BazyDanych

Wydanie 1.0

Bartosz Potoczny

05 lip 2025

Spis Treści

Autorzy: Bartkosz Potoczny **Data:** 05 lip 2025

Rozdzia 1

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

Autor
Bartosz Potoczny

Data
2025-06-12

1.1 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.

1.2 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.

1.3 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.).

1.4 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,
    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. 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).

1.5 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.

1.6 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 >=
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
↪'2024-01-01';

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

1.7 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.

1.8 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.

1.9 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.

1.10 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.

1.11 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.

1.12 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,
   ```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    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.

1.13 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/

Rozdzia 2

Wydajność, skalowanie i replikacja

Autorzy

- Mateusz Brokos
- Szymon Blatkowski
- Maciej Gołębiowski

2.1 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.

2.2 Buforowanie oraz zarządzanie połączeniami

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

2.2.1 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.

2.2.2 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.

2.3 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.

2.3.1 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.

2.3.2 1. Współbieżność w bazach danych

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

Na wysoką współbieżność wpływają:

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

Do głównych wyzwań należą:

- Hotspoty danych, czyli miejsca często jednocześnie odczytywane lub modyfikowane, tworzące wąskie gardła.

- Zakleszczenia, gdy transakcje wzajemnie się blokują, uniemożliwiając zakończenie pracy.
- Głód zasobów, kiedy niektóre operacje monopolizują zasoby, ograniczając dostęp innym procesom i obniżając wydajność.

2.3.3 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.

2.3.4 3. Responsywność bazy danych

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

Na czas reakcji bazy wpływają:

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

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

2.3.5 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.

2.3.6 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.

2.3.7 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.

2.3.8 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.

2.3.9 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.

2.3.10 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.

2.3.11 Unikanie wąskich gardeł:

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

2.4 Skalowanie

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

2.5 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.

2.5.1 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.

2.5.2 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.

2.5.3 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).

2.6 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.

2.7 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.

2.8 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.

2.9 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.

Rozdzia 3

Sprzęt dla baz danych

3.1 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

3.2 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.

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.2.5 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.

3.2.6 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.

3.3 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.

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.3.4 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.

3.3.5 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.

3.3.6 Chłodzenie

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

3.4 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.

Rozdzia 4

Sprawozdanie: Konfiguracja i Zarządzanie Bazą Danych

Authors

- Piotr Domagała
- Piotr Kotuła
- Dawid Pasikowski

4.1 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.

4.2 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.

4.3 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.

4.4 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.

4.5 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.

4.6 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
```

4.7 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

4.8 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
```

4.9 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.

4.9.1 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.

4.9.2 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.

4.9.3 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.

4.9.4 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).

4.10 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).

4.10.1 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/`).

4.10.2 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.

4.10.3 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';
   ```

4.10.4 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';
```

4.10.5 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).

4.10.6 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.

4.11 11. Planowanie

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

4.11.1 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`).

4.11.2 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.

4.11.3 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).

4.11.4 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.

4.11.5 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)
```

4.11.6 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.

4.12 12. Tabele – rozmiar, planowanie i monitorowanie

4.12.1 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.

4.12.2 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.

4.12.3 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.

4.12.4 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.

4.12.5 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`

4.12.6 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.

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

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

4.13.1 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;
```

4.13.2 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;
```

4.13.3 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.

4.13.4 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.

4.14 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.

4.14.1 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.

4.14.2 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).

4.14.3 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.

4.14.4 14.4 Znaczenie zarządzania rozmiarem

Niewłaściwe zarządzanie przestrzenią dyskową może prowadzić do:

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

4.15 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.

—

Rozdzia 5

Bezpieczeństwo

Autorzy

- Katarzyna Tarasek
- Błażej Uliasz

5.1 1. pg_hba.conf — opis pliku konfiguracyjnego PostgreSQL

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

5.1.1 Format pliku

Każdy wiersz odpowiada jednej regule dostępu:

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

Opis elementów:

5.1.2 Znaczenie Elementów

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

5.1.3 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.

5.1.4 Metody uwierzytelniania

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

5.1.5 Przykładowy wpis

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

5.1.6 Zmiany i przeładowanie

Po zmianach w pliku należy przeładować konfigurację PostgreSQL:

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

5.2 2. Uprawnienia użytkownika

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

5.2.1 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)

5.2.2 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

5.2.3 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

5.2.4 Poziom tabeli

Uprawnienia do całej tabeli :

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

5.2.5 Przykład

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

5.3 3. Zarządzanie użytkownikami a dane wprowadzone

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

5.3.1 Tworzenie i modyfikacja użytkowników

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

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

5.3.2 Usuwanie użytkowników

Do usuwania użytkowników, używamy polecenia “DROP USER‘:

```
DROP USER username;
```

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

5.3.3 Usunięcie użytkownika, a dane które posiadał

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

5.3.4 Usunięcie użytkowników, a obietky

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

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

Aby zapobiec takim błędom stosujemy poniższe rozwiazanie:

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

5.4 4. Zabezpieczenie połączenia przez SSL/TLS

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

5.4.1 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'
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
ssl_ca_file = 'root.crt'
ssl_min_protocol_version = 'TLSv1.3'
```

oraz «»pg_hba.conf»»:

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

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

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

Konfiguracja klienta: parametry SSL, których możemy użyć.

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

5.4.2 Monitorowanie i testowanie SSL/TLS

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

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

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

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

5.5 5. Szyfrowanie danych

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

5.5.1 Szyfrowanie transmisji

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

5.5.2 Szyfrowanie całego dysku

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

5.5.3 Szyfrowanie na poziomie kolumn z użyciem pgcrypto

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

- symetryczne szyfrowanie

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

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

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

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

5.5.4 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.

5.5.5 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.

Rozdzia 6

Kontrola i konserwacja baz danych

6.1 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.

6.2 Podział konserwacji baz danych

Autor: Bartłomiej Czyż

6.2.1 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.

6.2.2 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.

6.3 Różnice konserwacyjne w zależności od rodzaju bazy danych

Autor: Bartłomiej Czyż

6.3.1 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).

6.3.2 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.

6.3.3 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.

6.3.4 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.

6.4 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.

6.5 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.

6.5.1 Uruchamianie

Linux:

