

SQL

Set
vet

Peter A. Carter



100

najczęstszych błędów
i jak ich skutecznie unikać

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

© Helion S.A. 2026

Authorized translation of the English edition © 2025 Manning Publications. This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

<i>Wstęp</i>	11
<i>Podziękowania</i>	13
<i>O książce</i>	15
<i>O autorze</i>	19
1. Wprowadzenie do SQL Servera	21
1.1. Pomyłka indeksowa związana z SQL Serverem (pomyłka numer 0)	22
1.2. Przegląd SQL Servera	23
1.2.1. Przegląd silnika bazy danych	26
1.2.2. Platformy heterogeniczne	30
1.3. Dlaczego SQL Server jest wciąż wart zachodu?	32
1.4. Dlaczego właściwe korzystanie z SQL Servera jest ważne?	33
Podsumowanie	34
2. Standardy programowania	35
2.1. Przykładowa pomyłka	36
2.2. Numer 1 — nieopisowe nazwy obiektów	37
2.3. Numer 2 — używanie prefiksów obiektowych	44
2.4. Numer 3 — niesławny prefiks sp_	48
2.5. Numer 4 — nieznajdowanie czasu na standardy kodowania	52
2.6. Numer 5 — używanie porządkowej pozycji kolumny	54
Podsumowanie	55
3. Typy danych	56
3.1. Numer 6 — przechowywanie liczb całkowitych zawsze jako typu INT	58
3.2. Numer 7 — używanie wyłącznie łańcuchów o zmiennej długości	62
3.3. Numer 8 — pisanie własnego kodu hierarchii	64

3.4.	Numer 9 — nieprzechowywanie danych XML w natywnym formacie	73
3.4.1.	<i>Szatkowanie XML</i>	73
3.4.2.	<i>Rekonstruowanie XML</i>	80
3.4.3.	<i>Unikanie dodatkowych kosztów przez przechowywanie danych w formacie XML</i>	81
3.5.	Numer 10 — ignorowanie JSON	84
	Podsumowanie	88
4.	<i>Projekt bazy danych</i>	90
4.1.	Numer 11 — brak normalizacji	92
4.1.1.	<i>Projektowanie schematu na podstawie własnego osądu</i>	93
4.1.2.	<i>Problemy z naszym schematem bazy danych</i>	98
4.1.3.	<i>Projektowanie schematu bazy danych z wykorzystaniem normalizacji</i>	100
4.2.	Numer 12 — używanie szerokiego klucza podstawowego	117
4.3.	Numer 13 — nieużywanie kluczy obcych	123
	Podsumowanie	127
5.	<i>Programowanie w języku T-SQL</i>	128
5.1.	Numer 14 — niepoprawna obsługa wartości NULL	129
5.2.	Numer 15 — używanie NOLOCK jako sposobu na poprawę wydajności	131
5.3.	Numer 16 — standardowe stosowanie instrukcji SELECT *	133
5.4.	Numer 17 — niepotrzebne porządkowanie danych	136
5.5.	Numer 18 — używanie słowa kluczowego DISTINCT bez ważnej przyczyny	140
5.6.	Numer 19 — niepotrzebne stosowanie klauzuli UNION	143
5.7.	Numer 20 — używanie kursorów	145
5.8.	Numer 21 — usuwanie wielu wierszy w jednej transakcji	148
	Podsumowanie	151
6.	<i>Programowanie w SSIS</i>	152
6.1.	Numer 22 — odrzucanie niepoprawnych danych	156
6.2.	Numer 23 — nieoptymalizowanie wczytywania danych	162
6.3.	Numer 24 — używanie SSIS jako narzędzia do koordynacji T-SQL	165
6.3.1.	<i>Koordynowanie zadań Execute T-SQL Statement</i>	166
6.3.2.	<i>Przekształcanie zadań Execute T-SQL Statement w przepływ danych</i>	169

