Sprawozdanie

Wydanie 1.0.0

Szymon Piskorz

05 lip 2025

Spis treści:

ROZDZIAŁ 1

1. Wprowadzenie

Autor: Szymon Piskorz

Prowadzący: Piotr Czaja

Niniejsze sprawozdanie stanowi kompleksową dokumentację projektu bazodanowego, którego realizacja przyświecała dwóm głównym celom: dydaktycznemu oraz praktycznemu.

1.1 Cel dydaktyczny

Nadrzędnym celem było pogłębienie wiedzy i zdobycie praktycznych umiejętności z zakresu zaawansowanej administracji systemami baz danych. Proces ten obejmował badanie i aplikację następujących zagadnień:

- **Infrastruktura sprzętowa i konfiguracja:** Analiza wymagań sprzętowych oraz optymalna konfiguracja parametrów serwera bazy danych pod kątem wydajności i stabilności.
- **Kontrola, konserwacja i diagnostyka:** Poznanie narzędzi do monitorowania stanu bazy, diagnozowania wąskich gardeł oraz wdrażanie procedur konserwacyjnych, takich jak aktualizacja statystyk czy reindeksacja.
- **Wydajność, skalowanie i replikacja:** Techniki optymalizacji zapytań, strategie skalowania (pionowego i poziomego) oraz konfiguracja mechanizmów replikacji w celu zapewnienia wysokiej dostępności i rozłożenia obciążenia.
- **Partycjonowanie danych:** Zrozumienie i zastosowanie metod partycjonowania tabel w celu poprawy zarządzania dużymi zbiorami danych i zwiększenia wydajności zapytań.
- **Bezpieczeństwo:** Implementacja polityk bezpieczeństwa, zarządzanie użytkownikami i uprawnieniami, a także ochrona przed nieautoryzowanym dostępem.
- **Kopie zapasowe i odzyskiwanie danych:** Projektowanie i automatyzacja strategii tworzenia kopii zapasowych (pełnych, różnicowych, przyrostowych) oraz procedur odtwarzania danych po awarii.

1.2 Cel praktyczny i zakres projektu

Drugim celem była realizacja kompletnego projektu bazy danych o nazwie „Karnety na siłowni". Projekt ten ilustruje pełen cykl życia produktu bazodanowego, od analizy wymagań, przez modelowanie, aż po wdrożenie i analizę.

1.3 Wykorzystane technologie

- **System Bazy Danych:** PostgreSQL, SQLite
- **Język skryptowy:** Python 3.11 (z biblioteką *psycopg2*)
- **System dokumentacji:** Sphinx/Latex
- **System kontroli wersji:** Git / GitHub

1.4 Struktura sprawozdania

Dokument został podzielony na pięć rozdziałów. Po niniejszym wprowadzeniu, rozdział drugi odnosi się do zrealizowanego projektu grupowego. Rozdział trzeci szczegółowo opisuje projekt bazy danych „Karnety na siłowni", prezentując jego modele i procesy. Rozdział czwarty skupia się na analizie normalizacji, wydajności i bezpieczeństwa. Całość zamyka rozdział piąty, zawierający podsumowanie, wnioski oraz listę wykorzystanych repozytoriów.

ROZDZIAŁ 2

2. Projekt grupowy

Poniżej zamieszczono dokumentacje poszczególnych przeglądów literatury:

2.1 Sprzęt dla baz danych

2.1.1 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

2.1.2 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.

2.1.3 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.

2.1.4 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.

2.2 Sprawozdanie: Konfiguracja i Zarządzanie Bazą Danych

Authors

- Piotr Domagała
- Piotr Kotuła
- Dawid Pasikowski

2.2.1 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.

2.2.2 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.

2.2.3 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.

2.2.4 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.

2.2.5 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.

2.2.6 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
```

2.2.7 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

2.2.8 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
```

2.2.9 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).

2.2.10 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.

2.2.11 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.

2.2.12 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.

2.2.13 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.

2.2.14 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.

2.2.15 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.

—

2.3 Kopie zapasowe i odzyskiwanie danych w PostgreSQL

Autorzy

Miłosz Śmieja Szymon Piskorz Mateusz Wasilewicz

2.3.1 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.

2.3.2 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

2.3.3 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.

2.3.4 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

> **Informacja**
>
> 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

2.3.5 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
- Uproszczone procedury odzyskiwania

Ważne

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

Wskazówka

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 konsystentności 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

2.3.6 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

> **Ostrzeżenie**
>
> 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

2.4 Kontrola i konserwacja baz danych

2.4.1 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.

