# SQL Server

Peter A. Carter



# 100

najczęstszych błędów
i jak ich skutecznie unikać

Helion

Tytuł oryginału: 100 SQL Server Mistakes and How to Avoid Them

Tłumaczenie: Grzegorz Werner

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki
Materiały graficzne na okładce: Adobe Stock

ISBN: 978-83-289-2896-1



Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*helion.pl/user/opinie/ss100b*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

# Spis treści

# 5 Programowanie w języku T-SQL

**W tym rozdziale:**

- Pomyłki prowadzące do nieoczekiwanych rezultatów
- Pomyłki prowadzące do problemów z wydajnością
- Unikanie pętli z kursorami w T-SQL
- Usuwanie dużej liczby wierszy

SQL to standardowy język ANSI i ISO, który umożliwia programistom baz danych przeszukiwanie i przetwarzanie informacji zawartych w relacyjnych bazach danych. T-SQL jest dialektem SQL używanym w SQL Serverze i służy do interakcji z instancjami SQL Servera oraz hostowanymi w nich bazami danych.

W rozdziale 4. zaprojektowaliśmy i utworzyliśmy tabele dla nowej bazy danych MagicChoc. W tym rozdziale przyjrzymy się typowym pomyłkom, które mogą popełniać programiści mniej doświadczeni w języku T-SQL. Jako przykład posłuży nam firma MagicChoc, dla której mamy opracować logikę wykorzystywaną przez jej aplikacje frontendowe. Będzie to dla nas okazja, aby zacząć analizować najczęstsze pułapki, w które wpadają początkujący programiści T-SQL.

Opanowanie SQL może stanowić wyzwanie dla programistów, którzy są bardziej zaznajomieni z pisaniem kodu aplikacji w językach takich jak C# czy Visual Basic. Wynika to z dużej różnicy koncepcyjnej w sposobie działania tych języków. Na przykład stosowanie pętli w językach .NET jest powszechnie akceptowane, ale w opartym na zbiorach świecie SQL może powodować poważne problemy z wydajnością.

Większość pomyłek programistycznych w T-SQL powoduje problemy z wydajnością i to będzie głównym tematem tego rozdziału. W pierwszych dwóch podrozdziałach skupimy się jednak na błędach prowadzących do nieoczekiwanych wyników. Na koniec przyjrzymy się częstej pomyłce popełnianej przy usuwaniu dużej liczby wierszy.

# 5.1. Numer 14 — niepoprawna obsługa wartości NULL

Firma MagicChoc poprosiła nas o przeanalizowanie danych i ustalenie, ile podkategorii produktów nie ma opisu. W związku z tym wykonujemy kwerendę pokazaną na listingu 5.1.

**Listing 5.1. Niepoprawne zliczanie wartości NULL**

```
SELECT COUNT(*)
FROM dbo.ProductSubcategories
WHERE ProductSubcategoryDescription = NULL ;
```

Wynik zwrócony to 0. Świetnie. Każda podkategoria produktu powinna mieć opis, prawda? Wiemy, że łącznie mamy 16 podkategorii, więc sprawdźmy nasz wynik jeszcze raz, modyfikując kwerendę tak, aby policzyć liczbę wierszy, w których opis nie jest równy NULL. Użyjemy do tego kwerendy z listingu 5.2.

**Listing 5.2. Niepoprawne zliczanie wartości różnych od NULL**

```
SELECT COUNT(*)
FROM dbo.ProductSubcategories
WHERE ProductSubcategoryDescription <> NULL ;
```

Chwileczkę! Ta kwerenda również zwraca 0 jako wynik. Co się tutaj dzieje? Programiści, którzy dopiero zaczynają pracę z SQL, czasem nie zdają sobie sprawy, że NULL reprezentuje nieznaną wartość. W związku z tym w porównaniach jedno NULL nie jest równe innemu NULL.

Aby to zrozumieć, pomyśl o następującej analogii. Ile gwiazd jest w naszej galaktyce? Osobiście nie znam odpowiedzi na to pytanie. Ile ziarenek piasku jest na świecie? Znowu, nie mam pojęcia. Czy to oznacza, że liczba gwiazd w galaktyce jest równa liczbie ziarenek piasku na świecie? Oczywiście, że nie. Może być taka sama, a może nie być. Nie wiem. Podobnie jak ja nie mogę stwierdzić, czy dwie nieznane mi wartości są takie same czy różne, tak samo SQL Server nie jest w stanie określić, czy dwie nieznane mu wartości są identyczne czy odmienne.

Aby rozwiązać ten problem, wystarczy zmodyfikować składnię przy obsłudze wartości NULL, używając operatorów IS lub IS NOT zamiast = i <>. Na przykład skrypt z listingu 5.3 poprawnie zwraca liczbę podkategorii produktów, które nie mają opisu, a następnie liczbę podkategorii produktów, które mają opis.

**Listing 5.3. Poprawne zliczanie wartości NULL i różnych od NULL**

```
SELECT COUNT(*)
FROM dbo.ProductSubcategories
WHERE ProductSubcategoryDescription IS NULL ;

SELECT COUNT(*)
FROM dbo.ProductSubcategories
WHERE ProductSubcategoryDescription IS NOT NULL ;
```

Kolejnym aspektem pracy z wartościami NULL, który może być mylący dla początkujących użytkowników SQL, jest różnica między operatorem IS NULL a funkcją ISNULL(). Jak właśnie zobaczyliśmy, IS NULL stosuje się w klauzuli WHERE do filtrowania wyników kwerendy tak, aby zwracała ona tylko wiersze, w których dana kolumna zawiera wartości NULL.

Funkcja ISNULL() jest natomiast używana na liście SELECT, w klauzuli JOIN lub klauzuli SET instrukcji UPDATE do zastąpienia wartości NULL inną niepustą wartością. Na przykład kwerenda z listingu 5.4 zwróci pełną listę podkategorii produktów, ale opisy o wartości NULL zostaną zastąpione tekstem Brak opisu.

**Listing 5.4. Użycie funkcji ISNULL() do zastąpienia wartości NULL**

```
SELECT
      ProductSubcategoryName
    , ISNULL(ProductSubcategoryDescription, 'Brak opisu')
FROM dbo.ProductSubcategories ;
```

Zawsze zachowuj ostrożność przy pracy z wartościami NULL. Pamiętaj, że jedna wartość NULL nie jest równa innej wartości NULL. Pamiętaj też, że składnia IS NULL służy do filtrowania zapytań w celu zwrócenia wartości NULL, natomiast funkcja ISNULL() jest używana do zastąpienia wartości NULL inną, określoną wartością.

# 5.2. Numer 15 — używanie NOLOCK jako sposobu na poprawę wydajności

Aplikacja sprzedażowa MagicChoc zawiera pole rozwijane z adresami klientów, które umożliwia sprzedawcy wybór adresu dostawy. Jednak gdy otwiera się ekran z adresami dostawy, lista rozwijana wolno się wypełnia danymi. Poproszono nas o zwiększenie wydajności kwerendy odpowiedzialnej za tę operację.

Słyszeliśmy, że blokowanie może powodować problemy z wydajnością w SQL Serverze. Ktoś wspomniał o istnieniu wskazówki `NOLOCK`, która może poprawić wydajność. W związku z tym zmodyfikowaliśmy kwerendę, która wypełnia listę rozwijaną z adresami dostawy, w sposób pokazany na listingu 5.5.