6.4.	Numer 25 — pobieranie wszystkich danych, kiedy potrzebny jest tylko ich podzbiór	175
	Podsumowanie	177
7.	<i>Obsługa błędów, testowanie, kontrola wersji i wdrażanie</i>	<i>....179</i>
7.1.	Numer 26 — pisanie kodu bez obsługi błędów	180
7.2.	Numer 27 — niealarmowanie o błędach	194
7.3.	Numer 28 — nieużywanie funkcji debugowania	198
7.4.	Numer 29 — niestosowanie narzędzia Schema Compare	203
7.5.	Numer 30 — niepisanie testów jednostkowych	206
7.6.	Nowoczesne techniki programowania	209
7.6.1.	Numer 31 — niekorzystanie z kontroli wersji kodu źródłowego	211
7.6.2.	Numer 32 — nieużywanie potoku CI/CD do wdrażania kodu	215
	Podsumowanie	217
8.	<i>Instalacja SQL Servera</i>	<i>..... 219</i>
8.1.	Numer 33 — stosowanie niejasnych nazw instancji	220
8.2.	Numer 34 — bezkrytyczne używanie systemu Windows	221
8.3.	Numer 35 — zapominanie, jak użyteczne mogą być kontenery	224
8.4.	Numer 36 — niepotrzebne używanie środowiska Desktop Experience	226
8.5.	Numer 37 — bezkrytyczne stosowanie wersji Enterprise Edition	228
8.6.	Numer 38 — instalowanie instancji, kiedy wystarczyłoby rozwiązanie DBaaS lub PaaS	231
8.7.	Numer 39 — instalowanie wszystkich funkcji	233
8.8.	Numer 40 — niestosowanie skryptów do instalacji SQL Servera	236
8.9.	Numer 41 — zakładanie, że zarządzanie konfiguracją nie dotyczy SQL Servera	242
8.10.	Numer 42 — używanie chmurowych obrazów SQL Servera bez ich modyfikowania	248
	Podsumowanie	251
9.	<i>Zarządzanie instancją i bazą danych</i>	<i>.....254</i>
9.1.	Numer 43 — automatyczne zmniejszanie baz danych	255
9.2.	Numer 44 — zaniedbywanie odbudowy indeksów po zmniejszeniu pliku danych	257

9.3.	Numer 45 — poleganie na automatycznym powiększaniu	264
9.4.	Numer 46 — używanie wielu plików dziennika	266
9.5.	Numer 47 — dopuszczanie do fragmentacji dzienników	268
9.6.	Numer 48 — zaniedbywanie planowania wydajności	271
9.7.	Numer 49 — umieszczanie TempDB i plików dziennika zawsze na osobnych dyskach	275
9.8.	Numer 50 — zaniedbywanie regularnego sprawdzania bazy pod kątem uszkodzeń	276
9.9.	Numer 51 — zaniedbywanie automatyzacji	279
9.10.	Numer 52 — używanie kursorów do celów administracyjnych	280
9.11.	Numer 53 — nieinstalowanie poprawek	283
	Podsumowanie	285
10.	Optymalizacja	287
10.1.	Numer 54 — włączanie flag TF1117 i TF1118	288
10.2.	Numer 55 — niestosowanie natychmiastowej inicjalizacji plików	291
10.3.	Numer 56 — niepozostawianie wystarczającej ilości pamięci dla innych aplikacji	293
10.4.	Numer 57 — nieblokowanie stron w pamięci	294
10.5.	Numer 58 — działanie wbrew optymalizatorowi	296
10.6.	Numer 59 — niewykorzystywanie informacji zwrotnych DOP	300
10.7.	Numer 60 — niepartycjonowanie dużych tabel	303
10.8.	Numer 61 — niezrozumienie ograniczeń eliminowania partycji	308
10.9.	Numer 62 — niekompresowanie dużych tabel	311
10.10.	Numer 63 — używanie poziomu izolacji Read Uncommitted	315
10.11.	Numer 64 — używanie nadmiernie restrykcyjnych poziomów izolacji	316
10.12.	Numer 65 — nieuwzględnianie optymistycznych poziomów izolacji	320
10.13.	Numer 66 — rozwiązywanie problemów przez dokładanie sprzętu	322
	Podsumowanie	324

