POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Baz Danych

Inżynieria oprogramowania, procesów biznesowych i baz danych

Filip Socha

Nr albumu 31311

Wpływ partycjonowania na wydajność systemów bazodanowych

Praca magisterska napisana pod kierunkiem:

dr inż. Paweł Lenkiewicz

Warszawa, wrzesień, 2025

Streszczenie

Celem pracy magisterskiej była analiza wpływu różnych technik partycjonowania danych na wydajność systemów bazodanowych. Badania zostały przeprowadzone w trzech znanych środowiskach: Oracle Database, Microsoft SQL Server oraz PostgreSQL. Użyta metoda obejmowała stworzenie jednorodnych środowisk testowych, wdrożenie technik partycjonowania w każdej z baz danych oraz załadowanie tego samego zestawu danych do testów. Umożliwiło to na dokładne porównanie otrzymanych wyników. W części teoretycznej pracy ukazano podstawy działania systemów bazodanowych. Omówiono również zagadnienia związane z partycjonowaniem danych oraz wskazano jego zalety i wady. W części praktycznej dokonano implementacji różnych metod partycjonowania w wybranych systemach oraz przygotowano zestaw zapytań testowych. Obejmowały one operacje odczytu, wstawiania i aktualizacji danych. Badania przeprowadzono z wykorzystaniem narzędzia Apache JMeter. Pozwoliło ono na wykonanie testów wydajnościowych i obciążeniowych. W testach wydajnościowych analizowano różne metryki przy pojedynczych operacjach. Natomiast testy obciążeniowe pozwoliły na ocenę zachowania baz danych przy równoczesnym dostępie wielu użytkowników. Zdefiniowano metryki, takie jak średni czas odpowiedzi, odchylenie standardowe oraz przepustowość. Umożliwiły one porównanie i analizę wyników pomiędzy różnymi silnikami baz danych. Praca zawiera rzetelne omówienie wyników przeprowadzonych eksperymentów oraz podsumowanie, w którym zestawiono obserwacje i wnioski dotyczące wpływu partycjonowania na wydajność badanych systemów baz danych.

Kluczowe słowa:

bazy danych, partycjonowanie, wydajność

Spis treści

1. Wstęp

W rozdziale tym przedstawiono ogólne założenia pracy, jej cel, zakres oraz zastosowaną metodologię badań. Rozwinięcie zaprezentowanych tu zagadnień następuje w kolejnych częściach pracy.

1.1 Cel pracy

Celem pracy jest szczegółowa analiza wpływu różnych technik partycjonowania danych na wydajność wybranych systemów bazodanowych badając czas odpowiedzi zapytań, skalowalność i elastyczność systemów w warunkach zróżnicowanego obciążenia. Cel zostanie zrealizowany poprzez przygotowanie i implementację takich samych środowisk testowych dla każdego z analizowanych systemów bazodanowych, użycie wybranych metod partycjonowania, stworzenie zestawu zapytań testowych, a następnie przeprowadzenie serii pomiarów wydajnościowych i obciążeniowych. Wyniki będą porównane i poddane analizie statystycznej, co umożliwi wyciągnięcie wniosków na temat wpływu zastosowanego partycjonowania na wydajność oraz obciążenie systemów.

1.2 Zakres pracy

Zakres pracy obejmuje wprowadzenie teoretyczne dotyczące systemów bazodanowych, podstaw partycjonowania danych jak również ich wad i zalet. W pracy została przedstawiona praktyczna implementacja partycjonowania w systemach Oracle Database, Microsoft SQL Server i PostgresSQL oraz eksperymenty badawcze dotyczące wydajności tych implementacji. Część praktyczna skupia się na wdrożeniu różnych technik partycjonowania dostępnych w poszczególnych systemach baz danych. Natomiast część badawcza zawiera przeprowadzone testy wydajnościowe oraz obciążeniowe przy użyciu narzędzia Apache Jmeter. W kolejnym etapie pracy uzyskane wyniki testów zostały poddane metodzie porównawczej pod kątem wydajności. Ostatni etap pracy skupia się na podsumowaniu całej pracy magisterskiej i możliwych optymalizacji partycjonowania dla poszczególnych systemów baz danych.

1.3 Metodologia badań

Przed wykonaniem badań zostały przygotowane spójne środowiska bazodanowe dla trzech wybranych systemów: Oracle Database, Microsoft SQL Server i PostgresSQL. W każdym z nich zaimplementowano wybrane techniki partycjonowania i załadowano jednolite zestawy danych testowych, które zapewniły porównywalność wyników. Badania podzielono na dwa główne etapy: wydajnościowe i obciążeniowe. Zostały przeprowadzone przy użyciu narzędzia Apache Jmeter. Umożliwia ono symulowanie wielu różnych scenariuszy testowych.

Zostały również zdefiniowane metryki, które były używane do oceny wyników, takie jak czas odpowiedzi zapytań, odchylenie standardowe i przepustowość. Badania wydajnościowe wykonano za pomocą utworzonego skryptu aplikacji Apache Jmeter, generującego zapytania do bazy danych, które odwzorowują typowe operacje takie jak odczyt danych, aktualizacja oraz dodawanie rekordów. Badania obciążeniowe symulują rzeczywiste użytkowanie bazy danych poprzez równoczesny dostęp wielu użytkowników i wykonano je również poprzez uruchomienie skryptów Jmeter. W ramach tych testów odwzorowano warunki pracy systemu o zwiększonej liczbie zapytań, aby ocenić parametry przy zwiększonym obciążeniu, bez wchodzenia w zakres testów przeciążeniowych. Tak przygotowana metodyka badawcza umożliwia porównanie zachowania systemów bazodanowych w warunkach pracy pojedynczego użytkownika oraz przy równoczesnym wykonaniu zapytań, co pozwala na rzetelną ocenę wpływu zaimplementowanego partycjonowania wydajność i stabilność działania.

2. Podstawy teoretyczne

W tym rozdziale omówiono zagadnienia stanowiące teoretyczne fundamenty dla realizacji pracy. Podano ogólne informacje dotyczące systemów zarządzania bazami danych oraz wyjaśniono istotne elementy i procesy związane z partycjonowania danych. Szczególną uwagę poświęcono klasyfikacji metod partycjonowania oraz ich użyciu w kontekście zarządzania dużymi zbiorami danych. W ostatnim podrozdziale przeprowadzono analizę najważniejszych zalet oraz ograniczeń związanych z wdrażaniem partycjonowania w relacyjnych systemach bazodanowych.

2.1 Systemy baz danych

System zarządzania bazami danych (ang. Database Management System, DBMS) jest to zaawansowane oprogramowanie informatyczne, którego celem jest zbieranie, przechowywanie, organizowanie, przetwarzanie oraz udostępnianie danych w sposób bezpieczny, spójny i efektywny [1]. Główna rola DBMS polega na zarządzaniu ogromnymi zbiorami informacji, które mogą być wykorzystywane równocześnie przez wielu użytkowników oraz równoległe procesy systemowe. W ramach tej funkcjonalności systemy bazodanowe oferują mechanizmy automatyzacji i reakcji na różne zdarzenia. Na przykład w relacyjnych DBMS często stosuje się zaplanowane zadania (ang. jobs), które realizują operacje administracyjne lub przetwarzanie danych [2], oraz wyzwalacze (ang. triggers), uruchamiane w odpowiedzi na operacje wstawiania, aktualizacji czy usuwania rekordów [3]. W bazach nierelacyjnych mechanizmy te występują zazwyczaj w prostszej formie jako strumienie zdarzeń [4] czy funkcje reaktywne [5].

Silnik bazy danych odgrywa kluczową rolę w architekturze systemu, jest odpowiedzialny za fizyczne przechowywanie danych, optymalizację zapytań, kontrolę dostępu, obsługę transakcji oraz zarządzanie strukturami indeksów. Silnik stosuje różnorodne mechanizmy, które zapewniają trwałość danych, nawet w sytuacjach awarii lub nagłego zakończenia działania systemu. Struktury indeksowe, które są skonstruowane i optymalizowane przez silnik, znacznie przyspieszają proces wyszukiwania i filtrowania rekordów [6].

Systemy baz danych można podzielić na relacyjne (ang. Relational Database Management System, RDBMS) oraz nierelacyjne (ang. Non – Relational Database Management System, NRDBMS). W modelu relacyjnym dane są przechowywane w tabelach, które składają się

z kolumn określonych typów oraz wierszy z konkretnymi rekordami, a między tabelami mogą występować zdefiniowane relacje. Do najpopularniejszych systemów tego rodzaju należą MySQL, PostgreSQL, Oracle Database oraz Microsoft SQL Server. W przeciwieństwie do nich bazy nierelacyjne NoSQL (ang Not Only SQL) przechowują dane w mniej formalnych strukturach, takich jak dokumenty (np. baza MongoDB), pary klucz-wartość (np. Redis), grafy (np. Neo4j) czy kolumny (np. Apache Cassandra). Rozwiązania NoSQL są szczególnie przydatne w przypadku obsługi dużych, nieregularnych lub dynamicznie zmieniających się zbiorów danych [7].

Najczęściej interakcja z bazą danych odbywa się za pomocą języka SQL (ang. Structured Query Language), który pozwala na definiowanie struktury bazy, zarządzanie danymi oraz na wykonywanie operacji analitycznych. Dzięki temu można tworzyć, zmieniać, usuwać oraz pobierać dane bez konieczności działania na fizycznej warstwie. SQL stanowi standard dla większości relacyjnych baz danych. Oferuje on uniwersalną metodę komunikacji z serwerem bazodanowym. Oczywiście istnieją nieznaczne różnice języka SQL pomiędzy systemami baz danych, które są dostosowane do nich [8]. W przypadku baz nierelacyjnych nie istnieje jeden wspólny język zapytań. Poszczególne rozwiązania korzystają z odmiennych mechanizmów dostępu do danych, które są dostosowane do przyjętego modelu. W bazie MongoDB wykorzystuje się zapytania w formacie dokumentów JSON/BSON [9], Apache Cassandra posiada własny język CQL (ang. Cassandra Query Language) wzorowany na SQL [10]. Bazy grafowe, takie jak Neo4j, stosują język Cypher, pozwalający na wyszukiwanie zależności między węzłami [11]. Natomiast baza Redis opiera się na prostych komendach typu GET, SET lub DEL [12].

Integralną częścią systemów baz danych są mechanizmy transakcyjne, które w klasycznych relacyjnych DBMS zapewniają zgodność z zasadą ACID (ang. Atomicity, Consistency, Isolation, Durability). Dzięki niej operacje są grupowane w logiczne transakcje, które muszą zostać wykonane w całości lub wcale, co gwarantuje spójność i integralność danych nawet w przypadku błędów, konfliktów czy awarii. Takie podejście znajduje zastosowanie w systemach, w których priorytetem jest niezawodność i poprawność danych, czyli w bankowości czy systemach ERP. W bazach nierelacyjnych, szczególnie w środowiskach rozproszonych, często stosuje się odmienne podejście. Określa je się akronimem BASE (ang. Basically Available, Soft state, Eventually consistent). W tym modelu nacisk kładzie się na wysoką dostępność i możliwość pracy systemu nawet w sytuacji awarii części węzłów, kosztem

natychmiastowej spójności danych. Spójność traktowana jest jako cel do osiągnięcia. Oznacza to, że różne dane mogą przez pewien czas zawierać odmienne wersje obiektów, lecz ostatecznie ich stan zostaje ujednolicony. Taki kompromis wynika z twierdzenia CAP, które wskazuje, że w systemach rozproszonych możliwe jest jednoczesne zagwarantowanie tylko dwóch spośród trzech właściwości: spójności (ang. Consistency), dostępności (ang. Availability) oraz odporności na podziały sieciowe (ang. Partition tolerance) [6, rozdział 23.6.1].

Współczesne bazy danych muszą radzić sobie z wyzwaniami powstałymi z przetwarzania ogromnych zbiorów informacji, określanych jako big data. Kluczowym wyzwaniem jest skalowalność, która dotyczy zdolności do efektywnego rozkładu danych oraz obciążenia na sporą liczbę serwerów. Ponadto musi zapewnić wysokiej jakości wydajność zapytań w obliczu zwiększającej się liczby wpisów. Równie istotne jest zapewnienie spójności w systemach rozproszonych, ponieważ dane mogą być tam kopiowane i modyfikowane w różnych miejscach. Ponadto, niemałym wyzwaniem jest różnorodność formatów i struktur danych oraz potrzeba ich ochrony przed nieautoryzowanym dostępem. Należy również zadbać o jakość danych poprzez eliminację błędów, duplikatów oraz niespójności. Proces łączenia danych pochodzących z różnych źródeł bywa czasochłonny i wymaga dokładnego sprawdzenia. Aby sprostać tym wymaganiom, nowoczesne systemy zarządzania bazami danych korzystają z zaawansowanych technologii, takich jak bazy kolumnowe wspierające analitykę OLAP (ang. Online Analytical Processing), rozproszone systemy, rozwiązania chmurowe oraz analiza strumieniowa w czasie rzeczywistym. Bazy danych stały się kluczowym elementem działania współczesnych aplikacji i usług, obejmując takie obszary jak finanse, logistyka, e-commerce oraz zaawansowane analizy. Wraz z postępującą cyfryzacją gospodarek ich znaczenie w zapewnianiu dostępu do rzetelnych i aktualnych informacji rośnie.

2.2 Partycjonowanie danych

Partycjonowanie danych to technika dzielenia dużych zbiorów danych na mniejsze, łatwiejsze do zarządzania fragmenty, zwane partycjami. Każda z tych partycji działa jako odrębna jednostka danych, jednak razem tworzą one spójną całość w obrębie jednej tabeli lub indeksu. Głównym celem partycjonowania jest optymalizacja wydajności pracy z bazą danych oraz uproszczenie zarządzania danymi, szczególnie w przypadku bardzo obszernych zbiorów. Operacje na takich podzielonych fragmentach mogą przykładowo umożliwiać szybsze przeprowadzanie zapytań, ponieważ dotyczą jedynie wybranych partycji. Omija się przy tym

konieczność przeszukiwania całej tabeli. Partycjonowanie można używać zarówno w przypadku tabel, jak i indeksów. Zwiększa to możliwości zarządzania danymi. Umożliwia to również prostsze przeprowadzanie operacji takich jak archiwizacja, tworzenie kopii zapasowych, przywracanie lub przenoszenie starszych danych na tańsze nośniki, nie wpływając przy tym na dostępność bardziej aktywnych informacji. W realnym zastosowaniu partycjonowanie jest szeroko wykorzystywane w systemach, które muszą radzić sobie z ogromnymi zbiorami danych, takimi jak rozwiązania finansowe, e-commerce czy systemy analityczne [13].

Istnieje wiele różnych rodzajów technik partycjonowania. Poniżej zostały opisane najczęściej stosowane rozwiązania.

Partycjonowanie zakresowe polega na klasyfikacji rekordów do określonych partycji w zależności od ustalonych przedziałów wartości w kolumnie używanej do partycjonowania. Zwykle jest to kolumna z datami lub liczbami. To podejście podziału danych jest powszechnie stosowane w zarządzaniu danymi archaicznymi, ponieważ umożliwia łatwiejsze przeprowadzanie operacji na całych grupach, takich jak archiwizacja czy usuwanie.

Partycjonowanie listowe bazuje na dzieleniu danych w oparciu o konkretne wartości lub zbiory wartości. Każda partycja jest przypisana do określonych kategorii, takich jak regiony geograficzne lub rodzaje produktów. Metoda ta jest przydatna, gdy dane naturalnie dzielą się na niewielką liczbę stałych grup.

Partycjonowanie haszowe korzysta z funkcji haszującej, która rozdziela dane pomiędzy partycje na podstawie wartości klucza podziału. Celem jest zapewnienie równomiernego rozmieszczenia rekordów. Pomaga to zminimalizować obciążenie pojedynczych partycji. Takiego rodzaju partycjonowanie jest stosowane, kiedy dane nie są klarownie podzielone na zakresy lub kategorie.

Partycjonowanie kompozytowe łączy różne metody podziału danych. Ma to na celu wykorzystać zalety każdej z metod. Na przykład, można zastosować partycjonowanie listowe i dla każdej z tych utworzonych partycji wprowadzić dodatkowy podział zakresowy. Takie podejście pozwala lepiej dostosować strukturę przechowywania danych do wymagań systemu.

2.3 Zalety i wady partycjonowania

Stosowanie partycjonowania jest związane z wieloma korzyściami, które są szczególnie widoczne w systemach obsługujących duże ilości danych. Jednym z podstawowych atutów jest znaczna poprawa wydajności zapytań. Ogranicza ono przetwarzane dane jedynie do tych partycji, które odnoszą się do konkretnego zapytania, a system jest w stanie realizować operacje znacznie szybciej, co ma ogromny wpływ podczas pracy z dużymi tabelami, na przykład w przypadku zapytań dotyczących określonych zakresów czasowych lub kategorii danych. Korzyść ta wynika m.in. z eliminacji partycji (ang. partition pruning). Przez to redukowana jest ilość danych czytanych z dysku oraz pozwala optymalizatorowi zapytań pominąć nieistotne partycje [14]. Dodatkowo, wiele systemów bazodanowych umożliwia tzw. partition-wise joins, które polegają na dzieleniu złączeń dużych tabel na mniejsze operacje, które są wykonywane w ramach odpowiadających sobie partycji. Rozwiązanie to pozwala znacząco skrócić czas przetwarzania złożonych, skomplikowanych zapytań analitycznych, ponieważ zamiast przetwarzać całe zestawy danych, system działa tylko na ich mniejszych podzbiorach. W przypadku baz wspierających wielowątkowość lub przetwarzanie równoległe, każda partycja może być obsługiwana niezależnie przez inny proces. Zwiększa to przepustowość systemu i skraca czas odpowiedzi.

Ponadto, partycjonowanie ułatwia zarządzanie danymi, ponieważ pozwala administratorom realizować zadania takie jak tworzenie kopii zapasowych, odzyskiwanie danych, archiwizowanie czy usuwanie przestarzałych zbiorów jedynie na wyznaczonych partycjach, a nie na całej tabeli. Dzięki temu system nie jest zbytnio obciążony, a czas realizacji tych działań ulega skróceniu. Kolejną ważną zaletą partycjonowania jest poprawa możliwości skalowania, co umożliwia elastyczne rozbudowywanie bazy danych przez dodawanie nowych partycji, co jest bardzo istotne w systemach, gdzie ilość danych nieustannie wzrasta i wymaga dostosowania do tych zmian. W środowiskach hurtowni danych oraz systemach rozproszonych partycjonowanie stanowi podstawę tzw. skalowania poziomego, w którym obciążenie można rozdzielać pomiędzy wiele węzłów lub procesów. Daje to możliwość efektywnego przetwarzania rosnących wolumenów danych bez konieczności kosztownej przebudowy całej architektury systemu. Ostatnim równie ważnym atutem jest zwiększona dostępność danych. Rozdzielenie informacji między wiele partycji i serwerów ogranicza ryzyko wystąpienia pojedynczego punktu awarii. Awaria jednej partycji nie wpływa na dostępność całego systemu, który może nadal działać na innych fragmentach. W środowiskach chmurowych z wbudowaną

nadmiarowością znaczenie tego efektu jest mniejsze, ale w systemach lokalnych może to być kluczowe dla ciągłości działania. [15].

Mimo wielu zalet, partycjonowanie danych nie jest pozbawione wad, które należy wziąć pod uwagę już na etapie projektowania systemu. Po pierwsze, wprowadzenie partycjonowania zwiększa złożoność systemu bazodanowego, co wpływa na trudności w jego utrzymaniu oraz na potrzebę bardziej zaawansowanego zarządzania. Administratorzy bazy muszą bowiem dbać o spójność i integralność wielu partycji, co może prowadzić do zwiększenia ryzyka pojawienia się błędów lub problemów technicznych. Dodatkowy nakład związany z utworzeniem partycji wiąże się z większym zapotrzebowaniem na przestrzeń dyskową w bazie danych. Wynika to z faktu, że każda utworzona partycja jest osobnym segmentem danych w warstwie fizycznej. Oznacza to, że choć partycjonowanie przynosi korzyści w zakresie wydajności, należy uwzględnić jego wpływ na zużycie przestrzeni dyskowej i uważnie zaplanować zasoby. Warto zwrócić uwagę na ryzyko związane z wydajnością. Jeśli zapytania nie wykorzystują klucza partycjonowania do filtrowania lub posługują się wyrażeniami, które nie są odpowiednie do indeksowania optymalizator nie ma możliwości pominięcia nieistotnych partycji i może być zmuszony zeskanować wiele segmentów co niweluje korzyści płynące z partycjonowania. Zbyt duża liczba małych partycji podnosi koszty związane z planowaniem oraz operacjami na metadanych. Jest to szczególnie odczuwalne w systemach, które realizują wiele krótkich zapytań [16].