2.4.2 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.

2.4.3 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 silknika 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.

2.4.4 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 koniecznie po wykryciu błędu lub wystąpieniu awarii.

Konserwacja może obejmować m.in. zmianę parametrów konfiguracji bazy, przeprowadzenie procesu VACUUM, zmianę uprawnien 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.

2.4.5 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.

2.4.6 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 spowolniając 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 usename = '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.

2.4.7 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:

> autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor * liczba_wierszy

Podobnie dla ANALYZE:

> autovacuum_analyze_threshold + autovacuum_analyze_scale_factor * liczba_wierszy

2.4.8 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 teorzenia przestrzeni nazw,
- SQLite - pojedynczy schemat, uproszczony system typów,

- SQL Server - schemat jako logiczna przestrzeń obiektów, np. dbo, hr, finance.

2.4.9 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 eksplicytny SAVEPOINT.

2.4.10 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

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

Autor
Bartosz Potoczny

Data
2025-06-12

2.5.1 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.

2.5.2 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.

2.5.3 2. Definicja i cel partycjonowania

Partycjonowanie polega na logicznym podziale dużej tabeli na mniejsze, łatwiejsze w zarządzaniu fragmenty zwane partycjami. Mimo fizycznego rozdzielenia, 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.).

2.5.4 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);
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
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,
    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 telemetryczne.

3.4 Partycjonowanie wielopoziomowe (Composite/Hierarchical Partitioning)

PostgreSQL umożliwia tworzenie partycji podrzędnych, czyli partycjonowanie już partycjonowanych tabel (tzw. sub-partitioning).

Przykład:

```
CREATE TABLE measurements (
    id serial PRIMARY KEY,
    region text,
    measurement_date date,
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    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).

2.5.5 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.

2.5.6 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.
```

2.5.7 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.

2.5.8 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.

2.5.9 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.

2.5.10 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.

2.5.11 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.