11.	<i>Indeksy</i>	<i>327</i>
11.1.	Numer 67 — zakładanie, że fragmentacja wewnętrzna jest zawsze niekorzystna	329
11.2.	Numer 68 — pogląd, że fragmentacja zewnętrzna powoduje problemy we wszystkich kwerendach	333
11.3.	Numer 69 — reorganizowanie indeksów w celu poprawienia gęstości stron	336
11.4.	Numer 70 — błędne interpretowanie statystyk fragmentacji	339
11.5.	Numer 71 — nieprzebudowywanie indeksów	341
11.6.	Numer 72 — bezkrytyczne przebudowywanie wszystkich indeksów	344
11.7.	Numer 73 — aktualizowanie statystyk po przebudowaniu indeksów	346
11.8.	Numer 74 — nieoptymalizowanie konserwacji indeksów zgodnie z własnymi potrzebami	348
	11.8.1. Parametr MAXDOP	349
	11.8.2. Opcja SORT_IN_TEMPDB	350
	11.8.3. Opcja OPTIMIZE_FOR_SEQUENTIAL_KEY	351
11.9.	Numer 75 — niewyłączanie indeksów podczas hurtowego wczytywania danych	352
11.10.	Numer 76 — nadmierne poleganie na narzędziu Database Engine Tuning Advisor	355
11.11.	Numer 77 — nieużywanie indeksów kolumnowych	356
	Podsumowanie	360
12.	<i>Kopie zapasowe</i>	<i>363</i>
12.1.	Numer 78 — nieuwzględnianie RPO i RTO	365
12.2.	Numer 79 — stosowanie migawek bazy danych jako strategii przywracania	368
12.3.	Numer 80 — używanie migawek zachowujących spójność w razie awarii jako strategii przywracania danych	370
12.4.	Numer 81 — nietestowanie kopii zapasowych	372
	12.4.1. Weryfikacja poprawności wykonania kopii zapasowych	372
	12.4.2. Weryfikacja integralności kopii zapasowej	375
12.5.	Numer 82 — tworzenie kopii zapasowych podczas okna ETL	377
12.6.	Numer 83 — stosowanie wyłącznie modelu przywracania FULL w hurtowniach danych i systemach deweloperskich	382
12.7.	Numer 84 — używanie modelu SIMPLE dla baz OLTP	384

12.8.	Numer 85 — nietworzenie kopii zapasowej po zmianie modelu przywracania	388
12.9.	Numer 86 — planowanie backupu dziennika natychmiast po backupie pełnym	389
12.10.	Numer 87 — nieużywanie backupu COPY_ONLY do tworzenia doraźnych kopii zapasowych	391
12.11.	Numer 88 — zapominanie, że kopie zapasowe są częścią systemu bezpieczeństwa	392
	Podsumowanie	395
13.	<i>Dostępność</i>	397
13.1.	Numer 89 — mylenie HA z DR	398
13.2.	Numer 90 — nieprojektowanie architektury pod kątem wymagań	402
13.3.	Numer 91 — nietestowanie strategii DR	405
13.4.	Numer 92 — zakładanie, że grupy dostępności zawsze są właściwym rozwiązaniem	407
13.5.	Numer 93 — przeciążanie klastra	410
	Podsumowanie	412
14.	<i>Bezpieczeństwo</i>	414
14.1.	Numer 94 — niestosowanie zasady najmniejszych przywilejów	416
14.2.	Numer 95 — niewyłączanie konta sa	419
14.3.	Numer 96 — używanie niewłaściwej ziarnistości konta usługowego	421
14.4.	Numer 97 — włączanie procedury xp_cmdshell	422
14.5.	Numer 98 — nieaudytowanie działań administracyjnych	426
14.6.	Numer 99 — narażanie firmy na ataki polegające na podstawianiu całych wartości	430
	14.6.1. Przygotowywanie zaszyfrowanego środowiska	431
	14.6.2. Zapobieganie atakom polegającym na podstawianiu całych wartości	434
14.7.	Numer 100 — narażanie firmy na ataki typu SQL injection	436
	Podsumowanie	443
	<i>Skorowidz</i>	445

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 pisanie 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 sprawdzimy 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 nieznanne mi wartości są takie same czy różne, tak samo SQL Server nie jest w stanie określić, czy dwie nieznanne 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.