```
sudo systemctl start postgresql
```

Windows CMD:

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

Windows PowerShell

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

6.5.2 Zatrzymywanie

Linux:

```
sudo systemctl stop postgresql
```

Windows CMD:

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

Windows PowerShell

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

6.5.3 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.

6.6 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.

6.6.1 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;
```

6.6.2 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);
```

6.6.3 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';
```

6.6.4 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.

6.7 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.

6.7.1 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

6.8 Schemat bazy danych

Autor: Bartłomiej Czyż

6.8.1 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).

6.8.2 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.

6.8.3 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.

6.9 Transakcje

Autor: Bartłomiej Czyż

6.9.1 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.

6.9.2 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.

6.9.3 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.

6.9.4 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.

6.10 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

Rozdzia 7

Kopie zapasowe i odzyskiwanie danych w PostgreSQL

Autorzy

Miłosz Śmieja Szymon Piskorz Mateusz Wasilewicz

7.1 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.

7.2 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.

7.2.1 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

7.2.2 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

7.2.3 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

7.2.4 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

7.3 Mechanizmy wbudowane do tworzenia kopii zapasowych poszczególnych baz danych

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

7.3.1 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

7.3.2 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

7.3.3 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

7.3.4 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.

7.4 Odzyskiwanie usuniętych lub uszkodzonych danych

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

7.4.1 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

7.4.2 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.

7.4.3 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

7.4.4 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

7.4.5 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

7.5 Dedykowane oprogramowanie i skrypty zewnętrzne do automatyzacji

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

7.5.1 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.

7.5.2 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

7.5.3 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

7.5.4 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.

7.5.5 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

7.5.6 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

7.6 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

7.6.1 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.

7.6.2 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

Rozdzia 8

Projekt i Implementacja Bazy Danych

W niniejszym rozdziale przedstawiono kompletny projekt bazy danych „Sklep”. Proces został podzielony na trzy kluczowe etapy: od ogólnego modelu koncepcyjnego, przez szczegółowy model logiczny, aż po model fizyczny w postaci schematu SQL i kodu generatora.

8.1 Model Koncepcyjny

Model koncepcyjny definiuje kluczowe byty biznesowe oraz fundamentalne relacje między nimi. Stanowi on wysokopoziomową mapę systemu, która identyfikuje główne obiekty i ich wzajemne powiązania.

8.2 Model Logiczny

Model logiczny jest uszczegółowieniem modelu koncepcyjnego. Definiuje on strukturę tabel, atrybuty (kolumny) dla każdej z nich oraz klucze główne (PK) i obce (FK), które zapewniają integralność relacyjną. Jest on niezależny od konkretnego systemu zarządzania bazą danych.

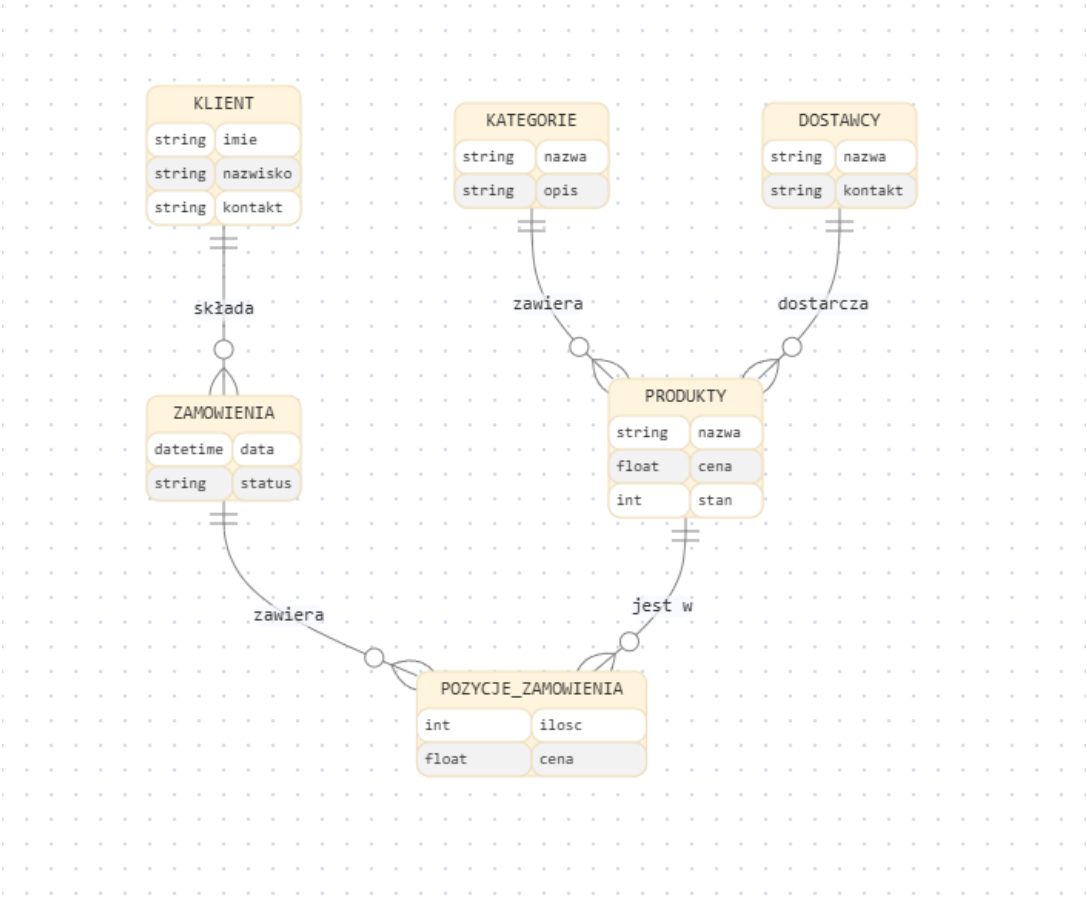
8.3 Model Fizyczny (Schemat SQL)