2.5.12 11. Przykład migracji niepartyconowanej 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');
   ```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
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.

2.5.13 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/

ROZDZIAŁ 3

3. Projekt bazy danych „Karnety na siłowni”

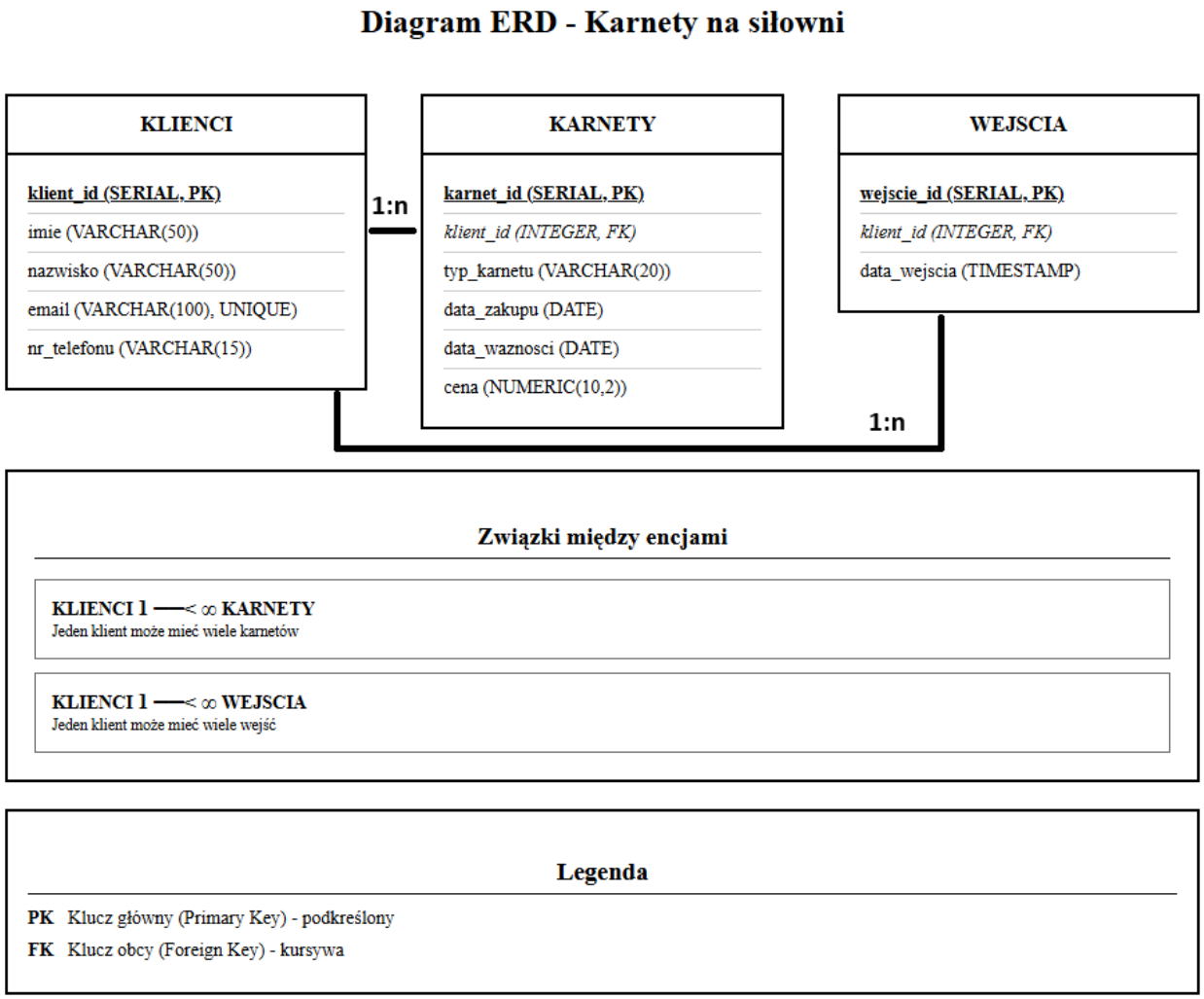
W tym rozdziale szczegółowo przedstawiono proces projektowania i implementacji bazy danych, od modelu koncepcyjnego po fizyczną realizację.

3.1 Opis procesów biznesowych

System bazodanowy został zaprojektowany w celu wsparcia trzech fundamentalnych procesów biznesowych siłowni:

1. **Rejestracja nowego klienta:** Proces polega na zebraniu podstawowych danych klienta (imię, nazwisko, dane kontaktowe) i zapisaniu ich w systemie. Każdy klient otrzymuje unikalny identyfikator. Proces ten jest realizowany poprzez operację *INSERT* na tabeli *Klienci*.
2. **Sprzedaż i aktywacja karnetu:** Klient może zakupić jeden z dostępnych karnetów. System rejestruje typ karnetu, datę zakupu, oblicza datę ważności i zapisuje cenę transakcji. Karnet jest jednoznacznie powiązany z klientem, który go zakupił. Proces ten obsługuje operacja *INSERT* na tabeli *Karnety*, z kluczem obcym wskazującym na klienta.
3. **Rejestracja wejścia na siłownię:** Przy każdej wizycie klienta, system weryfikuje, czy posiada on aktywny (ważny) karnet. Weryfikacja polega na wyszukaniu w tabeli *Karnety* rekordu powiązanego z danym klientem, którego *data_waznosci* jest późniejsza lub równa bieżącej dacie. Jeśli weryfikacja przebiegnie pomyślnie, system rejestruje wejście, zapisując identyfikator klienta i dokładny czas w tabeli *Wejscia*.

3.2 Model Koncepcyjny (ERD)



3.3 Model Logiczny

Model logiczny przekłada koncepcje na konkretną strukturę tabel, kolumn, typów danych i więzów integralności.

- **Tabela: Klienci**
  - *klient_id* (SERIAL, PK): Unikalny, automatycznie inkrementowany identyfikator klienta. Klucz główny.
  - *imie* (VARCHAR(50), NOT NULL): Imię klienta.
  - *nazwisko* (VARCHAR(50), NOT NULL): Nazwisko klienta.
  - *email* (VARCHAR(100), UNIQUE, NOT NULL): Adres e-mail, musi być unikalny, służy jako login lub do komunikacji.
  - *nr_telefonu* (VARCHAR(15)): Numer telefonu, opcjonalny.
- **Tabela: Karnety**
  - *karnet_id* (SERIAL, PK): Unikalny identyfikator transakcji zakupu karnetu.

  - *klient_id* (INTEGER, FK, NOT NULL): Klucz obcy wskazujący na klienta, który zakupił karnet.
  - *typ_karnetu* (VARCHAR(20), NOT NULL): Typ karnetu (np. «miesieczny»). Ograniczony więzem CHECK.
  - *data_zakupu* (DATE, NOT NULL): Data, w której karnet został sprzedany.
  - *data_waznosci* (DATE, NOT NULL): Data, do której karnet jest ważny.
  - *cena* (NUMERIC(10, 2), NOT NULL): Cena zapłacona za karnet. Użycie typu *NUMERIC* zapobiega błędom zaokrągleń typowym dla typów zmiennoprzecinkowych.

- **Tabela: Wejscia**
  - *wejscie_id* (SERIAL, PK): Unikalny identyfikator zdarzenia wejścia.
  - *klient_id* (INTEGER, FK, NOT NULL): Klucz obcy wskazujący na wchodzącego klienta.
  - *data_wejscia* (TIMESTAMP, NOT NULL): Dokładna data i godzina wejścia. Wartość domyślna to *CURRENT_TIMESTAMP*.

3.4 Model Fizyczny (Kod SQL)

Poniższy kod DDL (Data Definition Language) dla PostgreSQL tworzy opisaną strukturę bazy danych.

Listing 1: Skrypt tworzący strukturę bazy danych