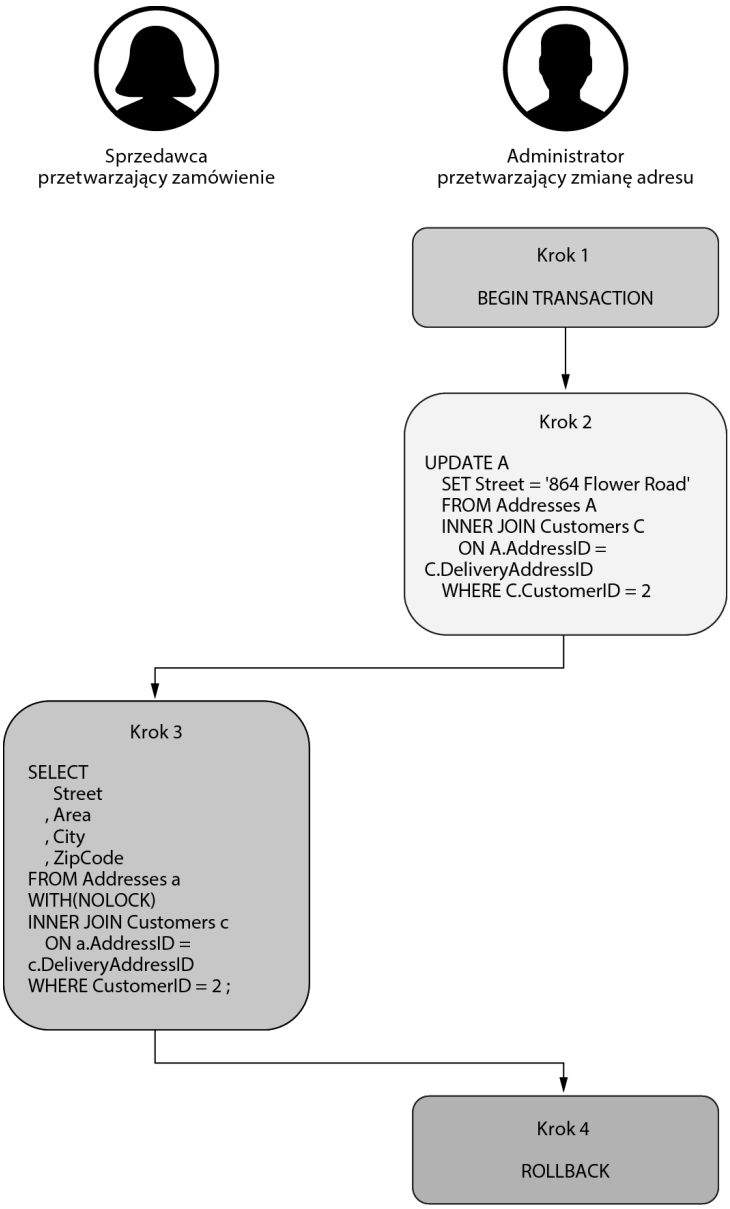
**Listing 5.5. Dodawanie wskazówki NOLOCK do kwerendy**

```
DECLARE @CustomerID INT ;
SET @CustomerID = 2 ;
SELECT
      Street
    , Area
    , City
    , ZipCode
FROM dbo.Addresses a WITH(NOLOCK)
INNER JOIN dbo.Customers c
    ON a.AddressID = c.DeliveryAddressID
WHERE CustomerID = @CustomerID ;
```

Przez pewien czas wszystko działa jak należy, ale pewnego dnia przesyłka trafia pod niewłaściwy adres i zostajemy poproszeni o zbadanie, jak mogło do tego dojść.

Wyobraźmy sobie następującą sytuację. Sprzedawca przetwarza zamówienie od firmy Cooking Schmooking. W tym samym czasie administrator rozmawia z innym członkiem zespołu Cooking Schmooking o aktualizacji różnych danych, w tym o zmianie adresu. Administrator orientuje się, że wprowadził nieprawidłowy adres, i anuluje aktualizację danych klienta przed jej zakończeniem. Administrator kontynuuje, wprowadzając poprawny adres. Przyjrzyjmy się sekwencji zdarzeń przedstawionej na rysunku 5.1.

SQL Server wykorzystuje blokady, aby zagwarantować, że transakcja nie będzie mogła odczytać danych, które są obecnie modyfikowane przez inną transakcję. Używając wskazówki `NOLOCK`, uniemożliwiamy naszemu zapytaniu `SELECT` założenie jakichkolwiek blokad. W rezultacie sprzedawca odczytuje adres z tabeli, podczas gdy transakcja wykonująca aktualizację jest w toku. Następnie transakcja aktualizacyjna zostaje wycofana. Prowadzi to do sytuacji, w której sprzedawca odczytuje adres dostawy, który nigdy nie został faktycznie zatwierdzony, a więc tak naprawdę nigdy nie istniał w bazie danych.




**Rysunek 5.1.** *Sekwencja zdarzeń*

Istnieje wiele sposobów optymalizacji wydajności SQL Servera. Jednym z nich jest dostrajanie blokad poprzez wykorzystanie poziomów izolacji transakcji, które wpływają na sposób zarządzania blokadami. Temat ten zostanie omówiony szczegółowo w rozdziale 10. Zdecydowanie odradzam stosowanie opcji `NOLOCK` w kwerendach. Jest to nieprzejrzysta optymalizacja, która zazwyczaj niesie ze sobą więcej ryzyka niż korzyści.

# 5.3. Numer 16 — standardowe stosowanie instrukcji SELECT *

Przyjrzyjmy się teraz pomyłkom, które mogą prowadzić do problemów z wydajnością. Zacznijmy od scenariusza, w którym tworzymy procedurę zwracającą informacje o obszarach sprzedaży. Nie pamiętamy dokładnie kolumn w tabeli ani jak wyglądają dane, więc wykonujemy następującą doraźną kwerendę:

```
SELECT *
FROM SalesAreas
```

Następnie kolega pyta nas, czy przechowujemy daty ważności produktów. Nie jesteśmy pewni, więc wykonujemy następującą doraźną kwerendą, żeby odpowiedzieć koledze:

```
SELECT *
FROM Products
```

Oba te przykłady reprezentują poprawne użycie kwerendy `SELECT *`. Problem, który chcę omówić, pojawia się wtedy, gdy programiści stosują `SELECT *` w kodzie, który planują wdrożyć i utrzymywać.

Wyobraźmy sobie, że nasza aplikacja ma przeprowadzić analizę czasu realizacji zamówień i liczby opóźnionych dostaw. Aby spełnić to wymaganie, musimy zwrócić do aplikacji frontendowej następujące kolumny:

- `SalesOrderDate`,
- `SalesOrderDeliveryDueDate`,
- `SalesOrderDeliveryActualDate`.

Zamiast wybierać te trzy konkretne kolumny, decydujemy się użyć `SELECT *`. Jest to błąd z trzech powodów: wydajności, utrzymania kodu i jego czytelności. Przyjrzyjmy się najpierw kwestii wydajności.

Rozważając użycie instrukcji `SELECT *`, powinniśmy wziąć pod uwagę dwa aspekty wydajności. Pierwszy dotyczy przesyłania danych do warstwy aplikacji. W tym przypadku nasza aplikacja potrzebuje tylko trzech kolumn, co daje łącznie 9 bajtów. Gdybyśmy przesłali wszystkie kolumny, rozmiar każdego wiersza mógłby wzrosnąć nawet do 61 bajtów. To dodatkowe 51 bajtów na wiersz. Wyobraźmy sobie, że w tabeli mamy 2,5 miliona zamówień. Oznacza to, że przesyłamy przez sieć dodatkowe 121 MB danych zupełnie bez potrzeby.

Wyobraź sobie teraz, że zastosowalibyśmy tę samą technikę dla tabeli zawierającej kilka kolumn typu `NVARCHAR(MAX)`, z których każda może przechowywać do 2 GB danych. Co by się stało, gdyby wielu użytkowników jednocześnie wykonało tę samą kwerendę? Łatwo zrozumieć, jak szybko mogłoby to stać się problemem.

Drugim aspektem wydajności, który należy rozważyć, są indeksy. Aby wyjaśnić ten problem, wyobraźmy sobie, że dokładnym wymaganiem jest zwrócenie trzech kolumn, filtrowanych według daty zamówienia. Aby zrealizować tę kwerendę w najbardziej efektywny sposób, moglibyśmy utworzyć tak zwany *indeks pokrywający* — czyli indeks zawierający wszystkie kolumny, które chcemy zwrócić.