Model fizyczny to konkretna implementacja modelu logicznego w wybranym systemie DBMS (w tym przypadku PostgreSQL i SQLite). Definiuje on precyzyjne typy danych (np. *VARCHAR(100)*, *NUMERIC(10,2)*), indeksy i inne elementy specyficzne dla danej technologii. Poniżej przedstawiono schemat dla bazy PostgreSQL.

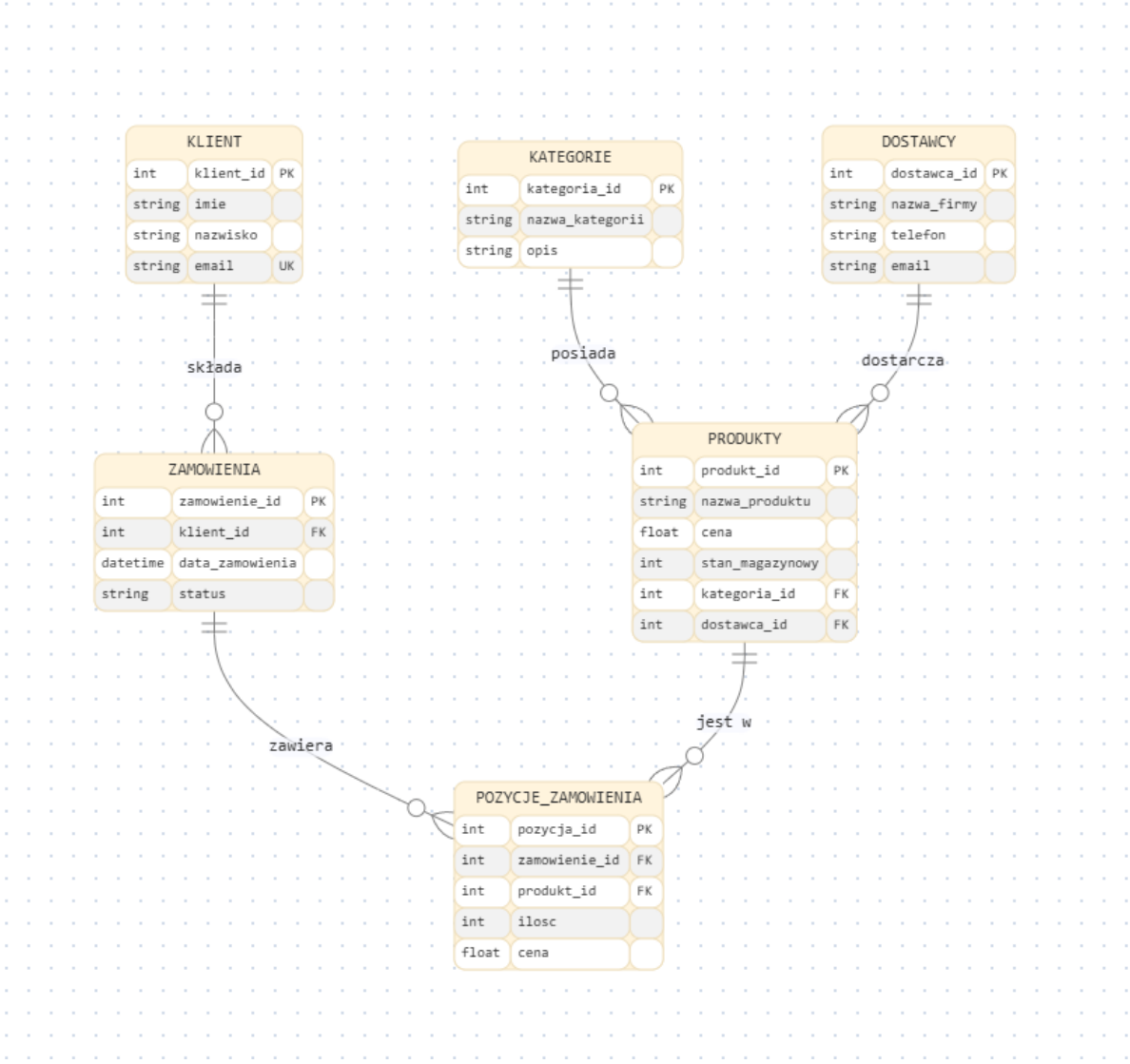
Listing 1: Fizyczny schemat bazy danych dla PostgreSQL

```
CREATE TABLE Kategorie (
    kategoria_id SERIAL PRIMARY KEY,
    nazwa_kategorii VARCHAR(100) NOT NULL,
    opis TEXT
);
CREATE TABLE Dostawcy (
    dostawca_id SERIAL PRIMARY KEY,
    nazwa_firmy VARCHAR(255) NOT NULL,
    telefon VARCHAR(20),
```

(ciąg dalszy na następnej stronie)



Rys. 1: **Rysunek 1: Model koncepcyjny.** Widoczne są główne encje: Klient, Zamówienie, Produkt, Kategoria, Dostawca oraz Pozycje Zamówienia, a także powiązania między nimi, takie jak „składa”, „zawiera” czy „dostarcza”.