```
-- Tabela przechowująca dane klientów siłowni.
-- Każdy klient jest unikalnie identyfikowany przez email.
CREATE TABLE Klienci (
    klient_id SERIAL PRIMARY KEY,
    imie VARCHAR(50) NOT NULL,
    nazwisko VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    nr_telefonu VARCHAR(15)
);

-- Tabela przechowująca informacje o zakupionych karnetach.
-- Więzy integralności (FOREIGN KEY z ON DELETE CASCADE) zapewniają,
-- że usunięcie klienta spowoduje usunięcie jego karnetów.
CREATE TABLE Karnety (
    karnet_id SERIAL PRIMARY KEY,
    klient_id INTEGER NOT NULL,
    typ_karnetu VARCHAR(20) NOT NULL,
    data_zakupu DATE NOT NULL,
    data_waznosci DATE NOT NULL,
    cena NUMERIC(10, 2) NOT NULL,

    CONSTRAINT fk_klient
        FOREIGN KEY(klient_id)
        REFERENCES Klienci(klient_id)
        ON DELETE CASCADE,

    CONSTRAINT chk_typ_karnetu
        CHECK (typ_karnetu IN ('miesieczny', 'trzymiesieczny', 'polroczny'))
);
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
-- Tabela rejestrująca wejścia klientów.
-- Każde wejście jest powiązane z istniejącym klientem.
CREATE TABLE Wejscia (
    wejscie_id SERIAL PRIMARY KEY,
    klient_id INTEGER NOT NULL,
    data_wejscia TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,