Na przykład kod z listingu 5.6 tworzy indeks oparty na kolumnie `SalesOrder↪Date`, ale następnie dołącza kolumny `SalesOrderDeliveryDueDate` i `SalesOrder↪DeliveryActualDate`. Gdy używasz składni `INCLUDE`, SQL Server generuje indeks na głównej kolumnie (lub kolumnach), a następnie dodaje wartości dołączonych kolumn na poziomie liści. Jest to szczególnie przydatne w przypadku kwerend pokrywających, kiedy musisz filtrować lub łączyć dane na podstawie określonej kolumny, ale w wynikach chcesz również zwrócić niewielki zbiór innych kolumn. Takie podejście minimalizuje rozmiar indeksu, jednocześnie zapobiegając konieczności odwoływania się do indeksu klastrowego w celu pobrania pozostałych kolumn.

**Listing 5.6. Tworzenie indeksu pokrywającego**

```
CREATE NONCLUSTERED INDEX [OrderDate-Including-DueDate-ActualDate]
    ON dbo.SalesOrderHeaders (SalesOrderDate)
INCLUDE(SalesOrderDeliveryDueDate,SalesOrderDeliveryActualDate) ;
```

Niestety, jeśli używasz instrukcji `SELECT *`, indeksy tego typu prawie na pewno nie zostaną wykorzystane, ponieważ nie obejmują wszystkich kolumn, które zwracasz. Oczywiście, można by uwzględnić każdą kolumnę w tabeli, ale spowodowałoby to utworzenie bardzo szerokiego i nieefektywnego indeksu. Co więcej, gdybyśmy dodali nową kolumnę do tabeli, indeks przestałby działać, chyba że pamiętalibyśmy o zaktualizowaniu jego definicji.

To prowadzi do drugiego problemu związanego z instrukcją `SELECT *`, jakim jest utrzymanie kodu. Wyobraźmy sobie aplikację działającą jako warstwa pośrednia. Pobiera ona daty zamówień z tabeli `SalesOrderHeaders` za pomocą takiej kwerendy, jak:

```
SELECT *
FROM dbo.SalesOrderHeaders
WHERE SalesOrderDate = '20230616' ;
```

Dane są następnie przekazywane do procedury składowanej w innej instancji, która przeprowadza analizę. Ponieważ przekazujemy wszystkie kolumny z tabeli, typ tabeli w instancji analitycznej jest tworzony w następujący sposób:

```
CREATE TYPE SalesOrdersForAnalysis AS TABLE
(
    SalesOrderNumber                NCHAR(12)       NOT NULL,
    SalesOrderDate                  DATE            NOT NULL,
    SalesPersonID                   INT             NOT NULL,
    SalesAreaID                     INT             NOT NULL,
    CustomerID                      INT             NOT NULL,
```

```
    SalesOrderDeliveryDueDate       DATE            NOT NULL,
    SalesOrderDeliveryActualDate    DATE            NULL,
    CurrierUsedforDelivery          NVARCHAR(32)    NOT NULL
) ;
```

Procedura składowana jest zadeklarowana w taki sposób:

```
CREATE PROCEDURE dbo.AsyncAnalysis
    @DatesForAnalysis SalesOrdersForAnalysis READONLY
AS
BEGIN
    SELECT *
    FROM @DatesForAnalysis ;

    --Tutaj logika analizy…
END
```

W tym przypadku, po dodaniu nowej kolumny do tabeli `SalesOrderHeaders`, będziemy musieli wykonać następujące kroki:

1. Usunięcie procedury składowanej `AsyncAnalysis`, ponieważ zależy ona od typu `SalesOrdersForAnalysis`.
2. Usunięcie typu `SalesOrderForAnalysis`.
3. Odtworzenie typu `SalesOrderForAnalysis`.
4. Odtworzenie procedury składowanej `AsyncAnalysis`.

Krótko mówiąc, aktualizacja kodu byłaby o wiele prostsza, gdybyśmy wykorzystali tylko te kolumny, które były nam potrzebne, a nie wszystkie kolumny w tabeli. W mojej karierze wielokrotnie spotykałem się z podobnymi problemami, które wykładniczo zwiększały czas potrzebny na wprowadzenie prostej zmiany.

Ostatni powód unikania podejścia `SELECT *` ma związek z czytelnością kodu. W rozdziale 2. omówiliśmy korzyści płynące z tworzenia samodokumentującego się kodu. Używanie `SELECT *` łamie zasadę samodokumentacji i sprawia, że kod staje się mniej przejrzysty, a także trudniejszy do utrzymania w przyszłości dla Ciebie i innych programistów, ponieważ cel, który chcesz osiągnąć, staje się mniej oczywisty.

Należy unikać stosowania instrukcji `SELECT *` w kodzie, który ma być wdrożony i utrzymywany. Może to negatywnie wpłynąć na wydajność poprzez zwiększenie obciążenia sieci i wymuszenie mniej efektywnych operacji indeksowania przy pobieraniu danych. Utrudnia to również utrzymanie kodu w przypadkach, gdy istnieją zależności w dalszych etapach przetwarzania. Ponadto sprawia, że kod jest mniej zrozumiały, i zaburza model samodokumentacji.

# 5.4. Numer 17 — niepotrzebne porządkowanie danych

Poproszono nas o przygotowanie raportów dotyczących kontaktów z klientami. Uznaliśmy, że dane mogą być bardziej użyteczne, jeśli uporządkujemy je według adresów e-mail. Zanim jednak przejdziemy do analizy, wygenerujmy przykładowe dane dla tabeli `CustomerContacts`. Użyjemy w tym celu skryptu przedstawionego na listingu 5.7, który utworzy 3,2 miliona wierszy danych.

**Listing 5.7. Generowanie danych dla tabeli CustomerContacts**

```
DECLARE @FirstName TABLE (FirstName NVARCHAR(32)) ;

DECLARE @LastName TABLE (LastName NVARCHAR(32)) ;

DECLARE @domain TABLE (Domain NVARCHAR(250)) ;

DECLARE @topleveldomain TABLE (TLD NVARCHAR(6)) ;

DECLARE @email TABLE (Email NVARCHAR(256)) ;

INSERT INTO @FirstName
VALUES
    ('Rachel'),
    ('Seth'),
    ('Tony'),
    ('Angel'),
    ('Isabell'),
    ('Robert'),
    ('Adelaide'),
    ('Jessie'),
    ('Paxton'),
    ('London'),
    ('Jadyn'),
    ('Corey'),
    ('Maximo'),
    ('Johan'),
    ('Mariah'),
    ('Raven'),
    ('Hamza'),
    ('Cristofer'),
    ('Molly'),
    ('Malcolm') ;

INSERT INTO @LastName
VALUES
    ('Hill'),
    ('Acosta'),
    ('Oconnell'),
    ('Jefferson'),
    ('Cross'),
    ('Patel'),
```

```
    ('House'),
    ('Price'),
    ('Morales'),
    ('Reeves'),
    ('Rice'),
    ('Drake'),
    ('Briggs'),
    ('Henry'),
    ('Aguilar'),
    ('Holloway'),
    ('Burnett'),
    ('Aguilar'),
    ('Simon'),
    ('Barry') ;

INSERT INTO @domain
SELECT
    CONCAT(FirstName, LastName)
FROM @FirstName
CROSS JOIN @LastName ;

 INSERT INTO @topleveldomain
 VALUES
    ('.net'),
    ('.com'),
    ('.co.uk'),
    ('.eu'),
    ('.ru'),
    ('.edu'),
    ('.gov'),
    ('.ninja'),
    ('.io'),
    ('.co'),
    ('.ai'),
    ('.ca'),
    ('.me'),
    ('.de'),
    ('.fr'),
    ('.ac'),
    ('.am'),
    ('.ax'),
    ('.ba'),
    ('.ch') ;

INSERT INTO @email
SELECT
    CONCAT(Domain, TLD)
FROM @domain
CROSS JOIN @topleveldomain ;

INSERT INTO dbo.CustomerContacts(
    CustomerContactFirstName,
    CustomerContactLastName,
    CustomerContactEmail
)
SELECT
      FirstName
    , LastName
```

```
    , Email
FROM @FirstName
CROSS JOIN @LastName
CROSS JOIN @email ;
```

Teraz, gdy mamy już trochę danych, zajmijmy się naszym zadaniem i napiszmy kwerendę, która zwróci kolumny `CustomerContactFirstName`, `CustomerContactLast`
`↪Name` i `CustomerContactEmail` z tabeli `CustomerContacts`. Przed uruchomieniem tej kwerendy skrypt z listingu 5.8 wykonuje polecenie `SET STATISTICS TIME ON`, które spowoduje wyświetlenie statystyk czasu wykonania w oknie komunikatów SSMS.