Rys. 2: **Rysunek 2: Model logiczny.** Szczegółowo przedstawia strukturę każdej tabeli, w tym nazwy pól, ich typy generyczne oraz powiązania zrealizowane za pomocą kluczy obcych.

(kontynuacja poprzedniej strony)

```
    email VARCHAR(255)
);
CREATE TABLE Klienci (
    klient_id SERIAL PRIMARY KEY,
    imie VARCHAR(100) NOT NULL,
    nazwisko VARCHAR(100) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE
);
CREATE TABLE Produkty (
    produkt_id SERIAL PRIMARY KEY,
    nazwa_produktu VARCHAR(255) NOT NULL,
    cena NUMERIC(10, 2) NOT NULL,
    stan_magazynowy INT NOT NULL,
    kategoria_id INT REFERENCES Kategorie(kategoria_id),
    dostawca_id INT REFERENCES Dostawcy(dostawca_id)
);
CREATE TABLE Zamowienia (
    zamowienie_id SERIAL PRIMARY KEY,
    klient_id INT NOT NULL REFERENCES Klienci(klient_id),
    data_zamowienia TIMESTAMP NOT NULL,
    status VARCHAR(50) NOT NULL
);
CREATE TABLE PozycjeZamowienia (
    pozycja_id SERIAL PRIMARY KEY,
    zamowienie_id INT REFERENCES Zamowienia(zamowienie_id),
    produkt_id INT REFERENCES Produkty(produkt_id),
    ilosc INT NOT NULL,
    cena DECIMAL(10, 2) NOT NULL
);
```

8.4 Opis Tabel

Poniżej znajduje się szczegółowy opis przeznaczenia każdej z tabel.

8.4.1 Klienci

Przechowuje informacje o klientach sklepu.

- `klient_id` (PK): Unikalny identyfikator klienta.
- `imie`, `nazwisko`: Dane osobowe.
- `email` (UNIQUE): Unikalny adres email służący do kontaktu i logowania.

8.4.2 Kategorie

Słownik kategorii, do których przypisane są produkty.

- `kategoria_id` (PK): Unikalny identyfikator kategorii.
- `nazwa_kategorii`: Nazwa, np. „Nabiał”, „Pieczywo”.
- `opis`: Dodatkowy opis kategorii.

8.4.3 Dostawcy

Tabela przechowująca dane o dostawcach towaru.

- `dostawca_id` (PK): Unikalny identyfikator dostawcy.
- `nazwa_firmy`, `telefon`, `email`: Dane kontaktowe dostawcy.

8.4.4 Produkty

Główna tabela z informacjami o wszystkich produktach dostępnych w sklepie.

- `produkt_id` (PK): Unikalny identyfikator produktu.
- `nazwa_produktu`, `cena`: Podstawowe informacje o produkcie.
- `stan_magazynowy`: Liczba dostępnych sztuk.
- `kategoria_id` (FK): Klucz obcy łączący z tabelą `Kategorie`.
- `dostawca_id` (FK): Klucz obcy łączący z tabelą `Dostawcy`.

8.4.5 Zamowienia

Przechowuje informacje o zamówieniach złożonych przez klientów.

- `zamowienie_id` (PK): Unikalny identyfikator zamówienia.
- `klient_id` (FK): Klucz obcy wskazujący, który klient złożył zamówienie.
- `data_zamowienia`: Data i godzina złożenia zamówienia.
- `status`: Aktualny status, np. „Złożone”, „Wysłane”, „Anulowane”.

8.4.6 PozycjeZamowienia

Tabela asocjacyjna (łącząca) zamówienia z produktami. Określa, jakie produkty i w jakiej ilości znalazły się w danym zamówieniu.

- `pozycja_id` (PK): Unikalny identyfikator pozycji.
- `zamowienie_id` (FK): Klucz obcy łączący z tabelą `Zamowienia`.
- `produkt_id` (FK): Klucz obcy łączący z tabelą `Produkty`.
- `ilosc`: Liczba sztuk danego produktu w zamówieniu.
- `cena`: Cena produktu w momencie zakupu (zapisana, aby uniknąć problemów przy zmianie ceny produktu w przyszłości).

8.5 Kod Źródłowy Generatora Bazy Danych

Pełna implementacja, wraz z logiką do generowania danych testowych, znajduje się w poniższym skrypcie. Został on użyty do automatycznego stworzenia schematów dla SQLite i PostgreSQL oraz wypełnienia ich danymi.