    CONSTRAINT fk_klient
        FOREIGN KEY(klient_id)
        REFERENCES Klienci(klient_id)
        ON DELETE CASCADE
);
```

ROZDZIAŁ 4

4. Normalizacja, Wydajność i Bezpieczeństwo

W tym rozdziale przeprowadzono analizę kluczowych niefunkcjonalnych aspektów zaprojektowanej bazy danych.

4.1 Analiza normalizacji

Normalizacja jest procesem projektowania schematu bazy danych w celu zminimalizowania redundancji danych i wyeliminowania niepożądanych charakterystyk, takich jak anomalie wstawiania, aktualizacji i usuwania. Zaproponowany schemat jest zgodny z **trzecią postacią normalną (3NF)**.

Zaczynamy od: jednej dużej tabeli (czyli dane nie są jeszcze uporządkowane)

Wyobraźmy sobie, że wszystko zapisujemy w jednej tabeli *Rejestr_Silowni*.

Tabela 1: Przykład nieuporządkowanej tabeli (*Rejestr_Silowni*)

Imie_Klienta	Nazwisko_Klienta	Email_Klienta	Typ_Karnetu	Cena_Karnetu	Daty_Wejsc	
Jan	Kowalski	jan.kowalski@ex.com	miesieczny	120.00	«2025-07-02, 05»	2025-07-
Anna	Nowak	anna.nowak@ex.com	polroczny	500.00	«2025-07-03»	
Jan	Kowalski	jan.kowalski@ex.com	trzymiesieczny	300.00	«2025-08-01, 04»	2025-08-

Problemy z taką tabelą:

- **Nie można dodać klienta**, jeśli jeszcze nic nie kupił.
- **Usunięcie jednego wejścia** może przypadkiem usunąć dane o kliencie.
- **Zmiana adresu e-mail** klienta wymaga edytowania kilku wierszy.
- **Dużo powtarzających się danych** — np. Jan Kowalski występuje kilka razy.

Krok 1: Pierwsza postać normalna (1NF)

Tabela jest w 1NF, jeśli nie ma list w kolumnach — każda komórka ma jedną wartość.

U nas kolumna *Daty_Wejsc* zawiera listy. Rozbijmy to:

1. Tworzymy tabelę *Klienci_Karnety* — każdy karnet to osobny wiersz.
2. Tworzymy tabelę *Wejscia*, gdzie każde wejście to jeden rekord.

Tabela 2: Tabela *Klienci_Karnety* (po 1NF)

KarnetID	Imie_Klienta	Nazwisko_Klienta	Email_Klienta	Typ_Karnetu
1	Jan	Kowalski	jan.kowalski@ex.com	miesieczny
2	Anna	Nowak	anna.nowak@ex.com	polroczny
3	Jan	Kowalski	jan.kowalski@ex.com	trzymiesieczny

Tabela 3: Tabela *Wejscia* (po 1NF)

KarnetID_FK	Data_Wejscia
1	2025-07-02
1	2025-07-05
2	2025-07-03
3	2025-08-01
3	2025-08-04

Problem: Nadal mamy powtarzające się dane o klientach w *Klienci_Karnety*.

Krok 2: Druga postać normalna (2NF)

W 2NF dane powinny zależeć od całego klucza głównego, a nie tylko części.

W naszej tabeli *Klienci_Karnety* dane o kliencie nie zależą od *KarnetID*, tylko od klienta. Trzeba to rozdzielić:

1. Tworzymy tabelę *Klienci* z unikalnym ID.
2. W tabeli *Karnety* trzymamy tylko typ karnetu i odwołanie do klienta.

Tabela 4: Tabela *Klienci* (po 2NF)

KlientID	Imie	Nazwisko	Email
101	Jan	Kowalski	jan.kowalski@ex.com
102	Anna	Nowak	anna.nowak@ex.com

Tabela 5: Tabela *Karnety* (po 2NF)

KarnetID	KlientID_FK	Typ_Karnetu
1	101	miesieczny
2	102	polroczny
3	101	trzymiesieczny

Tabela *Wejscia* zostaje bez zmian, ale możemy też rozważyć, czy nie lepiej byłoby wiązać ją bezpośrednio z klientem.

Krok 3: Trzecia postać normalna (3NF)

Tutaj chodzi o to, żeby dane nie zależały od innych danych niebędących kluczem.

Jeśli w *Karnety* dodamy np. *Cena*, która zależy od *Typ_Karnetu*, to mamy tzw. zależność przechodnią.

Zamiast tego możemy utworzyć tabelę *Cennik* z kolumnami *Typ_Karnetu* i *Cena*.

W naszym projekcie jednak **trzymamy cenę w tabeli `Karnety`**, ponieważ może się ona zmieniać w czasie — to celowe odstępstwo (denormalizacja), żeby zachować historię.

Podsumowanie

Zaczęliśmy od nieuporządkowanej tabeli, a skończyliśmy na trzech powiązanych:

- *Klienci*
- *Karnety*
- *Wejscia*

Dzięki temu dane nie powtarzają się, łatwo je edytować i są bezpieczne przed przypadkowymi błędami.

4.2 Analiza wydajności i indeksowanie

Wydajność zapytań jest kluczowa dla responsywności systemu. Podstawową techniką optymalizacji jest strategiczne stosowanie indeksów.

Identyfikacja kandydatów do indeksowania: * **Klucze obce:** Kolumny używane jako klucze obce (*Karnety.klient_id*, *Wejscia.klient_id*) są głównymi kandydatami do indeksowania. Indeksy te drastycznie przyspieszają operacje *JOIN* oraz wyszukiwanie rekordów powiązanych z danym klientem. PostgreSQL automatycznie nie tworzy indeksów na kluczach obcych, więc należy je dodać ręcznie. * **Często filtrowane kolumny:** Kolumna *Karnety.data_waznosci* będzie często używana w klauzuli *WHERE* do sprawdzania aktywnych karnetów. Dodanie na niej indeksu przyspieszy ten krytyczny proces biznesowy.

Przykładowa implementacja indeksów:

```
-- Indeks na kluczu obcym w tabeli Karnety
CREATE INDEX idx_karnety_klient_id ON Karnety(klient_id);

