Bazy Danych 2 – Lab 5

Trigger (wyzwalacz) jest obiektem bazy danych, który jest dołączony do tabeli. W wielu aspektach jest podobny do procedury składowanej. W rzeczywistości są często określane jako "specjalny rodzaj procedury składowanej". Główna różnica polega na tym, że ta pierwsza jest dołączona do tabeli i wykonywana tylko wtedy gdy wystąpią operacje takie jak: **INSERT**, **UPDATE** lub **DELETE**. Określasz działania modyfikujące, które powodują uruchomienie wyzwalacza.

Poniżej przedstawiono przykład wyzwalacza, który będzie wyświetlał bieżący czas systemowy w momencie dodania wiersza do tabeli do której tej wyzwalacz jest przypisany:

```
CREATE TABLE SampleDataTable(SampleID int IDENTITY, SampleName varchar(20))
GO
CREATE TRIGGER SampleDataTable_INSERT
ON SampleDataTable
FOR INSERT
AS
PRINT GETDATE()
GO
```

Wywołanie oraz rezultat:

```
INSERT SampleDataTable(SampleName) VALUES ('SampleData')

-- Aug 29 2017 10:31AM
```

Kiedy używać wyzwalaczy?

Zazwyczaj używamy ich tylko gdy trzeba wykonać określone działanie będące następstwem operacji takich jak dodanie, zmodyfikowanie czy usunięcie danych. Większość operacji manipulacji na danych przygotowywane jest w procedurach składowanych. Dzięki takiemu zabiegowi funkcjonalność wyzwalaczy może być zdefiniowana w tych procedurach. Dla przykładu, załóżmy, że chcemy wysłać wiadomość e-mail do managera sprzedaży, gdy do systemu zostało dodane zlecenie o dużym priorytecie. Kiedy takie zlecenie zostaje dodane do bazy danych **trigger** jest używany do określania priorytetu oraz wysłania wiadomości, kiedy kryteria są spełnione. Poniżej przedstawiam częściową implementację takiego zachowania:

```
CREATE TABLE Orders (OrdID INT, OrdPiority varchar(10))
GO
CREATE TRIGGER tr_Orders_Insert
ON Orders
FOR INSERT
AS
iF( SELECT COUNT(*) FROM inserted WHERE OrdPiority = 'High') = 1
  BEGIN
    PRINT 'Email Code Goes Here'
  END
GO
INSERT Orders (OrdPiority) VALUES ('High')
```

Oraz wynik dodania nowego pierwsza do tabeli **Orders**:

```
-- Email Code Goes Here
```

Kiedy jednak używana jest procedura składowana możemy przenieść do niej kod wyzwalacza:

```
CREATE PROCEDURE proc_Orders_INSERT @OrdPiority varchar(10)
AS
BEGIN TRANSACTION
  INSERT Orders (OrdPiority) VALUES (@OrdPiority)
  IF @@ERROR <> 0
    GOTO ErrorCode
  IF @OrdPiority = 'High'
    PRINT 'Email Code Goes Here'
  COMMIT TRANSACTION
ErrorCode:
  IF @@TRANCOUNT <> 0
    PRINT 'Error Code'
GO
```

Przyjrzymy się bardziej szczegółowo przykładowi wyzwalacza. Pierwszą rzeczą na którą należy zwrócić uwagę jest odwołanie **SELECT** do tabeli nazywanej **inserted**. Wyzwalacze korzystają z dwóch specjalnych tabel zwanych **inserted** oraz **deleted**. Pierwsza z nich odwołuje się do operacji **INSERT** zanim ta zostanie rzeczywiście wykonana. Usunięta tabela zawiera referencje do tabeli bazowej na której wykonujemy operację **DELETE**. Podobnie jak w poprzednim przypadku dotyczy to stanu sprzed operacji. W momencie przeprowadzenia operacji **UPDATE** obie z tych tabel są brane pod uwagę. Będąc bardziej precyzyjnym: nowe dane odnoszące się do operacji **UPDATE** odnoszą się do tabeli **inserted** a dane, które zostały zaktualizowane do danych zawartych w tabeli **deleted**.

W naszym przykładzie możemy zobaczyć, że dane dostaną dodane do tabeli. Instrukcja **IF** szuka wyniku równego 1. Oznacza to, że wyzwalacz zakłada, że do tabeli dodany jest tylko jeden wiersz. Jeżeli do tabeli zostanie dodany więcej niż jeden wiersz możesz pominać zamównie z priorytetem **High** ponieważ wyzwalacz uruchamia się tylko raz dla każdej skojarzonej instrukcji. Zdaje sobie sprawę, że może to być mylące dlatego spojrzmy na dwa przykłady. Poniżej możemy zobaczyć, że wyzwalacz uruchamia się dla każdej instrukcji **INSERT**:

```
INSERT Orders (OrdPiority) VALUES ('High')
INSERT Orders (OrdPiority) VALUES ('High')
-- Result:
--
-- Email Code Goes Here
-- Email Code Goes Here
```

Teraz w tabeli **Orders** mamy trzy wiersze z priorytetem **High**. Dodajmy teraz nowe wiersze w oparciu o bieżącą treść, aby pokazać jak zachowuje się wyzwalacz, gdy chcemy wykonać instrukcję składającą się z wielu poleceń:

```
INSERT INTO Orders (OrdPiority)
SELECT OrdPiority FROM Orders
```

Wiadomość **Email Code Goes Here** się nie wyświetliła miło, że to tabeli zostały dodane trzy nowe wierwsz z odpowiednim priorytetem. Wszystko dlatego, że nie zostały spełnione warunki instrukcji **IF**. Wyzwalacz uruchamia się tylko raz dla całego ciągu instrukcji dlatego wartość po zsumowaniu wynosiła 3 a nie oczekiwane 1. Poniżej przykład jak poradzić sobie z obsługą wielu instrukcji **INSERT** w jednym poleceniu:

```
ALTER TRIGGER tr_Orders_INSERT
ON Orders
FOR INSERT
AS
 IF EXISTS(SELECT * FROM inserted WHERE OrdPiority = 'High')
   BEGIN
     DECLARE @count tinyint
     SET @count = (SELECT COUNT(*) FROM inserted where OrdPiority = 'High')
     PRINT CAST(@count as varchar(3)) + ' row(s) with a priority of High were entered'
   END
GO
```

Możemy teraz przetestować nasz kod używając tego samego polecenia co w poprzednim przykładzie:

```
--6 row(s) with a priority of High were entered
```

Przykład

Osoby zaznajomione z tematem zarządzania witrynami sieciowymi wiedzą jak ważne jest sprawdzenie ruchu na stronie celem sprawdzenia, które obszary witryny są odwiedzane przez użytkowników. **IIS** posiada funkcje rejestrowania wielu atrybutów związanych z każdym użytkownikiem. Dla przykładu, za każdym razem kiedy użytkownik odwiedza daną witrynę i chce się zalogować. Domyślnie, dane te zapisywane są do pliku log ale można zmienić to zachowanie aby było zgodne ze standardem **ODBC**. Jest to interfejs pozwalający programom łączyć się z systemami zarządzającymi bazą danych.

To podejście jest używane dla jednego z klientów od pewnego czasu ponieważ chciał w prosty sposób śledzić każdy obszar swojej witryny. Główny obszar został zdefiniowany jako sekcje wymienione na głównym pasku nawigacyjnym, np. Strona główna, o nas, kontakt, etc. Celem było przygotowanie raportu, który wskazywałby miesięczne zestawienie odwiedzin każdego z głównych obszarów serwisu. Można się zastanawiać dlaczego wyzwalacz jest potrzebny do wdrożenia takiego rozwiązania. Przecież **SELECT** z klauzulą **WHERE** do filtrowania zakresu dat oraz **GROUP BY** do policzenia odwiedzin każdej z podkategorii powinien być wystarczający.

Powód dla którego zdecydowano się na użycie wyzwalacza był związany z niedopuszczalnym czasem przygotowania raportu. Nawet na stronach ze stosunkowo małych ruchem liczba wierszy związanych z odwiedzinami rośnie w zastraszającym tempie. Dla każdej strony odwiedzanej przez użytkownika należy liczyć jeden wiersz dodany do tabeli. Jeżeli strona zawiera dodatkowe odwołania, np. pliki graficzne, należy dodać kolejny wiersz.