Listing 2: generator_bazy_danych.py

```
"""
Generator Bazy Danych "Sklep"

Moduł ten dostarcza kompletne rozwiązanie do automatycznego tworzenia
i wypełniania danymi testowymi bazy danych dla dwóch różnych systemów:
SQLite oraz PostgreSQL.

Został zaprojektowany z myślą o braku zewnętrznych zależności, co gwarantuje
jego działanie w każdym standardowym środowisku Python 3.
"""
import sqlite3
import random
import os
from datetime import datetime, timedelta

# --- Konfiguracja parametrów skryptu ---
DB_NAME_SQLITE = "sklep.db"
OUTPUT_SQL_POSTGRES = "sklep_postgres.sql"

# Liczba rekordów do wygenerowania
NUM_KLIENCI = 50
NUM_PRODUKTY = 100
NUM_ZAMOWIENIA = 150

# --- Statyczne dane jako źródło dla generatora ---
IMIONA_M = ['Jan', 'Piotr', 'Krzysztof', 'Andrzej', 'Tomasz', 'Paweł', 'Marcin
↪', 'Michał']
IMIONA_K = ['Anna', 'Katarzyna', 'Maria', 'Małgorzata', 'Agnieszka', 'Barbara
↪', 'Ewa', 'Elżbieta']
NAZWISKA = ['Nowak', 'Kowalski', 'Wiśniewski', 'Wójcik', 'Kowalczyk',
↪'Kamiński', 'Lewandowski', 'Zieliński']
CZESCI_PRODUKTU_1 = ['Chleb', 'Ser', 'Mleko', 'Jogurt', 'Szynka', 'Sok', 'Woda
↪', 'Masło', 'Jajka', 'Makaron']
CZESCI_PRODUKTU_2 = ['wiejski', 'naturalny', 'świeży', 'tradycyjny',
↪'ekologiczny', 'pełnoziarnisty', 'owocowy', 'gazowana']

KATEGORIE = [
    (1, 'Pieczywo', 'Świeże chleby, bułki i wyroby cukiernicze.'),
    (2, 'Nabiał', 'Mleko, sery, jogurty i inne produkty mleczne.'),
    (3, 'Napoje', 'Soki, wody mineralne i napoje gazowane.'),
    (4, 'Produkty sypkie', 'Mąka, cukier, ryż, makarony.')
]
DOSTAWCY = [
    (1, 'Piekarnia "Złoty Kłos"', '111-222-333', 'kontakt@zlotyklos.pl'),
    (2, 'Mleczarnia "Łąka"', '444-555-666', 'biuro@mleczarnia-laka.com'),
    (3, 'Hurtownia "Napojex"', '777-888-999', 'zamowienia@napojex.pl'),
    (4, 'Rolnik "EkoZbiory"', '123-456-789', 'rolnik@ekozbiory.pl')
]

```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
# --- Definicje schematów SQL ---
SCHEMA_SQLITE = """
CREATE TABLE Kategorie (
    kategoria_id INTEGER PRIMARY KEY,
    nazwa_kategorii TEXT NOT NULL,
    opis TEXT
);
CREATE TABLE Dostawcy (
    dostawca_id INTEGER PRIMARY KEY,
    nazwa_firmy TEXT NOT NULL,
    telefon TEXT,
    email TEXT
);
CREATE TABLE Klienci (
    klient_id INTEGER PRIMARY KEY AUTOINCREMENT,
    imie TEXT NOT NULL,
    nazwisko TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE
);
CREATE TABLE Produkty (
    produkt_id INTEGER PRIMARY KEY AUTOINCREMENT,
    nazwa_produktu TEXT NOT NULL,
    cena REAL NOT NULL,
    stan_magazynowy INTEGER NOT NULL,
    kategoria_id INTEGER,
    dostawca_id INTEGER,
    FOREIGN KEY (kategoria_id) REFERENCES Kategorie(kategoria_id),
    FOREIGN KEY (dostawca_id) REFERENCES Dostawcy(dostawca_id)
);
CREATE TABLE Zamowienia (
    zamowienie_id INTEGER PRIMARY KEY AUTOINCREMENT,
    klient_id INTEGER NOT NULL,
    data_zamowienia DATETIME NOT NULL,
    status TEXT NOT NULL,
    FOREIGN KEY (klient_id) REFERENCES Klienci(klient_id)
);
CREATE TABLE PozycjeZamowienia (
    pozycja_id INTEGER PRIMARY KEY AUTOINCREMENT,
    zamowienie_id INTEGER NOT NULL,
    produkt_id INTEGER NOT NULL,
    ilosc INTEGER NOT NULL,
    cena REAL NOT NULL,
    FOREIGN KEY (zamowienie_id) REFERENCES Zamowienia(zamowienie_id),
    FOREIGN KEY (produkt_id) REFERENCES Produkty(produkt_id)
);
"""

SCHEMA_POSTGRES = """
DROP TABLE IF EXISTS PozycjeZamowienia, Zamowienia, Produkty, Klienci,␣
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    ↪Dostawcy, Kategorie CASCADE;
94  CREATE TABLE Kategorie (
95      kategoria_id SERIAL PRIMARY KEY,
96      nazwa_kategorii VARCHAR(100) NOT NULL,
97      opis TEXT
98  );
99  CREATE TABLE Dostawcy (
100     dostawca_id SERIAL PRIMARY KEY,
101     nazwa_firmy VARCHAR(255) NOT NULL,
102     telefon VARCHAR(20),
103     email VARCHAR(255)
104 );
105 CREATE TABLE Klienci (
106     klient_id SERIAL PRIMARY KEY,
107     imie VARCHAR(100) NOT NULL,
108     nazwisko VARCHAR(100) NOT NULL,
109     email VARCHAR(255) NOT NULL UNIQUE
110 );
111 CREATE TABLE Produkty (
112     produkt_id SERIAL PRIMARY KEY,
113     nazwa_produktu VARCHAR(255) NOT NULL,
114     cena NUMERIC(10, 2) NOT NULL,
115     stan_magazynowy INT NOT NULL,
116     kategoria_id INT REFERENCES Kategorie(kategoria_id),
117     dostawca_id INT REFERENCES Dostawcy(dostawca_id)
118 );
119 CREATE TABLE Zamowienia (
120     zamowienie_id SERIAL PRIMARY KEY,
121     klient_id INT NOT NULL REFERENCES Klienci(klient_id),
122     data_zamowienia TIMESTAMP NOT NULL,
123     status VARCHAR(50) NOT NULL
124 );
125 CREATE TABLE PozycjeZamowienia (
126     pozycja_id SERIAL PRIMARY KEY,
127     zamowienie_id INT REFERENCES Zamowienia(zamowienie_id),
128     produkt_id INT REFERENCES Produkty(produkt_id),
129     ilosc INT NOT NULL,
130     cena DECIMAL(10, 2) NOT NULL
131 );
132 """
133
134 def generuj_dane():
135     """
136     Przygotowuje w pamięci kompletny zestaw danych testowych.
137
138     Funkcja tworzy spójny zestaw list z danymi dla klientów, produktów,
139     zamówień oraz pozycji zamówień. Respektuje przy tym zdefiniowane
140     relacje i losowo generuje powiązania.
141
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    :returns: Słownik zawierający listy krotek dla każdej tabeli.
              Klucze: 'klienci', 'produkty', 'zamowienia', 'pozycje_zamowien'.
    :rtype: dict
    """
    klienci = []
    for i in range(NUM_KLIENCI):
        imie = random.choice(IMIONA_M + IMIONA_K)
        nazwisko = random.choice(NAZWISKA)
        email = f"{imie.lower()}.{nazwisko.lower()}{i}@example.com"
        klienci.append((i + 1, imie, nazwisko, email))