3. Partycjonowanie w wybranych systemach baz danych

W niniejszym rozdziale omówiono rozwiązania dotyczące partycjonowania danych w trzech wybranych systemach bazodanowych: Microsoft SQL Server, Oracle Database oraz PostgreSQL. Każdy z podrozdziałów przedstawia architekturę oraz dostępne rodzaje partycjonowania w danym środowisku.

3.1 Partycjonowanie w Microsoft SQL Server

W tym rozdziale omówiono architekturę partycjonowania jak i opis dostępnych rodzajów partycjonowania w systemie Microsoft SQL Server 2022.

3.1.1 Architektura partycjonowania

W tej części szczegółowo opisano kluczowe elementy architektury partycjonowani w Microsoft SQL Server. Polega ona na podziale danych w ramach jednej tabeli lub indeksu zarówno w aspekcie logicznym, jak i fizycznym. Głównym celem tego podejścia jest efektywniejsze zarządzanie dużymi zbiorami danych poprzez podział ich na mniejsze jednostki nazywane partycjami danych [17].

Jednym z podstawowych elementów tej architektury jest funkcja partycjonująca (ang. Partition Function), która określa sposób przypisywania danych do poszczególnych partycji na podstawie wartości klucza partycjonowania. Zazwyczaj opiera się ona na wartościach liczbowych, tekstowych bądź datowych i wyznacza zakresy, na podstawie których dane są rozmieszczane w odpowiednich partycjach. Może wykorzystywać podejście RANGE LEFT lub RANGE RIGHT, w zależności od tego czy graniczna wartość ma znajdować się w lewej czy prawej partycji.
Drugim istotnym elementem jest schemat partycjonowania (ang. Partition Scheme), odpowiedzialny za mapowanie partycji, utworzonych na podstawie funkcji partycjonującej, do logicznych grup plików. Każda partycja może być przypisana do oddzielnej grupy plików. Umożliwia to rozłożenie danych na różnych nośnikach oraz optymalizację wydajności.
Z kolei grupy plików (ang. Filegroups) stanowią logiczne jednostki przechowywania danych, które są powiązane z plikami fizycznymi na dysku. Każda partycja może być przypisana do innej grupy plików, co pozwala na selektywne wykonywanie kopii zapasowych lub przenoszenie danych pomiędzy dyskami. Umożliwiają one dystrybucję danych w różnych

lokalizacjach, co przyczynia się do zwiększenia wydajności operacji na dużych zbiorach danych.

Ważnym składnikiem architektury jest również tabela partycjonowana, czyli tabela przypisana do określonego schematu partycjonowania. Klucz partycjonowania, określony w definicji tabeli lub indeksu, musi odpowiadać typowi danych funkcji partycjonującej. Fizyczne podzielenie danych występuje w chwili tworzenia indeksu pogrupowanego zgodnego z kolumną partycjonującą i schematem.

Często stosuje się także kolumnę obliczeniową (ang. computed column), czyli taką kolumnę, której dane nie są wprowadzane ręcznie, lecz generowane automatycznie na podstawie wartości innych kolumn. W tematyce partycjonowania stosuje się ją często jako obejście ograniczeń MSSQL Servera, który nie posiada wsparcia dla kompozytowego partycjonowania. W tej sytuacji kolumna obliczeniowa łączy w sobie dwie wartości (np. rok zamówienia i region geograficzny) w jedną wartość, która jest następnie używana jako klucz do partycjonowania. Aby mogła być użyta w indeksach lub w schemacie partycjonowania, musi być oznaczona jako PERSISTED, co oznacza, że jej wartość jest na stałe zapisywana w tabeli [18].

Kolejnym istotnym elementem jest zestaw indeksów niepogrupowanych, które służą do przyspieszania operacji odczytu, które nie są bezpośrednio zależne od kolumny partycjonowania. Dobrze zaprojektowany zestaw indeksów niepogrupowanych pozwala na efektywne filtrowanie i sortowanie danych, nawet jeśli zapytanie nie trafia bezpośrednio w konkretną partycję.

Istotnym procesem w kontekście architektury jest import danych. Jest to proces, w którym do tabeli podzielonej na partycje dodawane są rekordy pochodzące z już istniejącego zbioru danych – zazwyczaj z tabeli, która nie jest podzielona na partycje. Ta czynność może być realizowana przez zapytanie: INSERT INTO … SELECT …, często z zastosowaniem różnych modyfikacji (na przykład nadawania unikalnych dat lub uzupełniania brakujących informacji). Dzięki wykorzystaniu kolumn obliczeniowych, wartości klucza partycjonowania są obliczane automatycznie w trakcie importu, co sprawia, że rekordy są przypisywane do odpowiednich partycji zgodnie z przyjętą funkcją i schematem partycjonowania. Odpowiednie przygotowanie danych w tym etapie jest niezwykle istotne dla późniejszej efektywności testów oraz wydajności wykonywanych operacji.

Dzięki tego rodzaju architekturze SQL Server wspiera poziome skalowanie w obrębie pojedynczej tabeli a także znacznie podnosi wydajność operacji takich jak odczyt, tworzenie kopii zapasowych, archiwizacja czy usuwanie danych w dużych zbiorach.

3.1.2 Rodzaje partycjonowania w Microsoft SQL Server 2022

W systemie Microsoft SQL Server 2022 są możliwe jedynie dwa podstawowe rodzaje partycjonowania: zakresowe (ang. Range Partitioning) oraz listowe (ang. List Partitioning). Pierwsze z nich jest najczęściej wykorzystywane i pozwala na przypisanie danych do partycji według ustalonych przedziałów według klucza partycjonowania. W praktyce najczęściej stosuje się je w podziale danych według dat, co ułatwia zarządzanie archiwizacją oraz planowanie przyszłych partycji.
Drugim wariantem jest partycjonowanie listowe, w którym dane przydzielane są do partycji na podstawie listy wartości. Choć jest to metoda stosunkowo mniej popularna, partycjonowanie listowe może być bardzo skuteczne w sytuacjach, gdy dane mają znaczną różnorodność z uwagi na konkretne cechy logiczne.

W Microsoft SQL Server nie można bezpośrednio zastosować partycjonowania kompozytowego (np. połączenia zakresowego i listowego) na jednej tabeli. SQL Server wspiera tylko jednowymiarowe partycjonowanie Oznacza to, że można wybrać tylko jeden rodzaj klucza partycjonowani**a** dla danej tabeli np. zakresowy lub listowy. Mimo że istnieją metody symulacji partycjonowania wielowymiarowego przy użyciu partycji podrzędnych, widoków partycjonowanych lub połączenia partycjonowania z partycjami plików, nie są one rozwiązaniami wbudowanymi. Wybór odpowiedniego rodzaju partycjonowania powinien być oparty na charakterystyce danych oraz rodzaju przeprowadzanych operacji. Badanie rozkładu danych, typowych zapytań i strategii archiwizacji jest kluczowe przy projektowaniu modelu partycjonowania, który zapewnia maksymalną wydajność i skalowalność.

3.2 Partycjonowanie w Oracle Database

Rozdział obejmuje omówienie architektury partycjonowania dostępnej w systemie Oracle Database 23c oraz przegląd typów partycjonowania.

3.2.1 Architektura partycjonowania

Oracle Database 23c udostępnia wbudowany mechanizm partycjonowania, który działa na poziomie fizycznej organizacji danych w tabeli. W tym systemie każda partycja może również zawierać subpartycje. Umożliwia to na jeszcze bardziej precyzyjne zarządzanie dużymi zbiorami danych i wiąże się z wykorzystaniem partycjonowania wielowymiarowego. Dane są

dzielone według dwóch lub też więcej kryteriów – np. kraju (ShipCountry) i daty (OrderDate). Partycjonowanie w tym systemie jest przejrzyste dla aplikacji co znaczy, że programy i zapytania SQL nie muszą być modyfikowane, aby korzystać z danych znajdujących się w wielu partycjach. Mechanizm ten jest w pełni zintegrowany z optymalizatorem zapytań, który ma umiejętność automatycznego stosowania przycinania partycji (ang. partition pruning) oraz równoległe przetwarzanie danych z wielu partycji [19].

Podstawowe elementy architektury partycjonowania w Oracle Database 23c [20] obejmują kilka kluczowych mechanizmów. Głównym elementem jest klucz partycjonowania (ang. Partition Key), czyli kolumna lub zestaw kolumn, na podstawie których Oracle decyduje, do której partycji zostanie zapisany dany wiersz. Klucz może być prosty składający się z jednej kolumny lub złożony składający się z kilku kolumn. W przypadku zastosowania partycjonowania kompozytowego jedna kolumna odpowiada za pierwszy poziom podziału partycji, a druga za poziom subpartycji.
Istotnym atrybutem architektury jest również definicja partycjonowania, która jest określana bezpośrednio w instrukcji CREATE TABLE lub CREATE INDEX. W tym miejscu wskazywana jest metoda podziału – rodzaj partycjonowania oraz granice partycji lub wartości przypisane do partycji. W przypadku zastosowania subpartycjonowania (drugi poziom partycjonowania) można użyć SUBPARTITION TEMPLATE, czyli szablon partycjonowania, który automatycznie utworzy identyczne zestawy subpartycji dla każdej partycji nadrzędnej.
Dane przechowywane są w partycjach i subpartycjach czyli fizycznych segmentach tworzonych zgodnie z definicją podziału. Oracle umożliwia zarządzanie partycjami i subpartycjami w niezależny sposób. Można je usuwać, dodawać i przenosić czy łączyć co daje dużą elastyczność w administrowaniu. Każda partycja przechowywana jest w odrębnym segmencie danych, który można przypisać do wybranej przestrzeni tabel (ang. Tablespaces). Daje to możliwość rozmieszczenia danych na rożnych dyskach lub na różnych zasobach pamięci masowej.
Ważną rolę odgrywają także indeksy partycjonowane. Oracle umożliwia tworzenie indeksów dopasowanych do struktury partycjonowania tabeli (tzw. indeksy lokalne) lub niezależnych od niej (tzw. indeksy globalne). Indeksy lokalne są dzielone zgodnie z podziałem tabeli, co ułatwia zarządzanie i zwiększa wydajność wyszukiwania danych w obrębie konkretnej partycji.
Dodatkowo system wyposażony jest w mechanizmy optymalizacji, takie jak Partition Pruning i Partition-wise Joins. Pierwszy z nich pozwala optymalizatorowi zapytań ograniczyć zakres przeszukiwania tylko do tych partycji, które spełniają warunki w klauzuli WHERE, co znacznie

redukuje liczbę odczytywanych danych. Drugi umożliwia wykonywanie złączeń w obrębie odpowiadających sobie partycji, co zmniejsza ilość przetwarzanych danych i przyspiesza operacje [21].
Na uwagę zasługuje również obsługa DML i DDL na poziomie partycji. W Oracle można wykonywać operacje wstawiania, aktualizacji, usuwania czy nawet zmiany struktury tabeli w obrębie wybranej partycji. Zwiększa to kontrolę nad operacjami na dużych zbiorach danych i minimalizuje wpływ na pozostałe partycje.

3.2.2 Rodzaje partycjonowania w Oracle Database 23c

W tym rozdziale opisano dokładnie wszystkie rodzaje partycjonowania dostępne na serwerze Oracle Database 23c. Oferuje on szeroki zestaw mechanizmów partycjonowania, gdzie każdy typ charakteryzuje się inną logiką przypisywania rekordów do partycji.

W systemie Oracle Database 23c dostępnych jest kilka metod partycjonowania [20, rozdział 2.3], z których część została już omówiona w rozdziale 2.2. Poniżej przedstawiono je w kontekście specyficznych możliwości oferowanych przez ten system.

Oracle umożliwia stosowanie partycjonowania zakresowego, które w praktyce stosuje się najczęściej w przypadku kolumn typu datowego i identyfikatorów liczbowych. Metoda ta jest szczególnie użyteczna, gdzie dane napływają sekwencyjnie - przykładowo zapisy dziennie lub tygodniowe. W tych przypadkach takie partycjonowanie pozwala na łatwe archiwizowanie lub usuwanie historycznych danych poprzez wykonanie operacji na całych partycjach.
Drugim obsługiwanym podejściem jest partycjonowanie hashowe (ang. Hash Partitioning). W tym sposobie partycjonowania wiersze są przypisywane do partycji na podstawie wyniku funkcji skrótu (ang. hash) obliczonej z wartości kolumny lub zestawu kolumn. Celem jest równomierne rozłożenie danych między partycje. Szczególnie korzystne jest to w systemach, w których dostęp do danych ma charakter losowy, a zapytania nie koncentrują się na konkretnych przedziałach czy kategoriach.
Partycjonowanie listowe pozwala na przypisanie danych do partycji według jednoznacznie określonych wartości lub grup wartości. W Oracle rozwiązanie to stosowane jest m.in. w danych kategorycznych, które naturalnie dzielą się na logiczne zbiory, takie jak kraje, regiony czy typy produktów.

Znacząca rolę w Oracle odgrywa również partycjonowanie kompozytowe (ang. Composite Partitioning). Łączy ono w sobie dwie różne metody partycjonowania, co pozwala jeszcze dokładniej odwzorować strukturę danych i zoptymalizować przetwarzanie danych. Jest to partycjonowanie wielowymiarowe, w którym dane dzielone są równocześnie według więcej niż jednego atrybutu – na przykład geograficznego (kraj) i czasowego (rok). Taki model pozwala osiągnąć wysoką selektywność i zminimalizować liczbę przeszukiwanych partycji, danych. Ta metoda znacznie poprawia efektywność odczytu i modyfikacji danych.

Oprócz standardowych metod system obsługuje także partycjonowanie systemowe (ang. System Partitioning). Nazywane jest również ręcznym i polega na tym, że administrator samodzielnie decyduje o tym, do jakiej partycji trafią poszczególne rekordy. Odbywa się to poprzez wskazanie docelowej partycji już na etapie ładowania danych za pomocą odpowiedniej składni polecenia INSERT z podaniem nazwy partycji. Nie ma w tej sytuacji żadnego mechanizmu do automatycznego przypisania danych. Jest to raczej rzadko wykorzystywana metoda, ale znajduje swoje zastosowanie w niestandardowych sytuacjach, gdzie należy dostosować partycjonowanie do nietypowych wymagań biznesowych.

Oracle oferuje również mechanizmy dynamicznego partycjonowania. Należy do nich partycjonowanie interwałowe (ang. Interval Partitioning). Jest to odmiana partycjonowania zakresowego, w której nowe partycje tworzone są automatycznie w miarę pojawienia się nowych danych wykraczających poza wcześniejsze zakresy partycjonowania. W odróżnieniu od partycjonowania zakresowego, definiowane są tylko wielkości interwału np. miesiąc, kwartał, rok. Baza danych sama tworzy kolejne partycje.

Drugim przykładem dynamicznego partycjonowania jest partycjonowanie referencyjne (ang. Reference Partitioning). Jest to rodzaj partycjonowania, które umożliwia automatyczne dziedziczenie struktury partycji przez tabelę podrzędną na podstawie klucza obcego do tabeli nadrzędnej. Oznacza to, że tabela podrzędna przyjmuje identyczny schemat partycjonowania, co tabela nadrzędna, bez konieczności osobnej konfiguracji. Eliminuje to konieczność synchronizacji definicji partycji w wielu tabelach.

3.3 Partycjonowanie w PostgreSQL

Rozdział obejmuje omówienie architektury partycjonowania w PostgresSQL, jak i opis dostępnych rodzajów partycjonowania. System PostgreSQL należy do najpopularniejszych baz danych typu open source, oferując użytkownikom rozbudowane możliwości partycjonowania tabel.

3.3.1 Architektura partycjonowania

PostgreSQL stosuje partycjonowanie poprzez wykorzystanie tabeli nadrzędnej (ang. parent table), która jest logicznym kontenerem dla danych, a fizyczne rekordy umieszczane są w tabelach potomnych (ang. partitions). Każda z tych partycji odpowiada konkretnemu podzbiorowi danych, który został wydzielony przy pomocy klucza partycjonowania. Tabela nadrzędna ma na celu jedynie przechowywanie definicji struktury, natomiast dane trafiają bezpośrednio do właściwej partycji w zależności od wartości klucza. Z punktu widzenia użytkownika i programisty mechanizm ten jest w pełni obsługiwany przez system bazodanowy. Zapytania do tabeli nadrzędnej są automatycznie przekierowywane do właściwych partycji. Nie ma zatem powodu do jawnego odwoływania się do tabel podrzędnych, ale w razie potrzeby mogą być one wykorzystywane bezpośrednio w celach administracyjnych.

Architektura partycjonowania w PostgreSQL oferuje możliwość wykorzystania zaawansowanych metod optymalizacji, takich jak ograniczanie zakresu wyszukiwania (partition pruning) do wybranych partycji, a także wsparcie dla połączeń między partycjami (partition-wise joins). System ten umożliwia automatyczne kierowanie operacjami INSERT, UPDATE oraz DELETE do odpowiednich partycji i zapewnia obsługę dedykowanych indeksów oraz ograniczeń na poziomie partycji. Dodatkowo istnieje możliwość łączenia różnych rodzajów partycjonowania w złożone struktury. Pozwala to dopasować struktury danych do potrzeb konkretnej aplikacji [22].

Podstawowe elementy architektury partycjonowania w PostgreSQL opierają się na koncepcji tabeli nadrzędnej (ang. Parent Table). Pełni ona funkcję logicznego kontenera, określając strukturę kolumn, klucz partycjonowania oraz zasady podziału danych. Tabela ta nie zawiera danych (z wyjątkiem przypadków, gdy ustawiona jest domyślna partycja), lecz kieruje operacjami do odpowiednich partycji podrzędnych. Jest to model, na podstawie którego zorganizowane są fizycznie partycje.
Partycje w PostgreSQL są to fizyczne tabele podrzędne, które przechowują dane według ustalonych kryteriów podziału. Każda z tych partycji funkcjonuje jako oddzielna tabela, co pozwala na niezależne ustalanie indeksów, ograniczeń oraz zasad przechowywania danych dla każdej z nich. Spójność całej struktury zapewnia klucz partycjonowania. Jest to jedna lub kilka kolumn, których wartości decydują, do której partycji rekord zostanie przypisany. Klucz ten

jest określony w tabeli głównej i musi być zgodny z definicją dla wszystkich partycji. Gwarantuje to spójna logikę podziału danych.

Ważnym rozszerzeniem mechanizmu jest partycja domyślna (ang. Default Partition). Jest to opcjonalna partycja, w której umieszczane są wszystkie rekordy, które nie pasują do żadnych z wcześniej określonych zakresów lub wartości listowych. Dzięki temu można elastycznie obsługiwać dane, które nie były wcześniej przewidziane, co zapobiega występowaniu błędów podczas operacji dodawania danych.

Istotnym elementem architektury jest również Partition Pruning, czyli technika optymalizacji zapytań, która znacząco poprawia wydajność pracy z tabelami partycjonowanymi. Mechanizm ten polega na eliminacji z planu zapytania tych partycji, które na pewno nie maja danych w warunkach zapytania. W PostgreSQL partition pruning może zachodzić w dwóch różnych etapach. Podczas planowania zapytania, gdy warunki filtracji są wcześniej znane, system ustala, które partycje mogą zawierać pasujące dane i wyklucza pozostałe z planu zapytania. To sprawia, że nie są one przeszukiwane w trakcie wykonania, co obniża ilość przetwarzanych danych i przyspiesza czas odpowiedzi na zapytanie. Partition pruning może zachodzić także podczas realizacji zapytania, gdy warunki filtrowania opierają się na wartościach znanych dopiero w trakcie działania zapytania. System jest w stanie na bieżąco wykluczać zbędne partycje podczas samego wykonywania zapytania.

3.3.2 Rodzaje partycjonowania w PostgreSQL 17.2

W PostgreSQL Version 17.2 dostępne są cztery podstawowe rodzaje partycjonowania: zakresowe, listowe, hashowe oraz kompozytowe [23]. Partycjonowanie zakresowe pozwala na dzielenie danych według przedziałów wartości wybranej kolumny, najczęściej czasu, daty lub liczby i znajduje zastosowanie przy zarządzaniu danymi historycznymi czy logami. Partycjonowanie listowe umożliwia przypisywanie rekordów do określonych partycji na podstawie zdefiniowanych wartości klucza, co pozwala grupować dane według np. regionów, typów zamówień czy kategorii produktów.

Partycjonowanie hashowe rozdziela dane między partycje przy pomocy funkcji skrótu, co zapewnia równomierne obciążenie i sprawdza się w systemach o losowym dostępie do danych. Z kolei partycjonowanie kompozytowe pozwala na zastosowanie kilku poziomów partycjonowania w obrębie jednej tabeli, co oznacza łączenie różnych metod klasyfikacji danych, takich jak zakresowe, listowe oraz haszowe. To daje możliwość tworzenia hierarchii partycji, które dokładnie odpowiadają strukturze danych oraz wymaganiom aplikacji przez

administratorów. Partycjonowanie kompozytowe znacznie ułatwia zarządzanie danymi wielowymiarowymi i poprawia wydajność zapytań, w szczególności dla operacji, gdzie kluczowe są różne kryteria przeszukiwania danych [24].