Najważniejsze z naszego punktu widzenia jest to, że wygenerowania raportu z takiej ilości danych trwałoby niezwykle długo. Aby skrócić czas potrzebny do wykonania raportu zdecydowano się na użycie tabeli podsumowywującej do liczenia odsłon stron odwiedzanych przez różnych użytkowników. Dzięki temu rozwiązaniu mamy tabelę zawierającą jedynie osiem (liczba zależna od głównych kategorii na stronie) wierszy a generowanie raportu trwa mniej niż jedną sekundę.

Poniżej poglądowy przykład użycia wspomnianego wcześniej wyzwalacza okrojonego do operowania jedynie na dwóch głównych obszarach serwisu:

```
CREATE TABLE CustomLog(ClientHost varchar(255), LogTime datetime, Target varchar(255))
GO
CREATE TABLE LogSummary(LogSumCategory varchar(30), LogSumCount int)
GO
INSERT LogSummary VALUES('O nas',0)
INSERT LogSummary VALUES('Kontakt', 0)
```

CustomLog to główna tabela logowania a **LogSummary** to tabela z podsumowaniem. Dwia główne obszary serwisu to ‘O nas’ oraz ‘Kontakt’. Celem przygotowanego wyzwalacza jest aktualizacja wartości kolumny **LogSumCount** za każdym razem gdy użytkownik odwiedzi strony **About.aspx** oraz **Contact.aspx**:

```
CREATE TRIGGER tr_CustomLog_INSERT
ON CustomLog
FOR INSERT
AS
BEGIN
  IF EXISTS (SELECT * FROM inserted WHERE Target = 'About.aspx')
  BEGIN
    UPDATE LogSummary
    SET LogSumCount = (
      SELECT COUNT(*) FROM CustomLog WHERE Target = 'About.aspx'
    )
    WHERE LogSumCategory = 'O nas'; -- <- dopasuj wartość
  END

  IF EXISTS (SELECT * FROM inserted WHERE Target = 'Contact.aspx')
  BEGIN
    UPDATE LogSummary
    SET LogSumCount = (
      SELECT COUNT(*) FROM CustomLog WHERE Target = 'Contact.aspx'
    )
    WHERE LogSumCategory = 'Kontakt'; -- <- dopasuj wartość
  END
END;
```

Wyzwalacz będzie oczekiwał nowych wartości w tabeli ale również będzie sprawdzał przygotowany przez nas warunek, aby logować informację w odpowiednich polach. Poniżej przykład użycia tak przygotowanego wyzwalacza:

```
INSERT INTO CustomLog VALUES('111.111.111.111', '4/1/29 12:00:50', 'Default.aspx')
INSERT INTO CustomLog VALUES('111.111.111.111', '4/1/29 12:00:50', 'About.aspx')
INSERT INTO CustomLog VALUES('111.111.111.111', '4/1/29 12:00:50', 'About.aspx')
INSERT INTO CustomLog VALUES('111.111.111.111', '4/1/29 12:00:50', 'Contact.aspx')
INSERT INTO CustomLog VALUES('111.111.111.111', '4/1/29 12:00:50', 'Contact.aspx')
INSERT INTO CustomLog VALUES('111.111.111.111', '4/1/29 12:00:50', 'About.aspx')
INSERT INTO CustomLog VALUES('111.111.111.111', '4/1/29 12:00:50', 'Contact.aspx')
INSERT INTO CustomLog VALUES('111.111.111.111', '4/1/29 12:00:50', 'About.aspx')
```

Oraz wynik z tabeli **LogSummary**:

```
LogSummary                LogSumCount
------------------------- -----------
O nas                     4
Kontakt                   3
```

Takie rozwiązanie w dzisiejszych czasach nie jest najlepsze do monitorowania ruchu użytkowników.

Wyzwalacz UPDATE

Jest to wyzwalacz, który uaktywnia się na przypisanej tabeli po wykonaniu akcji **UPDATE**. Poniższy przykład jest oparty na tabeli używanej w pierwszej cześci artykułu:

```
CREATE TRIGGER tr_Order_UPDATE
ON Orders
AFTER UPDATE
AS
-- sprawdzamy czy prioryter został zmieniony
IF NOT UPDATE(OrdPiority)
   RETURN
-- Sprawdzenie czy priorytet został zmieniony na wysoki
IF EXISTS(
SELECT * FROM inserted i
JOIN deleted d ON i.OrdID = d.OrdID
WHERE d.OrdPiority <> 'High'
AND i.OrdPiority = 'High')
BEGIN
   DECLARE @Count tinyint
   SET @Count = (
   SELECT COUNT(*) FROM inserted i
   JOIN deleted d ON i.OrdID = d.OrdID
   WHERE d.OrdPiority <> 'High'
   AND d.OrdPiority = 'High')
   PRINT CAST(@Count as varchar(3))+' row(s) where changed to a priority of High'
   END
go
```

Wyzwalacz **INSERT** obserwuje zamówienia z priorytetem **High**. Wyzwalacz **UPDATE** obserwuje zlecenia w których priorytet został zmieniony na inny niż **High**.

Instrukcja **IF** sprawdza czy wartość **OrdPiority** została zmieniona. Jeżeli nie, możemy zaoszczędzić trochę czasu poprzez natychmiastowe opuszczenie wyzwalacza.

Tabela **deleted** zawiera wartości sprzed operacji **UPDATE** a tabela **inserted** zawiera nowe wartości. Kiedy dochodzi do połączenia table bardzo łatwo stwierdzić, kiedy doszło do zmiany priorytetu z **High** na inny.

Wyzwalacz DELETE

Jeżeli rozumiemy działanie wyzwalacza **UPDATE** nie będzie problemu ze zrozumieniem jak działa **DELETE**. Jest łatwy do zaimplementowania. W poniższym przykładzie można zobaczyć obliczanie wierszy w tabeli **deleted** - pozwala to sprawdzić ile rekordów było zapisanych z priorytetem **High**.

```
CREATE TRIGGER tr_Orders_DELETE
ON Orders
AFTER DELETE
AS
-- sprawdzamy czy zlecenie z priorytetem High zostało skasowane
IF EXISTS (SELECT * FROM deleted WHERE OrdPiority = 'High')
  BEGIN
    DECLARE @Count tinyint
    SET @Count = (SELECT COUNT(*) FROM deleted where OrdPiority = 'High')
    PRINT CAST(@Count as varchar(3))+' row(s) where deleted whose priority was High'
  END
GO
```

Wyzwalacz INSTEAD OF

Wyzwalacz **INSTEAD OF** jest dostępny od wersji **SQL Server 2000**. Powodem ich wprowadzenia było ułatwienie aktualizacji widoków. Zobaczymy jak działają te wyzwalacze ale nie będziemy się skupiać na aktualizowaniu widoków. Wzywalacz ten jest używany do wykonania czynności zamiast tej, która to wywołanie spowodowała. Załóżmy, że przygotowaliśmy wyzwalacz **INSTAED OF INSERT** zdefiniowany na tabeli i dochodzi do wywołania operacji **INSERT**. Wiersz nie zostaje dodany do tabeli ale kod wyzwalacza jest uruchamiany. Poniżej przykład takiego zachowania:

```
CREATE TRIGGER tr_Orders_InsteadOfINSERT
ON ORDERS
INSTEAD OF INSERT
AS
PRINT 'Widoki z aktualizacjami wprowadzają bałagan'
Go
```

Taka wiadomość jest wyświetalana przez operację **INSERT**. Należy jednak pamiętać, że wiersz nie jest dodany do tabeli:

```
INSERT INTO Orders(OrdPiority) VALUES ('Nie dodano')
-- Widoki z aktualizacjami wprowadzaja balagan
```

Podstawowe aspekty wydajnościowe zapytań SQL

Jak zrozumieć znaczenie optymalizacji zapytań SQL

Optymalizacja zapytań SQL to kluczowy element, który może znacząco wpłynąć na wydajność aplikacji bazodanowych. Aby lepiej zrozumieć jej znaczenie, warto przyjrzeć się kilku kluczowym aspektom.

- **Wydajność systemu**: Dobrze zoptymalizowane zapytania przyspieszają czas odpowiedzi systemu, co jest istotne w przypadku aplikacji wymagających szybkiego przetwarzania danych.
- **skalowalność**: W miarę wzrostu ilości danych, zoptymalizowane zapytania pozwalają na płynne działanie systemu, nawet gdy obciążenie rośnie.
- **zużycie zasobów**: Efektywne zapytania minimalizują obciążenie serwera baz danych, co przekłada się na oszczędność zasobów i kosztów operacyjnych.