    produkty = []
    for i in range(NUM_PRODUKTY):
        nazwa = f"{random.choice(CZESCI_PRODUKTU_1)} {random.choice(CZESCI_
↪PRODUKTU_2)}"
        cena = round(random.uniform(2.5, 50.0), 2)
        stan = random.randint(0, 200)
        kat_id = random.choice(KATEGORIE)[0]
        dos_id = random.choice(DOSTAWCY)[0]
        produkty.append((i + 1, nazwa, cena, stan, kat_id, dos_id))

    zamowienia = []
    statusy = ['Złożone', 'Wysłane', 'Dostarczone', 'Anulowane']
    for i in range(NUM_ZAMOWIENIA):
        klient_id = random.randint(1, NUM_KLIENCI)
        data = datetime.now() - timedelta(days=random.randint(0, 365))
        status = random.choice(statusy)
        zamowienia.append((i + 1, klient_id, data, status))

    pozycje_zamowien = []
    pozycja_id_counter = 1
    for zamowienie in zamowienia:
        zamowienie_id = zamowienie[0]
        for _ in range(random.randint(1, 5)):
            produkt = random.choice(produkty)
            produkt_id = produkt[0]
            cena_w_chwili_zakupu = produkt[2]
            ilosc = random.randint(1, 10)
            pozycje_zamowien.append((pozycja_id_counter, zamowienie_id,␣
↪produkt_id, ilosc, cena_w_chwili_zakupu))
            pozycja_id_counter += 1

    return {
        'klienci': klienci,
        'produkty': produkty,
        'zamowienia': zamowienia,
        'pozycje_zamowien': pozycje_zamowien
    }

```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
189 def stworz_baze_sqlite(dane):
190     """
191     Tworzy i wypełnia bazę danych SQLite na podstawie dostarczonych danych.
192 
193     Funkcja usuwa istniejący plik bazy, tworzy nową strukturę
194     zgodnie ze schematem i wstawia dane za pomocą operacji masowych.
195 
196     :param dict dane: Słownik z danymi, wynik działania funkcji␣
    ↪:func:`generuj_dane`.
197     """
198     if os.path.exists(DB_NAME_SQLITE):
199         os.remove(DB_NAME_SQLITE)
200 
201     conn = sqlite3.connect(DB_NAME_SQLITE)
202     cursor = conn.cursor()
203 
204     cursor.executescript(SCHEMA_SQLITE)
205     cursor.executemany("INSERT INTO Kategorie VALUES (?, ?, ?)", KATEGORIE)
206     cursor.executemany("INSERT INTO Dostawcy VALUES (?, ?, ?, ?)", DOSTAWCY)
207     cursor.executemany("INSERT INTO Klienci (klient_id, imie, nazwisko,␣
    ↪email) VALUES (?, ?, ?, ?)", dane['klienci'])
208     cursor.executemany("INSERT INTO Produkty (produkt_id, nazwa_produktu,␣
    ↪cena, stan_magazynowy, kategoria_id, dostawca_id) VALUES (?, ?, ?, ?, ?, ?)
    ↪", dane['produkty'])
209     cursor.executemany("INSERT INTO Zamowienia (zamowienie_id, klient_id,␣
    ↪data_zamowienia, status) VALUES (?, ?, ?, ?)", dane['zamowienia'])
210     cursor.executemany("INSERT INTO PozycjeZamowienia (pozycja_id, zamowienie_
    ↪id, produkt_id, ilosc, cena) VALUES (?, ?, ?, ?, ?)", dane['pozycje_zamowien
    ↪'])
211 
212     conn.commit()
213     conn.close()
214     print(f"Baza SQLite '{DB_NAME_SQLITE}' została utworzona pomyślnie.")
215 
216 def stworz_skrypt_postgres(dane):
217     """
218     Generuje kompletny plik .sql dla bazy PostgreSQL.
219 
220     Funkcja tworzy plik .sql, który może być wykonany na serwerze
221     PostgreSQL w celu odtworzenia identycznej struktury i zawartości
222     bazy danych. Automatycznie formatuje wartości i aktualizuje sekwencje.
223 
224     :param dict dane: Słownik z danymi, wynik działania funkcji␣
    ↪:func:`generuj_dane`.
225     """
226     with open(OUTPUT_SQL_POSTGRES, 'w', encoding='utf-8') as f:
227         f.write("-- Wygenerowany skrypt SQL dla PostgreSQL\n")
228         f.write(SCHEMA_POSTGRES)
229         f.write("\n\n-- Wstawianie danych\n")
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
230
231         def format_sql(val):
232             if isinstance(val, str):
233                 return f"'{val.replace("'", "''")}'"
234             if isinstance(val, datetime):
235                 return f"'{val.strftime('%Y-%m-%d %H:%M:%S')}'"
236             return str(val)
237
238         for kat in KATEGORIE:
239             f.write(f"INSERT INTO Kategorie VALUES ({', '.join(map(format_sql,
↪ kat))});\n")
240         for dos in DOSTAWCY:
241             f.write(f"INSERT INTO Dostawcy VALUES ({', '.join(map(format_sql,␣
↪dos))});\n")
242         for k in dane['klienci']:
243             f.write(f"INSERT INTO Klienci (klient_id, imie, nazwisko, email)␣
↪VALUES ({', '.join(map(format_sql, k))});\n")
244         for p in dane['produkty']:
245              f.write(f"INSERT INTO Produkty (produkt_id, nazwa_produktu, cena,
↪ stan_magazynowy, kategoria_id, dostawca_id) VALUES ({', '.join(map(format_
↪sql, p))});\n")
246         for z in dane['zamowienia']:
247             f.write(f"INSERT INTO Zamowienia (zamowienie_id, klient_id, data_
↪zamowienia, status) VALUES ({', '.join(map(format_sql, z))});\n")
248         for pz in dane['pozycje_zamowien']:
249             f.write(f"INSERT INTO PozycjeZamowienia (pozycja_id, zamowienie_
↪id, produkt_id, ilosc, cena) VALUES ({', '.join(map(format_sql, pz))});\n")
250
251         f.write("\n-- Aktualizacja sekwencji kluczy głównych\n")
252         f.write("SELECT setval('kategorie_kategoria_id_seq', (SELECT␣
↪MAX(kategoria_id) FROM Kategorie));\n")
253         f.write("SELECT setval('dostawcy_dostawca_id_seq', (SELECT␣
↪MAX(dostawca_id) FROM Dostawcy));\n")
254         f.write("SELECT setval('klienci_klient_id_seq', (SELECT MAX(klient_
↪id) FROM Klienci));\n")
255         f.write("SELECT setval('produkty_produkt_id_seq', (SELECT MAX(produkt_
↪id) FROM Produkty));\n")
256         f.write("SELECT setval('zamowienia_zamowienie_id_seq', (SELECT␣
↪MAX(zamowienie_id) FROM Zamowienia));\n")
257         f.write("SELECT setval('pozycjezamowienia_pozycja_id_seq', (SELECT␣
↪MAX(pozycja_id) FROM PozycjeZamowienia));\n")
258
259     print(f"Skrypt dla PostgreSQL '{OUTPUT_SQL_POSTGRES}' został wygenerowany␣
↪pomyślnie.")
260
261 if __name__ == '__main__':
262     """
263     Główny blok wykonawczy skryptu.
264
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
265     Jego zadaniem jest orkiestracja całego procesu: wygenerowanie danych,
266     stworzenie bazy SQLite oraz wygenerowanie skryptu dla PostgreSQL.
267     """
268     print("Rozpoczynam generowanie baz danych.")
269     dane_wygenerowane = generuj_dane()
270     stworz_baze_sqlite(dane_wygenerowane)
271     stworz_skrypt_postgres(dane_wygenerowane)
272     print("\nZakończono. Utworzono plik bazy SQLite i plik .sql dla␣
↪PostgreSQL.")
```