**Listing 5.8. Pobieranie wymaganych danych z tabeli CustomerContacts**

```
SET STATISTICS TIME ON ;

SELECT
      CustomerContactFirstName
    , CustomerContactLastName
    , CustomerContactEmail
FROM dbo.CustomerContacts ;
```

Na moim stanowisku testowym ta kwerenda wykonywała się przez 20 679 ms. Teraz spróbujmy jeszcze raz, ale tym razem posortujemy dane według kolumny `CustomerContactEmailAddress`, co pokazano w przykładzie z listingu 5.9.

**Listing 5.9. Pobieranie z tabeli CustomerContacts danych uporządkowanych według adresu e-mail**

```
SET STATISTICS TIME ON ;

SELECT
      CustomerContactFirstName
    , CustomerContactLastName
    , CustomerContactEmail
FROM dbo.CustomerContacts
ORDER BY CustomerContactEmail ;
```

**WSKAZÓWKA** Jeśli przeprowadzasz pomiary wydajności u siebie, pamiętaj, że Twoje wyniki mogą być inne w zależności od możliwości sprzętowych oraz innych uruchomionych procesów. Odradzam również rejestrowanie planów wykonania jednocześnie z testowaniem wydajności, ponieważ może to wpłynąć na wyniki pomiaru czasu.

Na tym samym stanowisku testowym ta kwerenda wykonywała się przez 27 415 ms, czyli o 25% wolniej niż wersja bez sortowania. Wynika to z faktu, że bazy relacyjne opierają się na gałęzi matematyki zwanej *teorią zbiorów*, w której rozpatrujemy *zbiór* (czyli grupę różnych obiektów) oraz *multizbiór* (czyli kolekcję obiektów, która może zawierać duplikaty). W obu tych koncepcjach matematycznych kolejność wyników nie ma znaczenia. Dlatego sortowanie danych nie jest operacją bazującą na zbiorach, a jedynie operacją prezentacyjną. Z tego powodu powinieneś sortować dane tylko wtedy, gdy jest to naprawdę konieczne.

Możemy to zaobserwować w relatywnym koszcie operacji sortowania, jeśli przyjrzymy się planowi wykonania kwerendy. Plan wykonania to zestaw kroków lub operatorów, które optymalizator zapytań wybrał do realizacji danej kwerendy. Dostęp do planów wykonania można uzyskać na kilka sposobów, w tym poprzez Query Store (o którym będzie mowa w rozdziale 10.) lub metadane (również omówione w rozdziale 10.). Najprostszym sposobem jest kliknięcie przycisku *Include Actual Execution Plan* na pasku narzędzi w SQL Server Management Studio przed uruchomieniem kwerendy.

Plan wykonania kwerendy z listingu 5.9 pokazano na rysunku 5.2. Warto zwrócić uwagę, że operacja sortowania stanowi aż 91% szacowanego kosztu całej kwerendy.


**Rysunek 5.2.** *Koszt operacji sortowania w planie kwerendy*

### Koszt operacji sortowania w planie wykonania

W kontekście planu wykonania kwerendy należy pamiętać o kilku istotnych kwestiach. Po pierwsze, koszt jest szacowany w momencie kompilowania kwerendy, a SQL Server nie aktualizuje go po wykonaniu. Oznacza to, że nawet jeśli wyświetlisz rzeczywisty plan wykonania (w przeciwieństwie do planu szacowanego, który możesz obejrzeć przed wykonaniem kwerendy), zobaczysz koszty szacunkowe. Mogą one być niedokładne z powodu różnych czynników, takich jak nieaktualne statystyki.

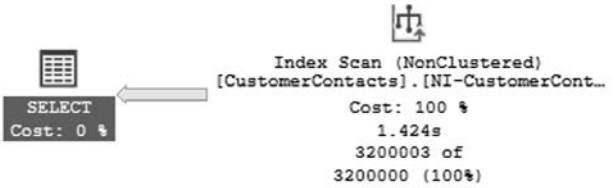
Po drugie, koszt nie jest bezpośrednim miernikiem wydajności. Jest to wartość ważona, obliczana na podstawie zastrzeżonego algorytmu, który szacuje względny koszt operacji procesora i wejścia-wyjścia, a następnie sumuje te wartości, aby uzyskać całkowity koszt operatora, znany jako *koszt poddrzewa*.

Jeśli pojawi się konieczność uporządkowania danych ze względów prezentacyjnych, pomocne mogą okazać się indeksy. W przypadku tej kwerendy idealny indeks pokrywający zostałby zbudowany na `CustomerContactEmail` jako kluczu indeksu z dodatkowymi kolumnami `CustomerContactFirstName` i `CustomerContact` ↪`LastName` na poziomie liści. Taki indeks, który można utworzyć za pomocą polecenia z listingu 5.10, jest już uporządkowany według kolumny `CustomerContact` ↪`Email`, więc nie jest wymagana dodatkowa operacja sortowania. Ponieważ pozostałe potrzebne kolumny znajdują się na poziomie liści, nie ma konieczności wykonywania operacji wyszukiwania w indeksie klastrowym.

**Listing 5.10. Tworzenie indeksu pokrywającego w celu poprawienia wydajności**

```
CREATE NONCLUSTERED INDEX
    [NI-CustomerContactEmail-Include-FirstName-LastName]
        ON dbo.CustomerContacts(CustomerContactEmail)
        INCLUDE(CustomerContactFirstName, CustomerContactLastName) ;
```

Teraz, gdy mamy indeks pokrywający, wykonajmy ponownie kwerendę z listingu 5.9 i zobaczmy, jak wpłynie to na plan wykonania (przedstawiony na rysunku 5.3) oraz czas realizacji. Jak widać, tym razem optymalizator wybrał skanowanie indeksu nieklastrowego. Na moim stanowisku testowym kwerenda wykonała się w ciągu 20 904 ms, co jest w przybliżeniu takim samym czasem jak w przypadku pierwotnej, nieuporządkowanej kwerendy.




**Rysunek 5.3.** *Plan wykonania ze skanowaniem indeksu nieklastrowego*

Choć utworzenie indeksu sprawdziło się dobrze w tej konkretnej kwerendzie, trzeba pamiętać, że nie ma nic za darmo. Indeks rozwiązał problem wydajności kwerendy sortującej dane według adresu e-mail, ale jego obecność spowolni operacje `INSERT`, `UPDATE` i `DELETE` wykonywane na tabeli. Wynika to stąd, że SQL Server będzie musiał aktualizować również indeks nieklastrowy przy każdej zmianie danych.

Dlatego też, jeśli nie jest to absolutnie konieczne, lepiej unikać porządkowania danych, ponieważ wpływa to negatywnie na wydajność. Jeśli jednak musisz uporządkować dane, na przykład ze względów prezentacyjnych, rozważ odpowiednią strategię indeksowania. Pamiętaj jednak, że będzie to miało wpływ na wydajność operacji zapisu do tabeli.