Choć wiele z pozoru prostych zapytań wydaje się szybko się wykonywać, to czasami ich optymalizacja może przynieść zaskakujące rezultaty. Analiza planu wykonania zapytania to dobry sposób na identyfikację potencjalnych wąskich gardeł.

Kilka kluczowych technik, które mogą znacznie poprawić wydajność zapytań:

Technika	**Opis**
Indeksy	Pomoże w szybkim wyszukiwaniu danych w dużych tabelach.
Użycie JOIN zamiast subzapytania	Może znacznie poprawić wydajność w przypadku skomplikowanych zapytań.
Limitowanie wyników	Redukuje ilość przetwarzanych danych, co przyspiesza zapytania.

Najczęstsze problemy z wydajnością zapytań SQL

Optymalizacja wydajności zapytań SQL to kluczowy element zarządzania bazami danych, który wpływa na ogólną efektywność aplikacji. Wiele osób napotyka na typowe problemy, które mogą prowadzić do spowolnienia pracy systemu. Poniżej przedstawiono najczęstsze z nich:

- **Brak indeksów:** Wiele zapytań nie korzysta z indeksów, co znacznie wydłuża czas ich wykonania. indeksy przyspieszają dostęp do danych, zmniejszając ilość przeszukiwanego tekstu w tabelach.
- **nieprawidłowe użycie operacji JOIN:** Złożone zapytania wykorzystujące wiele JOINów mogą prowadzić do dużych obciążeń sieciowych i wydajnościowych.Korzystanie z odpowiednich aliasów oraz ograniczanie liczby łączonych tabel są kluczowe.
- **Nieoptymalne filtry:** Używanie zbyt ogólnych lub nieoptymalnych warunków WHERE sprawia, że zapytania przeszukują zbędne dane. często lepiej jest ograniczać wyniki już na etapie tworzenia zapytania.
- **Niedopasowanie typów danych:** Używanie różnych typów danych w zapytaniach może powodować niepotrzebne konwersje, co dodatkowo obciąża bazę danych. Ważne jest, aby typy danych były zgodne.

Inną kwestią są zbyt długie zapytania, które mogą być trudne w zarządzaniu i prowadzić do błędów. Warto więc stosować praktyki pomagające w ich podziale na mniejsze, bardziej zrozumiałe jednostki. Oto przykładowa tabela ilustrująca różnice w wydajności pomiędzy długimi a krótkimi zapytaniami:

Typ zapytania	**Czas wykonania (ms)**	**Obciążenie CPU (%)**
Długie zapytanie	1200	85
Krótkie zapytanie	300	30

Należy pamiętać, że **monitorowanie i analiza wydajności zapytań** to proces ciągły. Stosując narzędzia takie jak EXPLAIN w SQL, można lepiej zrozumieć, jak zapytania są wykonywane i w jakich miejscach występują potencjalne wąskie gardła. Regularne przeglądanie zapytań i ich optymalizacja przynosi długofalowe korzyści dla całej bazy danych.

Podstawowe techniki optymalizacji zapytań SQL

Wydajność zapytań SQL można znacznie poprawić poprzez zastosowanie kilku fundamentalnych technik optymalizacji. Poniżej przedstawiamy najważniejsze z nich:

- **Indeksy:** Stosowanie odpowiednich indeksów to jedna z najskuteczniejszych metod przyspieszania zapytań. Indeksy umożliwiają szybkie wyszukiwanie danych, co znacznie redukuje czas potrzebny na przetwarzanie zapytań.
- **Unikanie SELECT *:** Zamiast pobierać wszystkie kolumny z tabeli, wybieraj tylko te, które są niezbędne. Zmniejsza to ilość przesyłanych danych i zwiększa szybkość wykonania zapytań.
- **Wydajne łączenie tabel:** Używamy odpowiednich typów łączeń (JOIN) i unikamy łączenia dużych tabel, jeżeli nie jest to konieczne. Zaleca się także używanie łączeń w taki sposób, aby korzystać z indeksów.
- **Zarządzanie transakcjami:** Dbamy o to, aby transakcje były jak najkrótsze. Użycie odpowiednich poziomów izolacji może pomóc w optymalizacji operacji w bazie danych.

Technika	**Korzyści**
Indeksy	Szybszy dostęp do danych
Select na poszczególnych kolumnach	Mniejsze obciążenie sieci
Optymalizacja łączenia	Redukcja czasu wykonania
Zarządzanie transakcjami	Lepsza wydajność

Analiza planu wykonania zapytania

Jest kluczowym krokiem w procesie optymalizacji zapytań SQL. To narzędzie umożliwia zrozumienie, jak silnik bazy danych interpretuje zapytanie i jakie strategie wykorzystuje do jego realizacji. Dobrze przeprowadzona analiza może znacznie poprawić wydajność zapytania i całej aplikacji.

Aby skutecznie przeanalizować plan wykonania zapytania, warto zwrócić uwagę na kilka istotnych elementów:

- **Rodzaj operacji:** Ważne jest, jakie operacje są wykonywane na danych (np. skanowanie, łączenie, sortowanie).
- **Koszt:** Każda operacja ma przypisany koszt, który pozwala ocenić, jak złożona jest cała operacja.
- **Indeksy:** Należy sprawdzić, czy zapytanie korzysta z odpowiednich indeksów, co może drastycznie zwiększyć szybkość wykonania.
- **Łączenia:** Typ, który jest używany do łączenia tabel, może znacząco wpływać na wydajność zapytania.

Warto także przeprowadzać regularne przeglądy zapytań, aby zidentyfikować te, które mogą wymagać optymalizacji. Można to zrobić, korzystając z narzędzi analitycznych dostępnych w systemie zarządzania bazą danych. Analizując zapytania, należy być bardzo uważnym na zmiany w strukturze danych, które mogą wpłynąć na wydajność.

Element	**Opis**
Indeks	Struktura, która przyspiesza wyszukiwanie danych w tabeli.
Plan wykonania	Reprezentacja kroków podejmowanych przez silnik bazy danych w celu zrealizowania zapytania.
koszt	Osobna miara, określająca zasoby potrzebne do wykonania zapytania.

Na końcu warto również badać wpływ zmian, które wprowadza się w zapytaniach lub strukturze bazy danych. Utrzymywanie aktualnej dokumentacji planów wykonania oraz testowanie nowych rozwiązań w środowisku deweloperskim pomoże uniknąć problemów w produkcji.

Rola indeksów w optymalizacji zapytań SQL

Indeksy stanowią jeden z najważniejszych elementów bazy danych, wpływających na wydajność zapytań SQL. Poprawne ich wykorzystanie może znacząco przyspieszyć wykonanie operacji, a ich brak często prowadzi do znacznych spowolnień, szczególnie w przypadku dużych zbiorów danych.

Rodzaje indeksów:

- **Indeks unikalny:** zapewnia, że wszystkie wartości w kolumnie są unikalne, co z kolei wspomaga proces wyszukiwania.
- **Indeks wielokolumnowy:** pozwala na indeksowanie kilku kolumn, co zwiększa wydajność złożonych zapytań.
- **Indeks pełnotekstowy:** umożliwia wyszukiwanie słów w tekście, co jest szczególnie przydatne w przypadku dużych baz danych zawierających teksty.

Warto jednak pamiętać, że nadmiar indeksów może przynieść odwrotny skutek. Każdy indeks wymaga dodatkowej przestrzeni na dysku oraz wpływa na czas, jaki jest potrzebny do wstawiania, aktualizowania lub usuwania danych. Z tego powodu, kluczowe jest zrozumienie, które kolumny wymagają indeksowania.

Przykład analizy zapytań z wykorzystaniem indeksów:

Zapytanie	**Czas wykonania bez indeksu**	**Czas wykonania z indeksem**
SELECT * FROM products WHERE category = 'Electronics’	200 ms	30 ms
UPDATE users SET last_login = NOW() WHERE id = 123	150 ms	80 ms

Odpowiednie dobieranie indeksów to złożony proces, który może wymagać analizy zapytań oraz statystyk wykonania. W narzędziach takich jak EXPLAIN w SQL możemy zobaczyć, jak baza danych planuje wykonać zapytania i które indeksy będą używane.