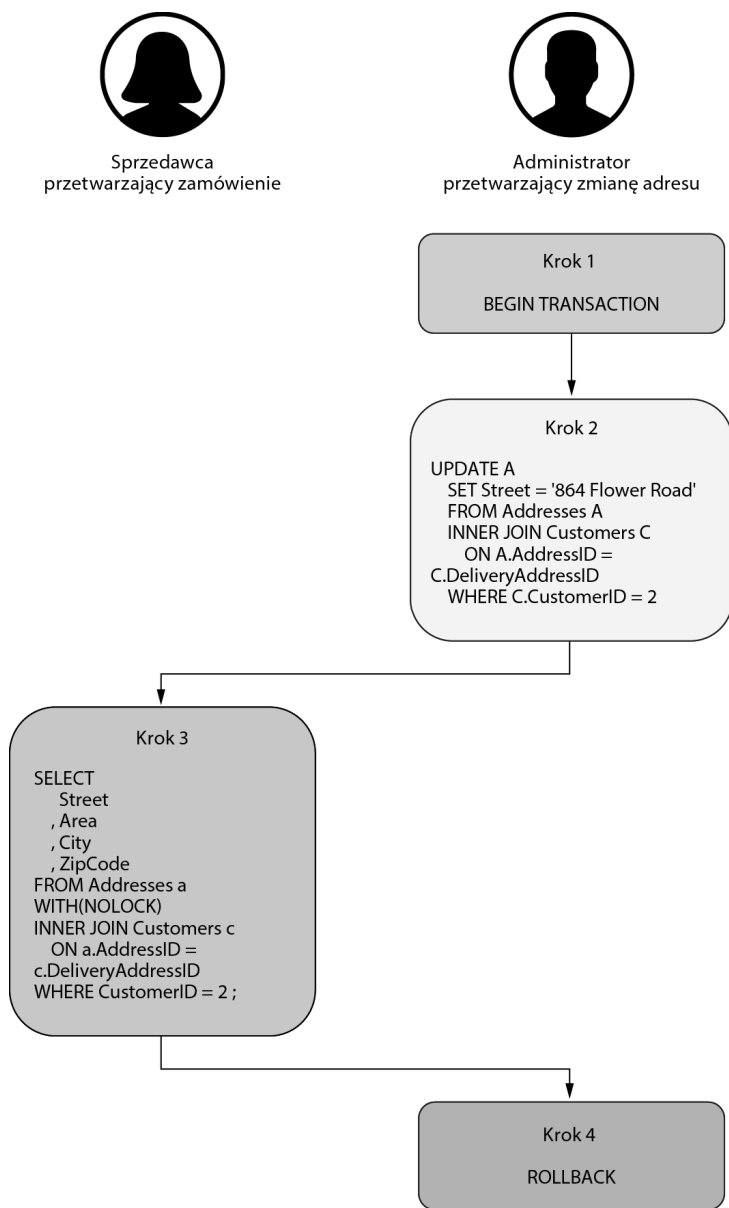
Listing 5.5. Dodawanie wskazówki NOLOCK do kwerendy

```
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. Zaczniemy 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 kwerendę, 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 SalesOrderDate, ale następnie dołącza kolumny SalesOrderDeliveryDueDate i SalesOrderDeliveryActualDate. 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,
CarrierUsedforDelivery        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, wygenerujemy 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, CustomerContactLastName 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 ;

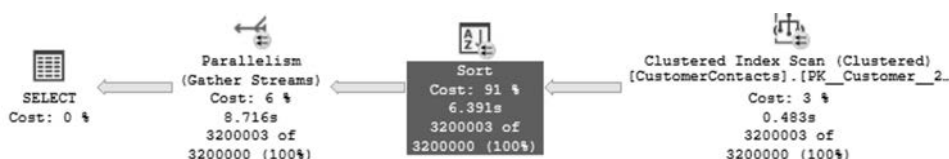
SELECT
    CustomerContactFirstName
    , CustomerContactLastName
    , CustomerContactEmail
FROM dbo.CustomerContacts
ORDER BY CustomerContactEmail ;
```

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.

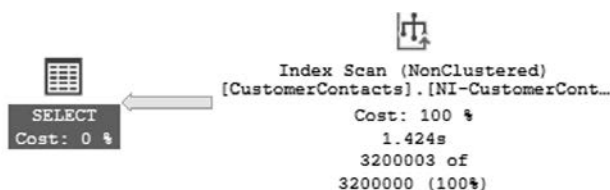
Po drugie, koszt nie jest bezpośrednim miernikiem wydajności. Jest to wartość ważona, obliczana na podstawie zastrzeżonego algorytmu, który szacuje względny koszt operacji procesora i wejścia-wyjścia, a następnie sumuje te wartości, aby uzyskać całkowity koszt operatora, znany jako *koszt poddrzewa*.

