
SELECT inhrelid::regclass AS partition
FROM pg_inherits
WHERE inhparent = 'measurements'::regclass;
```

### ### 5.3 Rozmiar partycji

```
SELECT relname AS "Partition", pg_size_pretty(pg_total_relation_size(relid))
↪ AS "Size"
FROM pg_catalog.pg_statio_user_tables
WHERE relname LIKE 'measurements%'
ORDER BY pg_total_relation_size(relid) DESC;
```

### ### 5.4 Analiza planu zapytania (partition pruning)

```
EXPLAIN ANALYZE
SELECT * FROM measurements WHERE region = 'Europe' AND measurement_date >=
↪ '2024-01-01';

-- W planie widać użycie tylko właściwych partycji.
```

### 2.8.7 6. Typowe scenariusze zastosowań

- **Przetwarzanie danych czasowych:** partycjonowanie zakresowe po dacie (logi, zamówienia, pomiary).
- **Dane geograficzne lub kategoryczne:** partycjonowanie listowe (kraj, region, kategoria produktu).
- **Systemy telemetryczne i IoT:** partycjonowanie haszowe lub wielopoziomowe (np. urządzenie + czas).
- **Duże systemy ERP/CRM:** partycjonowanie po kliencie, regionie, a następnie po dacie.

### 2.8.8 7. Dobre praktyki projektowania partycji

- **Dobór klucza partycjonowania:** Powinien odpowiadać najczęściej używanym warunkom w zapytaniach WHERE.
- **Optymalna liczba partycji:** Zbyt mała liczba partycji nie daje efektu, zbyt duża zwiększa narzut administracyjny.
- **Automatyzacja tworzenia partycji:** Skrypty lub narzędzia generujące nowe partycje np. na kolejne miesiące/lata.
- **Monitorowanie wydajności:** Regularne sprawdzanie rozmiarów partycji, statystyk oraz planów wykonania zapytań.
- **Bezpieczeństwo danych:** Możliwość szybkiego backupu lub usunięcia starych partycji.

### 2.8.9 8. Ograniczenia i potencjalne problemy

- Brak natywnych indeksów globalnych (stan na 2025) utrudnia niektóre zapytania przekrojowe.
- Operacje DDL na tabeli nadrzędnej mogą być kosztowne przy dużej liczbie partycji.
- Niektóre narzędzia zewnętrzne mogą nie obsługiwać partycji w pełni transparentnie.
- Przenoszenie danych między partycjami wymaga operacji INSERT + DELETE lub narzędzi specjalistycznych.

### 2.8.10 9. Podsumowanie i wnioski

Partycjonowanie danych w PostgreSQL jest zaawansowanym i elastycznym narzędziem, pozwalającym na istotną poprawę wydajności oraz ułatwiającym zarządzanie dużymi zbiorami danych. Właściwy dobór typu partycjonowania, klucza oraz liczby i organizacji partycji wymaga analizy charakterystyki danych i typowych zapytań. Zaleca się regularne monitorowanie i dostosowywanie architektury partycjonowania, zwłaszcza w przypadku dynamicznie rosnących zbiorów danych.

### 2.8.11 10. Krótkie porównanie partycjonowania w PostgreSQL i innych systemach bazodanowych

Partycjonowanie danych jest wspierane przez większość nowoczesnych systemów baz danych, jednak szczegóły implementacji i dostępne możliwości mogą się różnić:

- **PostgreSQL:** Umożliwia partycjonowanie zakresowe, listowe, haszowe oraz wielopoziomowe (od wersji 10). Partycje są w pełni zintegrowane z silnikiem (od wersji 10), a operacje na partycjonowanych tabelach są transparentne dla użytkownika. Nie obsługuje jeszcze natywnych indeksów globalnych (stan na 2025).



- **Oracle Database:** Bardzo rozbudowane opcje partycjonowania (RANGE, LIST, HASH, COMPOSITE), obsługuje indeksy lokalne i globalne, automatyczne zarządzanie partycjami, także partycjonowanie na poziomie fizycznym (np. partycjonowanie indeksów, tabel LOB). Mechanizmy zaawansowane, ale często dostępne tylko w płatnych edycjach.
- **MySQL (InnoDB):** Wspiera partycjonowanie RANGE, LIST, HASH, KEY. Możliwości są jednak bardziej ograniczone niż w PostgreSQL czy Oracle. Nie wszystkie operacje i typy indeksów są wspierane na partycjonowanych tabelach.
- **Microsoft SQL Server:** Umożliwia partycjonowanie tabel i indeksów przy użyciu tzw. partition schemes i partition functions. Pozwala na łatwe przenoszenie partycji oraz obsługuje indeksy globalne, co ułatwia optymalizację zapytań przekrojowych.

**Podsumowanie:** PostgreSQL oferuje bardzo elastyczne i wydajne partycjonowanie, jednak niektóre zaawansowane funkcje (np. partycjonowanie indeksów globalnych) są jeszcze w fazie rozwoju, podczas gdy w Oracle czy SQL Server są już dojrzałymi rozwiązaniami.

### 2.8.12 11. Przykład migracji niepartycjonowanej tabeli na partycjonowaną

Migracja istniejącej tabeli na partycjonowaną w PostgreSQL wymaga kilku kroków. Oto przykładowy proces dla tabeli orders:

Załóżmy, że mamy tabelę:

```
CREATE TABLE orders (
  id serial PRIMARY KEY,
  order_date date NOT NULL,
  customer_id int,
  amount numeric
);
```

Chcemy ją partycjonować po kolumnie ``order\_date`` (zakresy roczne):

1. Zmień nazwę oryginalnej tabeli:

```
ALTER TABLE orders RENAME TO orders_old;
```

2. Utwórz nową tabelę partycjonowaną:

```
CREATE TABLE orders (
  id serial PRIMARY KEY,
  order_date date NOT NULL,
  customer_id int,
  amount numeric
) PARTITION BY RANGE (order_date);

CREATE TABLE orders_2023 PARTITION OF orders
  FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');

CREATE TABLE orders_2024 PARTITION OF orders
  FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');
```

3. Skopiuj dane do partycji:

```
INSERT INTO orders (id, order_date, customer_id, amount)
SELECT id, order_date, customer_id, amount FROM orders_old;
```

4. Sprawdź, czy dane zostały poprawnie rozdzielone:

```
SELECT tableoid::regclass, COUNT(*) FROM orders GROUP BY tableoid;
```

5. Usuń starą tabelę po upewnieniu się, że wszystko działa:

```
DROP TABLE orders_old;
```

Można też użyć narzędzi automatyzujących migracje (np. `pg_partman`), jeśli tabel jest bardzo dużo lub są bardzo duże.

### 2.8.13 12. Bibliografia

1. Dokumentacja PostgreSQL: <https://www.postgresql.org/docs/current/ddl-partitioning.html>
2. “PostgreSQL. Zaawansowane techniki programistyczne”, Grzegorz Wójtowicz, Helion 2021
3. <https://wiki.postgresql.org/wiki/Partitioning>
4. Oficjalny blog PostgreSQL: <https://www.postgresql.org/about/news/>

## PROJEKTOWANIE BAZY DANYCH - MODELE

### author

Mateusz Brokos, Szymon Błatkowski

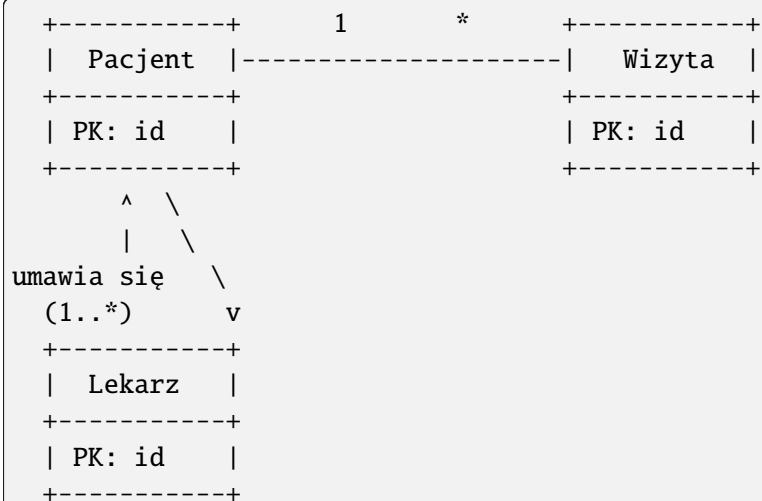
### 3.1 Wprowadzenie

Prowadzący: Piotr Czaja

Kurs: Bazy Danych 1

Celem tego raportu jest przedstawienie pełnego procesu projektowania i optymalizacji bazy danych wspierającej rejestrację oraz obsługę wizyt lekarskich w przychodni.

### 3.2 Model Konceptualny



#### Legenda

- Pacjent – może umawiać wiele wizyt (1..\*)
- Wizyta – dotyczy jednego pacjenta i jednego lekarza
- Lekarz – może prowadzić wiele wizyt (1..\*)

### 3.3 Model Logiczny

Pacjent	Lekarz	Wizyta
PK: patient_id	PK: doctor_id	PK: visit_id
first_name	first_name	patient_id (FK)
last_name	last_name	doctor_id (FK)
pesel	specialization_id (FK)	visit_datetime
birth_date		status
email	specialization_id → Specialization	notes
phone		

Dodatkowa encja:

Specialization
PK: specialization_id
name

#### Relacje

- Visit.patient\_id → Patient.patient\_id (1:N)
- Visit.doctor\_id → Doctor.doctor\_id (1:N)
- Doctor.specialization\_id → Specialization.specialization\_id (1:N)

### 3.4 Model Fizyczny