-- Indeks na kluczu obcym w tabeli Wejscia
CREATE INDEX idx_wejscia_klient_id ON Wejscia(klient_id);

-- Indeks wspomagający wyszukiwanie aktywnych karnetów
CREATE INDEX idx_karnety_data_waznosci ON Karnety(data_waznosci);
```

Analiza planu zapytania (`EXPLAIN ANALYZE`): Przed dodaniem indeksu *idx_karnety_klient_id*, zapytanie o wszystkie karnety danego klienta skutkowałoby pełnym skanowaniem tabeli (*Seq Scan*). Po jego dodaniu, planer zapytań PostgreSQL wykorzysta znacznie szybszy *Index Scan*, co przy dużej liczbie rekordów może skrócić czas wykonania zapytania z sekund do milisekund.

4.3 Zarządzanie bezpieczeństwem

Bezpieczeństwo danych osobowych i operacyjnych jest priorytetem. Zastosowano model bezpieczeństwa oparty na rolach (Role-Based Access Control).

Definicja ról: * **`rola_admin`**: Superużytkownik z pełnymi uprawnieniami do wszystkich tabel (CRUD - Create, Read, Update, Delete). Przeznaczona dla administratorów bazy danych. * **`rola_recepcja`**: Rola dla pracowników recepcji. Powinna mieć uprawnienia do:

- *SELECT* na *Klienci*.
- *INSERT* do *Klienci*.
- *SELECT*, *INSERT* na *Karnety*.
- *SELECT*, *INSERT* na *Wejscia*.

- Brak uprawnień *DELETE* i *UPDATE* na większości danych w celu ochrony przed przypadkowym usunięciem.
- **`rola_analityk`**: Rola tylko do odczytu (*SELECT*) na wszystkich tabelach. Przeznaczona dla analityków biznesowych generujących raporty.

Przykładowa implementacja ról i uprawnień:

```
-- Tworzenie ról
CREATE ROLE rola_recepcja;
CREATE ROLE rola_analityk;

-- Nadawanie uprawnień dla recepcji
GRANT SELECT, INSERT ON Klienci, Karnety, Wejscia TO rola_recepcja;
GRANT USAGE, SELECT ON SEQUENCE klienci_klient_id_seq, karnety_karnet_id_seq, wejscia_
↪wejscie_id_seq TO rola_recepcja;


-- Nadawanie uprawnień dla analityka
GRANT SELECT ON ALL TABLES IN SCHEMA public TO rola_analityk;

-- Tworzenie użytkowników i przypisywanie im ról
CREATE USER pracownik_recepcji WITH PASSWORD 'bezpieczne_haslo';
GRANT rola_recepcja TO pracownik_recepcji;
```

4.4 Skrypty wspomagające

Listing 1: Skrypty w PostgreSQL

```
import psycopg2
from datetime import date, timedelta

# ... (konfiguracja połączenia DB_CONFIG) ...

def generuj_raport_wygasajacych_karnetow(dni_do_konca=7):
    """
    Znajduje klientów, których karnety wygasają
    w ciągu najbliższych 'dni_do_konca' dni.
    """
    # ... (logika połączenia z bazą) ...
    query = """
    SELECT k.imie, k.nazwisko, k.email, kr.data_waznosci
    FROM Klienci k
    JOIN Karnety kr ON k.klient_id = kr.klient_id
    WHERE kr.data_waznosci BETWEEN %s AND %s
    ORDER BY kr.data_waznosci ASC;
    """
    dzis = date.today()
    data_koncowa = dzis + timedelta(days=dni_do_konca)
    cur.execute(query, (dzis, data_koncowa))
    # ... (logika wyświetlania raportu) ...

def znajdz_najaktywniejszych_klientow(data_od, data_do, limit=5):
 """Wyświetla listę najczęściej wchodzących klientów w danym okresie."""
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
 print(f"\n--- TOP {limit} najaktywniejszych klientów od {data_od} do {data_do} ---")
 conn = get_connection()
 query = """
 SELECT k.imie, k.nazwisko, COUNT(w.wejscie_id) AS liczba_wejsc
 FROM Wejscia w
 JOIN Klienci k ON w.klient_id = k.klient_id
 WHERE w.data_wejscia::date BETWEEN %s AND %s
 GROUP BY k.klient_id, k.imie, k.nazwisko
 ORDER BY liczba_wejsc DESC
 LIMIT %s;
 """
 with conn.cursor() as cur:
     cur.execute(query, (data_od, data_do, limit))
     for row in cur.fetchall():
         print(f"Klient: {row[0]} {row[1]}, Liczba wejść: {row[2]}")
 conn.close()