Rozdzia 9

Analiza i Implementacja Bazy Danych

Ten rozdział poświęcony jest szczegółowej analizie struktury bazy danych „Sklep”, procesowi jej normalizacji oraz prezentacji kluczowych skryptów i zapytań SQL, które umożliwiają interakcję z danymi.

9.1 Analiza Struktury i Normalizacja

Projekt bazy danych został oparty o model relacyjny, co gwarantuje spójność i integralność danych. Proces projektowania uwzględniał zasady normalizacji, aby wyeliminować redundancję i anomalie danych.

9.1.1 Pierwsza Postać Normalna (1NF)

Każda tabela w bazie posiada klucz główny, a wszystkie atrybuty w tabelach przechowują wartości atomowe (niepodzielne). Przykładowo, w tabeli *Klienci* adres email jest pojedynczą informacją, a w *Produktach* cena jest jedną liczbą. Nie ma pól, które zawierałyby listy czy zbiory danych.

9.1.2 Druga Postać Normalna (2NF)

Wszystkie atrybuty w tabelach, które mają klucze złożone (w naszym przypadku tylko tabela asocjacyjna *PozycjeZamowienia*), są w pełni zależne od całego klucza głównego. W pozostałych tabelach klucze główne są proste (jednopolowe), więc warunek 2NF jest automatycznie spełniony. W *PozycjeZamowienia* atrybuty *ilosc* i *cena* zależą zarówno od *zamowienie_id*, jak i *produkt_id*.

9.1.3 Trzecia Postać Normalna (3NF)

W bazie nie występują zależności przechodnie. Żaden atrybut niekluczowy nie jest zależny od innego atrybutu niekluczowego. Na przykład, w tabeli *Produkty* nie przechowujemy nazwy kategorii czy danych dostawcy – zamiast tego używamy kluczy obcych (*kategoria_id*, *dostawca_id*), które wskazują na odpowiednie rekordy w tabelach *Kategorie* i *Dostawcy*. Dzięki temu zmiana nazwy kategorii wymaga modyfikacji tylko jednego rekordu w tabeli *Kategorie*.

Podsumowując, schemat bazy danych jest zgodny z **trzecią postacią normalną (3NF)**, co jest standardem dla dobrze zaprojektowanych relacyjnych baz danych.

9.2 Skrypty SQL i Generowanie Danych

Do stworzenia i wypełnienia bazy danych przygotowano skrypty SQL dla dwóch popularnych systemów: SQLite oraz PostgreSQL.

9.2.1 Implementacja w SQLite

Poniższy fragment kodu SQL definiuje kompletną strukturę tabel dla bazy SQLite. Używa typów danych specyficznych dla tego systemu, takich jak *INTEGER PRIMARY KEY* dla automatycznie inkrementowanych kluczy.