Każdy typ partycjonowania realizowany jest poprzez odpowiednią składnię instrukcji CREATE TABLE, a PostgreSQL zapewnia narzędzia do zarządzania, monitorowania i modyfikacji układu partycji.

4. Badania

W tym rozdziale najpierw opisano narzędzie Docker umożliwiające uruchomienie systemów bazodanowych oraz narzędzie do testów Apache Jmeter. Następnie dokładnie omówiono etapy konfiguracji środowisk testowych jak i implementacji partycjonowań dla poszczególnych systemów. Później zawarte zostały opisy kroków badawczych i rodzaje przeprowadzonych badań. Ostatni etap przedstawia uzyskane wyniki dla wskazanych baz danych.

4.1 Narzędzia i technologie: Docker i Apache JMeter

Do uruchomienia środowisk testowych dla badanych systemów bazodanowych wykorzystano platformę Docker. Jest to rozwiązanie programistyczne oparte na technologii kontenerów, zaprojektowane z myślą o tworzeniu, dostarczaniu i uruchamianiu aplikacji w odizolowanym środowisku. Docker umożliwia przygotowanie dostosowanych kontenerów linuksowych z gotowych obrazów Dockera lub zgodnych ze standardem OCI (ang. Open Container Initiative). Obrazy są zbiorem oprogramowania uruchamiającego jako kontener, zawierający instrukcje tworzenia kontenera, który może działać na platformie Docker. Kontenery zawierają kompletne środowisko uruchomieniowe, w którego skład wchodzą aplikacje, biblioteki o raz konfiguracja systemowa. Pozwala to na szybkie wdrożenie systemów bazodanowych niezależnie od platformy sprzętowej i systemu operacyjnego [25]. Na potrzeby przeprowadzanych badań przygotowano osobne kontenery dla każdego z testowanych silników bazodanowych. Ograniczenia zasobów zostały skonfigurowane na poziomie Docker Engine, a więc były wspólne dla wszystkich uruchomionych kontenerów. Ustalono je na poziomie 5 rdzeni procesora, pamięć RAM na poziomie 8 GB oraz 1GB pamięci wymiany (swap). Takie ustawienia pozwoliły na zapewnić porównywalność warunków pracy środowisk testowych.

Do realizacji testów wydajnościowych i obciążeniowych zastosowano narzędzie Apache JMeter. Jest to popularne narzędzie otwartoźródłowe wykorzystywane do badania wydajności aplikacji, w tym systemów bazodanowych. Daje ono możliwość między innymi do definiowania scenariuszy testowych, równoczesnego wykonywania zapytań SQL, symulowania wielu równoległych użytkowników oraz pomiar kluczowych parametrów, takich jak czas odpowiedzi zapytania, odchylenie standardowe i przepustowość systemu [26]. W ramach badań przygotowano skrypty testowe w formacie jmx, które odwzorowywały typowe operacje baz danych, w tym odczyt danych, ich aktualizacja oraz wstawianie nowych rekordów. Apache Jmeter pozwolił rzetelnie odwzorować warunki pracy systemów

w środowisku rzeczywistym, co ma znaczenie dla uzyskania prawidłowych wyników pomiarowych.

4.2 Konfiguracja i implementacja

Ten podrozdział zawiera szczegółowe opisy sposobów przygotowania środowisk testowych, implementacji partycjonowań dla każdego testowego systemu bazodanowego.

4.2.1 Microsoft SQL Server – przygotowanie środowiska i implementacja partycjonowania

W celu przeprowadzenia testów partycjonowania w systemie Microsoft SQL Server, przygotowano środowisko testowe w oparciu o kontener Dockera. Utworzono wolumen danych „mssql-data". Pobrano oficjalny obraz Microsoft SQL Server 2022 z repozytorium Docker Hub i uruchomiono kontener z odpowiednimi parametrami startowymi. Konfiguracja obejmowała akceptacje licencji, ustawienie hasła dla użytkownika SA, mapowanie portu 1433 oraz wskazanie katalogu przeznaczonego na przechowywanie danych serwera. Środowisko testowe działało w oparciu o edycję Microsoft SQL Server 2022 Developer Edition (64-bit) w wersji 16.0.4095.4 (RTM-CU10, KB5031778). Do zarządzania bazą danych wykorzystano narzędzie Azure Data Studio, które umożliwia obsługę instancji SQL Server w trybie graficznym. Na potrzeby implementacji i testów użyto wbudowaną bazę Northwind, która zawiera przykładowe tabele i dane [27]. Tabela Orders zawarta w tej bazie została wykorzystana przy implementacji partycjonowania i testów wydajnościowych. Poniżej została przedstawiona struktura tabeli, założone indeksy i klucze obce.

	Field name	Data type	Nullable	Default value	Description
PK	OrderID	int			
FK	CustomerID	nchar (5)	Yes		
FK	EmployeeID	int	Yes		Same entry as in Employees table.
IX	OrderDate	datetime	Yes		
	RequiredDate	datetime	Yes		
IX	ShippedDate	datetime	Yes		
FK	ShipVia	int	Yes		Same as Shipper ID in Shippers table.
	Freight	money	Yes	(0)	
	ShipName	nvarchar (40)	Yes		Name of person or company to receive the shipment.
	ShipAddress	nvarchar (60)	Yes		Street address only -- no post-office box allowed.
	ShipCity	nvarchar (15)	Yes		
	ShipRegion	nvarchar (15)	Yes		State or province.
IX	ShipPostalCode	nvarchar (10)	Yes		
	ShipCountry	nvarchar (15)	Yes		

Rysunek 4.1 Struktura tabeli Orders

Źródło: [27]

Index name	Column name	Sort direction	Is unique	Index type
CustomerID	CustomerID	ASC		NONCLUSTERED
CustomersOrders	CustomerID	ASC		NONCLUSTERED
EmployeeID	EmployeeID	ASC		NONCLUSTERED
EmployeesOrders	EmployeeID	ASC		NONCLUSTERED
OrderDate	OrderDate	ASC		NONCLUSTERED
PK_Orders	OrderID	ASC	Yes	CLUSTERED
ShippedDate	ShippedDate	ASC		NONCLUSTERED
ShippersOrders	ShipVia	ASC		NONCLUSTERED
ShipPostalCode	ShipPostalCode	ASC		NONCLUSTERED

Rysunek 4.2 Indeksy tabeli Orders

Źródło: [27]

Tabela 4.1 Klucze obce w tabeli Orders

Nazwa klucza obcego	**Kolumna lokalna**	**Tabela powiązana**	**Klucz główny powiązanej tabeli**
FK_Orders_Customers	CustomerID	dbo.Customers	PK_Customers
FK_Orders_Employees	EmployeeID	dbo.Employee	PK_Employees
FK_Orders_Shippers	ShipVia	dbo.Shippers	PK_Shippers

Źródło: opracowanie własne.

Poniżej zostały przedstawione przykładowe dane zawarte w tabeli Orders.

Tabela 4.2 Przykładowe dane z tabeli Orders

OrderID	**CustomerID**	**EmployeeID**	**OrderDate**	**RequiredDate**	**ShippedDate**	**ShipVia**	**Freight**
10253	HANAR	3	1996-07-10 00:00:00.000	1996-07-24 00:00:00.000	1996-07-16 00:00:00.000	2	58,17
10256	WELLI	4	1996-07-15 00:00:00.000	1996-08-12 00:00:00.000	1996-07-17 00:00:00.000	3	13,97

Źródło: opracowanie własne.

Poniżej przedstawiona została kontynuacja tabeli 4.2 dla pozostałych kolumn.

Tabela 4.3 Kontynuacja przykładowych danych z tabeli Orders

ShipName	**ShipAddress**	**ShipCity**	**ShipRegion**	**ShipPostalCode**	**ShipCountry**
Hanari Carnes	Rua do Mercado, 12	Rio de Janeiro	RJ	05454-876	Brazil
Wellington	Rua da Panificadora, 12	San Cristóbal	SP	08737-363	Venezuela

Źródło: opracowanie własne.

Następnie dokonano aktualizacji danych zamieniającej w tabeli Orders pola ShipCountry równe „UK” na „Australia”. Miało to na celu umożliwić stworzenie partycji dla kontynentu Australia i dokonać poprawnych badań na partycjach dla tego kraju dla serwera MSSQL.

```
UPDATE Orders SET ShipCountry = 'Australia'
WHERE ShipCountry = 'UK';
```

Rysunek 4.3 Aktualizacja pól ShipCountry z UK na Australia

Źródło: opracowanie własne.

W celu zwiększenia rozmiaru danych powielono rekordy w tabeli Orders przy wykorzystaniu skryptu „Powielenie_danych_Orders.sql". Celem było osiągnięcie co najmniej 12 milionów rekordów w tabeli Orders. Skrypt działa na zasadzie iteracyjnego kopiowania istniejących danych w tabeli do momentu, aż liczba rekordów przekroczy wartość określoną w zmiennej @targetCount = 12 000 000. Warto podkreślić, że ostateczna liczba rekordów osiągnęła 13 598 720. To rezultat logicznej konsekwencji algorytmu kopiowania, w którym ostatnia iteracja zdublowała dotychczasową liczbę wierszy wynoszącą 6 799 360, co sprawiło, że przekroczono ustalony próg. Powielenie danych umożliwiło przeprowadzenie bardziej wiarygodnych testów partycjonowania.

Poniżej zawarty jest dokładny opis działania skryptu „Powielenie_danych_Orders.sql". Najpierw zostały zadeklarowane zmienne widoczne na rysunku 4.4. Obiektowi @targetCount przypisano wartość 12 000 000. Jest to zmienna, która określa próg liczby rekordów. Obiekt @currentCount określa aktualną liczbę rekordów w tabeli Orders.

```
DECLARE @targetCount INT = 12000000;
DECLARE @currentCount INT;
```

Rysunek 4.4 Deklaracja zmiennych

Źródło: opracowanie własne.

Następnie w skrypcie została zliczona aktualna liczba rekordów w tabeli Orders i przypisano ją do zmiennej @currentCount.

```
SELECT @currentCount = COUNT(*) FROM Orders;
```

Rysunek 4.5 Przypisanie wartości do currentCount

Źródło: opracowanie własne.

Jeśli bieżąca liczba rekordów jest mniejsza niż 12 miliony wykonywana jest pętla WHILE zobrazowana na rysunku 4.6. Kopiuje ona istniejące dane w tabeli Orders do niej samej. W każdej iteracji wstawiana jest taka sama liczba rekordów, jaka już znajduje się w tabeli.

Zmienna OrderDate jest losową datą większą bądź równą dacie ‘01-01-1996’. Najpierw za pomocą funkcji NEWID() generowany jest unikalny identyfikator typu GUID. Funkcja CHECKSUM() przekształca wygenerowany GUID na wartość liczbową. Następnie wykonywana jest operacja modulo a z wyniku brana jest wartość bezwzględna. Ostatni etap to dodanie za pomocą funkcji DATEADD() wyliczonej liczby dni (0–9130) do daty początkowej '01-01-1996’. Po każdym załadowaniu rekordów aktualizowana jest wartość zmiennej @currentCount.

```
WHILE @currentCount < @targetCount
BEGIN
    INSERT INTO Orders (CustomerID, EmployeeID, OrderDate, ShipVia, Freight,
    ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry)
    SELECT
        CustomerID,
        EmployeeID,
        DATEADD(DAY, ABS(CHECKSUM(NEWID()) % 9131), '1996-01-01') AS OrderDate,
        ShipVia,
        Freight,
        ShipName,
        ShipAddress,
        ShipCity,
        ShipRegion,
        ShipPostalCode,
        ShipCountry
    FROM Orders;

    -- Aktualizacja liczby rekordów
    SELECT @currentCount = COUNT(*) FROM Orders;
END;
```

Rysunek 4.6 Pętla while

Źródło: opracowanie własne.

Po zakończeniu procesu powielania danych skrypt przechodzi do wypełniania brakujących wartości pola ShippedDate. Wykorzystana została pętla WHILE widoczna na rysunku 4.7. Dopóki są puste pola dla ShippedDate pętla się nie zakończy. W każdej iteracji aktualizowanych jest maksymalnie 50 000 rekordów. Wartość ShippedDate jest generowana losowo, z przesunięciem od 1 do 10 dni względem OrderDate.

```sql
WHILE EXISTS (SELECT 1 FROM Orders WHERE ShippedDate IS NULL)
BEGIN
    UPDATE TOP (50000) Orders
    SET ShippedDate = DATEADD(DAY, ABS(CHECKSUM(NEWID()) % 10) + 1, OrderDate)
    WHERE ShippedDate IS NULL;
END;
```

Rysunek 4.7 Pętla while exist dla ShippedDate

Źródło: opracowanie własne.

Następnie skrypt przechodzi do kolejnej pętli WHILE pokazanej na rysunku 4.8 i uzupełnia puste pola dla kolumny RequiredDate i dopiero kończy działanie, gdy wszystkie te pola zostaną uzupełnione. Również jak w poprzedniej pętli aktualizowanych jest maksymalnie 50 000 rekordów. Natomiast RequiredDate jest datą losowo wybraną z zakresu od 1 do 40 dni po ShippedDate.

```sql
WHILE EXISTS (SELECT 1 FROM Orders WHERE RequiredDate IS NULL)
BEGIN
    UPDATE TOP (50000) Orders
    SET RequiredDate = DATEADD(DAY, ABS(CHECKSUM(NEWID()) % 40) + 1, ShippedDate)
    WHERE RequiredDate IS NULL;
END;
```

Rysunek 4.8 Pętla while exist dla RequiredDate

Źródło: opracowanie własne.

Po przygotowaniu środowiska testowego, można było zająć się tematem implementacji partycjonowania. SQL Server nie wspiera natywnego kompozytowego partycjonowania łączącego jednocześnie kryteria listowe i zakresowe, dlatego zastosowano obejście pozwalające symulować podobny efekt. Kluczową rolę odgrywa tutaj stworzona kolumna obliczeniowa nazwana PartitionKey, zawierająca połączoną informację o regionie geograficznym oraz roku zamówienia. Dzięki takiej konstrukcji możliwe jest wykorzystanie jednowymiarowego partycjonowania, przy jednoczesnym uwzględnieniu dwóch atrybutów danych (listy i zakresu). W poniższym rozdziale przedstawiono szczegółowy proces implementacji tego podejścia.

Pierwszym krokiem było zdefiniowanie funkcji partycjonującej PF_Orders_CompCol, która operuje na kolumnie typu varchar(6) odpowiadającej wartościom w formacie X_RRRR (gdzie X to symbol grupy regionalnej, a RRRR to czterocyfrowy rok zamówienia). Funkcja ta dzieli

dane na logiczne przedziały, do których będą przypisywane one na podstawie wartości klucza partycjonowania. Funkcja wykorzystuje strategię RANGE LEFT, co oznacza, że każda wartość klucza przypisywana jest do partycji, której górna granica zawiera daną wartość. Zakresy zostały stworzone oddzielnie dla czterech regionów – Australia, Ameryka Północna, Ameryka Południowa i Europa, z podziałem na lata od 1996 do 2024. Pozwala to uzyskać bardzo dokładną segmentację danych.

```
CREATE PARTITION FUNCTION PF_Orders_CompCol (varchar(6))
AS RANGE LEFT FOR VALUES
(
    -- Grupa A (Australia)
    'A_1996', 'A_1997', 'A_1998', 'A_1999', 'A_2000', 'A_2001', 'A_2002', 'A_2003', 'A_2004', 'A_2005',
    'A_2006', 'A_2007', 'A_2008', 'A_2009', 'A_2010', 'A_2011', 'A_2012', 'A_2013', 'A_2014', 'A_2015',
    'A_2016', 'A_2017', 'A_2018', 'A_2019', 'A_2020', 'A_2021', 'A_2022', 'A_2023', 'A_2024',

    -- Grupa N (Ameryka Północna)
    'N_1996', 'N_1997', 'N_1998', 'N_1999', 'N_2000', 'N_2001', 'N_2002', 'N_2003', 'N_2004', 'N_2005',
    'N_2006', 'N_2007', 'N_2008', 'N_2009', 'N_2010', 'N_2011', 'N_2012', 'N_2013', 'N_2014', 'N_2015',
    'N_2016', 'N_2017', 'N_2018', 'N_2019', 'N_2020', 'N_2021', 'N_2022', 'N_2023', 'N_2024',

    -- Grupa S (Ameryka Południowa)
    'S_1996', 'S_1997', 'S_1998', 'S_1999', 'S_2000', 'S_2001', 'S_2002', 'S_2003', 'S_2004', 'S_2005',
    'S_2006', 'S_2007', 'S_2008', 'S_2009', 'S_2010', 'S_2011', 'S_2012', 'S_2013', 'S_2014', 'S_2015',
    'S_2016', 'S_2017', 'S_2018', 'S_2019', 'S_2020', 'S_2021', 'S_2022', 'S_2023', 'S_2024',

    -- Grupa E (Europa)
    'E_1996', 'E_1997', 'E_1998', 'E_1999', 'E_2000', 'E_2001', 'E_2002', 'E_2003', 'E_2004', 'E_2005',
    'E_2006', 'E_2007', 'E_2008', 'E_2009', 'E_2010', 'E_2011', 'E_2012', 'E_2013', 'E_2014', 'E_2015',
    'E_2016', 'E_2017', 'E_2018', 'E_2019', 'E_2020', 'E_2021', 'E_2022', 'E_2023', 'E_2024'

);
```

Rysunek 4.9 Funkcja partycjonująca PF_Orders_CompCol

Źródło: opracowanie własne.

Kolejnym krokiem było utworzenie schematu partycjonowania PS_Orders_CompCol, który mapuje partycje na logiczną grupę plików. W przypadku tej implementacji wszystkie dane przechowywane są w jednej grupie plików o nazwie PRIMARY. Mimo że fizycznie wszystkie dane znajdują się w jednej lokalizacji, logiczne przypisanie do partycji umożliwia serwerowi stosowanie selektywnych operacji oraz optymalizację zapytań, zwłaszcza w kontekście przetwarzania danych historycznych i geograficznych.

W SQL Server liczba partycji, które tworzy schemat partycjonowania, jest zawsze o jedną większa niż liczba wartości podanych w funkcji partycjonującej. Każda wartość w funkcji partycjonującej działa jak granica, która oddziela przedziały. Dlatego właśnie funkcja PF_Orders_CompCol z 87 wartościami tworzy 88 partycji w tym schemacie.

Na rysunku 4.10 jest widoczna poglądowa struktura kodu tworząca schemat dla jednej grupy plików.

```
CREATE PARTITION SCHEME PS_Orders_CompCol
AS PARTITION PF_Orders_CompCol
TO
(
    [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY],  [PRIMARY],
    [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY],  [PRIMARY],
    [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY],  [PRIMARY],
    [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY],  [PRIMARY],
    [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY],  [PRIMARY],
    [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY],  [PRIMARY],
    [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY],  [PRIMARY],
    [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY],  [PRIMARY],
    [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY], [PRIMARY]
);
```

Rysunek 4.10 Poglądowy schemat PF_Orders_CompCol

Źródło: opracowanie własne.

Końcowa, uproszczona struktura kodu tworząca schemat PS_Orders_CompCol została przedstawiona poniżej na rysunku 4.11.

```
CREATE PARTITION SCHEME PS_Orders_CompCol
AS PARTITION PF_Orders_CompCol
ALL TO ([PRIMARY]);
```

Rysunek 4.11 Końcowy schemat PS_Orders_CompCol

Źródło: opracowanie własne.

Następnie została zaprojektowana tabela OrdersPartitioned, która stanowi zmodyfikowaną wersję klasycznej tabeli Orders znanej z bazy Northwind. Wprowadzono w niej kilka zasadniczych zmian, dostosowując strukturę danych do potrzeb projektowych. Tabela OrdersPartitioned została utworzona z zachowaniem wszystkich istotnych kolumn z oryginalnej tabeli Orders i pokazano jej strukturę na rysunku 4.12. Obejmuje ona dane dotyczące ID zamówień, klientów, pracowników, dat realizacji, informacji o dostawie oraz kraju wysyłki.

```
OrderID INT IDENTITY (1, 1) NOT NULL,
CustomerID NCHAR(5),
EmployeeID INT,
OrderDate DATETIME NOT NULL,
RequiredDate DATETIME,
ShippedDate DATETIME,
ShipVia INT,
Freight MONEY,
ShipName NVARCHAR(40),
ShipAddress NVARCHAR(60),
ShipCity NVARCHAR(15),
ShipRegion NVARCHAR(15),
ShipPostalCode NVARCHAR(10),
ShipCountry NVARCHAR(15),
```

Rysunek 4.12 Struktura kolumn w tabeli OrdersPartitioned w MSSQL

Źródło: opracowanie własne.