Optymalizacja zapytań z użyciem klauzuli WHERE

Klauzula **WHERE** umożliwia precyzyjne filtrowanie wyników. Odpowiednie wykorzystanie tej klauzuli może znacząco wpłynąć na wydajność zapytania, zwłaszcza w przypadku dużych zbiorów danych. Poniżej przedstawiamy kilka kluczowych wskazówek dotyczących optymalizacji zapytań z użyciem **WHERE**.

- **Używaj indeksów:** Utworzenie indeksu na kolumnach, które często pojawiają się w klauzuli WHERE, może przyspieszyć czas wykonania zapytania. Indeksy pozwalają na szybszy dostęp do danych, co minimalizuje potrzebny czas przeszukiwania bazy.
- **Unikaj funkcji w klauzuli WHERE:** Funkcje, takie jak UPPER() lub LOWER(), mogą spowodować, że baza danych nie będzie mogła skorzystać z indeksu. Zamiast tego, lepiej przekształcić dane przed wykonaniem zapytania lub posługiwać się zapytaniami oparte na zestawach.
- **Używaj porównania:** Warunki porównawcze, takie jak =, <, czy >, są przeważnie bardziej wydajne niż operacje logiczne, np. LIKE, które wymagają pełnego przeszukiwania tekstu.
- **Unikaj złożonych subzapytania:** Możliwe, że lepiej zastąpić skomplikowane subzapytania prostymi połączeniami, które są zazwyczaj szybsze i łatwiejsze do zrozumienia.

W praktyce może to wyglądać tak:

Zapytanie z klauzulą WHERE	**Wydajność**
SELECT * FROM zamówienia WHERE status = 'zrealizowane'	Lepsza, dzięki indeksowi na kolumnie **status**
SELECT * FROM klienci WHERE UPPER(nazwisko) = 'KOWALSKI'	Gorsza, brak możliwości użycia indeksu
SELECT * FROM produkty WHERE cena > 100	Lepsza, szybkie porównanie

Znaczenie ograniczeń w zapytaniach SQL

W kontekście optymalizacji zapytań SQL, ograniczenia odgrywają kluczową rolę, wpływając na wydajność oraz integralność danych w bazach. Dzięki ich zastosowaniu, możliwe jest nie tylko wprowadzenie porządku w przechowywanych danych, ale także zwiększenie efektywności wykonywanych zapytań.

- **Ochrona integralności danych:** Ograniczenia, takie jak klucze główne, klucze obce czy unikalne, pomagają w utrzymaniu spójności danych. Zapobiegają one na przykład wprowadzeniu zduplikowanych rekordów czy błędnych relacji między tabelami.
- **Poprawa wydajności zapytań:** Ograniczenia mogą znacząco przyspieszyć okres wykonywanie zapytań. Na przykład, użycie indeksów w połączeniu z ograniczeniami może umożliwić szybsze znajdowanie rekordów w dużych zbiorach danych.
- **Lepsza czytelność kodu:** Zastosowanie ograniczeń czyni kod SQL bardziej czytelnym i zrozumiałym. Ograniczenia jasno definiują relacje oraz zasady, które muszą być przestrzegane podczas operacji na danych.

Warto również zwrócić uwagę na różne typy ograniczeń, które można zastosować w SQL:

Typ ograniczenia	**Opis**
PRIMARY KEY	Unikalny identyfikator rekordu w tabeli.
FOREIGN KEY	Relacja między dwiema tabelami, zapewniająca spójność danych.
UNIQUE	Zapobiega wprowadzeniu zduplikowanych wartości w kolumnie.
CHECK	Weryfikuje, czy wartości w kolumnie spełniają określone warunki.
DEFAULT	Ustala domyślną wartość dla kolumny, jeśli nie jest ona określona.

Implementacja ograniczeń powinna być starannie przemyślana, aby uniknąć problemów z wydajnością. Odpowiednia konfiguracja ograniczeń pozwala na minimalizację błędów i zwiększa efektywność zarządzania danymi. Dlatego ważne jest, aby programiści i administratorzy baz danych mieli świadomość znaczenia tych narzędzi w kontekście optymalizacji zapytań SQL.

Jak zmniejszyć liczbę złożonych joinów

W obliczu złożoności baz danych, efektywne zarządzanie zapytaniami SQL jest kluczowe. Zmniejszenie liczby skomplikowanych joinów to jeden ze sposobów na poprawę wydajności zapytań.

- **Użyj subzapytania:** Czasami warto wykorzystać subzapytania, co może uprościć główne zapytanie i zmniejszyć liczbę joinów. Dzięki temu możliwe jest zredukowanie liczby rekordów do przetworzenia.
- **Denormalizacja danych:** jeżeli to możliwe, rozważ denormalizację baz danych. Może to być korzystne w przypadku częstych zapytań, które wymagają wielu joinów. Dodanie kolumn z wymaganymi danymi do jednej tabeli może znacznie przyspieszyć proces.
- **Wykorzystanie agregacji:** Stosowanie funkcji agregujących, takich jak COUNT czy SUM, pozwala na obliczenia w jednej tabeli zamiast łączenia wielu. To podejście może zredukować liczbę wymaganych joinów.
- **Indeksowanie kolumn:** Indeksowanie kolumn używanych w warunkach joinu może znacznie poprawić wydajność zapytań. Dobre indeksy pozwolą na szybsze przeszukiwanie danych bez konieczności łączenia wielu tabel.

Oto przykład, w jaki sposób można zoptymalizować tabelę, aby zredukować liczbę joinów:

Tabela przed optymalizacją	**Tabela po optymalizacji**
Użytkownicy (ID, Imię, Nazwisko)	Użytkownicy (ID, Imię, Nazwisko, Email, Telefon)
Zakupy (ID, Użytkownik_ID, Produkt)	zakupy (ID, Użytkownik_ID, Produkt, Email, telefon)

Przykład powyższy pokazuje, jak poprzez dodanie kilku kolumn do tabeli zakupów, można zredukować liczbę joinów, umożliwiając jednocześnie łatwiejszy dostęp do potrzebnych danych.

Wykluczanie zbędnych kolumn z wyników

W procesie optymalizacji zapytań SQL jednym z kluczowych kroków jest eliminacja zbędnych kolumn w wynikach. Wiele zapytań niepotrzebnie zwraca całe tabele lub ich duże fragmenty, co prowadzi do wydłużenia czasu wykonania i zwiększonego obciążenia bazy danych. Aby poprawić efektywność, warto zastosować kilka prostych zasad:

- **Selekcja kolumn** – Zamiast używać ogólnej klauzuli SELECT *, lepiej określić dokładnie, które kolumny są nam potrzebne. Na przykład, zamiast:

SELECT * FROM users;

- możemy użyć:

SELECT id, name, email FROM users;

- **analiza danych** – Przed napisaniem zapytania, warto zastanowić się, jakie informacje są rzeczywiście potrzebne do osiągnięcia zamierzonego celu. Im mniej danych, tym szybciej zapytanie zostanie przetworzone.
- **Użycie aliasów** – Gdy zapytanie obejmuje zwracanie danych z połączeń wielu tabel, użycie aliasów dla kolumn może ułatwić ich identyfikację oraz zmniejszyć ich objętość w wynikach.

Przydatna może być również analiza użycia kolumn w różnych kontekstach. Przykładowa tabela poniżej ilustruje różnice w wydajności zapytań w zależności od liczby wybieranych kolumn:

Rodzaj zapytania	**Liczba kolumn**	**Czas wykonania (ms)**
Zapytanie 1	5	15
Zapytanie 2	10	25
Zapytanie 3	20	50

Ostatecznie, zwracając uwagę na wybór kolumn, możemy znacznie poprawić wydajność bazy danych i skrócić czas oczekiwania na wyniki.

Zastosowanie techniki EXISTS zamiast IN

W kontekście optymalizacji zapytań SQL jednym z kluczowych aspektów, które warto wziąć pod uwagę, jest wybór odpowiednich operatorów do filtrowania danych. Istnieje wiele sytuacji, w których wykorzystanie techniki **EXISTS** może przynieść znaczną poprawę wydajności w porównaniu do tradycyjnego korzystania z operatora **IN**. Zrozumienie tych różnic jest pierwszym krokiem do zwiększenia efektywności naszych zapytań.

Operator **IN** działa na zasadzie porównywania wartości kolumny z zestawem wartości, co może być efektywne w przypadku niewielkich zbiorów danych. Jak jednak pokazują liczne testy, w przypadku większych zbiorów, szczególnie w środowiskach z dużą ilością danych, jego efektywność znacząco maleje. Dzieje się tak, ponieważ baza danych musi przeszukać cały zbiór wartości, co zwiększa czas wykonania zapytania.