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 CustomerContactLastName na poziomie liści. Taki indeks, który można utworzyć za pomocą polecenia z listingu 5.10, jest już uporządkowany według kolumny CustomerContactEmail, 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,

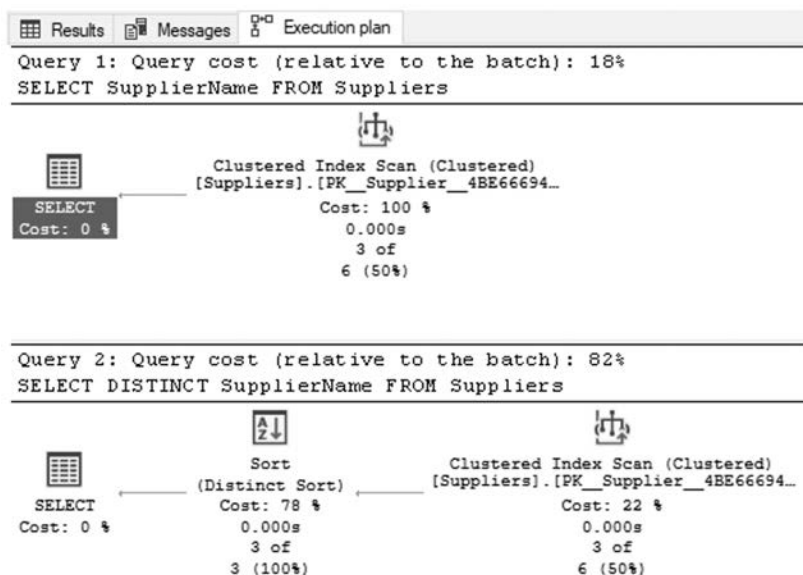
że nie mamy poważnego problemu z jakością danych) ten sam dostawca nie będzie wymieniony dwukrotnie w naszej tabeli, zastosowanie słowa kluczowego *DISTINCT* jest zbędne.

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 zębaki 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 uwzględniła 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 Mogłbym 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.

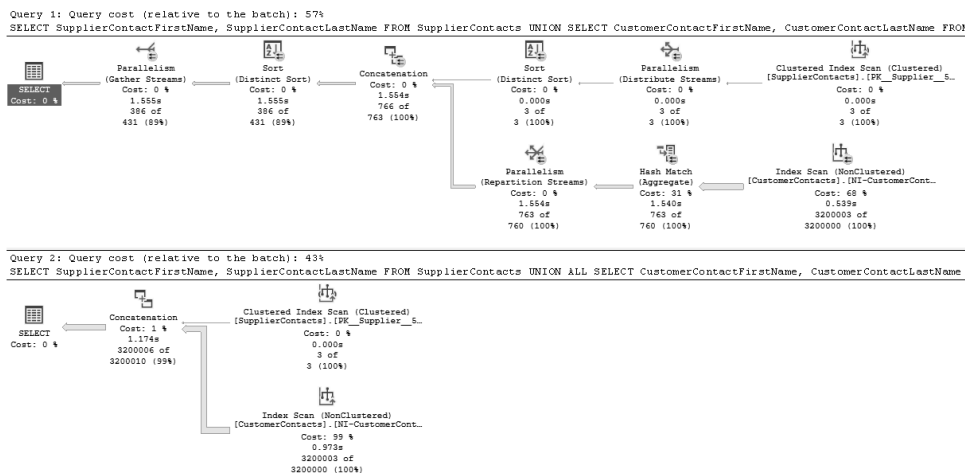
WSKAZÓWKA Istnieją dodatkowe operatory łączenia poziomego o nazwach INTERSECT i EXCEPT. INTERSECT zwraca wyniki z pierwszej kwerendy, które pojawiają się również w wynikach drugiej kwerendy. Z kolei EXCEPT zwraca wyniki z pierwszej kwerendy, których nie ma w wynikach drugiej kwerendy.

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 przedstawiania 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. Przetwarzanie danych za pomocą operatora PIVOT

```

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

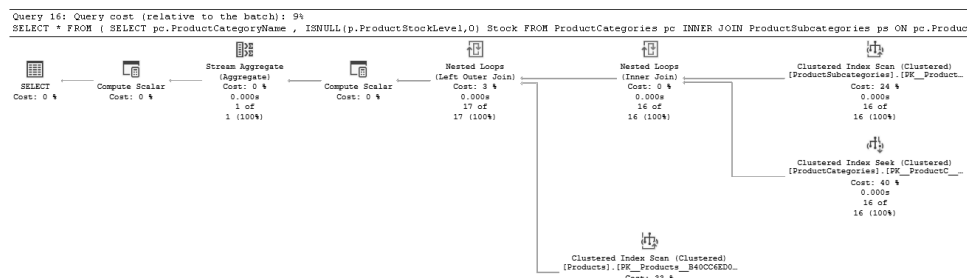
```

```

, 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 skalającej 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 ;

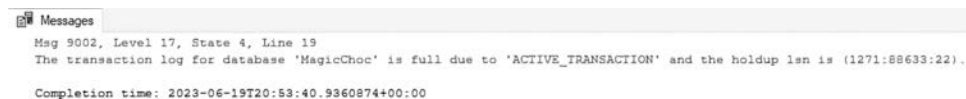
    SET @LoopCounter = @LoopCounter + 1 ;
END
```