Najważniejszą zmiana względem oryginalnej tabeli Orders jest dodanie kolumny obliczeniowej PartitionKey, która została zdefiniowana jako trwała kolumna obliczeniowa (PERSISTED). Domyślnie kolumna obliczeniowa nie jest zapisywana fizycznie w tabeli. Jest ona obliczana na bieżąco przy każdym odczycie danych. Nie można jej indeksować, jeśli nie jest deterministyczna. Natomiast użycie PERSISTED umożliwia zapisać dane w tabeli oraz dokonać indeksowania tej kolumny. Ponadto zwiększa wydajność, bo serwer nie musi każdorazowo przeliczać wartości, choć zajmuje miejsce na dysku jak zwykła kolumna.
Jej wartość wyliczana jest dynamicznie na podstawie dwóch pól:

- ShipCountry – kraj dostawy, który decyduje o przypisaniu do jednego z czterech kontynentów:
  - 'A' – Australia
  - 'N' – Ameryka Północna (USA, Kanada, Meksyk)
  - 'S' – Ameryka Południowa (Brazylia, Argentyna, Wenezuela)
  - 'E' – Europa (15 wybranych państw)
- OrderDate – z tej daty wyodrębniany jest czterocyfrowy rok.

Połączone one razem tworzą wartość typu np. A_1999, N_2012, S_2003, E_1997, która następnie służy jako klucz partycjonowania.

```
PartitionKey AS
(
    CASE
        WHEN ShipCountry = 'Australia' THEN 'A'
        WHEN ShipCountry IN ('USA','Canada','Mexico') THEN 'N'
        WHEN ShipCountry IN ('Brazil','Argentina','Venezuela') THEN 'S'
        WHEN ShipCountry IN ('Germany','France','Belgium','Poland','Spain',
                             'Italy','Austria','Denmark','Finland','Sweden',
                             'Switzerland','Ireland','Portugal','Norway') THEN 'E'
        ELSE NULL
    END
    + '_'
    + CONVERT(char(4), YEAR(OrderDate))
)
PERSISTED
```

Rysunek 4.13 Struktury kolumny obliczeniowej PartitionKey

Źródło: opracowanie własne.

Tworząc tabelę OrdersPartitioned założono klucz główny (PRIMARY KEY) jako indeks pogrupowany oparty na dwóch kolumnach: PartitionKey i OrderID.
Taki układ spełnia dwa cele:

- Fizycznie uruchamia partycjonowanie danych w oparciu o klucz PartitionKey,
- Gwarantuje unikalność rekordów dzięki drugiemu sortowaniu po OrderID.

```
CONSTRAINT [PK_OrdersPartitioned] PRIMARY KEY CLUSTERED (PartitionKey, OrderID),
```

Rysunek 4.14 Indeks pogrupowany na tabeli OrdersPartitioned w MSSQL

Źródło: opracowanie własne.

Zachowano wszystkie relacje logiczne znane z oryginalnej tabeli Orders, dodając klucze obce do tabel:

- Customers (CustomerID),
- Employees (EmployeeID),
- Shippers (ShipVia).

Dzięki temu integralność referencyjna danych została w pełni zachowana z tabelą Orders.

```
CONSTRAINT [FK_OrdersPartitioned_Customers] FOREIGN KEY ([CustomerID]) REFERENCES [dbo].[Customers] ([CustomerID]),
CONSTRAINT [FK_OrdersPartitioned_Employees] FOREIGN KEY ([EmployeeID]) REFERENCES [dbo].[Employees] ([EmployeeID]),
CONSTRAINT [FK_OrdersPartitioned_Shippers] FOREIGN KEY ([ShipVia]) REFERENCES [dbo].[Shippers] ([ShipperID]),
```

Rysunek 4.15 Klucze obce na tabeli OrdersPartitioned w MSSQL

Źródło: opracowanie własne.

Tabela została przypisana do wcześniej utworzonego schematu partycjonowania PS_Orders_CompCol, który opiera się na funkcji PF_Orders_CompCol. W wyniku tego każda wartość PartitionKey jest przypisywana do odpowiedniej partycji.

```
CREATE TABLE dbo.OrdersPartitioned
(--STRUKTURA TABELI
) ON PS_Orders_CompCol (PartitionKey);
```

Rysunek 4.16 Obrazowe przedstawienie tworzenia tabeli OrdersPartitioned bez pełnej struktury tabeli

Źródło: opracowanie własne.

Przedostatnim krokiem było utworzenie 6 indeksów niepogrupowanych odwzorowanych zgodnie z tabelą Orders. Celem utworzenia poniższych indeksów było zachowanie porównywalnych warunków wydajnościowych pomiędzy tabelą partycjonowaną a jej odpowiednikiem niepartycjonowanym. Te indeksy pozwalają optymalizatorowi bazy danych na szybsze wyszukiwanie danych według kolumn najczęściej wykorzystywanych w warunkach filtrujących. Takie kroki umożliwiają na rzetelne porównywanie wpływu samego partycjonowania na czas wykonania operacji.

Tabela 4.4 Indeksy niepogrupowane w OrdersPartitioned w MSSQL

Nazwa indeksu	**Kolumna**
OrdersPartitioned_Date	OrderDate
Shippers_OrdersPartitioned	ShipVia
ShipPostalCode_OrdersPartitioned	ShipPostalCode
CustomerID_OrdersPartitioned	CustomerID
ShippedDate_OrdersPartitioned	ShippedDate
EmployeeID_OrdersPartitioned	EmployeeID

Źródło: opracowanie własne.

Na koniec wykonano operację importu wszystkich danych z oryginalnej tabeli Orders do nowej struktury OrdersPartitioned. Przekopiowano łącznie 13 598 720 rekordów. Podczas kopiowania danych kolumna PartitionKey została automatycznie obliczona dla każdego wiersza na podstawie wartości w kolumnach ShipCountry oraz OrderDate, dzięki czemu rekordy zostały od razu przypisane do odpowiednich partycji.

```
INSERT INTO dbo.OrdersPartitioned
(
    CustomerID, EmployeeID, OrderDate, RequiredDate,
    ShippedDate, ShipVia, Freight, ShipName,
    ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry
)
SELECT
    CustomerID, EmployeeID, OrderDate, RequiredDate,
    ShippedDate, ShipVia, Freight, ShipName,
    ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry
FROM dbo.Orders;
```

Rysunek 4.17 Import danych do tabeli OrdersPartitioned w MSSQL

Źródło: opracowanie własne.

4.2.2 Oracle Database – przygotowanie środowiska i implementacja partycjonowania

W celu przeprowadzenia testów partycjonowania w systemie Oracle Database przygotowano środowisko testowe oparte na technologii Docker. Z repozytorium Docker Hub pobrano darmowy obraz bazy Oracle Database 23c Free. Potem w systemie lokalnym utworzono wolumen danych „oracle23-data” do przechowywania danych bazy poza kontenerem. Następnie uruchomiono kontener z odpowiednią konfiguracją obejmującą m.in. ustawienia hasła użytkownika oraz mapowanie portu komunikacyjnego serwera. Do obsługi środowiska Oracle wykorzystano narzędzie graficzne SQL Developer. Schemat bazy danych Northwind wygenerowano poprzez skrypt cr_Nortwind-for-23c.sql pobrany z repozytorium: https://github.com/Fxztam/DemoNorthwind4Oracle/tree/master. Tabela Orders została utworzona z odpowiednimi typami danych zgodnymi z Oracle i zmodyfikowana w celu dopasowania do struktur tabel zawartych na bazie Microsoft SQL Server. Dane do tabeli Orders zostały zaimportowane z pliku Orders.csv, który uprzednio wygenerowano z systemu MSSQL. Plik zawiera wszystkie rekordy zawarte w tabeli Orders w MSSQL. Pozostałe tabele, m.in. Customers, Employees, Products, Shippers, zostały zasilone danymi poprzez wykonanie skryptu ins_Nortwind.sql zawartego w tym samym repozytorium co schemat bazy. Ze skryptu usunięto ładowanie danych do tabeli Orders.

Poniżej na rysunku 4.17 przedstawiono strukturę tabeli OrdersPartitioned. Została utworzona z zachowaniem wszystkich kolumn z oryginalnej tabeli Orders.

```
OrderID NUMBER(9) GENERATED BY DEFAULT AS IDENTITY START WITH 1 INCREMENT BY 1,
CustomerID VARCHAR2(5) NOT NULL,
EmployeeID NUMBER(9) NOT NULL,
OrderDate DATE NOT NULL,
RequiredDate DATE,
ShippedDate DATE,
ShipVia NUMBER(9),
Freight NUMBER(10,2) DEFAULT 0,
ShipName VARCHAR2(40),
ShipAddress VARCHAR2(60),
ShipCity VARCHAR2(15),
ShipRegion VARCHAR2(15),
ShipPostalCode VARCHAR2(10),
ShipCountry VARCHAR2(15),
```

Rysunek 4.18 Struktura tabeli OrdersPartitioned w Oracle

Źródło: opracowanie własne.

Utworzono również takie same referencje kluczy obcych jak na pierwotnej tabeli Orders. Natomiast klucz główny przygotowano inaczej, według rysunku 4.18. W Oracle w tabeli partycjonowanej musi on zawierać ShipCountry i OrderDate czyli kolumny klucza partycjonowania.

```
CONSTRAINT PK_OrdersPartitioned PRIMARY KEY (ShipCountry, OrderDate, OrderID),

CONSTRAINT FK_OrdersPartitioned_Customers FOREIGN KEY (CustomerID)
  REFERENCES Customers (Customer_ID),
CONSTRAINT FK_OrdersPartitioned_Employees FOREIGN KEY (EmployeeID)
  REFERENCES Employees (Employee_ID),
CONSTRAINT FK_OrdersPartitioned_Shippers FOREIGN KEY (ShipVia)
  REFERENCES Shippers (Shipper_ID)
```

Rysunek 4.19 Indeksy tabeli OrdersPartitioned w Oracle

Źródło: opracowanie własne.

W ramach implementacji partycjonowania w systemie Oracle Database 23c przygotowano tabelę OrdersPartitioned, która opiera się na zmienionej wersji tabeli Orders. W tabeli tej zastosowano kompozytowe partycjonowanie – połączenie partycjonowania listowego (LIST) według kraju (ShipCountry) oraz zakresowego interwałowego (INTERVAL RANGE) według daty zamówienia (OrderDate). W kodzie zdefniowano 21 odrębnych partycji listowych widocznych na rysunku 4.19, z nazwami w formacie p_nazwa_kraju (np. p_argentina, p_france), gdzie każda z nich przechowuje wyłącznie rekordy zamówień dla wskazanego państwa.

```
PARTITION p_argentina VALUES ('Argentina'),
PARTITION p_australia VALUES ('Australia'),
PARTITION p_austria VALUES ('Austria'),
PARTITION p_belgium VALUES ('Belgium'),
PARTITION p_brazil VALUES ('Brazil'),
PARTITION p_canada VALUES ('Canada'),
PARTITION p_denmark VALUES ('Denmark'),
PARTITION p_finland VALUES ('Finland'),
PARTITION p_france VALUES ('France'),
PARTITION p_germany VALUES ('Germany'),
PARTITION p_ireland VALUES ('Ireland'),
PARTITION p_italy VALUES ('Italy'),
PARTITION p_mexico VALUES ('Mexico'),
PARTITION p_norway VALUES ('Norway'),
PARTITION p_poland VALUES ('Poland'),
PARTITION p_portugal VALUES ('Portugal'),
PARTITION p_spain VALUES ('Spain'),
PARTITION p_sweden VALUES ('Sweden'),
PARTITION p_switzerland VALUES ('Switzerland'),
PARTITION p_usa VALUES ('USA'),
PARTITION p_venezuela VALUES ('Venezuela')
```

Rysunek 4.20 Partycje listowe w Oracle

Źródło: opracowanie własne.

Każda z partycji listowych została dodatkowo podzielona na podpartycje zakresowe. W tym celu zastosowano szablon podpartycji (SUBPARTITION TEMPLATE), który jest wspólny dla wszystkich partycji. Zakresy zostały określone rocznie – od 1996 do 2024 roku. Dodatkowa podpartycja y_max przechowuje dane wykraczające poza ostatni zdefiniowany przedział czasowy.

```
PARTITION BY LIST (ShipCountry)
SUBPARTITION BY RANGE (OrderDate)
SUBPARTITION TEMPLATE (
  SUBPARTITION y1996 VALUES LESS THAN (DATE '1997-01-01'),
  SUBPARTITION y1997 VALUES LESS THAN (DATE '1998-01-01'),
  SUBPARTITION y1998 VALUES LESS THAN (DATE '1999-01-01'),
  SUBPARTITION y1999 VALUES LESS THAN (DATE '2000-01-01'),
  SUBPARTITION y2000 VALUES LESS THAN (DATE '2001-01-01'),
  SUBPARTITION y2001 VALUES LESS THAN (DATE '2002-01-01'),
  SUBPARTITION y2002 VALUES LESS THAN (DATE '2003-01-01'),
  SUBPARTITION y2003 VALUES LESS THAN (DATE '2004-01-01'),
  SUBPARTITION y2004 VALUES LESS THAN (DATE '2005-01-01'),
  SUBPARTITION y2005 VALUES LESS THAN (DATE '2006-01-01'),
  SUBPARTITION y2006 VALUES LESS THAN (DATE '2007-01-01'),
  SUBPARTITION y2007 VALUES LESS THAN (DATE '2008-01-01'),
  SUBPARTITION y2008 VALUES LESS THAN (DATE '2009-01-01'),
  SUBPARTITION y2009 VALUES LESS THAN (DATE '2010-01-01'),
  SUBPARTITION y2010 VALUES LESS THAN (DATE '2011-01-01'),
  SUBPARTITION y2011 VALUES LESS THAN (DATE '2012-01-01'),
  SUBPARTITION y2012 VALUES LESS THAN (DATE '2013-01-01'),
  SUBPARTITION y2013 VALUES LESS THAN (DATE '2014-01-01'),
  SUBPARTITION y2014 VALUES LESS THAN (DATE '2015-01-01'),
  SUBPARTITION y2015 VALUES LESS THAN (DATE '2016-01-01'),
  SUBPARTITION y2016 VALUES LESS THAN (DATE '2017-01-01'),
  SUBPARTITION y2017 VALUES LESS THAN (DATE '2018-01-01'),
  SUBPARTITION y2018 VALUES LESS THAN (DATE '2019-01-01'),
  SUBPARTITION y2019 VALUES LESS THAN (DATE '2020-01-01'),
  SUBPARTITION y2020 VALUES LESS THAN (DATE '2021-01-01'),
  SUBPARTITION y2021 VALUES LESS THAN (DATE '2022-01-01'),
  SUBPARTITION y2022 VALUES LESS THAN (DATE '2023-01-01'),
  SUBPARTITION y2023 VALUES LESS THAN (DATE '2024-01-01'),
  SUBPARTITION y2024 VALUES LESS THAN (DATE '2025-01-01'),
  SUBPARTITION y_max  VALUES LESS THAN (MAXVALUE)
```

Rysunek 4.21 Subpartycje zakresowe w Oracle

Źródło: opracowanie własne.

Dla zwiększenia wydajności i odwzorowania warunków z tabeli z bazy MSSQL utworzono sześć indeksów na kolumnach (CustomerID, EmployeeID, ShipVia, OrderDate, ShippedDate, ShipPostalCode) na tabeli OrdersPartitioned. Poniżej przedstawiono operacje DDL (ang. Data Definition Language), które je stworzyły.

```
CREATE INDEX IDX_ORDERSPARTITIONED_CUSTOMER_ID       ON OrdersPartitioned(CustomerID);
CREATE INDEX IDX_ORDERSPARTITIONED_EMPLOYEE_ID       ON OrdersPartitioned(EmployeeID);
CREATE INDEX IDX_ORDERSPARTITIONED_SHIPPER_ID        ON OrdersPartitioned(ShipVia);
CREATE INDEX IDX_ORDERSPARTITIONED_ORDER_DATE        ON OrdersPartitioned(OrderDate);
CREATE INDEX IDX_ORDERSPARTITIONED_SHIPPED_DATE      ON OrdersPartitioned(ShippedDate);
CREATE INDEX IDX_ORDERSPARTITIONED_SHIP_POSTAL_CODE  ON OrdersPartitioned(ShipPostalCode);
```

Rysunek 4.22 Indeksy na tabeli OrdersPartitioned w Oracle

Źródło: opracowanie własne.

Gdy już zaimplementowano partycjonowanie na tabeli OrdersPartitioned wykonano ostatni krok, którym było załadowanie danych do tabeli z pliku Orders.csv.

4.2.3 PostgreSQL – przygotowanie środowiska i implementacja partycjonowania

W celu przeprowadzenia testów partycjonowania w systemie PostgresSQL przygotowano środowisko testowe oparte na kontenerze Dockera. Z repozytorium Docker Hub pobrano oficjalny obraz PostgresSQL 17.2. Utworzono wolumen danych „postgres-data", przeznaczony do przechowywania plików bazy poza kontenerem. Ustawiono parametr pamięci współdzielonej (ang. shared memory) na poziomie 3GB dla kontenera. Taki zabieg pozwala uruchamiać zapytania testowe równolegle bez błędów. Jako narzędzie graficzne do obsługi środowiska PostgresSQL wykorzystano pgAdmin4. W celu odtworzenia danych bazy Northwind wykorzystano skrypt northwind.sql pobrany z repozytorium: https://github.com/pthom/northwind_psql/blob/master/northwind.sql gdzie kod został nieznacznie zmieniony na potrzeby projektu. Zmodyfikowany plik dołączono do folderu projektu. Z serwera MSSQL pobrano zbiór danych tabeli Orders w celu wykonania badań na dokładnie tych samych danych. Dokonano eksportu do pliku Orders.csv. Następnie zbiór załadowano do tabeli Orders na bazie Postgres.

Do implementacji partycjonowania na serwerze Postgres zastosowano partycjonowanie kompozytowe. Tabela nadrzędna OrdersPartitioned została zdefiniowana z kompozytowym kluczem głównym (OrderID, ShipCountry, OrderDate), zgodnym z wymogami unikalności w środowisku partycjonowanym dla serwera Postgres oraz partycjonowaniem listowym po kraju wysyłki (ShipCountry), co stanowi pierwszy poziom podziału. Poniżej została

przedstawiona struktura tabeli OrdersPartitioned zawarta w skrypcie OrdersPartitioned_PostgreSQL_Script.sql.

```
CREATE TABLE OrdersPartitioned (
    OrderID        INT,
    CustomerID     VARCHAR(10),
    EmployeeID     INT,
    OrderDate      DATE     NOT NULL,
    RequiredDate   DATE,
    ShippedDate    DATE,
    ShipVia        INT,
    Freight        NUMERIC,
    ShipName       VARCHAR(100),
    ShipAddress    VARCHAR(255),
    ShipCity       VARCHAR(100),
    ShipRegion     VARCHAR(100),
    ShipPostalCode VARCHAR(20),
    ShipCountry    VARCHAR(100) NOT NULL,
    PRIMARY KEY (OrderID, ShipCountry, OrderDate)
) PARTITION BY LIST (ShipCountry);
```

Rysunek 4.23 Struktura tabeli OrdersPartitioned w Postgres

Źródło: opracowanie własne.

Kolejny krok realizuje blok anonimowy w PL/pgSQL widoczny na rysunku 4.23, który w sposób deklaratywno- proceduralny wygenerował fizyczne partycje. Dla każdej wartości z listy krajów tworzona jest partycja listowa orders_<Kraj> oraz wewnątrz niej partycje zakresowe po dacie zamówienia w układzie rocznym orders_<Kraj>_<rok> dla lat 1996-2025. Dzięki temu uzyskano dwa poziomy partycjonowania odpowiadające implementacji wielowymiarowej.

```
DO $$
DECLARE
    country  TEXT;
    yr       INT;
    start_d   DATE;
    end_d     DATE;
    countries CONSTANT TEXT[] := ARRAY[
        'Argentina','Australia','Austria','Belgium','Brazil','Canada',
        'Denmark','Finland','France','Germany','Ireland','Italy','Mexico',
        'Norway','Poland','Portugal','Spain','Sweden','Switzerland',
        'USA','Venezuela'
    ];
BEGIN
    FOREACH country IN ARRAY countries LOOP
        EXECUTE format(
            'CREATE TABLE IF NOT EXISTS %I
               PARTITION OF OrdersPartitioned
               FOR VALUES IN (%L)
               PARTITION BY RANGE (OrderDate);',
            'orders_' || country, country);

        FOR yr IN 1996..2025 LOOP
            start_d := make_date(yr, 1, 1);
            end_d   := make_date(yr + 1, 1, 1);

            EXECUTE format(
                'CREATE TABLE IF NOT EXISTS %I
                   PARTITION OF %I
                   FOR VALUES FROM (%L) TO (%L);',
                'orders_' || country || '_' || yr, 'orders_' || country, start_d, end_d);
        END LOOP;
    END LOOP;
END $$;
```

Rysunek 4.24 Kod generujący partycje w Postgres

Źródło: opracowanie własne.