W porównaniu do tego, **EXISTS** zwraca **TRUE** lub **FALSE** na podstawie istnienia przynajmniej jednego wiersza w podzestawie danych. To podejście jest szczególnie korzystne, gdy korzystamy z podzapytań, ponieważ zapytanie kończy się natychmiast, gdy zostanie znaleziony pierwszy pasujący wiersz. Mniejsza liczba przeszukiwań w bazie danych prowadzi do szybszych czasów odpowiedzi, zwłaszcza w przypadku dużych zbiorów danych.

Oto małe porównanie zastosowania obu operatorów:

Operator	**Wydajność**	**Zastosowanie**
IN	Może być wolne przy dużych zbiorach	Małe, statyczne zbiory danych
EXISTS	Szybsze przy dużych zbiorach	Podzapytania, dynamiczne zbiory danych

Efektywne korzystanie z funkcji agregujących

Funkcje agregujące w SQL są niezwykle pomocne w przetwarzaniu danych, ale ich efektywne wykorzystanie wymaga zrozumienia kilku kluczowych aspektów. **Właściwe dobranie funkcji** pozwala nie tylko na uzyskanie pożądanych wyników, ale również na optymalizację wydajności zapytań.

- **Wybieraj odpowiednie kolumny:** Zminimalizuj liczbę przetwarzanych danych poprzez selekcję tylko tych kolumn, które są istotne dla wyników końcowych.
- **Używaj klauzuli GROUP BY:** Dobrze użyta klauzula GROUP BY pozwala na agregowanie danych na podstawie specyficznych kryteriów, co zwiększa czytelność wyników.
- **Filtruj wyniki z WHERE:** Zastosowanie klauzuli WHERE przed grupowaniem danych znacząco redukuje liczbę analizowanych wierszy, co wpływa na czas wykonania zapytania.

przykład zastosowania funkcji agregujących może obejmować analizę sprzedaży w czasie. Poniższa tabela ilustruje, jak można wykorzystać funkcję SUM() i GROUP BY do podsumowania wartości sprzedaży według miesiąca:

Miesiąc	**Łączna Sprzedaż**
Styczeń	5000 zł
Luty	7000 zł
marzec	6000 zł

Wykorzystanie podzapytań w optymalizacji

- **Redukcja liczby połączeń:** Zamiast wykonywać kilka zapytań, które zwracają wyniki na podstawie różnych tabel, można skorzystać z podzapytania, które zwraca wszystkie potrzebne dane w jednym kroku.
- **Eliminacja niepotrzebnych danych:** Podzapyty mogą pomóc w filtrowaniu danych na wcześniejszym etapie, co oznacza, że późniejsze operacje będą wykonywane na mniejszej ilości danych, co automatycznie przyspiesza całe zapytanie.
- **Lepsza czytelność:** Dzięki zastosowaniu podzapytań kod SQL staje się bardziej przejrzysty, co ułatwia jego zrozumienie i późniejsze modyfikacje.
- **Możliwość użycia wyników w różnych kontekstach:** Wyniki podzapytania mogą być wykorzystywane w różnych fragmentach głównego zapytania, co zwiększa elastyczność i możliwości przetwarzania danych.

Warto jednak pamiętać, że nadmierne korzystanie z podzapytań może prowadzić do spadku wydajności. Kluczem do optymalizacji jest umiejętne wyważenie pomiędzy ich wykorzystaniem a efektywnością zapytań. Przykład źle skonstruowanego podzapytania, które zamiast przyspieszyć działanie, spowalnia cały proces:

Rodzaj zapytania	**Czas wykonania (ms)**
Podzapytanie w SELECT	250
Podzapytanie w WHERE	400
Odwołania do JOIN	150

Wnioskując, odpowiednie wykorzystanie podzapytań może znacząco wpłynąć na optymalizację baz danych oraz zapewnić lepszą wydajność zapytań.

Cache'owanie wyników zapytań – czy warto?

Cache'owanie wyników zapytań to jeden z kluczowych elementów optymalizacji baz danych, który może znacznie zwiększyć wydajność aplikacji. Każde zapytanie do bazy danych generuje obciążenie, które może przyczynić się do wolniejszego działania całego systemu. Wykorzystanie mechanizmu cache'owania pozwala na przechowywanie wyników najczęściej wykonywanych zapytań, co z kolei zmniejsza czas odpowiedzi i obciążenie serwera.

Jedną z największych zalet cache'owania jest **znacząca redukcja obciążenia bazy danych**. Gdy wyniki zapytania są już zapisane w pamięci podręcznej, następne podobne zapytania mogą być obsługiwane bez potrzeby odwoływania się do bazy, co znacznie przyspiesza cały proces. Dodatkowo, zmniejsza to koszty operacyjne, a także umożliwia lepsze zarządzanie zasobami serwera.

Warto jednak pamiętać, że cache'owanie nie jest rozwiązaniem idealnym dla każdego scenariusza. Przykłady, w których warto rozważyć tę technikę to:

- Zapytania generujące duże ilości danych, które są często wykorzystywane.
- Dane statyczne, które rzadko się zmieniają.
- Wysoka roczna liczba powtórzeń zapytań o podobnej strukturze.

Istotne jest również monitorowanie skuteczności cache'owania. Można to osiągnąć poprzez analizowanie czasu wykonania zapytań przed i po wdrożeniu tej techniki. Poniższa tabela przedstawia przykładowe metryki, które warto obserwować:

Metryka	**Przed cache'owaniem**	**Po cache'owaniu**
Czas odpowiedzi (ms)	300	50
Obciążenie serwera (%)	80	30
Liczba unikalnych zapytań	1000	400

Analiza i optymalizacja zapytań w kontekście dużych zbiorów danych

Analiza zapytań SQL w kontekście dużych zbiorów danych to kluczowy krok w procesie optymalizacji wydajności baz danych. Wydajność zapytania może znacząco wpłynąć na czas reakcji aplikacji oraz na ogólną efektywność operacji wykonywanych na danych. Warto zwrócić uwagę na kilka istotnych czynników, które mogą pomóc w poprawieniu jakości zapytań.

Przede wszystkim, należy przeprowadzić analizę planu wykonania zapytania, aby zrozumieć, które operacje zajmują najwięcej czasu. Plan wykonania pokazuje, w jaki sposób silnik bazy danych zamierza wykonać dane zapytanie. Korzystając z narzędzi takich jak EXPLAIN w PostgreSQL lub SQL Server, można zidentyfikować potencjalne wąskie gardła i nieefektywne operacje.

oto kilka praktycznych wskazówek dotyczących optymalizacji:

- **Indeksowanie**: Tworzenie indeksów na kolumnach używanych w klauzulach WHERE,JOIN i ORDER BY może znacznie przyspieszyć zapytania.
- **Używanie odpowiednich typów danych**: Wybór najbardziej efektywnych typów danych dla przechowywanych wartości może zmniejszyć ilość przetwarzanych danych.
- **Unikanie złożonych podzapytań**: Gdy to możliwe, zamiast podzapytań, lepiej stosować JOIN, co może poprawić wydajność zapytań.
- **Optymalizacja klauzuli SELECT**: Warto pobierać tylko te kolumny, które są rzeczywiście potrzebne, zamiast używać SELECT *.

W przypadku dużych zbiorów danych, chwilowe zachowanie się systemu może być monitorowane za pomocą zapytań analitycznych, które ukazują, jak różne operacje wpływają na wydajność. Analiza statystyk dotyczących obciążenia bazy danych i zapytań pozwala na wczesne wykrycie problemów oraz podejmowanie działań naprawczych.

Istotne jest również regularne aktualizowanie statystyk bazy danych, co umożliwia optymalizatorowi zapytań podejmowanie lepszych decyzji dotyczących planu wykonania. Warto zainwestować czas w stworzenie i skonfigurowanie odpowiednich harmonogramów aktualizacji oraz analizy danych.

Ostatecznie, budowanie dzienników, które rejestrują czas wykonania zapytań, pozwala na bieżąco śledzić i analizować ich wydajność. Regularna analiza tych danych może ujawniać wzorce oraz wskazywać miejsca, w których optymalizacja zapytań jest niezbędna do poprawy działania całego systemu.