## 5.5. Numer 18 — używanie słowa kluczowego DISTINCT bez ważnej przyczyny

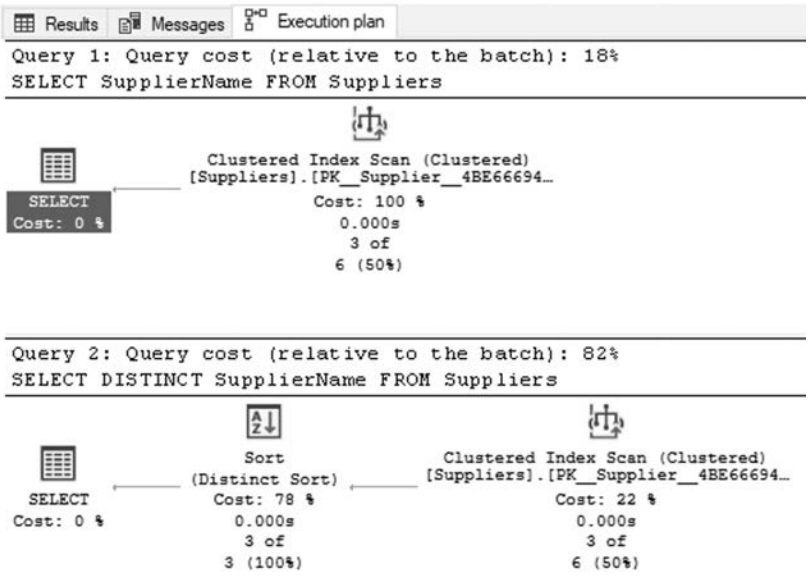
Poproszono nas o napisanie kwerendy zwracającej listę unikatowych dostawców. Zdarza się, że mniej doświadczeni programiści używają słowa kluczowego `DISTINCT`, aby upewnić się, że wyniki są unikatowe. Aby zbadać to zagadnienie, moglibyśmy zastosować dowolną kwerendę z listingu 5.11. Ponieważ (pod warunkiem,

że nie mamy poważnego problemu z jakością danych) ten sam dostawca nie będzie wymieniony dwukrotnie w naszej tabeli, zastosowanie słowa kluczowego `DISTINCT` jest zbędne.

**Listing 5.11. Zwracanie dostawców z użyciem lub bez użycia słowa kluczowego DISTINCT**

```
SELECT SupplierName
FROM dbo.Suppliers ;

SELECT DISTINCT SupplierName
FROM dbo.Suppliers ;
```

Choć słowo kluczowe `DISTINCT` jest tu niepotrzebne, czy zastosowanie go sprawia jakąś różnicę? Aby odpowiedzieć na to pytanie, przyjrzyjmy się planowi wykonania przedstawionemu na rysunku 5.4. Gdyby obie kwerendy miały taki sam koszt, w każdym planie ich względny koszt wynosiłby 50% całej operacji. W tym przypadku jednak widać, że kwerenda z użyciem `DISTINCT` ma względny koszt 82%, co oznacza, że jest znacznie mniej wydajna. Można to zaobserwować w operatorze `Distinct Sort`, który pojawia się w planie wykonania.




**Rysunek 5.4.** *Plany wykonania z użyciem i bez użycia słowa kluczowego DISTINCT*

W niektórych sytuacjach uzyskanie unikatowych wyników może być konieczne. Wyobraźmy sobie na przykład, że mamy za zadanie zwrócić listę unikatowych dostawców, od których firma MagicChoc zakupiła duże zębatki w czerwcu 2023 roku. Możemy uzyskać taką listę dostawców za pomocą kwerendy przedstawionej na listingu 5.12.

**Listing 5.12. Zwracanie listy dostawców dużych zębatek**

```
SELECT DISTINCT s.SupplierName
FROM dbo.Suppliers s
INNER JOIN dbo.PurchaseOrderHeaders poh
    ON poh.SupplierID = s.SupplierID
INNER JOIN dbo.PurchaseOrderDetails pod
    ON pod.PurchaseOrderNumber = poh.PurchaseOrderNumber
WHERE MONTH(poh.PurchaseOrderDate) = 6
    AND YEAR(poh.PurchaseOrderDate) = 2023
AND pod.ProductID = 4 ;
```

Problem w tym, że ponieważ w tym okresie kupiliśmy ten produkt dwukrotnie od firmy Unknown Engineering, kwerenda dwa razy uwzględnia ją w wynikach. Moglibyśmy oczywiście użyć słowa kluczowego `DISTINCT`, ale wiemy, że obniży to wydajność. Czy istnieją jakieś inne rozwiązania?

Trzy kwerendy przedstawione na listingu 5.13 są funkcjonalnie równoważne. Pierwsza wykorzystuje słowo kluczowe `DISTINCT` do usunięcia duplikatów z wyników, druga używa klauzuli `GROUP BY`, a trzecia korzysta z funkcji okienkowej `ROW_NUMBER()`.

**Listing 5.13. Stosowanie słowa kluczowego DISTINCT**

```
SELECT DISTINCT s.SupplierName
FROM dbo.Suppliers s
INNER JOIN dbo.PurchaseOrderHeaders poh
    ON poh.SupplierID = s.SupplierID
INNER JOIN dbo.PurchaseOrderDetails pod
    ON pod.PurchaseOrderNumber = poh.PurchaseOrderNumber
WHERE MONTH(poh.PurchaseOrderDate) = 6
    AND YEAR(poh.PurchaseOrderDate) = 2023
AND pod.ProductID = 4 ;

SELECT s.SupplierName
FROM dbo.Suppliers s
INNER JOIN dbo.PurchaseOrderHeaders poh
    ON poh.SupplierID = s.SupplierID
INNER JOIN dbo.PurchaseOrderDetails pod
    ON pod.PurchaseOrderNumber = poh.PurchaseOrderNumber
WHERE MONTH(poh.PurchaseOrderDate) = 6
    AND YEAR(poh.PurchaseOrderDate) = 2023
AND pod.ProductID = 4
GROUP BY s.SupplierName ;

SELECT SupplierName FROM (
    SELECT s.SupplierName, ROW_NUMBER() OVER(ORDER BY s.SupplierName) AS rn
    FROM dbo.Suppliers s
    INNER JOIN dbo.PurchaseOrderHeaders poh
        ON poh.SupplierID = s.SupplierID
    INNER JOIN dbo.PurchaseOrderDetails pod
        ON pod.PurchaseOrderNumber = poh.PurchaseOrderNumber
    WHERE MONTH(poh.PurchaseOrderDate) = 6
        AND YEAR(poh.PurchaseOrderDate) = 2023
    AND pod.ProductID = 4
) a WHERE rn = 1 ;
```

W tym konkretnym przypadku, ze względu na niewielką liczbę wierszy oraz brak obciążenia mojego środowiska testowego, nie zaobserwowałem różnicy w wydajności między trzema wariantami. W wersjach wykorzystujących `DISTINCT` i `GROUP BY` wygenerowano identyczny plan wykonania kwerendy.

W środowisku produkcyjnym, gdzie mamy do czynienia ze złożonymi kwerendami i dużymi ilościami danych, może się okazać, że otrzymujemy trzy zupełnie różne plany wykonania, a niektóre z nich są znacznie wydajniejsze od pozostałych. Jeśli więc napotkasz problemy z wydajnością przy użyciu słowa kluczowego `DISTINCT`, wypróbuj dwa pozostałe podejścia, aby sprawdzić, czy możliwe jest poprawienie wydajności.

**WSKAZÓWKA** Mógłbym sprawić, że kwerenda z funkcją `ROW_NUMBER()` przewyższyłaby wydajnością kwerendy `DISTINCT` i `GROUP BY` z poprzedniego przykładu — wystarczyłoby dodać więcej danych do tabeli. Ten przykład pokazuje jednak, jak ważne jest testowanie wydajności na realistycznych danych. Zagadnienie to omówimy szerzej w rozdziale 7.

Nie powinniśmy używać słowa kluczowego `DISTINCT` bez wyraźnej potrzeby. Jeśli jednak wyniki kwerendy rzeczywiście muszą być unikatowe, a `DISTINCT` powoduje problemy z wydajnością, możemy rozważyć inne podejścia. Jeśli słowo kluczowe `DISTINCT` jest naprawdę niezbędne i nie zaobserwowano żadnych problemów z wydajnością, zalecałbym trzymanie się tego rozwiązania, ponieważ jest ono najbardziej przejrzyste spośród trzech opcji. Od razu widać, co robimy, a to pomaga w tworzeniu samodokumentującego się kodu.

**WSKAZÓWKA** Rzeczywista potrzeba użycia `DISTINCT` może wynikać z problemów w strukturze bazy danych. Więcej informacji na ten temat znajdziesz w rozdziale 4.

## 5.6. Numer 19 — niepotrzebne stosowanie klauzuli UNION

Podobnie jak w przypadku niepotrzebnego porządkowania danych czy usuwania duplikatów, początkujący programiści SQL Servera często popełniają pomyłkę przy korzystaniu z klauzuli `UNION`. Unia to sposób na poziome łączenie dwóch zbiorów wyników, co można uzyskać na dwa różne sposoby. Ściślej, można użyć klauzuli `UNION` lub `UNION ALL`.