Następnie skrypt odtwarza zestaw nieunikalnych indeksów na kolumnach: ShippedDate, EmployeeID, ShipVia, ShipPostalCode, CustomerID i OrderDate.

```
CREATE INDEX IF NOT EXISTS "ShippedDate" ON OrdersPartitioned (ShippedDate);
CREATE INDEX IF NOT EXISTS "EmployeeID" ON OrdersPartitioned (EmployeeID);
CREATE INDEX IF NOT EXISTS "ShippersOrders" ON OrdersPartitioned (ShipVia);
CREATE INDEX IF NOT EXISTS "ShipPostalCode" ON OrdersPartitioned (ShipPostalCode);
CREATE INDEX IF NOT EXISTS "CustomerID" ON OrdersPartitioned (CustomerID);
CREATE INDEX IF NOT EXISTS "OrderDate" ON OrdersPartitioned (OrderDate);
```

Rysunek 4.25 Indeksy na tabeli OrdersPartitioned w Postgres

Źródło: opracowanie własne.

Na samym końcu skryptu dodane są więzy kluczy obcych do tabel referencyjnych shippers (shippedid), employees (employeeid) i customers (customerid).

Gdy już w pełni dokonano zmian na tabeli OrdersPartitoned zasilono ja danymi tymi samymi co tabele Orders z pliku Orders.csv.

4.3 Metodologia badawcza

Na potrzeby badań przygotowano zestaw zapytań testowych dotyczących manipulacji danymi, obejmujący operacje SELECT, INSERT, UPDATE, które należą do grona działań DML (ang. Data Manipulation Language). Celem tych zapytań było porównanie wydajności działania systemu bazodanowego w zależności od użycia mechanizmu partycjonowania danych. Zarówno dla niepartycjonowanej tabeli Orders, jak i dla partycjonowanej tabeli OrdersPartitioned skonstruowano analogiczne zapytania z uwzględnieniem możliwości trafienia lub nietrafienia w konkretną partycję.

Dodatkowo, ze względu na różnice implementacyjne w systemach zarządzania bazami danych (MSSQL, Oracle, Postgres), zapytania w testach wydajnościowych różniły się między sobą. W szczególności w Microsoft SQL Server wykorzystano obliczeniową kolumnę PartitionKey, która stanowiła podstawę partycjonowania, co wpłynęło na konieczność zmiany struktury zapytań w porównaniu do zapytań używanych w Oracle czy Postgres. W rezultacie dla MSSQL stosowano uproszczone zapytania z warunkiem na PartitionKey, podczas gdy w pozostałych systemach konieczne było użycie filtrów na kolumny ShipCountry i OrderDate. Dzieki temu zachowano porównywalność logiki zapytań przy jednoczesnym uwzględnieniu specyfikacji poszczególnych środowisk testowych.

Środowiska testowe dla poszczególnych systemów bazodanowych zostały przygotowane w technologii konteneryzacji Docker, co umożliwiło na uruchomienie każdej bazy w odseparowanym i kontrolowanym środowisku z takimi samymi parametrami sprzętowymi.

Do realizacji scenariuszy testowych wykorzystano narzędzie Apache Jmeter, które umożliwia wykonywanie zapytań SQL z określoną liczba wątków i rejestrowanie szczegółowych metryk wydajnościowych.

Badania wykonano w dwóch głównych trybach:

- Testy wydajnościowe – operacje pozwalające na ocenę czasu odpowiedzi pojedynczych zapytań w sposób powtarzalny.
- Testy obciążeniowe – scenariusze z równoczesnym wykonywaniem zapytań przez wiele wątków, symulujące rzeczywiste środowisko pracy systemu.

W celu porównania wydajności/obciążenia zastosowano następujące miary porównawcze:

- Średni czas wykonania – uzyskany poprzez 10 krotne wykonanie każdego zapytania lub 100 krotne wykonanie równoległych zapytań i wyliczenie średniej.

- Odchylenie standardowe – umożliwia określić stabilność działania systemu przy powtarzalnych zapytaniach.
- Przepustowość – liczba wykonanych zapytań na sekundę.

Z zamiarem zapewnienia powtarzalności i porównywalności wyników wszystkie scenariusze testowe wykonywano na identycznych danych wejściowych, przy stałych parametrach środowisk kontenerowych oraz nie zmienianej konfiguracji silników baz danych. Wyjątkiem było ustawienie zwiększonej pamięci współdzielonej w PostgreSQL, co było konieczne do poprawnego działania testów wielowątkowych. Aby zapewnić spójność prezentowanych wyników testów, przyjęto zasadę podawania wartości z dokładnością do dwóch miejsc po przecinku w przypadku odchylenia standardowego oraz przepustowości. Pozwala to zachować równowagę pomiędzy czytelnością a precyzją danych. Natomiast czasy wykonania zapytań prezentowane były w milisekundach, zgodnie z formatem zwracanym przez narzędzie Apache Jmeter, bez dodatkowego zaokrąglania.

4.4 Rodzaje badań

Przeprowadzone badania zostały podzielone na dwie główne grupy testów, z których każda miała na celu analizę innego aspektu działania systemów bazodanowych: testy wydajnościowe oraz testy obciążeniowe, co pozwoliło na ocenę zarówno szybkości wykonywania pojedynczych zapytań, jak i stabilności działania systemu w warunkach wielodostępu.

Testy wydajnościowe przeprowadzone przy użyciu Apache Jmeter miały na celu określenie wpływu mechanizmu partycjonowania na czas wykonywania operacji w warunkach niskiego obciążenia. Każde zapytanie było wykonywane 10 razy w każdym środowisku testowym, zarówno w tabeli niepartycjonowanej Orders, jak i partycjonowanej Orderspartitioned. W konfiguracji narzędzia Jmeter zastosowano grupę wątków z jednym użytkownikiem (1 thread), który sekwencyjnie wykonywał 10 powtórzeń zapytania, co pozwoliło wyeliminować wpływ równoległości i skupić się wyłącznie na pomiarze czasu odpowiedzi. Analiza wyników obejmowała wyliczenie średniego czasu wykonywania zapytań, odchylenie standardowe i przepustowości. Zestawy zapytań obejmowały operacje SELECT, INSERT, UPDATE. Przygotowano je w dwóch wariantach: dla tabeli Orders, gdzie nie występowało partycjonowanie oraz dla tabeli OrdersPartitioned gdzie filtrowanie umożliwiało trafienie w partycje.

Testy obciążeniowe służyły do oceny zachowania systemu przy jednoczesnym dostępie wielu użytkowników. W tym celu również wykorzystano narzędzie Jmeter, które umożliwiło równoczesne wykonywanie zapytań przez 100 wirtualnych użytkowników. Konfiguracja grupy wątków zakładała uruchomienie 100 wątków w ciągu 10 sekund, co pozwoliło na stopniowe zainicjowanie obciążenia i symulację bardziej rzeczywistego scenariusza pracy wielu użytkowników. Każdy ze scenariuszy był uruchamiany wielokrotnie z uwzględnieniem operacji odczytu jak i modyfikacji danych. Rejestrowano te same parametry co w testach wydajnościowych.

Poniżej w tabeli 4.5 przedstawiono zbiór zapytań SELECT, INSERT i UPDATE dla testów wydajnościowych. Testy DELETE zostały pominięte z uwagi na ich niski wpływ na praktyczne wykorzystanie partycjonowania oraz brak powtarzalności wyników bez ponownego ładowania danych. Zamiast tego skupiono się na testach SELECT, INSERT oraz UPDATE, które lepiej oddają realne operacje w systemach transakcyjnych. Przygotowano pięć zestawów zapytań SELECT i po jednym zestawie dla aktualizacji i dodania danych.

Tabela 4.5 Zbiór zapytań dla testów wydajnościowych

ID zapytania	Rodzaj serwera bazodanowego	Typ	Tabela	Zapytanie	Czy trafia w partycję?
TW_SEL_Ord_1	MSSQL, Oracle, Postgres	SELECT	Orders	`SELECT * FROM Orders WHERE OrderDate BETWEEN '2016-01-01' AND '2016-12-31' AND ShipCountry = 'Australia';`	NIE
TW_SEL_OrdPar_1	Oracle, Postgres	SELECT	OrdersPartitioned	`SELECT * FROM OrdersPartitioned WHERE OrderDate BETWEEN '2016-01-01' AND '2016-12-31' AND ShipCountry = 'Australia';`	TAK
TW_SEL_MSSQL_OrdPar_1	MSSQL	SELECT	OrdersPartitioned	`SELECT * FROM OrdersPartitioned WHERE PartitionKey = 'A_2016'`	TAK
TW_SEL_MSSQL_Ord_2	MSSQL	SELECT	Orders	`SELECT YEAR(OrderDate) AS Rok, COUNT(*) AS LiczbaZamowien FROM Orders`	NIE

				WHERE OrderDate BETWEEN '2017-01-01' AND '2019-12-31' AND ShipCountry = 'Australia' GROUP BY YEAR(OrderDate);	
TW_SEL_Ord_2	Oracle, Postgres	SELECT	Orders	SELECT EXTRACT(YEAR FROM OrderDate) AS Rok, COUNT(*) AS LiczbaZamowien FROM Orders WHERE OrderDate BETWEEN '2017-01-01' AND '2019-12-31' AND ShipCountry = 'Australia' GROUP BY Rok;	NIE
TW_SEL_MSSQL_OrdPar_2	MSSQL	SELECT	OrdersPartitioned	SELECT YEAR(OrderDate) AS Rok, COUNT(*) AS LiczbaZamowien FROM OrdersPartitioned WHERE PartitionKey IN ('A_2017', 'A_2018', 'A_2019') GROUP BY YEAR(OrderDate);	TAK
TW_SEL_OrdPar_2	Oracle, Postgres	SELECT	OrdersPartitioned	SELECT EXTRACT(YEAR FROM OrderDate) AS Rok, COUNT(*) AS LiczbaZamowien FROM OrdersPartitioned WHERE OrderDate BETWEEN '2017-01-01' AND '2019-12-31' AND ShipCountry = 'Australia' GROUP BY Rok;	TAK
TW_SEL_Ord_3	MSSQL, Oracle, Postgres	SELECT	Orders	SELECT * FROM Orders WHERE ShipCountry = 'Australia';	NIE
TW_SEL_OrdPar_3	Oracle, Postgres	SELECT	OrdersPartitioned	SELECT * FROM OrdersPartitioned WHERE ShipCountry = 'Australia';	TAK
TW_SEL_MSSQL_OrdPar_3	MSSQL	SELECT	OrdersPartitioned	SELECT * FROM OrdersPartitioned WHERE PartitionKey LIKE 'A_%';	TAK
TW_SEL_Ord_4	MSSQL, Oracle, Postgres	SELECT	Orders	SELECT * FROM Orders WHERE Freight > 100	NIE

				AND OrderDate BETWEEN '2020-01-01' AND '2020-12-31' AND ShipCountry = 'Brazil';	
TW_SEL_OrdPar_4	Oracle, Postgres	SELECT	OrdersPartitioned	SELECT * FROM OrdersPartitioned WHERE Freight > 100 AND OrderDate BETWEEN '2020-01-01' AND '2020-12-31' AND ShipCountry = 'Brazil';	TAK
TW_SEL_MSSQL_OrdPar_4	MSSQL	SELECT	OrdersPartitioned	SELECT * FROM OrdersPartitioned WHERE Freight > 100 AND PartitionKey = 'S_2020' AND ShipCountry = 'Brazil';	TAK
TW_SEL_Ord_5	MSSQL, Postgres	SELECT	Orders	SELECT o.OrderID, o.OrderDate, c.CompanyName FROM Orders o JOIN Customers c ON o.CustomerID = c.CustomerID WHERE o.OrderDate BETWEEN '2021-01-01' AND '2021-12-31' AND o.ShipCountry = 'Australia';	NIE
TW_SEL_Oracle_Ord_5	Oracle	SELECT	Orders	SELECT o.OrderID, o.OrderDate, c.Company_Name FROM Orders o JOIN Customers c ON o.CustomerID = c.Customer_ID WHERE o.OrderDate BETWEEN '2021-01-01' AND '2021-12-31' AND o.ShipCountry = 'Australia';	NIE
TW_SEL_Oracle_OrdPar_5	Oracle	SELECT	OrdersPartitioned	SELECT o.OrderID, o.OrderDate, c.Company_Name FROM OrdersPartitioned o JOIN Customers c ON o.CustomerID = c.Customer_ID WHERE o.OrderDate BETWEEN '2021-01-01' AND '2021-12-31' AND o.ShipCountry = 'Australia';	TAK

TW_SEL_OrdPar_5	Postgres	SELECT	OrdersPartitioned	SELECT o.OrderID, o.OrderDate, c.CompanyName FROM OrdersPartitioned o JOIN Customers c ON o.CustomerID = c.CustomerID WHERE o.OrderDate BETWEEN '2021-01-01' AND '2021-12-31' AND o.ShipCountry = 'Australia';	TAK
TW_SEL_MSSQL_OrdPar_5	MSSQL	SELECT	OrdersPartitioned	SELECT o.OrderID, o.OrderDate, c.CompanyName FROM OrdersPartitioned o JOIN Customers c ON o.CustomerID = c.CustomerID WHERE o.PartitionKey = 'A_2021';	TAK
TW_INS_MSSQL_Ord_1	MSSQL	INSERT	Orders	INSERT INTO dbo.Orders (CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry) SELECT TOP (50000) CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry FROM dbo.Orders;	NIE
TW_INS_ Oracle _Ord_1	Oracle	INSERT	Orders	INSERT INTO Orders (CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName,	NIE

				ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry) SELECT CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry FROM Orders FETCH FIRST 50000 ROWS ONLY;	
TW_INS_ Postgres _Ord_1	Postgres	INSERT	Orders	INSERT INTO Orders (CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry) SELECT CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry FROM Orders LIMIT 50000;	NIE
TW_INS_MSSQL_OrdPar_1	MSSQL	INSERT	OrdersPartitioned	INSERT INTO dbo.OrdersPartitioned (CustomerID, EmployeeID, OrderDate,	NIE

				RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry) SELECT TOP (50000) CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry FROM dbo. OrdersPartitioned;	
TW_INS_ Oracle _OrdPar_1	Oracle	INSERT	OrdersPartitioned	INSERT INTO OrdersPartitioned (CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry) SELECT CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry FROM OrdersPartitioned FETCH FIRST 50000 ROWS ONLY;	NIE
TW_INS_ Postgres _OrdPar_1	Postgres	INSERT	OrdersPartitioned	INSERT INTO OrdersPartitioned (	NIE

				CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry) SELECT CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry FROM OrdersPartitioned LIMIT 50000;	
TW_UPD_Ord_1	MSSQL, Oracle, Postgres	UPDATE	Orders	UPDATE Orders SET Freight = Freight + 1 WHERE ShipCountry = 'Australia' AND OrderDate BETWEEN '2020-01-01' AND '2020-12-31';	NIE
TW_UPD_OrdPar_1	Oracle, Postgres	UPDATE	OrdersPartitioned	UPDATE OrdersPartitioned SET Freight = Freight + 1 WHERE ShipCountry = 'Australia' AND OrderDate BETWEEN '2020-01-01' AND '2020-12-31';	TAK
TW_UPD_MSSQL_OrdPar_1	MSSQL	UPDATE	OrdersPartitioned	UPDATE OrdersPartitioned SET Freight = Freight + 1 WHERE PartitionKey = 'A_2021';	TAK

Źródło: opracowanie własne.

Poniżej w tabeli 4.6 przedstawiono zbiór zapytań SELECT, INSERT i UPDATE dla testów obciążeniowych, podzielonych według typu operacji, rodzaju testu oraz systemu bazodanowego. Przygotowano dwa zestawy zapytań dla odczytu danych i po jednym zestawie dla dodania i aktualizacji rekordów w bazie.

Tabela 4.6 Zbiór zapytań dla testów obciążeniowych

ID zapytania	Rodzaj serwera bazodanowego	Typ	Tabela	Zapytanie	Czy trafia w partycję?
TO_SEL_Ord_1	MSSQL, Oracle, Postgres	SELECT	Orders	`SELECT OrderID, OrderDate, Freight FROM Orders WHERE ShipCountry = 'Australia' AND OrderDate BETWEEN '2013-01-01' AND '2013-12-31';`	NIE
TO_SEL_OrdPar_1	Oracle, Postgres	SELECT	OrdersPartitioned	`SELECT OrderID, OrderDate, Freight FROM OrdersPartitioned WHERE ShipCountry = 'Australia' AND OrderDate BETWEEN '2013-01-01' AND '2013-12-31';`	TAK
TO_SEL_MSSQL_OrdPar_1	MSSQL	SELECT	OrdersPartitioned	`SELECT OrderID, OrderDate, Freight FROM OrdersPartitioned WHERE PartitionKey = 'A_2013';`	TAK
TO_SEL_Ord_2	MSSQL, Oracle, Postgres	SELECT	Orders	`SELECT OrderID, OrderDate, Freight, ShipCity FROM Orders WHERE OrderDate BETWEEN '2008-01-01' AND '2008-12-31';`	NIE
TO_SEL_OrdPar_2	Oracle, Postgres	SELECT	OrdersPartitioned	`SELECT OrderID, OrderDate, Freight, ShipCity FROM OrdersPartitioned WHERE OrderDate BETWEEN '2008-01-01' AND '2008-12-31';`	TAK
TO_SEL_MSSQL_OrdPar_2	MSSQL	SELECT	OrdersPartitioned	`SELECT OrderID, OrderDate, Freight, ShipCity FROM OrdersPartitioned WHERE PartitionKey IN ('A_2008','N_2008', 'S_2008', 'E_2008');`	TAK
TO_INS_MSSQL_Ord_1	MSSQL	INSERT	Orders	`INSERT INTO Orders (`	NIE

				CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry) SELECT TOP (5000) CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry FROM Orders;	
TO_INS_MSSQL_OrdPar_1	MSSQL	INSERT	OrdersPartitioned	INSERT INTO OrdersPartitioned (CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry) SELECT TOP (5000) CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry FROM OrdersPartitioned;	NIE
TO_INS_Oracle_Ord_1	Oracle	INSERT	Orders	INSERT INTO Orders (CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity,	NIE

				ShipRegion, ShipPostalCode, ShipCountry) SELECT CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry FROM Orders FETCH FIRST 5000 ROWS ONLY;	
TO_INS_Oracle_OrdPar_1	Oracle	INSERT	OrdersPartitioned	INSERT INTO OrdersPartitioned (CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry) SELECT CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry FROM OrdersPartitioned FETCH FIRST 5000 ROWS ONLY;	NIE
TO_INS_Postgres_Ord_1	Postgres	INSERT	Orders	INSERT INTO Orders (CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity,	NIE

				ShipRegion, ShipPostalCode, ShipCountry) SELECT CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry FROM Orders LIMIT 5000;	
TO_INS_Postgres_OrdPar_1	Postgres	INSERT	OrdersPartitioned	INSERT INTO OrdersPartitioned (CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry) SELECT CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry FROM OrdersPartitioned LIMIT 5000;	NIE
TO_UPD_Ord_1	MSSQL, Oracle, Postgres	UPDATE	Orders	UPDATE Orders SET ShippedDate = ShippedDate + 1 WHERE OrderID BETWEEN 12345 AND 20345 AND ShipCountry = 'Australia';	NIE
TO_UPD_OrdPar_1	Oracle, Postgres	UPDATE	OrdersPartitioned	UPDATE OrdersPartitioned SET ShippedDate = ShippedDate + 1	TAK

				`WHERE OrderID BETWEEN 12345 AND 20345 AND ShipCountry = 'Australia';`	
TO_UPD_MSSQL_OrdPar_1	MSSQL	UPDATE	OrdersPartitioned	`UPDATE OrdersPartitioned SET ShippedDate = ShippedDate + 1 WHERE OrderID BETWEEN 12345 AND 20345 AND PartitionKey LIKE 'A_%';`	TAK

Źródło: opracowanie własne.

4.4.1 Microsoft SQL Server

W tej części przedstawiono wyniki testów wydajnościowych przeprowadzonych dla systemu Microsoft SQL Server. Pomiarom poddano trzy typy operacji: SELECT, INSERT i UPDATE. Wykonywano je zarówno na tabeli niepartycjonowanej (Orders), jak i partycjonowanej (OrdersPartitioned). Każde z zapytań zostało oznaczone unikalnym identyfikatorem (ID zapytania), który odpowiada zestawieniu zawartemu w tabeli 4.5 z opisem zapytań wydajnościowych. W poniższych tabelach 4.7, 4.8 i 4.9 zawarte są dane uzyskane w ramach 10 krotnego wykonania każdego zapytania, które posłużą do dalszej analizy porównawczej w kolejnym rozdziale pracy.