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 zapelnieniu 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 zapelnieniu 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 @RowCount = (SELECT COUNT(*) FROM dbo.VeryLargeTable) ;
PRINT @RowCount ;
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 antywzorec 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

ACID, atomic, consistent, isolated, durable, 183, 217
alarmy, 217, 424, 443
 tworzenie, 197
analiza
 kodu, 199
 porównawcza schematów, 205
aplikacje DAC, 215
atak
 poprzez podstawienie całej wartości, 430, 434, 444
 typu brute force, 419
 typu SQL injection, 422, 436, 441, 444
atomowość, atomic, 183
atrybut podstawowy, 103
atrybuty, 92
audyt
 bazy danych, 429
 typy akcji, 428
audytowanie działań, 426, 443
automatyczna aktualizacja statystyk, 361
Azure
 Event Grid, 231
 Functions, 231
 Kubernetes Service, 231
 SQL Database, 232
 Synapse, 409

B

backup, 363
baza danych
 automatyczne
 powiększanie, 264
 zmniejszanie, 255
 skalowanie, 232

bezpieczeństwo, 414
błędne decyzje projektowe, 98
brak normalizacji, 92
brak poprawek, 283
jako usługa, DBaaS, 31
migawki, 368
model przywracania, 267
odbudowa indeksów, 257
regularne sprawdzanie, 276
tworzenie obiektów, 114
używanie kursorów, 280
własny projekt, 93, 98
wykorzystanie normalizacji, 100
zaniedbywanie automatyzacji, 279
zmniejszenie rozmiaru, 285

B-drzewo, 118

bezpieczeństwo

 SQL Servera, 414
 terminologia, 415, 416

blok TRY...CATCH, 185, 187

błąd 9002, 150

błędy

 brak alarmów, 194
 doraźne, 187
 niestandardowe, 192
 poziomy krytyczności, 186
 projektowe, 90
 rejestrowanie, 192
 w oknie Error List, 200

C

CTE, common table expression, 67
czas wykonania kwerendy, 138, 164, 165

D

DAC, data-tier application, 215
 dane niepoprawne, 156
 Database Mail, 195, 217
 New Alert, 197
 DBA, database administrator, 21
 DBaaS, Database as a Service, 31, 231, 252
 DDL, data definition language, 268
 debugowanie, 198
 opcje nawigacji, 200
 Desktop Experience, 226
 diagram
 C4, 24
 komponentów, 38, 51
 kontenerowy, 25
 krajobrazu systemu, 23
 znormalizowanego projektu, 113
 związków encji, ERD, 60, 92–98, 127
 DMF, dynamic management function, 261, 337
 DML, data manipulation language, 268
 docelowy
 czas przywracania, RTO, 364
 punkt przywracania, RPO, 364
 Docker, 30
 DOP, 300, 325
 DQS, Data Quality Services, 26
 DR, disaster recovery, 397–401, 412
 architektura, 402
 nietestowanie strategii, 405
 przesyłanie dzienników, 409
 topologia, 400
 druga postać normalna, 2NF, 107
 duplikaty, 144
 dynamiczna funkcja administracyjna,
 DMF, 261, 337
 dzienniki
 błędów, 192
 fragmentacja, 268
 transakcji, 151, 268

E

edycja
 Developer SQL Server, 231, 252
 Enterprise SQL Server, 228, 252
 Standard SQL Server, 231, 252

edytor

 miejsca docelowego, 158
 transformacji agregacyjnych, 173
 źródła pliku płaskiego, 160
 eliminowanie partycji, 308
 encje, 92
 ERD, Entity Relationship Diagram, 60, 92
 ETL, extract, transform, load, 152, 264, 327, 377–381