Monitorowanie wydajności bazy danych

Wydajność bazy danych ma kluczowe znaczenie dla funkcjonowania aplikacji.Aby efektywnie monitorować tę wydajność, warto wdrożyć kilka sprawdzonych praktyk:

- **Użycie narzędzi do monitorowania**: Wybierz odpowiednie oprogramowanie, które pozwoli na śledzenie zarówno zapytań, jak i stanu systemu. Narzędzia takie jak Nagios, Prometheus czy Grafana mogą być niezwykle pomocne.
- **Analiza logów**: Regularne przeglądanie logów bazy danych pomoże zidentyfikować wolno działające zapytania i inne problemy.
- **Monitorowanie obciążenia**: Obserwowanie CPU, pamięci oraz I/O to kluczowe elementy, które mogą wskazywać na problemy z wydajnością bazy danych.

Aby lepiej zrozumieć, gdzie mogą pojawiać się wąskie gardła, warto korzystać z odpowiednich metryk. Oto przykładowe metryki, które powinny być regularnie monitorowane:

Metryka	**Opis**
Czas wykonania zapytania	Czas potrzebny na wykonanie danego zapytania SQL.
Obciążenie CPU	Procent użycia procesora przez bazę danych.
wykorzystanie pamięci	Ilość pamięci wykorzystywanej przez serwer bazy danych.
Operacje dyskowe	Liczba operacji odczytu i zapisu na dysk.

Wymaga także zweryfikowania indeksów oraz struktury tabel. Indeksy mogą znacząco wpłynąć na szybkość zapytań, dlatego należy upewnić się, że są one aktualne i adekwatne do używanych zapytań. Rozważ także:

- **Usuwanie nieużywanych indeksów**: Zbyt wiele indeksów może spowolnić operacje zapisu.
- **Stosowanie odpowiednich typów danych**: Używanie bardziej zoptymalizowanych typów danych pozwala zaoszczędzić miejsce i poprawić wydajność.
- **Analiza planu zapytania**: Zrozumienie, jak baza danych wykonuje zapytania, pomoże w ich optymalizacji.

Wykorzystanie narzędzi do profilowania zapytań SQL

Profilowanie zapytań SQL jest kluczowym etapem w procesie optymalizacji. Dobrze dobrane narzędzia mogą znacząco przyspieszyć analizę wydajności zapytań oraz wykrywanie potencjalnych problemów. Istnieje wiele dostępnych rozwiązań, które pozwalają na monitorowanie, analizowanie i optymalizowanie zapytań w bazach danych. Warto zapoznać się z kilkoma z nich:

- **EXPLAIN** - to podstawowe polecenie, które pozwala na analizę planu wykonania zapytania. Dzięki niemu można zrozumieć, jak baza danych przetwarza nasze zapytania.
- **SQL Profiler** - narzędzie dostępne w SQL Server, które umożliwia monitorowanie zapytań w czasie rzeczywistym. Dzięki niemu można śledzić wykonanie zapytań oraz identyfikować te najbardziej obciążające system.
- **pg_stat_statements** - rozszerzenie dla PostgreSQL, które zbiera statystyki dotyczące wykonanych zapytań. Umożliwia analizę najczęściej wykonywanych zapytań oraz ich czasów wykonania.
- **MySQL Slow Query log** - mechanizm w MySQL, który loguje zapytania, które przekraczają określony czas wykonania. To doskonałe narzędzie do identyfikacji problematycznych zapytań.

Wszystkie te narzędzia można zintegrować z procesem ciągłego monitorowania wydajności baz danych. Genialną praktyką jest regularne analizowanie wyników, aby móc reagować na zmieniające się potrzeby użytkowników i dostosowywać zapytania do aktualnej struktury bazy danych.

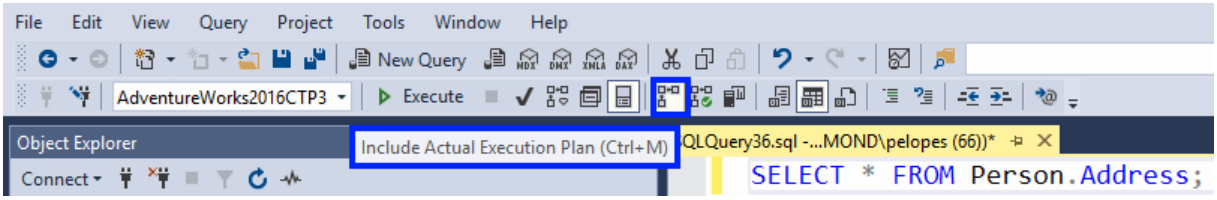
Plan wykonania (Execution Plan)

Uwzględnienie planu wykonania zapytania podczas jego uruchamiania

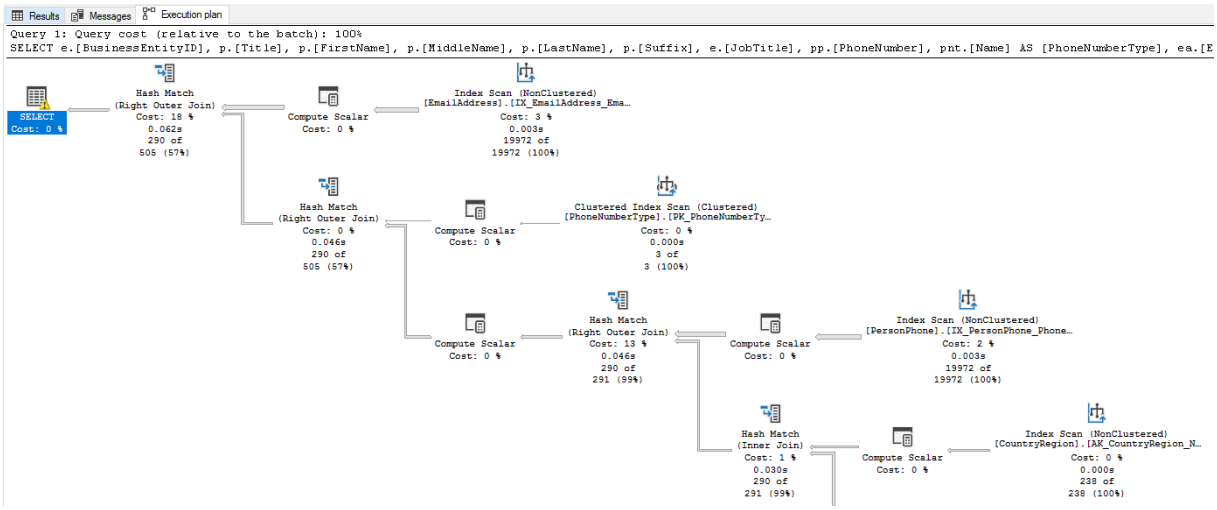
1. Na pasku narzędzi programu SQL Server Management Studio wybierz **Database Engine Query** (Zapytanie do silnika bazy danych). Możesz również otworzyć istniejące zapytanie i wyświetlić szacowany plan wykonania, klikając przycisk **Otwórz plik** na pasku narzędzi i wskazując odpowiedni plik zapytania.Enter the query for which you would like to display the actual execution plan.

2. Wprowadź zapytanie, dla którego chcesz wyświetlić rzeczywisty plan wykonania.

3. W menu **Query** (Zapytanie) wybierz **Include Actual Execution Plan** (Uwzględnij rzeczywisty plan wykonania) lub kliknij odpowiedni przycisk na pasku narzędzi.

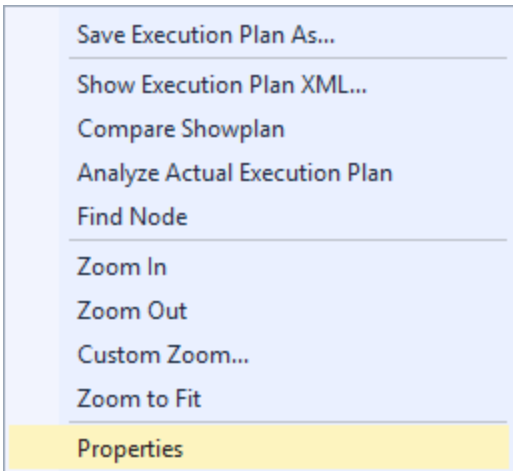


4. Uruchom zapytanie, klikając przycisk **Execute** (Wykonaj) na pasku narzędzi. Plan wykorzystany przez optymalizator zapytań zostanie wyświetlony na karcie **Execution Plan** (Plan wykonania) w panelu wyników.