Tabela 4.7 Testy wydajnościowe – MSSQL – zapytania SELECT

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TW_SEL_Ord_1	SELECT	Orders	10	2710	2250	3237	253,56	0,37
TW_SEL_MSSQL_OrdPar_1	SELECT	OrdersPartitioned	10	200	116	909	236,31	4,99
TW_SEL_MSSQL_Ord_2	SELECT	Orders	10	1941	522	9140	2515,21	0,51
TW_SEL_MSSQL_OrdPar_2	SELECT	OrdersPartitioned	10	81	59	273	63,75	12,20
TW_SEL_Ord_3	SELECT	Orders	10	4041	2549	7314	1710,35	0,25
TW_SEL_MSSQL_OrdPar_3	SELECT	OrdersPartitioned	10	2843	2608	3860	359,63	0,35
TW_SEL_Ord_4	SELECT	Orders	10	2366	504	14243	3994,74	0,42
TW_SEL_MSSQL_OrdPar_4	SELECT	OrdersPartitioned	10	193	29	1591	465,87	5,16
TW_SEL_Ord_5	SELECT	Orders	10	3798	2551	12205	2810	0,26
TW_SEL_MSSQL_OrdPar_5	SELECT	OrdersPartitioned	10	297	40	2507	736,66	3,36

Źródło: opracowanie własne.

Tabela 4.8 Testy wydajnościowe – MSSQL – zapytania INSERT

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TW_INS_MSSQL_Ord_1	INSERT	Orders	10	2409	2032	3780	530,63	0,42
TW_INS_MSSQL_OrdPar_1	INSERT	OrdersPartitioned	10	2640	2300	4937	769,31	0,38

Źródło: opracowanie własne.

Tabela 4.9 Testy wydajnościowe – MSSQL – zapytania UPDATE

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TW_UPD_Ord_1	UPDATE	Orders	10	4926	4336	5493	338,08	0,20
TW_UPD_MSSQL_OrdPar_1	UPDATE	OrdersPartitioned	10	220	66	532	143,67	4,53

Źródło: opracowanie własne.

W kolejnej części ukazano rezultaty testów obciążeniowych przeprowadzonych na systemie Microsoft SQL Server. Ich głównym celem było zbadanie, jak baza danych funkcjonuje przy równoczesnym dostępie dużej liczby użytkowników. W tym przypadku zastosowano narzędzie Apache JMeter, które pozwoliło na jednoczesne wykonywanie zapytań przez 100 wirtualnych użytkowników, uruchamianych w ustawionym przedziale czasowym równym 10 sekund. Podobnie jak w testach wydajności, każde zapytanie zostało przypisane do odpowiedniego identyfikatora (ID zapytania), zgodnego z listą testowych zapytań w tabeli 4.6. Wyniki zawierają wartości uzyskane podczas jednego przebiegu testu obciążeniowego, które następnie zostały uśrednione, aby uzyskać bardziej precyzyjny obraz działania systemu. W przedstawionych tabelach 4.10, 4.11 i 4.12 można znaleźć szczegółowe rezultaty dotyczące zapytań typu SELECT, INSERT oraz UPDATE, które zostały wykonane na tabelach Orders oraz OrdersPartitioned.

Tabela 4.10 Testy obciążeniowe – MSSQL – zapytania SELECT

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TO_SEL_Ord_1	SELECT	Orders	100	73113	67308	90733	7665,18	1,10
TO_SEL_MSSQL_OrdPar_1	SELECT	OrdersPartitioned	100	1346	352	2119	434,5	40,40
TO_SEL_Ord_2	SELECT	Orders	100	63519	57002	77765	5773,25	1,15
TO_SEL_MSSQL_OrdPar_2	SELECT	OrdersPartitioned	100	15674	3070	18646	3843,2	4,02

Źródło: opracowanie własne.

Tabela 4.11 Testy obciążeniowe – MSSQL – zapytania INSERT

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TO_INS_MSSQL_Ord_1	INSERT	Orders	100	72576	23488	81427	9907,06	1,11
TO_INS_MSSQL_OrdPar_1	INSERT	OrdersPartitioned	100	34101	4389	46880	11527,98	1,76

Źródło: opracowanie własne.

Tabela 4.12 Testy obciążeniowe – MSSQL – zapytania UPDATE

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TO_UPD_Ord_1	UPDATE	Orders	100	3744	338	6778	1822,59	5,99
TO_UPD_MSSQL_OrdPar_1	UPDATE	OrdersPartitioned	100	100652	3125	206566	57735,52	0,46

Źródło: opracowanie własne.

4.4.2 Oracle Database

W tej części przedstawiono wyniki testów wydajnościowych realizowanych w środowisku Oracle Database 23c. Metoda testowa była taka sama jak w systemie Microsoft SQL Server. Każde zapytanie zostało uruchomione 10 razy dla zarówno tabeli Orders, jak i OrdersPartitioned. Wyniki zawierają te same wskaźniki pomiarowe, takie jak średni czas odpowiedzi, wartości minimalne i maksymalne, odchylenie standardowe oraz wydajność

zapytań. Poniżej zaprezentowano tabele wynikowe 4.13, 4.14 i 4.15 odpowiednio dla operacji SELECT, INSERT i UPDATE, które stanowią podstawę do dalszej analizy wyników.

Tabela 4.13 Testy wydajnościowe – Oracle – zapytania SELECT

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TW_SEL_Ord_1	SELECT	Orders	10	13408	10095	20197	2632,49	0,07
TW_SEL_OrdPar_1	SELECT	OrdersPartitioned	10	466	254	787	144,35	2,14
TW_SEL_Ord_2	SELECT	Orders	10	6104	1451	24273	6315,67	0,16
TW_SEL_OrdPar_2	SELECT	OrdersPartitioned	10	34	18	173	46,14	28,90
TW_SEL_Ord_3	SELECT	Orders	10	18007	15229	31489	4670,47	0,06
TW_SEL_OrdPar_3	SELECT	OrdersPartitioned	10	38	1	373	111,53	26,04
TW_SEL_Ord_4	SELECT	Orders	10	4566	1431	10360	2751,61	0,22
TW_SEL_OrdPar_4	SELECT	OrdersPartitioned	10	80	48	287	69,02	12,39
TW_SEL_Oracle_Ord_5	SELECT	Orders	10	4271	1442	6939	1921,63	0,23
TW_SEL_Oracle_OrdPar_5	SELECT	OrdersPartitioned	10	92	65	232	47,46	10,85

Źródło: opracowanie własne.

Tabela 4.14 Testy wydajnościowe – Oracle – zapytania INSERT

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TW_INS_Oracle_Ord_1	INSERT	Orders	10	8961	7019	12226	1548,61	0,11
TW_INS_Oracle_OrdPar_1	INSERT	OrdersPartitioned	10	4202	3563	5932	692,52	0,24

Źródło: opracowanie własne.

Tabela 4.15 Testy wydajnościowe – Oracle – zapytania INSERT

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TW_UPD_Ord_1	UPDATE	Orders	10	24245	22826	25114	538,6	0,04
TW_UPD_OrdPar_1	UPDATE	OrdersPartitioned	10	413	300	769	130,47	2,42

Źródło: opracowanie własne.

W następnym kroku analizy pokazano rezultaty testów obciążeniowych wykonanych w systemie Oracle Database 23c. Metoda była również taka sama jak ta użyta w Microsoft SQL Server. Testy były przeprowadzane z wykorzystaniem narzędzia Apache JMeter, które pozwoliło na jednoczesne uruchamianie zapytań przez 100 wirtualnych użytkowników. Wyniki obejmują operacje SELECT, UPDATE i INSERT i są zawarte odpowiednio w tabelach 4.16, 4.17 i 4.18.

Tabela 4.16 Testy obciążeniowe – Oracle – zapytania SELECT

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TO_SEL_Ord_1	SELECT	Orders	100	19702	5415	23915	5444,49	3,25
TO_SEL_OrdPar_1	SELECT	OrdersPartitioned	100	169	143	486	41,59	9,94
TO_SEL_Ord_2	SELECT	Orders	100	76412	49125	82877	8449,76	1,14
TO_SEL_OrdPar_2	SELECT	OrdersPartitioned	100	11910	3592	14418	3006,55	4,62

Źródło: opracowanie własne.

Tabela 4.17 Testy obciążeniowe – Oracle – zapytania INSERT

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TO_INS_Oracle_Ord_1	INSERT	Orders	100	34388	17378	40032	6789,96	2,14
TO_INS_Oracle_OrdPar_1	INSERT	OrdersPartitioned	100	8696	621	13950	4577,17	4,76

Źródło: opracowanie własne.

Tabela 4.18 Testy obciążeniowe – Oracle – zapytania UPDATE

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TO_UPD_Ord_1	UPDATE	Orders	100	37515	2260	70156	20067,52	1,25
TO_UPD_OrdPar_1	UPDATE	OrdersPartitioned	100	311026	11429	604216	174382,02	0,16

Źródło: opracowanie własne.

4.4.3 PostgresSQL

W tej części zaprezentowano w tabelach 4.19, 4.20 i 4.21 wyniki testów wydajności, które zostały przeprowadzone w środowisku PostgreSQL. Procura była analogiczna jak w pozostałych systemach: każde polecenie wykonywano 10 krotnie dla tabeli Orders oraz OrdersPartitioned. W JMeterze ustalono jednego użytkownika, który w sposób sekwencyjny wykonał 10 powtórzeń. Miało to na celu zminimalizowanie wpływu wielowątkowości i skupienie się na mierzeniu czasu pojedynczego żądania.

Tabela 4.19 Testy wydajnościowe – PostgreSQL – zapytania SELECT

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TW_SEL_Ord_1	SELECT	Orders	10	2499	1983	3450	495,63	0,40
TW_SEL_OrdPar_1	SELECT	OrdersPartitioned	10	179	94	707	177,59	5,57
TW_SEL_Ord_2	SELECT	Orders	10	2284	2195	2810	176,82	0,44
TW_SEL_OrdPar_2	SELECT	OrdersPartitioned	10	45	27	203	52,44	21,74
TW_SEL_Ord_3	SELECT	Orders	10	4240	3799	4797	278,89	0,24
TW_SEL_OrdPar_3	SELECT	OrdersPartitioned	10	2793	2681	2941	91,02	0,36
TW_SEL_Ord_4	SELECT	Orders	10	1402	1274	2177	261,43	0,71
TW_SEL_OrdPar_4	SELECT	OrdersPartitioned	10	36	21	144	35,83	27,10
TW_SEL_Ord_5	SELECT	Orders	10	1623	1436	2361	266,05	0,62
TW_SEL_OrdPar_5	SELECT	OrdersPartitioned	10	40	27	116	25,54	24,57

Źródło: opracowanie własne.

Tabela 4.20 Testy wydajnościowe – PostgreSQL – zapytania INSERT

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TW_INS_Postgres_Ord_1	INSERT	Orders	10	1455	1325	1686	107,05	0,69
TW_INS_Postgres_OrdPar_1	INSERT	OrdersPartitioned	10	1034	730	3095	691,61	0,97

Źródło: opracowanie własne.

Tabela 4.21 Testy wydajnościowe – PostgreSQL – zapytania UPDATE

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TW_UPD_Ord_1	UPDATE	Orders	10	7145	4086	21029	4863,90	0,14
TW_UPD_OrdPar_1	UPDATE	OrdersPartitioned	10	875	649	1048	96,15	1,14

Źródło: opracowanie własne.

W kolejnym etapie przedstawiono rezultaty testów obciążeniowych dla PostgreSQL, które miały na celu zbadanie, jak baza danych zachowuje się przy jednoczesnym dostępie wielu użytkowników. Testy przeprowadzono za pomocą Apache JMeter, uruchamiając 100 wątków w przedziale czasowym wynoszącym 10 sekund, co umożliwiło stopniowe zainicjowanie obciążenia. Poniższe tabele 4.22, 4.23 i 4.24 przedstawiają te same miary, co w testach wydajnościowych. Jednak przepustowość należy rozpatrywać jako efekt równoległego przetwarzania żądań przez wielu użytkowników.

Tabela 4.22 Testy obciążeniowe – PostgreSQL – zapytania SELECT

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TO_SEL_Ord_1	SELECT	Orders	100	40294	24	52533	17546,74	1,86
TO_SEL_OrdPar_1	SELECT	OrdersPartitioned	100	75	5	213	19,12	10,09
TO_SEL_Ord_2	SELECT	Orders	100	232464	227053	238751	3118,51	0,21
TO_SEL_OrdPar_2	SELECT	OrdersPartitioned	100	2679	1283	3765	738,00	4,10

Źródło: opracowanie własne.

Tabela 4.23 Testy obciążeniowe – PostgreSQL – zapytania INSERT

ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TO_INS_Postgres_Ord_1	INSERT	Orders	100	902	4	2310	730,09	10,10
TO_INS_Postgres_OrdPar_1	INSERT	OrdersPartitioned	100	6877	8	11309	3169,68	5,54

Źródło: opracowanie własne.

Tabela 4.24 Testy obciążeniowe – PostgreSQL – zapytania UPDATE

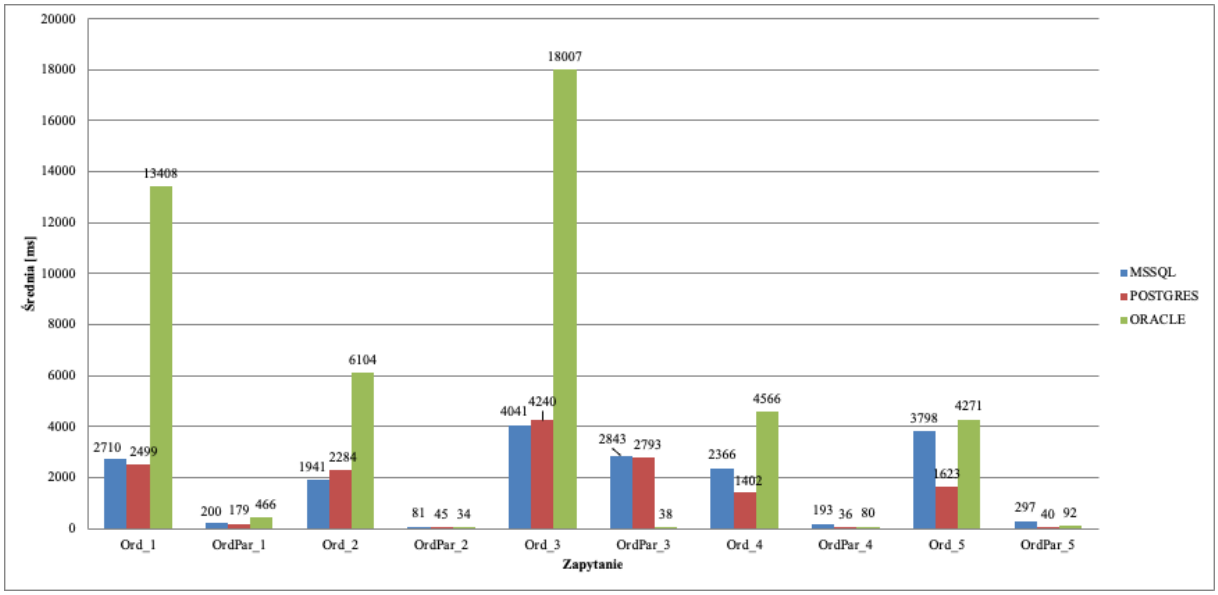
ID zapytania	Typ	Tabela	Liczba próbek	Średnia [ms]	Minimum [ms]	Maksimum [ms]	Odch. std.	Przepustowość [zapytania/s]
TO_UPD_Ord_1	UPDATE	Orders	100	22178	5	61599	18153,20	1,55
TO_UPD_OrdPar_1	UPDATE	OrdersPartitioned	100	15701	5	56668	18719,60	1,51

Źródło: opracowanie własne.

5. Porównanie wydajności systemów baz danych

W tym rozdziale dokonano szczegółowego omówienia wyników uzyskanych podczas przeprowadzonych testów. Celem analiz było porównanie wydajności wykonanych zapytań w trzech systemach baz danych: Microsoft SQL Server, PostgreSQL oraz Oracle Database. Badania zostały przeprowadzone zgodnie z przyjętą metodologią. Natomiast dane pomiarowe przedstawiono w formie wykresów słupkowych.

Na początku przedstawione zostały rezultaty testów wydajnościowych na rysunkach 5.1, 5.2, 5.3 i 5.4. Obejmują one pomiary średnich czasów wykonania zapytań typu SELECT, UPDATE oraz INSERT, a także przepustowości zapytań odczytujących. Potem omówione zostały wyniki testów obciążeniowych, pozwalających na ocenę zachowania systemów przy zwiększonej liczbie równoległych użytkowników. W analizie przyjęto regułę, w której oznaczenia Ord_i odnoszą się do zapytań wykonywanych na tabeli niepartycjonowanej. OrdPar_i dotyczy zapytań uruchamianych na tabeli partycjonowanej. Co więcej indeks „i" wskazuje numer kolejnego zapytania testowego odnoszącego się do tabeli 4.5 dla testów wydajnościowych i do tabeli 4.6 dla testów obciążeniowych.



Rysunek 5.1 Testy wydajnościowe – średnie czasy zapytań SELECT w MSSQL vs Postgres vs Oracle

Źródło: opracowanie własne.

Na rysunku 5.1 przedstawiono średnie czasy wykonania pięciu zapytań SELECT dla tabel niepartycjonowanych i partycjonowanych w trzech systemach: MSSQL 2022, PostgreSQL 17.2 oraz Oracle Database 23c Free. W testach bez partycjonowania najczęściej najkrótsze czasy uzyskiwał PostgreSQL, który w zapytaniach Ord_1, Ord_4 i Ord_5 osiągnął odpowiednio

około 2,50 s, 1,40 s i 1,62 s, podczas gdy MSSQL w tych samych przypadkach notował około 2,71 s, 2,37 s i 3,80 s. Oracle wypadł wyraźnie gorzej. W Ord_1 czas sięgał około 13,41 s, w Ord_2 około 6,10 s, a w Ord_3 dochodził do około 18,01 s. Należy podkreślić, że wyniki Oracle były bezpośrednio powiązane z ograniczeniami wersji Free, w której pamięć SGA + PGA jest ograniczona do maksymalnie 2 GB, a mechanizm Parallel Execution jest wyłączony lub mocno ograniczony. W praktyce oznacza to brak możliwości wykorzystania równoległego skanowania segmentów (tak jak w MSSQL czy PostgreSQL), co w przypadku dużych tabel znacząco wydłuża czas wykonania zapytań [28].

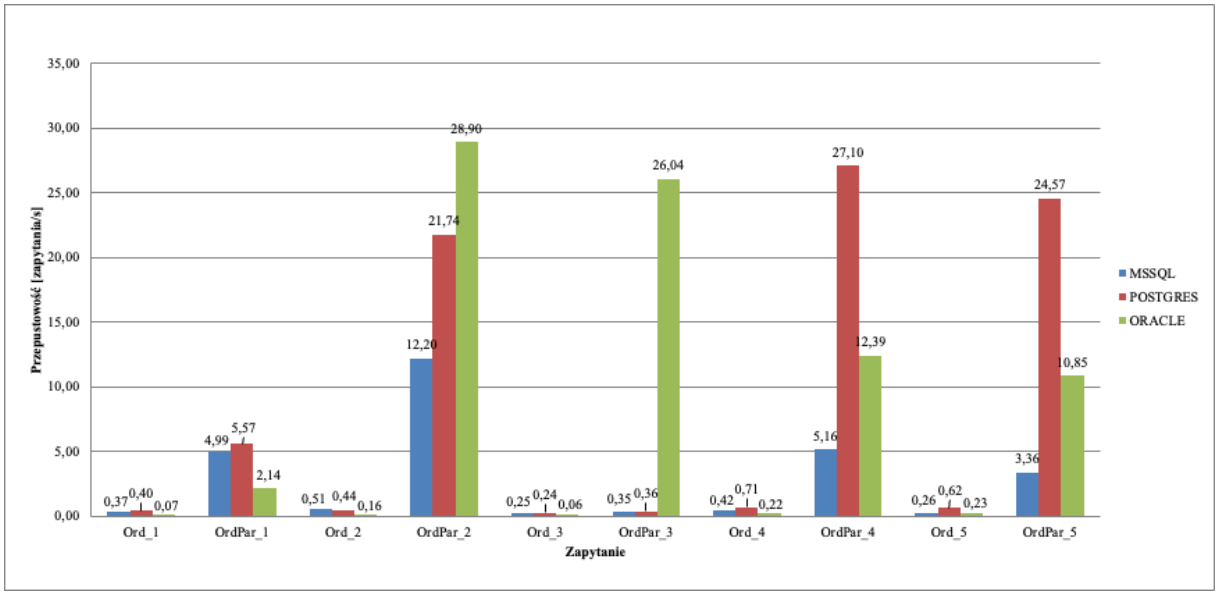
Zastosowanie partycjonowania znacznie wpływa na uzyskane wyniki i w każdym z systemów prowadzi do skrócenia czasu odpowiedzi. W OrdPar_1 czasy spadają do 200 ms w MSSQL, 179 ms w PostgreSQL i 466 ms w Oracle. Wskazuje to na skuteczną eliminację nieistotnych partycji oraz operowanie na mniejszych fragmentach danych. Zastosowane warunki filtrujące obejmują jedynie kolumny partycjonujące tzn. zakres dat ograniczony do roku 2016 oraz kraj wysyłki równy Australia (w przypadku MSSQL odpowiadające wartości kolumny obliczeniowej PartitionKey = 'A_2016'). Dzięki temu optymalizator mógł poprawnie zawęzić przetwarzanie do jednej partycji, co bezpośrednio przełożyło się na bardzo krótkie czasy odpowiedzi.