```
CREATE TABLE Specialization (
    specialization_id SERIAL PRIMARY KEY,
    name VARCHAR(100) UNIQUE
);

CREATE TABLE Doctor (
    doctor_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    license_number VARCHAR(20) UNIQUE NOT NULL,
    specialization_id INTEGER
    REFERENCES Specialization(specialization_id)
    ON UPDATE CASCADE ON DELETE SET NULL
);

CREATE TABLE Patient (
    patient_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
```

(continues on next page)

(continued from previous page)

```

pesel CHAR(11) UNIQUE NOT NULL,
birth_date DATE NOT NULL,
email VARCHAR(100),
phone VARCHAR(20)
);

CREATE TABLE Visit (
  visit_id SERIAL PRIMARY KEY,
  patient_id INTEGER NOT NULL
    REFERENCES Patient(patient_id)
    ON UPDATE CASCADE ON DELETE CASCADE,
  doctor_id INTEGER NOT NULL
    REFERENCES Doctor(doctor_id)
    ON UPDATE CASCADE ON DELETE CASCADE,
  visit_datetime TIMESTAMP NOT NULL,
  status VARCHAR(20) NOT NULL,
  notes TEXT
);

```

## 3.5 Przykładowe rekordy

### 3.5.1 Tabela Specialization

specialization_id	name
1	Internista
2	Pediatra
3	Kardiolog

### 3.5.2 Tabela Doctor

doctor_id	first_name	last_name	license_number	specialization_id
1	Anna	Nowak	PWZ123456	1
2	Paweł	Kowalski	PWZ654321	2

### 3.5.3 Tabela Patient

pa- tient_id	first_name	last_name	pesel	birth_date	email	phone
1	Maria	Wiśniewska	90010112345	1990-01-01	maria@example.com	+48123123123
2	Tomasz	Dąbrowski	85050554321	1985-05-05	tomasz@example.co	+48987654321

### 3.5.4 Tabela Visit

visit_id	patient_id	doctor_id	visit_datetime	status	notes
1	1	1	2025-06-01 10:30:00	zaplanowana	“Pierwsza wizyta”
2	2	2	2025-06-02 14:00:00	odbyta	“Kontrola po leczeniu”

## ANALIZA BAZY DANYCH I OPTYMALIZACJA ZAPYTAŃ

### 4.1 Analiza normalizacji

#### 4.1.1 Model logiczny jest w 3NF:

- Każda tabela ma pojedynczy klucz główny.
- Atrybuty niekluczowe zależą wyłącznie od klucza.
- Brak zależności przechodnich (specjalizacja wydzielona osobno).

### 4.2 Potencjalne problemy wydajnościowe

- Brak indeksów poza kluczami głównymi.
- Częste filtrowanie po *visit\_datetime* i *doctor\_id* wymaga skanów.

### 4.3 Strategie optymalizacji

#### 1. Indeksy

```
CREATE INDEX idx_visit_patient    ON Visit(patient_id);
CREATE INDEX idx_visit_doctor    ON Visit(doctor_id);
CREATE INDEX idx_visit_date      ON Visit(visit_datetime);
CREATE INDEX idx_visit_doc_date  ON Visit(doctor_id, visit_datetime);
```

#### 2. Partycjonowanie

- Partycjonuj tabelę *Visit* według miesiąca *visit\_datetime*.

#### 3. Widoki materializowane

```
CREATE MATERIALIZED VIEW mv_daily_visits AS
SELECT DATE(visit_datetime) AS day,
       doctor_id,
       COUNT(*) AS total
FROM Visit
GROUP BY day, doctor_id;
```

#### 4. Optymalizacja zapytań

- Używaj *EXPLAIN (ANALYZE, BUFFERS)* do analizy planów.
- Unikaj *SELECT \**, wybieraj konkretne kolumny.

Przykład optymalizacji:

```
-- Przed optymalizacją:
SELECT p.first_name, p.last_name, COUNT(*) AS cnt
FROM Visit v
JOIN Patient p ON v.patient_id = p.patient_id
WHERE v.visit_datetime BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY p.first_name, p.last_name;

-- Po (z indeksem na visit_datetime):
EXPLAIN ANALYZE
SELECT p.first_name, p.last_name, COUNT(*) AS cnt
FROM Visit v
JOIN Patient p ON v.patient_id = p.patient_id
WHERE v.visit_datetime BETWEEN '2025-06-01' AND '2025-12-31'
GROUP BY p.first_name, p.last_name
LIMIT 10;
```

## 4.4 Prezentacja skryptów wspomagających

```
import sqlite3
import pandas as pd

class ClinicDB:
    """
    Klasa do obsługi bazy danych kliniki SQLite z wykorzystaniem Pandas.