Różnica między klauzulami `UNION` a `UNION ALL` polega na tym, że `UNION ALL` zwraca wszystkie wyniki z obu kwerend, natomiast `UNION` usuwa duplikaty z wyników. Wyobraźmy sobie na przykład, że poproszono nas o zestawienie listy kontaktów firmy MagicChoc z tabel `CustomerContacts` i `SupplierContacts`. Pierwsza kwerenda z listingu 5.14 wykorzystuje klauzulę `UNION` do utworzenia unikatowej

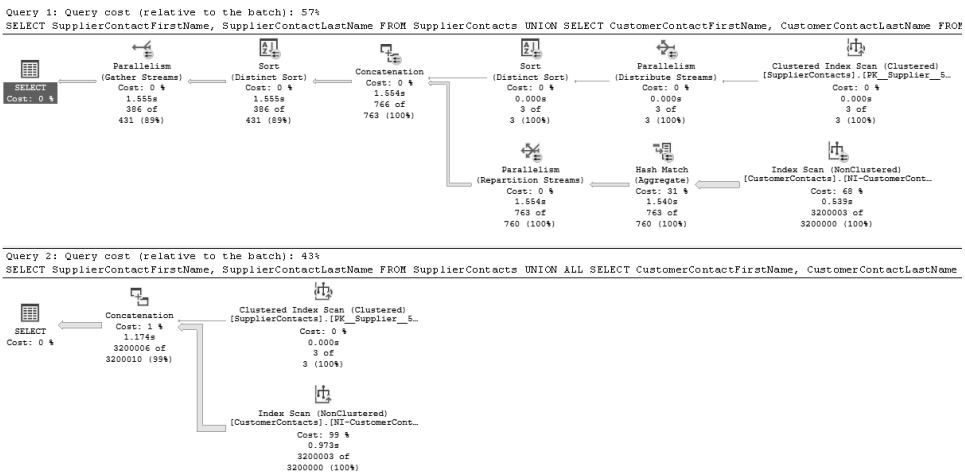
listy kontaktów. Druga używa `UNION ALL` do utworzenia listy, która może zawierać duplikaty.

> **WSKAZÓWKA** Istnieją dodatkowe operatory łączenia poziomego o nazwach `INTERSECT` i `EXCEPT`. `INTERSECT` zwraca wyniki z pierwszej kwerendy, które pojawiają się również w wynikach drugiej kwerendy. Z kolei `EXCEPT` zwraca wyniki z pierwszej kwerendy, których nie ma w wynikach drugiej kwerendy.

**Listing 5.14. Tworzenie list kontaktów z duplikatami i bez duplikatów**

```
SELECT SupplierContactFirstName, SupplierContactLastName
FROM dbo.SupplierContacts
UNION
SELECT CustomerContactFirstName, CustomerContactLastName
FROM dbo.CustomerContacts ;

SELECT SupplierContactFirstName, SupplierContactLastName
FROM dbo.SupplierContacts
UNION ALL
SELECT CustomerContactFirstName, CustomerContactLastName
FROM dbo.CustomerContacts ;
```

Analizując plany wykonania przedstawione na rysunku 5.5, możemy zauważyć, że koszt usuwania duplikatów z listy był znacznie wyższy. Warto więc zastanowić się nad zasadnością tego wymagania. Czy rzeczywiście musimy eliminować powtórzenia? Jeśli nie jest to konieczne, nie powinniśmy tego robić tylko dla zasady.




**Rysunek 5.5.** *Plany wykonania dla złączeń poziomych*

Czasem trzeba użyć operacji `UNION`, ale należy to robić tylko wtedy, gdy jest to naprawdę konieczne. Jeśli duplikaty nie są istotne lub nie mogą wystąpić ze względu na logikę biznesową, lepiej zastosować `UNION ALL`. Jest to bardziej wydajne rozwiązanie, ponieważ pomija krok usuwania duplikatów.

# 5.7. Numer 20 — używanie kursorów

Kierownik działu zaopatrzenia w firmie MagicChoc poprosił nas o przygotowanie raportu pokazującego ilość produktów w magazynie, pogrupowanych według kategorii. Zamiast standardowego układu pionowego, raport ma być przedstawiony w formacie poziomym, gdzie nazwy kolumn odpowiadają kategoriom produktów, a pojedynczy wiersz zawiera informacje o ilości produktów w magazynie dla każdej kategorii.

Pomyłką, którą wielu programistów popełnia na tym etapie, jest użycie kursora. Kursor to mechanizm w T-SQL służący do przetwarzania zbioru wierszy, jeden po drugim. Kursorów można używać do różnych celów, w tym do przestawiania danych oraz generowania i wykonywania dynamicznych skryptów T-SQL, takich jak uruchamianie polecenia dla każdej tabeli. Można ich również użyć do wyszukiwania wartości w dowolnej kolumnie tabeli lub do tworzenia rankingu danych.

Problem w tym, że kursory są niezwykle nieefektywnym sposobem przetwarzania danych relacyjnych. Każda iteracja kursora wiąże się z takim samym narzutem jak wykonanie pojedynczego polecenia. Na przykład, jeśli masz kursor, który przechodzi przez milion wierszy, będzie to miało taki sam koszt jak wykonanie miliona osobnych kwerend. Co więcej, rozwój języka T-SQL w ciągu ostatnich 25 lat sprawił, że kursory stały się niepotrzebne. Nie przychodzi mi do głowy ani jedna sytuacja, w której kursor byłby niezbędny do osiągnięcia pożądanego rezultatu. Zresztą (miejmy nadzieję) prawdopodobnie Twoje standardy kodowania i tak zabraniają stosowania kursorów.

> **WSKAZÓWKA** Nawet administratorzy, którzy wcześniej używali kursorów do iteracji po obiektach w bazie danych, nie mają już powodu, żeby to robić. Omówimy to szerzej w rozdziale 9.

Wracając do naszego scenariusza — gdybyśmy chcieli użyć kursora do wygenerowania raportu przestawnego, moglibyśmy zastosować skrypt podobny do tego z listingu 5.15. Skrypt ten najpierw tworzy tymczasową tabelę o strukturze odpowiadającej naszemu końcowemu raportowi. Następnie wstawiamy wiersz z zerami w każdej kolumnie, który będzie podstawą do późniejszej aktualizacji. Podczas deklarowania zmiennych definiujemy również kursor, który będzie zawierał pełny zestaw wyników do przetworzenia. W naszym przypadku jest to lista kategorii produktów i ilości w formacie tabelarycznym. Otwieramy kursor i używamy polecenia `FETCH`, aby pobrać pierwszy wiersz. Pętla `WHILE` określa działania, które chcemy wykonać na kursorze — w tym przypadku aktualizację odpowiedniej kolumny w tabeli tymczasowej na podstawie wartości

z kursora. Na końcu pętli WHILE pobieramy kolejny wiersz do kursora. Pętla kończy się, gdy @@FETCH_STATUS = 0, co oznacza, że nie ma już więcej wierszy do pobrania. Na koniec po prostu wykonujemy kwerendę SELECT z tabeli tymczasowej i usuwamy obiekty tymczasowe, aby nie zaśmiecały pamięci.