Listing 1: Schemat bazy danych dla SQLite

```
CREATE TABLE Kategorie (
    kategoria_id INTEGER PRIMARY KEY,
    nazwa_kategorii TEXT NOT NULL,
    opis TEXT
);
CREATE TABLE Produkty (
    produkt_id INTEGER PRIMARY KEY AUTOINCREMENT,
    nazwa_produktu TEXT NOT NULL,
    cena REAL NOT NULL,
    stan_magazynowy INTEGER NOT NULL,
    kategoria_id INTEGER,
    dostawca_id INTEGER,
    FOREIGN KEY (kategoria_id) REFERENCES Kategorie(kategoria_id),
    FOREIGN KEY (dostawca_id) REFERENCES Dostawcy(dostawca_id)
);
-- ... pozostałe tabele ...
```

9.2.2 Implementacja w PostgreSQL

Dla systemu PostgreSQL schemat wykorzystuje bardziej rygorystyczne typy danych (*VARCHAR*, *NUMERIC*, *TIMESTAMP*) oraz sekwencje (*SERIAL*) do automatycznego generowania kluczy głównych.

Listing 2: Schemat bazy danych dla PostgreSQL

```
CREATE TABLE Kategorie (
    kategoria_id SERIAL PRIMARY KEY,
    nazwa_kategorii VARCHAR(100) NOT NULL,
    opis TEXT
);
CREATE TABLE Produkty (
    produkt_id SERIAL PRIMARY KEY,
    nazwa_produktu VARCHAR(255) NOT NULL,
    cena NUMERIC(10, 2) NOT NULL,
    stan_magazynowy INT NOT NULL,
    kategoria_id INT REFERENCES Kategorie(kategoria_id),
    dostawca_id INT REFERENCES Dostawcy(dostawca_id)
);
-- ... pozostałe tabele ...
```

9.3 Przykładowe Zapytania i Optymalizacja

Poniżej znajdują się przykłady zapytań SQL, które można wykonać na bazie „Sklep”, wraz z omówieniem potencjalnych optymalizacji.

9.3.1 Zapytanie 1: Suma wartości zamówień dla każdego klienta

To zapytanie oblicza łączną kwotę wydaną przez każdego klienta.

```
SELECT
    k.imie,
    k.nazwisko,
    k.email,
    SUM(pz.ilosc * pz.cena) AS laczna_wartosc_zamowien
FROM Klienci k
JOIN Zamowienia z ON k.klient_id = z.klient_id
JOIN PozycjeZamowienia pz ON z.zamowienie_id = pz.zamowienie_id
GROUP BY k.klient_id, k.imie, k.nazwisko, k.email
ORDER BY laczna_wartosc_zamowien DESC;
```

Optymalizacja: Zapytanie wykorzystuje złączenia (JOIN) tabel. Aby przyspieszyć jego wykonanie, kluczowe jest posiadanie **indeksów** na kolumnach używanych do złączeń, czyli *Klienci(klient_id)*, *Zamowienia(klient_id)*, *Zamowienia(zamowienie_id)* oraz *PozycjeZamowienia(zamowienie_id)*. W naszym schemacie kolumny te są kluczami głównymi lub obcymi, na których systemy bazodanowe zazwyczaj automatycznie tworzą indeksy.

9.3.2 Zapytanie 2: Znalezienie 5 najpopularniejszych produktów

To zapytanie zlicza, ile razy każdy produkt został zamówiony.

```
SELECT
    p.nazwa_produktu,
    COUNT(pz.produkt_id) AS liczba_zamowien
FROM Produkty p
JOIN PozycjeZamowienia pz ON p.produkt_id = pz.produkt_id
GROUP BY p.produkt_id, p.nazwa_produktu
ORDER BY liczba_zamowien DESC
LIMIT 5;
```

Optymalizacja: Podobnie jak w poprzednim przypadku, wydajność zależy od indeksów na kolumnach *Produkty(produkt_id)* i *PozycjeZamowienia(produkt_id)*. Przy bardzo dużej liczbie pozycji w zamówieniach, wydajność można by dalej poprawić poprzez denormalizację, np. dodając licznik zamówień bezpośrednio w tabeli *Produkty*, jednak odbyłoby się to kosztem utrzymania spójności danych. Dla obecnej struktury indeksy są wystarczające.

Rozdzia 10

Struktura Repozytoriów Projektowych

10.1 Projekt został zrealizowany w oparciu o rozproszony system kontroli wersji Git, a platforma GitHub posłużyła jako centralne miejsce do przechowywania kodu, koordynacji prac oraz integracji poszczególnych części sprawozdania. Poniżej przedstawiono kompletną listę repozytoriów wykorzystanych w projekcie.

10.2 Repozytoria z Opracowaniami Tematycznymi

Każdy z tematów teoretycznych został opracowany w osobnym repozytorium, co pozwoliło na równoległą pracę członków zespołu.

1. **Temat:** Sprzęt dla bazy danych * * **Link:** oszczeda/Sprzet-dla-bazy-danych
2. **Temat:** Konfiguracja bazy danych * * **Link:** Chaiolites/Konfiguracja_baz_danych
3. **Temat:** Kontrola i konserwacja * * **Link:** Pi0trM/Kontrola_i_konserwacja
4. **Temat:** Monitorowanie i diagnostyka * * **Link:** GrzegorzSzczepanek/repo-wspolne
5. **Temat:** Wydajność, skalowanie i replikacja * * **Link:** Broksonn/Wydajnosc_Skalowanie_i_Replikacja
6. **Temat:** Partycjonowanie danych
   - **Link:** BartekHen/Partycjonowanie-danych
7. **Temat:** Bezpieczeństwo
   - **Link:** BlazejUl/bezpieczenstwo
8. **Temat:** Kopie zapasowe i odzyskiwanie danych
   - **Link:** m-smieja/Kopie_zapasowe_i_odzyskiwanie_danych