    Atrybuty:
        conn (sqlite3.Connection): Połączenie z bazą.
    """

    def __init__(self, db_path='clinic.db'):
        """
        Inicjalizuje połączenie z bazą.
        """
        self.conn = sqlite3.connect(db_path)

    def get_all_patients(self):
        """
        Zwraca DataFrame ze wszystkimi pacjentami.
        """
        return pd.read_sql("SELECT * FROM Patient", self.conn)

    def find_patients_by_name(self, name_part):
        """
        Wyszukuje pacjentów po fragmencie imienia/nazwiska.
        """
        q = "SELECT * FROM Patient WHERE first_name LIKE ? OR last_name LIKE ?"
        return pd.read_sql(q, self.conn, params=(f"%{name_part}%",)*2)
```

(continues on next page)



(continued from previous page)

```

.. code-block:: python

import sqlite3
import time

def measure_sqlite_queries(db_path, queries):
    """
    Mierzy czas wykonania zapytań SQL na SQLite.
    """
    conn = sqlite3.connect(db_path)
    cur = conn.cursor()
    for q in queries:
        t0 = time.time()
        cur.execute(q)
        rows = cur.fetchall()
        print(f"Czas: {time.time()-t0:.4f}s, wierszy: {len(rows)}")
    conn.close()

.. code-block:: python

import sqlite3
import pandas as pd
import matplotlib.pyplot as plt

def generate_reports(db_path="clinic.db"):
    """
    Generuje raporty i wykresy z danych kliniki.
    """
    conn = sqlite3.connect(db_path)
    df = pd.read_sql("SELECT * FROM Visit", conn)
    # ... wykresy ...
    conn.close()

```

## 4.5 Wnioski

- Model w 3NF minimalizuje redundancję i ułatwia utrzymanie.
- Indeksy i widoki materializowane znacząco przyspieszą zapytania analityczne.
- Regularne analizowanie planów (*EXPLAIN ANALYZE*) pozwoli wychwycić wąskie gardła.

## PODSUMOWANIE

Przeprowadzony projekt oraz analiza wykazały, że zaprojektowana baza danych spełnia założone wymagania funkcjonalne i wydajnościowe. Model danych został znormalizowany do trzeciej postaci normalnej, co pozwoliło ograniczyć redundancję i uprościć strukturę tabel. Implementacja indeksów oraz widoków materializowanych przyczyniła się do poprawy efektywności wykonywania zapytań. Dodatkowo zastosowanie narzędzi do dokumentacji i generowania raportów umożliwiło przygotowanie przejrzystego zestawienia kodów źródłowych oraz wizualizacji danych. Wyniki testów potwierdziły prawidłowe działanie systemu i jego zgodność ze specyfikacją założoną na etapie projektowania.

### 5.1 Spis repozytoriów

1. Sprawozdanie i kod: <https://github.com/Broksonn/Sprawozdanie.git>

Badania literaturowe:

1. Wydajność skalowanie i replikacja: [https://github.com/Broksonn/Wydajnosci\\_Skalowanie\\_i\\_Replikacja.git](https://github.com/Broksonn/Wydajnosci_Skalowanie_i_Replikacja.git)
2. Sprzęt dla bazy danych: <https://github.com/oszczeda/Sprzet-dla-bazy-danych.git>
3. Konfiguracja baz danych: [https://github.com/Chaiolites/Konfiguracja\\_baz\\_danych.git](https://github.com/Chaiolites/Konfiguracja_baz_danych.git)
4. Bezpieczeństwo: <https://github.com/BlazejUI/bezpieczenstwo.git>
5. Monitorowanie i diagnostyka: <https://github.com/GrzegorzSzczepanek/repo-wspolne.git>
6. Kopie zapasowe i odzyskiwanie danych: [https://github.com/m-smieja/Kopie\\_zapasowe\\_i\\_odzyskiwanie\\_danych.git](https://github.com/m-smieja/Kopie_zapasowe_i_odzyskiwanie_danych.git)
7. Kontrola i konserwacja: [https://github.com/Pi0trM/Kontrola\\_i\\_konserwacja.git](https://github.com/Pi0trM/Kontrola_i_konserwacja.git)
8. Partycjonowanie danych: <https://github.com/BartekHen/Partycjonowanie-danych.git>