**Listing 5.15. Używanie kursora do przestawienia danych**

```
CREATE TABLE #Categories (
    [Raw Ingredients]               INT,
    [Machine Parts]                 INT,
    [Misc]                          INT,
    [Confectionary Products]        INT,
    [Non-confectionary Products]    INT
) ;

INSERT INTO #Categories
VALUES (0,0,0,0,0) ;

DECLARE @Category as varchar(32) ;
DECLARE @Stock as varchar(32) ;

DECLARE product_cursor CURSOR FOR
SELECT
    pc.ProductCategoryName
    , SUM(ISNULL(p.ProductStockLevel,0)) Stock
FROM dbo.ProductCategories pc
INNER JOIN dbo.ProductSubcategories ps
    ON pc.ProductCategoryID = ps.ProductCategoryID
LEFT JOIN dbo.Products p
    ON ps.ProductSubcategoryID = p.ProductSubcategoryID
GROUP BY ProductCategoryName ;

OPEN product_cursor  ;

FETCH NEXT FROM product_cursor INTO @Category, @Stock  ;

WHILE @@FETCH_STATUS = 0
BEGIN
    IF @Category = 'Raw Ingredients'
        UPDATE #Categories
        SET [Raw Ingredients] =
            [Raw Ingredients] + @Stock
    ELSE IF @Category = 'Machine Parts'
        UPDATE #Categories
        SET [Machine Parts] =
            [Machine Parts] + @Stock
    ELSE IF @Category = 'Misc'
        UPDATE #Categories
        SET [Misc] = [Misc] + @Stock
    ELSE IF @Category = 'Confectionary Products'
        UPDATE #Categories
        SET [Confectionary Products] =
            [Confectionary Products] + @Stock
    ELSE IF @Category = 'Non-confectionary Products'
        UPDATE #Categories
        SET [Non-confectionary Products] =
            [Non-confectionary Products] + @Stock ;
```

```
    FETCH NEXT FROM product_cursor INTO @Category, @Stock  ;
END

SELECT
      [Raw Ingredients]
    , [Machine Parts], [Misc]
    , [Confectionary Products]
    , [Non-confectionary Products]
FROM #Categories ;

CLOSE product_cursor ;

DEALLOCATE product_cursor ;

DROP TABLE #Categories ;
```

Zamiast kosztownego kursora moglibyśmy wykorzystać operator `PIVOT`. Wykona on za nas całą trudną pracę i zrobi to jako operację na zbiorach, co będzie znacznie wydajniejsze niż kursor.

### Przykład z życia wzięty

Około dziesięciu lat temu współpracowałem z jedną z największych na świecie firm reklamowych, która operowała na ogromnych zbiorach danych pozyskiwanych z wyszukiwarek internetowych i od dostawców usług cookie. Poproszono mnie o przyjrzenie się problemowi z wydajnością, który wystąpił w zadaniu ETL przekształcającym duży zbiór danych.

W pierwotnej implementacji wykorzystano kursor. Przepisałem ten proces, używając instrukcji `PIVOT` zamiast kursora. Dzięki temu czas wykonania skrócił się z ponad 3 godzin do zaledwie 48 sekund!

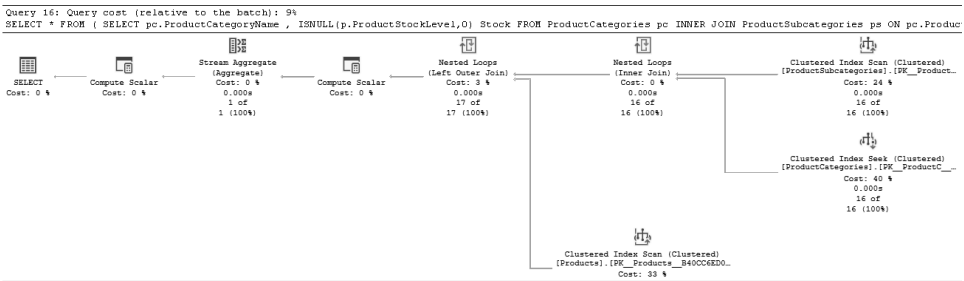
Pojedyncza kwerenda z listingu 5.16 osiąga ten sam rezultat co kursor. Kwerenda zewnętrzna określa kolumny, które chcemy zwrócić. Możemy użyć gwiazdki (*) lub podać listę konkretnych kolumn. Jeśli użyjemy listy kolumn, nie ma obowiązku wybierania wszystkich przestawionych kolumn. Podkwerenda pobiera płaską listę kategorii i stanów magazynowych. Na końcu operator `PIVOT` definiuje ostateczny zbiór wyników, określając agregację, którą chcemy zastosować do kolumny ze stanami magazynowymi, oraz wartości z kolumny `ProductCategoryName`, które mają stać się naszymi przestawionymi kolumnami.

**Listing 5.16. Przestawianie danych za pomocą operatora PIVOT**

```
SELECT
      [Raw Ingredients]
    , [Machine Parts]
    , [Misc]
    , [Confectionary Products]
    , [Non-confectionary Products]
FROM (
    SELECT
          pc.ProductCategoryName
```

```
        , ISNULL(p.ProductStockLevel,0) Stock
    FROM dbo.ProductCategories pc
    INNER JOIN dbo.ProductSubcategories ps
        ON pc.ProductCategoryID = ps.ProductCategoryID
    LEFT JOIN dbo.Products p
        ON ps.ProductSubcategoryID = p.ProductSubcategoryID
) AS WorkingTable
PIVOT
(
    SUM(Stock)
    FOR ProductCategoryName IN (
        [Raw Ingredients],
        [Machine Parts],
        [Misc],
        [Confectionary Products],
        [Non-confectionary Products]
    )
) AS PivotTable ;
```

Aby zademonstrować różnicę w koszcie kwerend, możemy skopiować kwerendę `PIVOT` do tego samego okna co nasza operacja z kursorem. Następnie, analizując plan wykonania dla tego polecenia, zauważymy, że stanowi ono jedynie 9% kosztu całej partii, co oznacza że podejście oparte na kursorach zostało oszacowane jako ponad dziesięciokrotnie mniej wydajne niż użycie operatora `PIVOT`. Odpowiedni fragment planu wykonania pokazano na rysunku 5.6.




**Rysunek 5.6.** *Koszt kwerendy używającej operatora PIVOT względem równoważnego kursora*

# 5.8. Numer 21 — usuwanie wielu wierszy w jednej transakcji

Programiści SQL często są proszeni o usunięcie starych danych z tabel. Zwykle po takiej operacji następuje zmniejszenie rozmiaru bazy danych, co pozwala odzyskać miejsce na dysku. Do tematu zmniejszania baz danych powrócimy jeszcze w rozdziałach 9. i 11.

Najbardziej efektywnym sposobem usunięcia dużej ilości danych z tabeli jest użycie polecenia `TRUNCATE TABLE`. Polecenie to usuwa dane z tabeli, zachowując

jednocześnie jej strukturę poprzez zwolnienie stron danych. Problem polega na tym, że operacja TRUNCATE nie pozwala na zastosowanie klauzuli WHERE. Jest to operacja „wszystko albo nic", co oznacza, że musimy usunąć wszystkie wiersze naraz.

Nawet jeśli chcesz usunąć wszystkie wiersze z tabeli, istnieją pewne ograniczenia związane z operacją TRUNCATE. Nie możesz na przykład wykonać tej operacji, jeśli kolumna w tabeli jest powiązana kluczem obcym lub *ograniczeniami krawędziowymi*, które egzekwują semantykę i zapewniają integralność *tabel krawędziowych* reprezentujących relacje w grafowej bazie danych. Ponadto nie można użyć TRUNCATE, jeśli tabela jest tabelą bazową w widoku indeksowanym, uczestniczy w replikacji transakcyjnej lub scalającej albo jest *tabelą temporalną z wersjonowaniem systemowym*, która służy do śledzenia pełnej historii zmian danych innej tabeli w celu umożliwienia analizy w określonym punkcie w czasie.

Te ograniczenia często zmuszają programistów do korzystania z instrukcji DELETE w celu usunięcia dużych ilości wierszy z tabel. Błąd, który wielokrotnie obserwowałem, polega na próbie usunięcia wszystkich wierszy z tabeli w ramach jednej transakcji. Wyobraźmy sobie na przykład tabelę zawierającą wiele milionów (lub nawet miliardów) wierszy.

Skrypt przedstawiony na listingu 5.17 tworzy tabelę i wypełnia ją ogromną liczbą wierszy. Liczba wygenerowanych wierszy będzie się różnić w zależności od tabel i kolumn w Twojej bazie danych, ale na moim stanowisku testowym jest to niespełna 3,5 miliarda wierszy.