W OrdPar_2 najlepszy wynik uzyskuje Oracle z czasem 34 ms, wyprzedzając PostgreSQL z 45 ms i MSSQL z 81 ms. Warunki filtrowania ograniczały zakres danych do lat 2017–2019 oraz kraju Australia. W schemacie partycjonowania odpowiadało to trzem partycjom. Dodatkowo zastosowano operację grupowania po roku, która została wykonana już na ograniczonym zbiorze danych i w dużym stopniu przyczyniła się do skrócenia czasu odpowiedzi.

Największą zmianę widać w OrdPar_3, gdzie w MSSQL i PostgreSQL poprawa jest umiarkowana odpowiednio z około 4,04 s do 2,84 s oraz z 4,24 s do około 2,79 s. Natomiast w Oracle czas ulega redukcji z około 18,01 s do aż 38 ms. W tym przypadku warunek filtrujący obejmował jedynie kolumnę ShipCountry = Australia i bez zawężenia do konkretnego zakresu dat. Efektem była konieczność przeszukania wszystkich partycji odpowiadających kolejnym latom. W MSSQL zastosowano filtrację w postaci PartitionKey LIKE 'A_%'. Dlatego w MSSQL i PostgreSQL uzyskano jedynie niewielką poprawę. Natomiast w Oracle optymalizator był w stanie efektywniej wykorzystać klucz partycjonowania i ograniczyć koszty dostępu. Pozwoliło to uzyskać bardzo krótki czas odpowiedzi dla tego systemu.

W zapytaniach OrdPar_4 i OrdPar_5 wszystkie trzy systemy znacząco poprawiają swoje wyniki, a najkrótsze czasy uzyskuje PostgreSQL z odpowiednio 36 ms i 40 ms. Oracle poradził sobie równie dobrze z wynikami 80 ms i 92 ms. MSSQL osiąga czasy 193 ms i 297 ms.

W przypadku zapytania OrdPar_4 we wszystkich bazach wprowadzono dodatkową filtrację po wartości kosztu dostawy (Freight > 100). Dla PostgreSQL i Oracle zakres danych został zawężony przez warunki na OrderDate i ShipCountry. W MSSQL wykorzystano filtrację po kolumnie obliczeniowej PartitionKey = S_2020 wraz z warunkiem kraju, co pozwoliło na wskazanie właściwej partycji. W zapytaniu OrdPar_5 dostęp ograniczał się do jednej partycji odpowiadającej zamówieniom z roku 2021 w Australii, jednak dodatkowym elementem było złączenie z tabelą Customers. Zwiększyło to trochę koszt wykonania zapytania. Mimo to, dzięki precyzyjnemu filtrowaniu po kolumnach partycjonujących, wszystkie systemy osiągnęły bardzo krótkie czasy realizacji. Szczegółowa analiza pokazuje, że skuteczność partycjonowania zależy przede wszystkim od tego, czy warunki zapytania odwołują się wprost do kolumny będącej kluczem partycjonowania. W takich przypadkach wszystkie trzy systemy wielokrotnie skracają czasy odpowiedzi zapytań. Wyniki zmniejszone są nawet do wartości rzędu milisekund. Jednak, gdy zapytanie obejmuje wiele partycji albo warunek utrudnia ich odrzucenie, zysk maleje znacznie. W Oracle partycjonowanie pozwoliło całkowicie zniwelować ograniczenia środowiskowe tej wersji serwera, a PostgreSQL po podziale danych najczęściej osiągał najkrótsze czasy wykonania.



Rysunek 5.2 Testy wydajnościowe – przepustowość zapytań SELECT w MSSQL vs Postgres vs Oracle

Źródło: opracowanie własne.

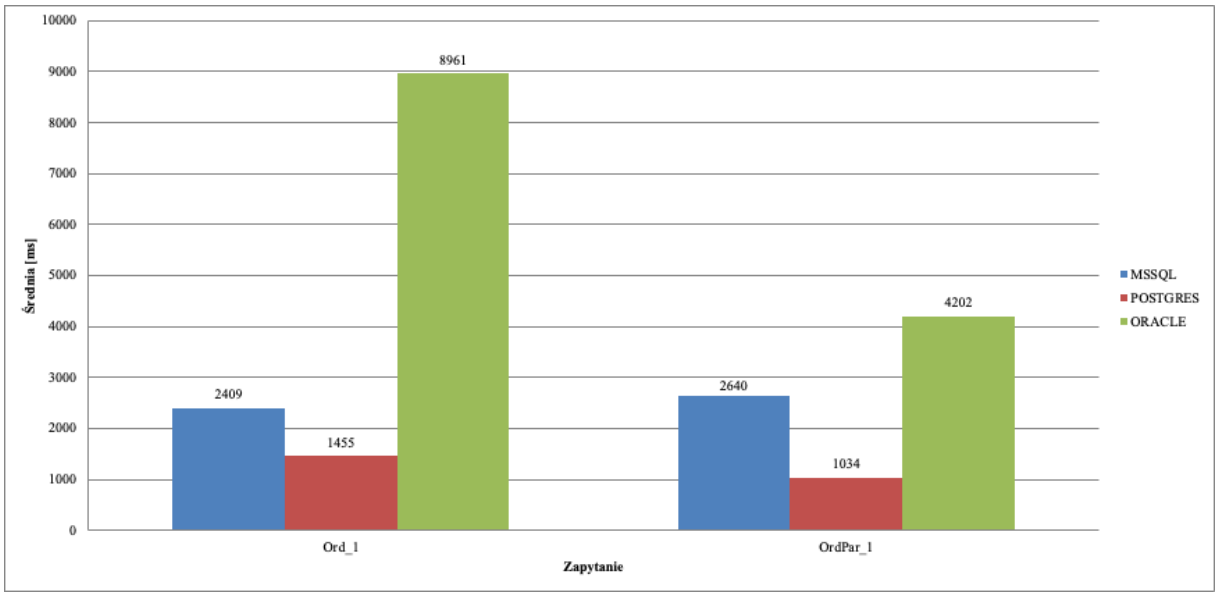
Drugi rysunek 5.2 przedstawia przepustowość zapytań SELECT, czyli liczbę zapytań obsłużonych w jednostce czasu. Dla tabel niepartycjonowanych wyniki przepustowości były bardzo niskie, poniżej 1 zapytania na sekundę. W przypadku zapytań Ord_1, Ord_2 czy Ord_3

przepustowość wahała się od 0,06 do 0,51. Wynikało to bezpośrednio z długich czasów przetwarzania.

Zastosowanie partycjonowania wyraźnie poprawiło sytuację. W przypadku zapytań OrdPar_1, OrdPar_2, OrdPar_4 i OrdPar_5 przepustowość wzrosła wielokrotnie, szczególnie w systemach PostgreSQL i Oracle, które osiągnęły wartości rzędu kilkunastu do nawet niemal trzydziestu zapytań na sekundę odpowiednio 21,74 i 28,90 dla OrdPar_2 oraz 27,10 i 12,39 dla OrdPar_4. MSSQL również poprawił wyniki, choć jego wzrost był mniejszy i zwykle mieścił się w przedziale od kilku do pięciu zapytań na sekundę.

Na szczególną uwagę zasługuje zapytanie OrdPar_3, w którym w PostgreSQL i MSSQL przepustowość wyniosła około 0,35 zapytania na sekundę. Te wartości prawie nie różnią się od wartości uzyskanych dla tabel niepartycjonowanych. Wynik ten wskazuje, że filtracja zastosowana w tym zapytaniu nie była bezpośrednio zgodna z kluczem partycjonowania, bo obejmowała jedynie kolumnę ShipCountry, a optymalizator w obu systemach musiał przeszukać wiele partycji. Natomiast dla Oracle przepustowość wzrosła do ponad 26 zapytań na sekundę. Mówi nam to o efektywnym wykorzystaniu klucza partycjonowania przez optymalizator, tak jak to miało miejsce wcześniej przy analizie średnich czasów wykonania zapytań na rysunku 5.1.

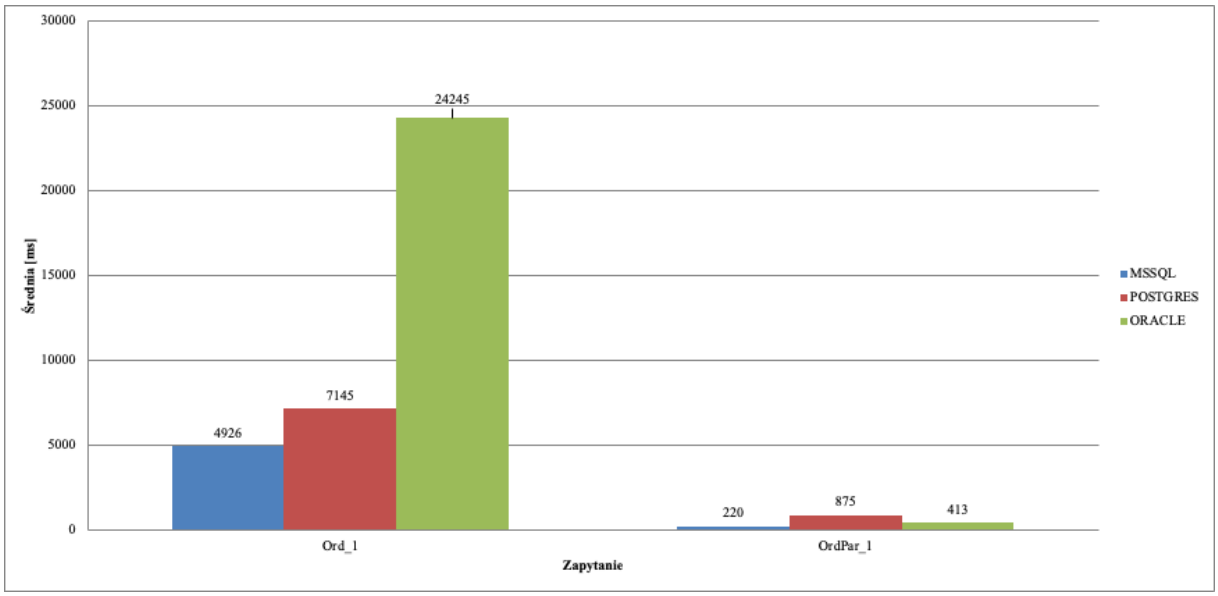
Warto również zwrócić uwagę na wartości odchylenia standardowego widoczne w tabelach 4.7, 4.13 i 4.19, które wskazują na stabilność czasów wykonania zapytań. W wielu przypadkach, szczególnie przy tabelach partycjonowanych, odchylenia były niewielkie. Dla zapytań OrdPar_2 w PostgreSQL wyszło 52 ms, a w OrdPar_5 w Oracle uzyskano 46 ms. Potwierdza to oczywiście powtarzalność działania optymalizatora. Z kolei przy niepartycjonowanych tabelach obserwowano znacznie wyższe odchylenia. Dla zapytania Ord_2 w Oracle uzyskano ponad 6315 ms. Może to wynikać z czynników zewnętrznych, takich jak obciążenie środowiska testowego, mechanizmy buforowania czy zarządzanie pamięcią, a nie z braku stabilności systemu.



Rysunek 5.3 Testy wydajnościowe – średnie czasy zapytań INSERT w MSSQL vs Postgres vs Oracle

Źródło: opracowanie własne.

Na ilustracji 5.3 przedstawiono średnie czasy zapytań typu INSERT, które pokazują istotne różnice pomiędzy poszczególnymi systemami bazodanowymi. W każdym przypadku dane ładowane były dziesięciokrotnie, a pojedynczy zestaw obejmował 50 tys. rekordów. W przypadku tabeli niepartycjonowanej najlepszy wynik uzyskał PostgreSQL z czasem 1455 ms, podczas gdy MSSQL potrzebował 2409 ms, a Oracle 23c Free aż 8961 ms. Po zastosowaniu partycjonowania czasy wykonania w MSSQL nieznacznie wzrosły do 2540 ms. W PostgreSQL uległy one dalszej poprawie i spadły do 1034 ms, co czyni go najbardziej efektywnym systemem w tym scenariuszu. Największą względną zmianę zauważono w Oracle, gdzie średni czas wstawiania danych skrócił się ponad dwukrotnie – z 8961 ms do 4202 ms. Wynik ten jest efektem tego, że w wersji bez partycjonowania Oracle Free realizował zapis do dużego segmentu tabeli przy ograniczonej pamięci SGA i PGA (łącznie maksymalnie 2 GB) oraz bez wsparcia dla równoległego DML. Znacząco wydłużało to czas przetwarzania. Dopiero rozdzielenie danych na mniejsze segmenty i lokalne indeksy w tabeli partycjonowanej pozwoliło uzyskać zauważalną poprawę. Należy jednak podkreślić, że mimo tej optymalizacji Oracle nadal radził sobie wyraźnie gorzej niż MSSQL i PostgreSQL, które zachowały krótsze czasy wstawiania danych.

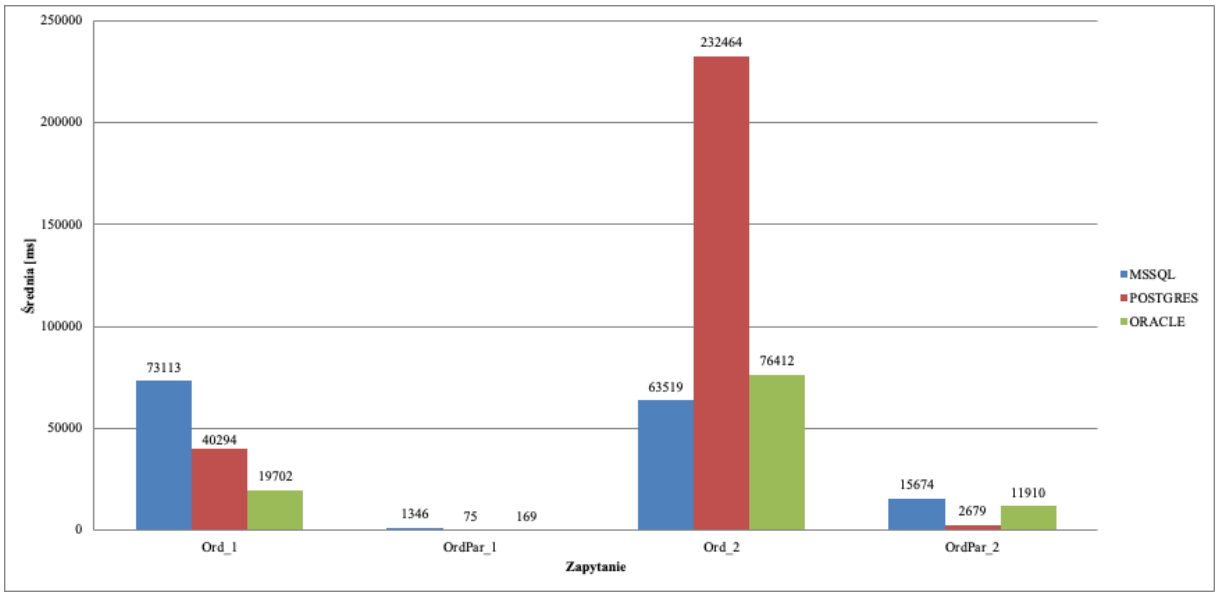


Rysunek 5.4 Testy wydajnościowe – średnie czasy zapytań UPDATE w MSSQL vs Postgres vs Oracle

Źródło: opracowanie własne.

Na rysunku 5.4 przedstawiono średnie czasy zapytań typu UPDATE. W tym przypadku zastosowanie partycjonowania przyniosło bardzo wyraźne korzyści we wszystkich analizowanych systemach. Dla tabel niepartycjonowanych czas aktualizacji wynosił 4926 ms w MSSQL, 7145 ms w PostgreSQL oraz aż 24 245 ms w Oracle. To wskazuje, że operacje modyfikacji na dużych tabelach były szczególnie kosztowne. Dla tabel z partycjonowaniem wartości te uległy redukcji odpowiednio do 220 ms w MSSQL, 875 ms w PostgreSQL i 413 ms w Oracle. Tak znaczna poprawa wynika z faktu, że warunki aktualizacji zostały powiązane bezpośrednio z kluczem partycjonowania i pozwoliło to optymalizatorom zawęzić operację do jednej partycji, zamiast skanować całą tabelę. W efekcie aktualizacja obejmowała niewielki fragment danych, a jej czas wykonania zmniejszył się nawet kilkudziesięciokrotnie. Warto zauważyć, że Oracle, mimo bardzo długiego czasu w wariancie zapytania z niepartycjonowaniem, po zastosowaniu partycjonowania uzyskał jeden z najlepszych wyników a dokładnie 413 ms. Potwierdza to oczywiście, że poprawnie zaprojektowane partycjonowanie może w ogromny sposób niwelować ograniczenia środowiskowe tego serwera. MSSQL osiągnął najkrótszy czas równy 220 ms, natomiast PostgreSQL, mimo że poprawa była wyraźna, utrzymał wynik wyższy równy 875 ms. Uzyskane rezultaty pokazują, że operacje aktualizacji należą do tych, które w największym stopniu zyskują na zastosowaniu partycjonowania.

Na poniższych rysunkach 5.5, 5.6, 5.7 i 5.8 zostały przedstawione wyniki testów obciążeniowych, w których analizowano zachowanie baz danych przy liczbie 100 równoległych użytkowników. Te analizy umożliwiają zrozumienie, jak systemy funkcjonują w sytuacjach z większą rywalizacją o zasoby sprzętowe. Umożliwia to na lepsze odwzorowanie rzeczywistych warunków działania aplikacji.



Rysunek 5.5 Testy obciążeniowe – średnie czasy zapytań SELECT w MSSQL vs Postgres vs Oracle

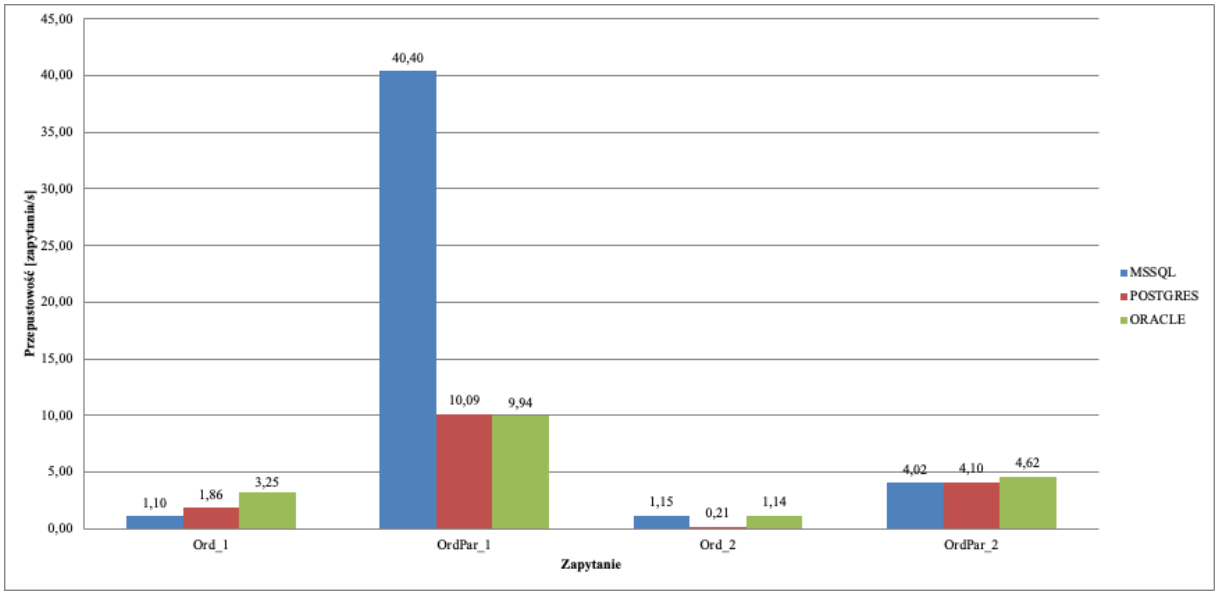
Źródło: opracowanie własne.

Na rysunku 5.5 zaprezentowano średnie czasy wykonywania zapytań SELECT w teście obciążeniowym, w którym jednocześnie pracowało wielu użytkowników. W porównaniu do testów wydajnościowych czasy realizacji bardzo wzrosły. Widać, że znacząco obciąża to system, gdy wzrasta liczba równoległych zapytań. W przypadku zapytania Ord_1 średni czas wyniósł 73113 ms w MSSQL, 40294 ms w PostgreSQL i 19702 ms w Oracle. Ponadto rozrzut wyników był znaczący. Odchylenie standardowe zawarte w tabelach 4.10, 4.16 i 4.22 wyniosło około 7665 ms w MSSQL, 5444 ms w Oracle i aż 17546 ms PostgreSQL. Wskazuje to na zdecydowanie mniejszą stabilność czasów odpowiedzi w tych systemach przy dużym obciążeniu.