 def generuj_raport_sprzedazy(data_od, data_do):
 """Oblicza sumę sprzedaży i liczbę sprzedanych karnetów w danym okresie."""
 print(f"\n--- Raport sprzedaży od {data_od} do {data_do} ---")
 conn = get_connection()
 query = "SELECT COUNT(karnet_id), SUM(cena) FROM Karnety WHERE data_zakupu BETWEEN %s␣
↪AND %s;"
 with conn.cursor() as cur:
     cur.execute(query, (data_od, data_do))
     result = cur.fetchone()
     print(f"Liczba sprzedanych karnetów: {result[0] or 0}")
     print(f"Łączna kwota sprzedaży: {result[1] or 0.00} PLN")
 conn.close()
```

Listing 2: Skrypty w SQLite

```
import psycopg2
from datetime import date, timedelta

# --- KONFIGURACJA ---
DB_FILE = "silownia.db" # Nazwa pliku bazy danych

def get_connection():
 """Nawiązuje połączenie z bazą danych SQLite."""
 return sqlite3.connect(DB_FILE)

 def znajdz_klientow_z_wygaslym_karnetem_sqlite(dni_od_wyga_do_wyga):
 """Znajduje klientów, których ostatni karnet wygasł w zadanym przedziale dni temu."""
 print(f"\n--- [SQLite] Klienci, których karnet wygasł od {dni_od_wyga_do_wyga[0]} do
↪{dni_od_wyga_do_wyga[1]} dni temu ---")
 conn = get_connection()
 # W SQLite do znalezienia ostatniego karnetu używamy podzapytania z GROUP BY i MAX()
 query = """
 SELECT k.imie, k.nazwisko, k.email, sub.max_data
 FROM Klienci k
 JOIN (
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    SELECT klient_id, MAX(data_waznosci) as max_data FROM Karnety GROUP BY klient_id
) AS sub ON k.klient_id = sub.klient_id
WHERE sub.max_data BETWEEN ? AND ?;
"""
date_to = (date.today() - timedelta(days=dni_od_wyga_do_wyga[0])).isoformat()
date_from = (date.today() - timedelta(days=dni_od_wyga_do_wyga[1])).isoformat()

with conn: # Użycie `with conn` automatycznie zarządza transakcjami
    cur = conn.cursor()
    cur.execute(query, (date_from, date_to))
    for row in cur.fetchall():
        print(f"Klient: {row[0]} {row[1]}, Email: {row[2]}, Karnet wygasł: {row[3]}")

def generuj_raport_sprzedazy_sqlite(data_od, data_do):
"""Oblicza sumę sprzedaży i liczbę sprzedanych karnetów w danym okresie."""
print(f"\n--- [SQLite] Raport sprzedaży od {data_od} do {data_do} ---")
conn = get_connection()
query = "SELECT COUNT(karnet_id), SUM(cena) FROM Karnety WHERE data_zakupu BETWEEN ?␣
↪AND ?;"
with conn:
    cur = conn.cursor()
    cur.execute(query, (data_od, data_do))
    result = cur.fetchone()
    print(f"Liczba sprzedanych karnetów: {result[0] or 0}")
    print(f"Łączna kwota sprzedaży: {result[1] or 0.00} PLN")