F

flagi śledzenia, 288, 324
 format
 JSON, 84, 86, 89
 XML, 81, 84, 89
 fragmentacja, 261
 dzienników, 268
 indeksów, 256, 263, 327
 błędne interpretowanie, 339
 wewnętrzna, 330, 360
 zewnętrzna, 329, 333, 361
 wewnętrzna, 259
 zewnętrzna, 259
 funkcja
 CAST(), 59
 DATALENGTH(), 59
 DECRYPTBYKEY(), 434
 DMF, 337
 ISNULL(), 130
 JSON_VALUE(), 87, 88
 NEWID(), 122
 OPENJSON(), 87
 OPENXML(), 76, 77, 87
 RAISERROR(), 185, 190
 ROW_NUMBER(), 70
 funkcje
 agregujące, 173
 automatycznego powiększania, 285
 automatycznego zmniejszania, 285
 do obsługi błędów, 186, 198
 SQL Servera, 235
 zarządzania dynamicznego, DMF, 261, 285

G

generalizacja danych, 105, 127
 generowanie danych, 136
 gęstość
 stron, 360
 zapisu na stronach, 337
 Git, 211
 GitHub
 tworzenie repozytorium, 212
 grupy dostępności, 401, 407

H

HA, high availability, 397–401, 412
 architektura, 402
 przełączanie awaryjne, 408
 topologia, 399
 hierarchia, 64, 69
 JSON, 86

I

IaaS, Infrastructure as a Service, 31
 IAM, Index Allocation Map, 117, 328
 IDE, 227
 identyfikator
 GUID, 122, 162, 170
 hierarchii, 70
 indeksu, 120
 PID, 238
 produktu, 184
 ilość pamięci, 293
 indeks
 klastrowy, 117, 329
 nieklastrowy, 118, 119, 329
 indeksowanie z wypełnieniem, 333
 indeksy, 327
 aktualizowanie statystyk, 346
 B-drzewo, 328, 360
 bezkrytyczne przebudowywanie, 344
 brak optymalizowania konserwacji, 348
 brak przebudowywania, 341
 fragmentacja, 327, 329, 361
 kolumnowe, 356
 niewyłączanie, 352
 pokrywające, 134, 140
 poziom fragmentacji, 338
 przebudowywanie, 361

przeszukiwanie, 334, 361
 reorganizowanie, 336
 skanowanie, 334, 361
 sprawdzanie, 335
 wyłączanie, 354
 informacje zwrotne DOP, 300, 325
 infrastruktura jako usługa, IaaS, 31
 inicjalizacja natychmiastowa plików, 291
 instalowanie
 instancji, 231
 SQL Servera, 219
 wszystkich funkcji, 233
 integralność referencyjna, 126
 izolacja, isolated, 183

J

język T-SQL, 85, 128
 JSON, 84, 86, 89

K

Kanban, 41
 kandydat na klucz, 103
 klastery aktywny-aktywny, 411
 klastry, 401
 klauzula, *Patrz także* polecenie,
 słowo kluczowe
 DELETE, 150
 FOR JSON, 86
 FOR JSON AUTO, 86
 FOR XML PATH, 80
 FROM, 86
 ORDER BY, 54
 PARTITION BY, 374
 SELECT, 54
 SELECT *, 133, 151
 TRUNCATE, 148, 149
 UNION, 143, 151
 UNION ALL, 151
 WITH, 76, 88
 WITH CHECKSUM, 376
 klucze
 główne, 93, 122
 naturalne, 111
 obce, 93, 123
 podstawowe, 117
 produktu, 239

klucze

- symetryczne i asymetryczne, 432
- sztuczne, 111
- złożone, 94, 103

kolumna uwierzytelniająca, 435

komentarze, 37

kompilowanie konfiguracji, 246

komponenty SQL Servera, 25, 26

kompresja

- danych, 325
- dużych tabel, 311

komunikaty o błędach, 190, 217

konfiguracja

- DSC, 245
- GitHuba, 212
- HA/DR, 405
- sa, 420
- źródła danych, 175

kontenery, 224, 251

konto

- gMSA, 443
- sa, 419, 443
- usługowe, 421, 443

kontrola wersji, 211

konwencje nazewnictwa, 36

koordynacja T-SQL, 165

kopie zapasowe, 363

- a proces ETL, 377, 378
- błędny harmonogram, 390
- certyfikatu, 394
- doraźne, 391, 396
- dziennika transakcji, 389, 396
- po zmianie modelu przywracania, 388
- rodzaje, 366
- strategia backupu, 393
- szyfrowanie, 393
- testowanie, 372
- typu COPY_ONLY, 392
- usuwanie, 368
- weryfikacja integralności, 375
- weryfikacja poprawności, 372