**OSTRZEŻENIE** Wykonanie tego skryptu może zająć dużo czasu.

**Listing 5.17. Tworzenie bardzo dużej tabeli**

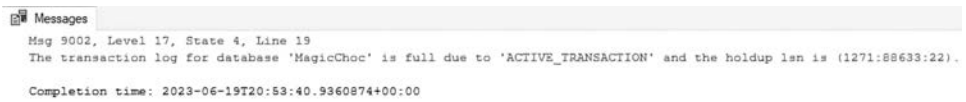
```
CREATE TABLE dbo.VeryLargeTable (
    ID         BIGINT            IDENTITY    PRIMARY KEY,
    TextCol    NVARCHAR(4000)
) ;

DECLARE @LoopCounter INT = 0 ;

WHILE @LoopCounter < 2000
BEGIN
    INSERT INTO dbo.VeryLargeTable (TextCol)
    SELECT 'Kolejny wiersz w bardzo, bardzo, bardzo dużej tabeli. W rzeczywistości
    ↪tworzenie tej tabeli zajmie bardzo dużo czasu i nie będziesz mógł usunąć wszystkich
    ↪wierszy za jednym zamachem!'
    FROM sys.columns c1
    CROSS APPLY sys.columns c2 ;

    SET @LoopCounter = @LoopCounter + 1 ;
END
```

Spróbujmy usunąć wszystkie wiersze z tej tabeli w ramach jednej transakcji, używając kwerendy przedstawionej na listingu 5.18. Pamiętaj, że jeśli nie rozpoczniemy jawnej transakcji, każde polecenie zostanie wykonane w ramach transakcji automatycznie zatwierdzanej. Innymi słowy, najmniejszą jednostką transakcji jest pojedyncze polecenie.

**Listing 5.18. Usuwanie wierszy w pojedynczej transakcji**

```
DELETE FROM dbo.VeryLargeTable ;
```

Ta kwerenda spowoduje utworzenie bardzo dużej transakcji. Gdy transakcja jest otwarta, dziennik transakcji nie może zostać przycięty w celu zwolnienia miejsca. Dotyczy to nawet modelu przywracania SIMPLE (o modelach przywracania będziemy mówić w rozdziale 12.). Transakcja stanie się tak duża, że wyczerpie całe dostępne miejsce w dzienniku, co spowoduje jej wycofanie i wygenerowanie błędu 9002, jak pokazano na rysunku 5.7.




**Rysunek 5.7.** *Błąd 9002 zgłaszany po zapełnieniu dziennika transakcji*

Zamiast tego musimy podzielić operację DELETE na wiele mniejszych instrukcji, a tym samym na wiele transakcji. Przy założeniu, że baza danych działa w modelu przywracania SIMPLE, zapobiegnie to zapełnieniu dziennika transakcji, ponieważ będzie można go przycinać pomiędzy wykonywaniem poszczególnych instrukcji. Jest to jeden z niewielu scenariuszy, w których rozważyłbym użycie pętli WHILE w środowisku produkcyjnym, ale tylko w przypadku skryptów doraźnych. W kodzie przeznaczonym do wdrożenia zawsze unikam stosowania pętli WHILE, z tych samych powodów, dla których unikam używania kursorów. W przykładzie z listingu 5.19 ustawiliśmy skrypt tak, aby usuwał wiersze partiami po 250 000. Możesz dostosować tę liczbę, aby zoptymalizować wydajność w zależności od specyfiki Twojego środowiska.

**OSTRZEŻENIE** Wykonanie tego skryptu zajmie dużo czasu.

**Listing 5.19. Usuwanie wierszy partiami**

```
DECLARE @RowCounter BIGINT ;

SET @RowCounter = 1 ;

WHILE @RowCounter > 0
BEGIN
    DELETE TOP(250000)
```

```
    FROM dbo.VeryLargeTable ;

    SET @RowCounter = (SELECT COUNT(*) FROM dbo.VeryLargeTable) ;
    PRINT @RowCounter ;
END
```

Operacje modyfikujące dane, takie jak operacje `DELETE`, mogą spowodować zapełnienie dziennika transakcji, gdy są wykonywane na dużych tabelach. Aby uniknąć tego problemu, należy podzielić operację na mniejsze części i wykonywać je iteracyjnie.

# Podsumowanie

- Pamiętaj, że `NULL` jest wartością nieznaną i dlatego nie jest równa innej wartości `NULL`.
- Unikaj stosowania wskazówki `NOLOCK` w celu zoptymalizowania wydajności, ponieważ może to prowadzić do nieoczekiwanych rezultatów, zwracając dane, które nigdy nie istniały w bazie danych.
- Unikaj używania instrukcji `SELECT *` w kwerendach innych niż doraźne, ponieważ może to powodować problemy z wydajnością i utrudniać konserwację kodu. Jest to również antywzorzec w kontekście tworzenia samodokumentującego się kodu.
- Porządkowanie danych to cecha prezentacyjna, a nie oparta na zbiorach. Dane należy porządkować tylko wtedy, gdy jest to absolutnie konieczne.
- Nie używaj `DISTINCT` do usuwania duplikatów, jeśli nie jest to niezbędne. Jeśli `DISTINCT` powoduje problemy z wydajnością, rozważ inne techniki, takie jak `GROUP BY` lub `ROW_NUMBER()`.
- Instrukcja `UNION` jest bardziej kosztowna obliczeniowo niż `UNION ALL`, ponieważ usuwa duplikaty. Dlatego jeśli duplikaty nie są istotne lub nie mogą wystąpić, lepiej zastosować `UNION ALL` zamiast `UNION`.
- Należy unikać stosowania kursorów. Są one bardzo kosztowne, a w nowoczesnych wersjach SQL Servera nie ma operacji, których nie można by wykonać innymi metodami.
- Usuwanie wielu wierszy z tabeli w ramach jednej transakcji może spowodować zapełnienie dziennika transakcji. Aby tego uniknąć, podziel operację usuwania na mniejsze partie.

# Skorowidz

## A



## B



## C

## D



## E



## F

## G



## H



## I



## J



## K

## L



## Ł



## M

## N



## O



## P

## Q



## R



## S

## Ś



## T

## U



## V



## W

## X



## Z



## Ź

Dlaczego zapytania T-SQL działają tak wolno? Czy mamy dostęp do odpowiednich danych? Czy jesteśmy zabezpieczeni przed kradzieżą danych? Co z nową instancją serwera, którą musimy zarządzać? Takie pytania zadają sobie administratorzy SQL Servera i nawet najlepszym z nich zdarza się popełniać kosztowne błędy.

*Praktyczny przewodnik po pułapkach administracyjnych i programistycznych!*

— **Josephine Bush**, sqlkitty.com

Tę książkę docenią administratorzy, programiści i specjaliści, którzy pracują z SQL Serverem. W odróżnieniu od większości pozycji na ten temat nie opisuje ona pracy systemu w idealnych warunkach. Zamiast tego zwraca uwagę na powszechne błędne przekonania i nieprawidłowe konfiguracje. Znajdziesz tu wyjaśnienie, gdzie są źródła problemów i jak ich unikać. Dzięki praktycznym przykładom, fragmentom kodu i przejrzystym diagramom łatwiej zrozumiesz każdy błąd i jego rozwiązanie. Dowiesz się, jak pisać wydajny kod, projektować efektywne schematy baz danych, implementować obsługę błędów, pracować ze złożonymi typami danych — i wiele więcej. Wszystko to w przyjaznym formacie problem – rozwiązanie.

*Warta przeczytania dla samego rozdziału o testowaniu i debugowaniu!*

— **Mike McQuillan**, McQTech Ltd

## W książce:

- **programowanie w T-SQL**
- **instalacja, administrowanie i optymalizacja**
- **wysoka dostępność i bezpieczeństwo**

*Szybkie, praktyczne rady!*

— **Ruben Vandeginste**, PeopleWare

**Peter A. Carter** ma dwudziestoletnie doświadczenie jako programista, administrator, architekt, instruktor i autor książek o SQL Serverze. Obecnie kieruje zespołem inżynierów platformy. Napisał wiele książek poświęconych różnym aspektom SQL Servera, od administracji i bezpieczeństwa po automatyzację i zaawansowane typy danych.

*Mistrzowska robota! Oto obowiązkowe do opanowania niuanse SQL Servera!*

— **Allen White**, MVP SQL Server w latach 2007–2022

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-2896-1
9 788328 928961
Cena: 119,00 zł