5. Zatrzymaj kursor myszy nad operatorami logicznymi i fizycznymi, aby wyświetlić ich opisy oraz właściwości w dymkach informacyjnych. Możesz również zobaczyć właściwości całego planu wykonania, klikając główny operator (np. węzeł **SELECT** na ilustracji powyżej).

Alternatywnie, możesz wyświetlić właściwości operatorów w oknie **Właściwości**. Jeśli okno to nie jest widoczne, kliknij prawym przyciskiem myszy wybrany operator i wybierz **Properties** (Właściwości). Następnie wybierz operator, aby zobaczyć jego właściwości.



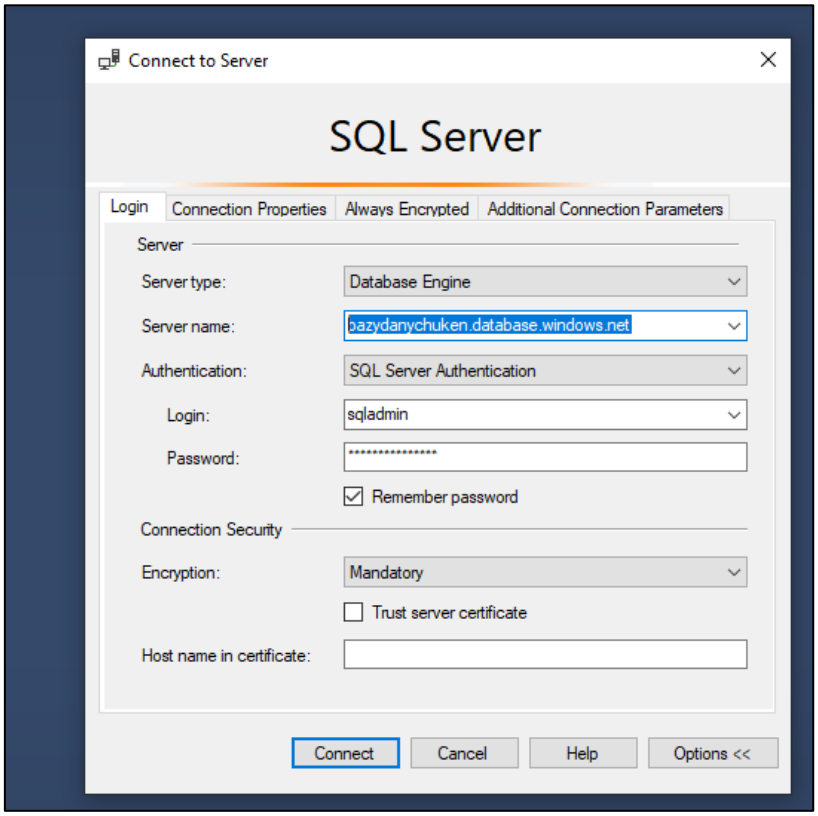
6. Możesz zmienić sposób wyświetlania planu wykonania, klikając prawym przyciskiem myszy na planie i wybierając **Zoom In** (Powiększ), **Zoom Out** (Pomniejsz), **Custom Zoom** (Własne powiększenie) lub **Zoom to Fit** (Dopasuj do okna). Opcje „Zoom In" i „Zoom Out" pozwalają na przybliżenie lub oddalenie widoku planu, natomiast „Custom Zoom" umożliwia ustawienie własnej skali powiększenia (np. 80%). „Zoom to Fit" dopasowuje cały plan do widocznego obszaru w panelu wyników. Alternatywnie, możesz przytrzymać klawisz **CTRL** i przewijać kółkiem myszy, aby dynamicznie zmieniać skalę.
7. Aby poruszać się po planie wykonania, skorzystaj z pionowego i poziomego paska przewijania, lub kliknij i przytrzymaj pusty obszar planu, a następnie przeciągnij mysz. Możesz również kliknąć i przytrzymać znak **+** w prawym dolnym rogu okna planu wykonania, aby wyświetlić miniaturową mapę całego planu.

Materiały dodatkowe

- https://www.cs.put.poznan.pl/wandrzejewski/wp-content/uploads/bd/PLSQL_06_Wyzwalacze.pdf
- https://www.geeksforgeeks.org/sql-trigger-student-database/
- https://www.datacamp.com/tutorial/sql-triggers
- https://www.geeksforgeeks.org/sql-trigger-student-database/
- https://www.tutorialspoint.com/plsql/plsql_triggers.htm
- https://www.cs.put.poznan.pl/rwrembel/dydaktykaPDF/RST_distrib_query_opt.pdf
- https://www.tutorialspoint.com/sql/sql-indexes.htm
- https://dbplus.tech/pl/2024/09/26/od-klastrowych-po-specjalistyczne-rodzaje-indeksow-sql-i-kiedy-ich-uzywac/
- https://www.w3schools.com/sql/sql_create_index.asp
- https://www.youtube.com/watch?v=TAeqAS-SG9M
- https://www.sqlpedia.pl/aspekty-wydajnosciowe-zapytan-sql/
- https://www.sqlpedia.pl/pomiar-wydajnosci-zapytan-sql-server/
- https://learn.microsoft.com/en-us/sql/relational-databases/performance/display-an-actual-execution-plan?view=sql-server-ver16
- https://learn.microsoft.com/en-us/sql/relational-databases/performance/execution-plans?view=sql-server-ver16
- https://www.sqlshack.com/execution-plans-in-sql-server/
- https://www.geeksforgeeks.org/query-execution-plan-in-sql/

Zadania praktyczne

Uruchom SQL Server Management Studio (https://learn.microsoft.com/en-us/ssms/download-sql-server-management-studio-ssms) i skonfiguruj podstawową bazę danych (https://www.youtube.com/watch?v=_12OOgKzi7I). Można wykorzystać gotową bazę danych:



Server name: bazydanychuken.database.windows.net

Login: sqladmin

Password: BazyDanych1234.!

Tworzenie wyzwalaczy w SQL

Zadanie 1: Logowanie nowych klientów

Typ: AFTER INSERT
Cel: Po dodaniu nowego klienta, wyświetl komunikat z jego imieniem i nazwiskiem.

Podpowiedź: użyj tabeli SalesLT.Customer i kolumn FirstName, LastName.

Zadanie 2: Ostrzeżenie przy wstawieniu produktu o wysokiej cenie

Typ: AFTER INSERT
Cel: Stwórz trigger, który po dodaniu nowego produktu o cenie ListPrice powyżej 1000 zł wypisuje komunikat: 'UWAGA: Produkt o wysokiej cenie został dodany!'.

Tabelka: SalesLT.Product

Zadanie 3: Zablokuj produkty z nazwą zawierającą słowo "test"

Typ: INSTEAD OF INSERT
Cel: Nie pozwól na dodanie produktu, którego Name zawiera słowo „test". Zamiast tego wyświetl komunikat.

Użyj funkcji LIKE w warunku IF EXISTS.

Zadanie 4: Reagowanie na zmianę statusu zamówienia

Typ: AFTER UPDATE
Cel: Utwórz trigger, który wykryje zmianę wartości w kolumnie Status w tabeli SalesLT.SalesOrderHeader i wypisze komunikat, ile zamówień zmieniło status.

Użyj tabel inserted, deleted oraz funkcji UPDATE().

Zadanie 5: Zliczanie usuniętych zamówień o wartości powyżej 500 zł

Typ: AFTER DELETE
Cel: Po usunięciu zamówienia, jeśli TotalDue > 500, wypisz liczbę takich usuniętych rekordów.

Tabela: SalesLT.SalesOrderHeader

Zadanie 6: Automatyczne logowanie działań

Typ: AFTER INSERT
Cel: Stwórz nową tabelę CustomerLog i zapisz tam dane nowo dodanego klienta (np. CustomerID, FirstName, LogTime).

Użyj INSERT INTO ... SELECT ... FROM inserted.

Zadanie 7: Uniemożliwienie zmiany ceny na 0

Typ: INSTEAD OF UPDATE
Cel: Zapobiegaj aktualizacji kolumny ListPrice na 0 w tabeli SalesLT.Product. Jeśli ktoś spróbuje to zrobić – wyświetl komunikat i nie wykonuj zmiany.

*Użyj warunku IF EXISTS (SELECT * FROM inserted WHERE ListPrice = 0).*

Zadanie 8: Automatyczne ustawianie domyślnego koloru

Typ: AFTER INSERT
Cel: Jeśli nowy produkt nie ma przypisanego koloru (Color IS NULL), automatycznie uzupełnij go jako 'Brak koloru'.