Jeszcze większe różnice odnotowano dla zapytania Ord_2, w którym brak warunku z ShipCountry powodował przetwarzanie większego zakresu danych. Tutaj PostgreSQL osiągnął średni czas aż 232464 ms, co jest wartością wielokrotnie wyższą niż w teście wydajnościowym. Warto dodać, że odchylenie standardowe dla tego zapytania wyniosło ponad

3118 ms, co potwierdzało dużą zmienność wyników przy wysokim obciążeniu. MSSQL i Oracle radziły sobie lepiej, choć również wykazały wzrosty. Średnie czasy wynosiły dla nich odpowiednio 63519 ms i 76412 ms. Tak duża rozbieżność pomiędzy PostgreSQL a pozostałymi systemami sugeruje, że brak dopasowania zapytania do klucza partycjonowania w tym silniku przy dużej liczbie równoległych operacji generował wyjątkowo duże opóźnienia.

Po wprowadzeniu partycjonowania sytuacja znacząco się poprawiła. W zapytaniach OrdPar_1 i OrdPar_2 czasy realizacji spadły do wartości z zakresu 75 – 15700 ms. W szczególności w przypadku OrdPar_1, gdzie warunki odwoływały się bezpośrednio do kolumn partycjonujących (OrderDate oraz ShipCountry albo kolumny obliczeniowej PartitionKey). Optymalizatory baz danych, w każdym z systemów mogły skutecznie ograniczyć przetwarzanie do jednej partycji. To przełożyło się na czasy wynoszące zaledwie 1346 ms w MSSQL, 75 ms w PostgreSQL i 169 ms w Oracle, a także bardzo niskie odchylenia standardowe rzędu kilkunastu – kilkuset ms. W przypadku OrdPar_2, które obejmowało zakres dat dla roku 2008 i wymagało przetwarzania wielu partycji. Również tutaj poprawa była zauważalna choć w mniejszym stopniu. Średnie czasy spadły do 15674 ms w MSSQL, 2679 ms w PostgreSQL i 11910 ms w Oracle.



Rysunek 5.6 Testy obciążeniowe – przepustowość zapytań SELECT w MSSQL vs Postgres vs Oracle

Źródło: opracowanie własne.

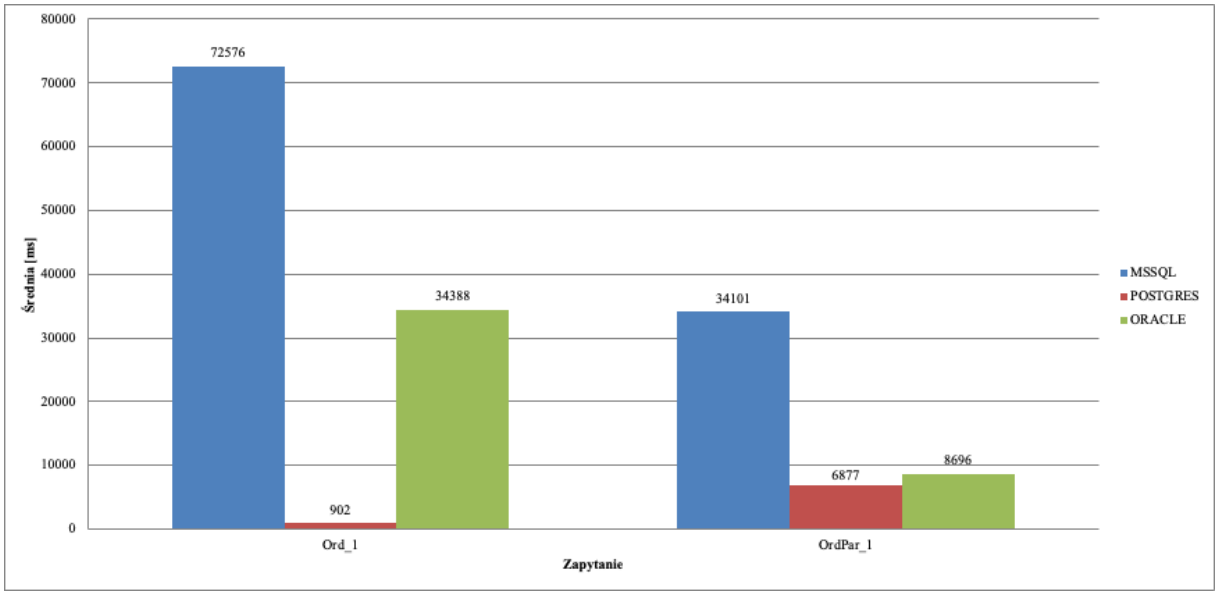
Na rysunku 5.6 przedstawiono przepustowość zapytań SELECT w warunkach dużego obciążenia, czyli liczbę obsłużonych zapytań na sekundę. Zauważalnie niższe są wartości niż w testach wydajnościowych. Jest to oczywiście naturalnym skutkiem jednoczesnej pracy wielu

użytkowników. Jednocześnie wyniki pokazują, że partycjonowanie poprawia zdolność systemów do obsługi równoległych zapytań.

Przepustowość w zapytaniu Ord_1 utrzymywała się na bardzo niskim poziomie od 1,10 do 3,25 zapytania/s. Było to wynikiem długich czasów odpowiedzi i dużego zakresu danych. Wartości te wzrosły wielokrotnie po zastosowaniu partycjonowania w zapytaniu OrdPar_1. Najlepszy rezultat uzyskał MSSQL z około 40 zapytaniami na sekundę. PostgreSQL i Oracle osiągnęły odpowiednio około 10 zapytań na sekundę. Stanowiło to również istotną poprawę względem zapytań z niepartycjonowaną tabelą, ale jednak nie na takim poziomie jak w MSSQL.

Dla zapytania Ord_2 przepustowość była niska we wszystkich systemach i spadła nawet poniżej 1 zapytania na sekundę w PostgreSQL. Brak warunku ShipCountry wymuszał dostęp do wielu partycji i tym samym zwiększał koszty przetwarzania przy równoległym obciążeniu.

Dopiero w przypadku OrdPar_2, gdzie zapytania mogły być częściowo zoptymalizowane przez ograniczenie zakresu partycji, przepustowość wzrosła do 4–5 zapytań na sekundę. Choć nie były to wartości tak wysokie jak w pierwszym zestawie, to i tak stanowiły wielokrotną poprawę w stosunku do przypadku bez partycjonowania.



Rysunek 5.7 Testy obciążeniowe – średnie czasy zapytań INSERT w MSSQL vs Postgres vs Oracle

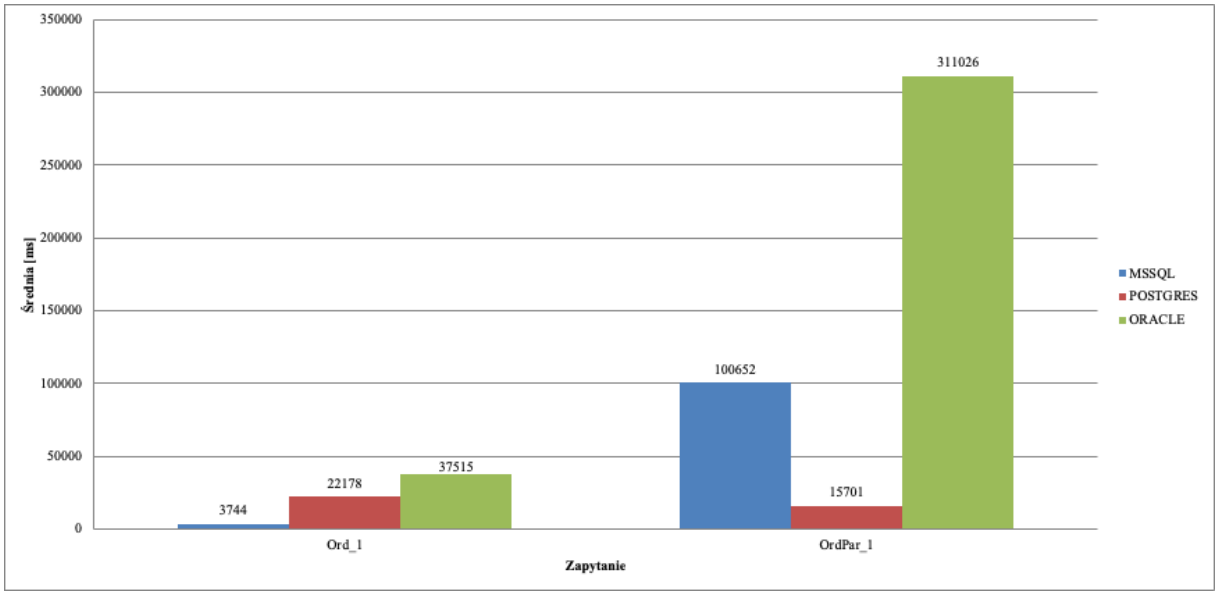
Źródło: opracowanie własne.

Na powyższym rysunku 5.7 widać średnie czasy zapytań typu INSERT dla testów obciążeniowych. Warto zaznaczyć, że operacje INSERT polegały na ładowaniu partii 5000 rekordów do tabel. Dla zapytań w tabeli niepartycjonowanej najdłuższy czas osiągnął MSSQL około 72 tys. ms, a także Oracle 34 tys. ms, podczas gdy PostgreSQL poradził sobie znacznie

lepiej, uzyskując zaledwie 902 ms. Po zastosowaniu partycjonowania wyniki uległy poprawie prawie dla wszystkich systemów. W MSSQL czas spadł do 34 tys. ms, czyli niemal o połowę w stosunku do tabeli niepartycjonowanej, w Oracle obniżył się ponad 4 krotnie do około 8,6 tys. ms. Natomiast w PostgreSQL średni czas w tabeli partycjonowanej wzrósł do około 6,8 tys. ms.

Analiza przepustowości zawarta w tabelach 4.11, 4.17 i 4.23 potwierdza jak najbardziej te wnioski. W przypadku tabel niepartycjonowanych PostgreSQL osiągnął najwyższy wynik z 10,10 zapytań na sekundę. MSSQL i Oracle były znacznie wolniejsze odpowiednio 1,11 i 2,14 zapytań na sekundę. Przepustowość w Oracle po zastosowaniu partycjonowania poprawiła się do 4,76 zapytania/s, a w MSSQL wzrosła nieznacznie do 1,76 zapytania/s. Natomiast w PostgreSQL spadła do 5,54 zapytania/s. Pokazuje to, że mimo lepszych wyników w wariancie bez partycjonowania, jego efektywność spadała, gdy dane były rozdzielane na partycje.

Podsumowując, testy obciążeniowe dla operacji INSERT ukazały, że PostgreSQL najlepiej radzi sobie z masowym równoległym wstawianiem rekordów w tabelach niepartycjonowanych, zapewniając tym samym krótkie czasy, jak i wysoką stabilność. Oracle i MSSQL w wariancie niepartycjonowanym osiągały znacznie gorsze wyniki, ale po użyciu partycjonowania ich wydajność uległa wyraźnej poprawie. Oracle, redukując średni czas ponad czterokrotnie, pokazał duże korzyści z zastosowania tej techniki.



Rysunek 5.8 Testy obciążeniowe – średnie czasy zapytań UPDATE w MSSQL vs Postgres vs Oracle

Źródło: opracowanie własne.

Na powyższym rysunku 5.8 widać średnie czasy zapytań typu UPDATE dla testów obciążeniowych. Dla zapytań w tabeli niepartycjonowanej najkrótszy czas uzyskał MSSQL około 3,7tys ms, PostgreSQL potrzebował więcej czasu około 22 tys. ms. Natomiast najdłużej trwały aktualizacje w Oracle, gdzie czas sięgał aż 37 tys. ms. Po włączeniu partycjonowania sytuacja uległa zmianie. PostgreSQL poradził sobie jako jedyny lepiej niż wcześniej, skracając czas do około 15,7 tys. ms. W MSSQL i Oracle średnie czasy drastycznie wzrosły. W MSSQL przekroczyły 100 tys. ms, a w Oracle sięgnęły aż ponad 311 tys. ms. Przy wielu równoległych aktualizacjach pokazuje to, że partycjonowanie nie zawsze przynosi korzyści.

Porównanie wyników testów wydajnościowych (rysunek 5.4) i obciążeniowych (rysunek 5.8) demonstruje wyraźny kontrast. W scenariuszu pojedynczych zapytań, czyli testów wydajnościowych wszystkie trzy systemy po zastosowaniu partycjonowania osiągnęły bardzo krótkie czasy aktualizacji 220 ms w MSSQL, 875 ms w PostgreSQL i 413 ms w Oracle. Natomiast w teście wielu równoległych użytkowników tendencja się odwróciła. W MSSQL i Oracle czasy aktualizacji zasadniczo wzrosły, a jedynie PostgreSQL utrzymał przewagę nad wariantem niepartycjonowanym. Różnice te wynikają z odmiennych mechanizmów kontroli współbieżności. PostgreSQL wykorzystuje podejście MVCC (ang. Multi-Version Concurrency Control), które pozwala uniknąć klasycznych blokad podczas modyfikacji danych, dzięki czemu operacje aktualizacji wykonywane są sprawniej przy dużej liczbie transakcji. Natomiast w MSSQL i Oracle zastosowane mechanizmy blokad powodują opóźnienia, co w tym scenariuszu dla równoległych zapytań istotnie wydłuża czas przetwarzania. Obrazuje to, że korzyści z partycjonowania są silnie zależne od charakteru pracy. Dla pojedynczych aktualizacji daje ono duże przyspieszenie, ale przy wielu równoległych transakcjach może generować dodatkowe koszty.

6. Podsumowanie

Celem niniejszej pracy było zbadanie jak mechanizm partycjonowania wpływa na wydajność systemów bazodanowych. Analizowano 3 różne silniki bazodanowe: Microsoft SQL Server, PostgreSQL oraz Oracle Database. Badania były oparte na zmodyfikowanej wersji schematu Northwind, a eksperymenty obejmowały testy wydajnościowe i obciążeniowe. W ramach tych testów mierzono średnie czasy wykonania zapytań, przepustowość oraz odchylenie standardowe. Uwzględniono trzy podstawowe typy operacji: SELECT, INSERT i UPDATE.

Przeprowadzone eksperymenty potwierdziły, że partycjonowanie jest skutecznym mechanizm optymalizacji zapytań, szczególnie w momentach, w których warunki filtrujące odpowiadają kluczowi partycjonowania. W testach wydajnościowych wszystkie systemy działały szybciej po zastosowaniu partycjonowania. Największe zyski pojawiły się w zapytaniach SELECT, gdzie czas odpowiedzi spadał z kilku sekund do zaledwie milisekund. W operacjach UPDATE poprawa była jeszcze bardziej wyraźna. Pojedyncze aktualizacje wykonywały się nawet kilkadziesiąt razy szybciej. Najlepsze wyniki w przypadku zapytań INSERT uzyskał PostgreSQL, ale również Oracle, mimo ograniczeń wersji Free, zyskał dzięki partycjonowaniu. Testy obciążeniowe pokazały, że efekty partycjonowania w dużym stopniu zależą od poziomu obciążenia i struktury zapytań. Dla zapytań SELECT przy wielu równoległych użytkownikach zastosowanie partycjonowania wyraźnie skracało czas odpowiedzi i zwiększało przepustowość, szczególnie na bazie MSSQL. Operacje INSERT także zyskały w Oracle i MSSQL, natomiast w PostgreSQL odnotowano gorsze rezultaty. Wskazuje to, że masowe ładowanie danych do tabel partycjonowanych może wiązać się z dodatkowymi kosztami. Największe różnice wystąpiły w zapytaniach UPDATE. Podczas gdy w testach wydajnościowych partycjonowanie przynosiło bardzo duże korzyści, to w zapytaniach aktualizacji przy dużym obciążeniu równoległym w MSSQL i Oracle prowadziło do ogromnego wydłużenia czasów. Tylko PostgreSQL utrzymał lepsze wyniki w przypadku z zastosowaniem partycjonowania.

Podsumowując, przeprowadzone badania potwierdziły, że partycjonowanie może znacząco poprawiać wydajność systemów bazodanowych. Jednak jego skuteczność zależy od wielu czynników takich jak: rodzaju zastosowanego systemu bazodanowego, struktury zapytań, poziomu obciążenia oraz użytej metody partycjonowania. Wyniki jasno wskazują, że prawidłowe zaprojektowanie schematu partycjonowania oraz dostosowanie zapytań do klucza partycjonującego są niezbędne dla osiągnięcia pozytywnych efektów.

Kierunki dalszych badań i możliwości optymalizacji obejmują między innymi zastosowanie dodatkowych metod, takich jak indeksy lokalne i globalne czy inne techniki partycjonowania, które mogą dodatkowo zwiększyć, usprawnić dostęp do danych. Istotnym obszarem dalszych badań jest także porównanie partycjonowania z innymi metodami skalowania wykorzystywanymi we współczesnych systemach baz danych. Przydatne byłoby zestawienie relacyjnych baz z partycjonowaniem z bazami rozproszonymi typu NoSQL, które wspierają poziome skalowanie i pracę w środowiskach o dużej dostępności. Pozwoliłoby to dokładniej określić, kiedy relacyjne systemy bazodanowe pozostają lepsze, a kiedy korzystniejsze są inne rozwiązania dla pracy z dużymi wolumenami danych.

7. Wykaz tabel

8. Wykaz rysunków

9. Bibliografia

[1] R. Elmasri i S. B. Navathe, Fundamentals of Database Systems, 7th Edition, 2016.

[2] „SQL Server Agent," [Online]. Available: https://learn.microsoft.com/pl-pl/ssms/agent/sql-server-agent.

[3] „Wyzwalacze," [Online]. Available: https://gakko.pjwstk.edu.pl/mat/118/lec/w14.html.

[4] „NoSQL Data Streaming," [Online]. Available: https://severalnines.com/blog/nosql-data-streaming-mongodb-kafka/.

[5] „Change Streams," [Online]. Available: https://www.mongodb.com/docs/manual/changeStreams/.

[6] A. Silberschatz, H. F. Korth i S. Sudarshan, Database System Concepts, 7th Edition, 2020.

[7] „Relacyjne i nierelacyjne bazy danych," [Online]. Available: https://www.ovhcloud.com/pl/learn/relational-vs-non-relational-databases/.

[8] „SQL podstawy," [Online]. Available: https://www.cognity.pl/blog-sql-podstawy-skladnia-i-zastosowanie.

[9] „MongoDB Database Manual," [Online]. Available: https://www.mongodb.com/docs/manual/.

[10] „Cassandra Documentation CQL," [Online]. Available: https://cassandra.apache.org/doc/4.0/cassandra/cql/.

[11] „Cypher Manual," [Online]. Available: https://neo4j.com/docs/cypher-manual/current/introduction/.

[12] „Redis Commands," [Online]. Available: https://redis.io/docs/latest/commands/.

[13] „Jak działa partycjonowanie danych w SQL?," [Online]. Available: https://programistajava.pl/2025/03/12/jak-dziala-partycjonowanie-danych-w-sql/.

[14] „Partition pruning," [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/21/vldbg/partition-pruning.html?utm_source.

[15] „Wskazówki dotyczące partycjonowania danych," [Online]. Available: https://learn.microsoft.com/pl-pl/azure/architecture/best-practices/data-partitioning.

[16] „partycjonowanie," [Online]. Available: https://blog.consdata.tech/2025/01/07/czy-wiesz-czym-jest-partycjonowanie.html.

[17] „Partitioned tables and indexes," [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/partitions/partitioned-tables-and-indexes?view=sql-server-ver17.

[18] „Specify computed columns in a table," [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/tables/specify-computed-columns-in-a-table?view=sql-server-ver17.

[19] „oracle.com," [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/23/vldbg/partition-availability.html#GUID-E677C85E-C5E3-4927-B3DF-684007A7B05D.

[20] „Partitioning Concepts," [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/23/vldbg/partition-concepts.html.

[21] „Partition-Wise Operations,” [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/19/vldbg/partition-wise-joins.html.

[22] „Postgres 17 Documentation,” [Online]. Available: https://www.postgresql.org/docs/17/ddl-partitioning.html.

[23] „PostgreSQL Partitioning,” [Online]. Available: https://stormatics.tech/blogs/improving-postgresql-performance-with-partitioning.

[24] „OpenSource postgres partitioning,” [Online]. Available: https://opensource-db.com/postgresql-partitioning-made-easy-features-benefits-and-tips/.

[25] S. P. Kane i K. Matthias, Docker niezawodne kontenery produkcyjne, Praktyczne zastosowania.

[26] „Testowanie wydajnościowe z Apache JMeter.,” [Online]. Available: https://programistajava.pl/2024/12/21/testowanie-wydajnosciowe-z-apache-jmeter/.

[27] „Northwind database,” [Online]. Available: https://dbdesc.com/output_samples/html_northwind.html.

[28] „Oracle Licensing Restrictions,” [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/23/xeinl/licensing-restrictions.html?utm_source.