def znajdz_najaktywniejszych_klientow_sqlite(data_od, data_do, limit=5):
"""Wyświetla listę najczęściej wchodzących klientów w danym okresie."""
print(f"\n--- [SQLite] TOP {limit} najaktywniejszych klientów od {data_od} do {data_do}␣
↪---")
conn = get_connection()
# Używamy funkcji DATE() do wyciągnięcia daty z pełnego timestampa
query = """
SELECT k.imie, k.nazwisko, COUNT(w.wejscie_id) AS liczba_wejsc
FROM Wejscia w
JOIN Klienci k ON w.klient_id = k.klient_id
WHERE DATE(w.data_wejscia) BETWEEN ? AND ?
GROUP BY k.klient_id
ORDER BY liczba_wejsc DESC
LIMIT ?;
"""
with conn:
    cur = conn.cursor()
    cur.execute(query, (data_od, data_do, limit))
    for row in cur.fetchall():
        print(f"Klient: {row[0]} {row[1]}, Liczba wejść: {row[2]}")
```

ROZDZIAŁ 5

5. Podsumowanie, Wnioski i Repozytoria

5.1 Podsumowanie projektu

Niniejszy projekt stanowił pełne ćwiczenie inżynierskie, obejmujące cały cykl życia systemu bazodanowego. Rozpoczynając od analizy potrzeb biznesowych fikcyjnej siłowni, poprzez staranne modelowanie danych, aż po fizyczną implementację i analizę aspektów niefunkcjonalnych, projekt ten z powodzeniem przełożył wymagania na działające, bezpieczne i wydajne rozwiązanie. Zastosowanie normalizacji zapewniło integralność danych, podczas gdy świadome planowanie indeksów i polityk bezpieczeństwa przygotowało system do działania w rzeczywistym środowisku.

5.2 Wnioski

Realizacja projektu pozwoliła na sformułowanie następujących wniosków:

1. **Modelowanie jest kluczowe:** Czas poświęcony na staranne stworzenie modelu koncepcyjnego i logicznego procentuje na etapie implementacji i późniejszej konserwacji. Dobrze zaprojektowany schemat jest intuicyjny i łatwy do rozbudowy.
2. **Normalizacja to kompromis:** Chociaż dążenie do wyższych postaci normalnych jest teoretycznie pożądane, w praktyce należy uwzględniać również wydajność i logikę biznesową. Celowe, udokumentowane odstępstwa (jak przechowywanie ceny w momencie transakcji) są często uzasadnione.
3. **Wydajność i bezpieczeństwo nie są opcjonalne:** Aspekty te muszą być uwzględniane od samego początku procesu projektowego, a nie dodawane jako „łatki” na końcu. Strategiczne indeksowanie i model bezpieczeństwa oparty na rolach to fundamenty stabilnego systemu.
4. **Praktyka utrwala teorię:** Projekt ten był nieocenionym doświadczeniem, które pozwoliło na praktyczne zastosowanie i głębsze zrozumienie teoretycznych koncepcji omawianych na zajęciach, takich jak replikacja, partycjonowanie czy zaawansowane strategie backupu.

5.3 Możliwe kierunki dalszego rozwoju

Zaprojektowana baza danych stanowi solidny fundament, który można rozwijać w wielu kierunkach, aby zwiększyć jej wartość biznesową:

- **Moduł rezerwacji zajęć:** Dodanie tabel *Zajecia*, *Instruktorzy* oraz *Rezerwacje* w celu umożliwienia klientom rezerwacji miejsc na zajęciach grupowych.
- **Integracja z systemem płatności:** Połączenie z bramką płatniczą w celu automatyzacji sprzedaży karnetów online.
- **Aplikacja kliencka:** Stworzenie aplikacji mobilnej lub webowej dla klientów, gdzie mogliby sprawdzać ważność swojego karnetu, historię wejść i rezerwować zajęcia.
- **Zaawansowana analityka:** Budowa hurtowni danych i wykorzystanie narzędzi BI (Business Intelligence) do analizy trendów, np. godzin największego obłożenia siłowni, najpopularniejszych typów karnetów czy segmentacji klientów.

5.4 Spis repozytoriów

1. **Repozytorium niniejszego sprawozdania:** https://github.com/HoszeQ/karnety_silownia_sprawozdanie
2. **Repozytorium projektu grupowego:** https://github.com/m-smieja/Kopie_zapasowe_i_odzyskiwanie_danych
3. **Repozytorium pracy pt. Konfiguracja_baz_danych:** https://github.com/Chaiolites/Konfiguracja_baz_danych.git
4. **Repozytorium pracy pt. Kontrola_i_konserwacja:** https://github.com/Pi0trM/Kontrola_i_konserwacja.git
5. **Repozytorium pracy pt. Partycjonowanie-danych:** https://github.com/BartekHen/Partycjonowanie-danych.git
6. **Repozytorium pracy pt. Sprzet-dla-bazy-danych:** https://github.com/oszczeda/Sprzet-dla-bazy-danych.git
7. **Repozytorium pracy pt. Wydajnosc-Skalowanie-i-Replikacja:** https://github.com/Broksonn/Wydajnosc_Skalowanie_i_Replikacja.git
8. **Repozytorium pracy pt. Bezpieczenstwo:** https://github.com/BlazejUl/bezpieczenstwo.git
9. **Repozytorium pracy pt. Monitorowanie-i-diagnostyka:** https://github.com/GrzegorzSzczepanek/repo-wspolne.git