Wymaga użycia UPDATE SalesLT.Product SET Color = 'Brak koloru' WHERE ProductID IN (...).

Zadanie 9: Zakaz usuwania ostatniego adresu klienta

Typ: INSTEAD OF DELETE
Cel: Zabezpiecz tabelę SalesLT.CustomerAddress – nie pozwól usunąć ostatniego adresu przypisanego do danego klienta.

Sprawdź w deleted, ile adresów zostało klientowi – jeśli tylko jeden, zablokuj operację.

Zadanie 10: Monitorowanie aktualizacji danych kontaktowych

Typ: AFTER UPDATE
Cel: Po zmianie numeru telefonu (Phone) w tabeli SalesLT.Customer, wyświetl komunikat: 'Telefon klienta został zmieniony'.

Użyj IF UPDATE(Phone).

Zadanie 11: Zliczanie wszystkich produktów dodanych z kolorem „Red”

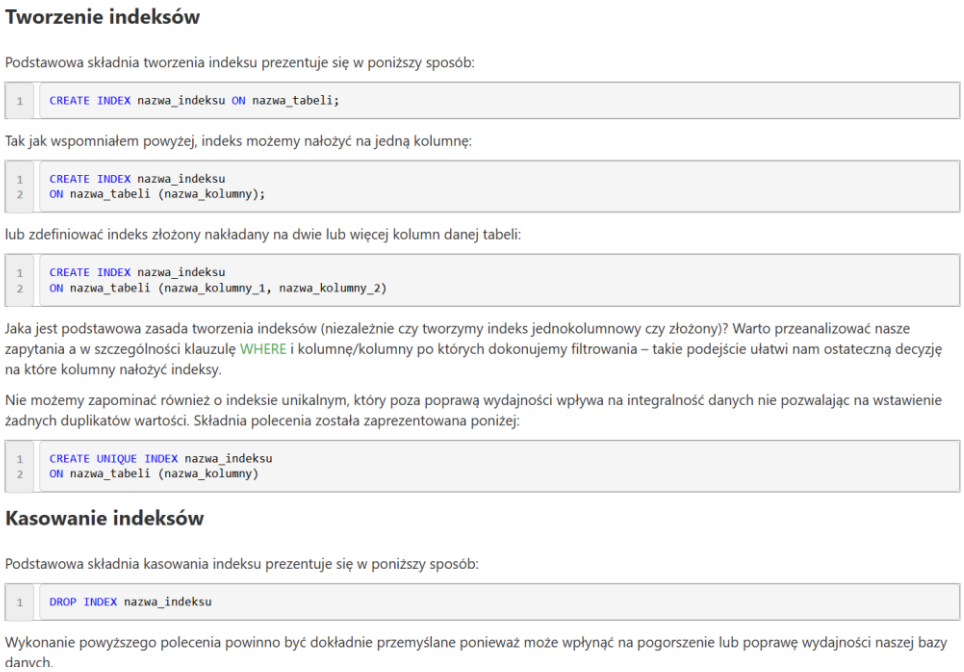
Typ: AFTER INSERT
Cel: Po dodaniu produktów w kolorze Red, zlicz ich liczbę i wyświetl komunikat typu: 'Dodano X czerwonych produktów'.

Użyj SELECT COUNT(*) FROM inserted WHERE Color = 'Red'.

Tworzenie indeksów w SQL

https://www.w3schools.com/sql/sql_create_index.asp

https://www.w3schools.com/sql/sql_create_index.asp



Tworzenie indeksów

Podstawowa składnia tworzenia indeksu prezentuje się w poniższy sposób:

```
CREATE INDEX nazwa_indeksu ON nazwa_tabeli;
```

Tak jak wspomniałem powyżej, indeks możemy nałożyć na jedną kolumnę:

```
CREATE INDEX nazwa_indeksu
ON nazwa_tabeli (nazwa_kolumny);
```

lub zdefiniować indeks złożony nakładany na dwie lub więcej kolumn danej tabeli:

```
CREATE INDEX nazwa_indeksu
ON nazwa_tabeli (nazwa_kolumny_1, nazwa_kolumny_2)
```

Jaka jest podstawowa zasada tworzenia indeksów (niezależnie czy tworzymy indeks jednokolumnowy czy złożony)? Warto przeanalizować nasze zapytania a w szczególności klauzulę WHERE i kolumnę/kolumny po których dokonujemy filtrowania – takie podejście ułatwi nam ostateczną decyzję na które kolumny nałożyć indeksy.

Nie możemy zapominać również o indeksie unikalnym, który poza poprawą wydajności wpływa na integralność danych nie pozwalając na wstawienie żadnych duplikatów wartości. Składnia polecenia została zaprezentowana poniżej:

```
CREATE UNIQUE INDEX nazwa_indeksu
ON nazwa_tabeli (nazwa_kolumny)
```

Kasowanie indeksów

Podstawowa składnia kasowania indeksu prezentuje się w poniższy sposób:

```
DROP INDEX nazwa_indeksu
```

Wykonanie powyższego polecenia powinno być dokładnie przemyślane ponieważ może wpłynąć na pogorszenie lub poprawę wydajności naszej bazy danych.

Zadanie 1: Indeks na kolumnie wyszukiwanej w zapytaniach

Cel: Utwórz indeks przyspieszający wyszukiwanie klientów po adresie e-mail.

Tabela: SalesLT.Customer
Kolumna: EmailAddress

Zadanie 2: Indeks złożony (wielokolumnowy)

Cel: Utwórz indeks, który przyspieszy zapytania sortujące produkty wg Color i ListPrice.

Tabela: SalesLT.Product
Kolumny: Color, ListPrice

Zadanie 3: Indeks unikalny

Cel: Utwórz indeks, który zabezpieczy unikalność numerów produktów.

Tabela: SalesLT.Product
Kolumna: ProductNumber

Execution Plan – analiza zapytań w SSMS

https://www.sqlpedia.pl/aspekty-wydajnosciowe-zapytan-sql/

https://www.sqlpedia.pl/pomiar-wydajnosci-zapytan-sql-server/

Zadanie 1: Analiza bez indeksu (Table Scan)

Polecenie:

1. Uruchom zapytanie:

```
SELECT * FROM SalesLT.Customer WHERE Phone = '123-456-789';
```

2. Włącz w SSMS: **"Include Actual Execution Plan"** (lub użyj Ctrl + M)
3. Zobacz, że występuje **pełne skanowanie tabeli (Table Scan)** – czyli przeszukiwanie każdego wiersza.

Zadanie 2: Dodanie indeksu i ponowna analiza

Polecenie:

1. Utwórz indeks:

```
CREATE INDEX IX_Customer_Phone ON SalesLT.Customer(Phone);
```

2. Ponownie uruchom zapytanie z Zadania 1.

Sprawdź w Execution Plan:

- powinno pojawić się **Index Seek**
- zobaczysz, że **koszt wykonania zapytania spadł**

Zadanie 3: Analiza złączeń (JOIN)

Polecenie:

1. Uruchom zapytanie:

```
SELECT c.FirstName, c.LastName, soh.SalesOrderID, soh.TotalDue
FROM SalesLT.Customer c
JOIN SalesLT.SalesOrderHeader soh ON c.CustomerID = soh.CustomerID;
```

2. Zbadaj, jaki typ złączenia został użyty w planie (np. Nested Loops).

Dodatkowe polecenie (dla indeksu):

```
CREATE INDEX IX_SalesOrderHeader_CustomerID
ON SalesLT.SalesOrderHeader (CustomerID);
```

3. Porównaj plan **przed i po** dodaniu indeksu.

Zadanie 4: Porównanie filtrów i sortowania

Polecenie:

1. Uruchom zapytanie:

```
SELECT * FROM SalesLT.Product WHERE ListPrice > 500 ORDER BY ListPrice DESC;
```

2. Zwróć uwagę w planie na użycie sortowania (**Sort**) oraz na sposób filtrowania (**Index Seek** lub **Scan**).

Zadanie 5: Parametryzacja i ponowne użycie planu

Polecenie:

1. Uruchom kilka razy poniższe zapytania z różnymi parametrami:

```
SELECT * FROM SalesLT.Product WHERE Color = 'Red';

SELECT * FROM SalesLT.Product WHERE Color = 'Black';
```

2. Sprawdź, czy SQL używa tego samego planu dla obu zapytań