koszt operacji sortowania, 139

koszty przestojów, 404

kreatywne planowanie, 380

Kubernetes, 30

kursory, 145

L

Linux, 30

Ł

łańcuch przywracania, 366, 367, 395

łańcuchy

- o stałej długości, 64
- o zmiennej długości, 62

M

macierz RAID, 267, 286

mapa alokacji indeksów, IAM, 117, 328

marketing internetowy, 156

MDS, Master Data Services, 26

menedżer

- blokad, 29
- buforów, 29
- połączeń
 - ADO, 166
 - OLE DB, 157
 - plików płaskich, 157
- transakcji, 29

metadane, 55, 139

indeksu, 285

metoda

- GetAncestor(), 71
- GetLevel(), 72
- GetRoot(), 70
- IsDescendantOf(), 71
- nodes(), 76
- value(), 76, 82

migawki, 276

bazy danych, 368, 395

pamięci masowej, 395

spójne

- na poziomie aplikacji, 371
- w razie awarii, 370

model przywracania

- pełny, FULL, 382, 395
- prosty, SIMPLE, 382, 384, 395
- z hurtowym rejestrowaniem, BULK LOGGED, 382
- zmienianie, 388

modelowanie

- adresów, 84
- hierarchii, 64, 69
- multizbiór, 138

N

nadklucz, 103
 narzędzie
 Database Engine Tuning Advisor, 355, 362
 Database Mail, 195
 Desired State Configuration, 257
 Schema Compare, 203
 SQL Agent, 373, 389
 SQL Server Data Tools, 152
 xp_cmdshell, 414
 nawiasy
 klamrowe {}, 86
 kwadratowe [], 86
 nazwy
 baz danych, 36
 instancji, 220
 kolumn, 55
 kolumn kluczy głównych, 41
 obiektów, 37
 procedur, 43
 nieprzechodniość zależności, 108
 niewłaściwe wykorzystanie XML, 40
 NOLOCK, 27, 29, 131, 151
 normalizacja
 druga postać normalna, 2NF, 107
 pierwsza postać normalna, 102
 schemat bez normalizacji, 92
 schemat z użyciem normalizacji, 100
 trzecia postać normalna, 3NF, 108
 zastosowanie, 100
 numery porządkowe kolumn, 54

O

obiekty
 bazy danych, 114
 systemowe, 50
 obraz SQL Server AMI, 249
 obsługa
 błędów, 77, 180, 185
 OLE DB, 154
 wartości NULL, 129
 odczyty
 fantomowe, 316
 niepowtarzalne, 316

ograniczenie
 klucza obcego, 127
 niepowodzenia, 153
 sukcesu, 153
 ukończenia, 153
 OLE DB, 154
 miejsce docelowe, 157, 158, 161, 177
 nieudany przepływ danych, 159
 OLTP, 92, 384, 387
 opcja
 OPTIMIZE_FOR_SEQUENTIAL_KEY, 351
 SORT_IN_TEMPDB, 350
 XACT_ABORT, 185, 217
 operacje ETL, 152, 174
 operator
 EXCEPT, 144
 INTERSECT, 144
 IS, 130
 Nested Loops, 298
 PIVOT, 147
 operatory złączeń, 297
 optymalizacja, 287
 pakietów, 163
 wczytywania danych, 162
 optymalizator kwerend, 27, 296, 325

P

PaaS, Platform as a Service, 31, 231
 pamięć
 blokowanie stron, 294, 324
 dla innych aplikacji, 293
 maksymalna serwera, 325
 podręczna, 304
 buforów, 29
 dziennika, 29
 planów, 29
 parametr MAXDOP, 349
 parser
 MSXML, 76
 XML, 76
 partycje, 308
 partycjonowanie dużych tabel, 303
 pętla WHILE, 150
 pierwsza postać normalna, 1NF, 102
 plan wykonania kwerendy, 139
 planowanie wydajności, 271
 platforma jako usługa, PaaS, 31, 231

pliki
 CSV, 154, 156
 dziennika, 266, 275
 MOF, 245
 natychmiastowa inicjalizacja, 291
 rozrzedzone, 369
 VLF, 268, 285, 386, 387

pochodzenie danych, 101

podzbiór, 175

podziały stron, 331

polecenie
 DBCC IND, 121
 DBCC PAGE, 121, 122
 DBCC WRITEPAGE, 277
 GOTO, 188

pomyłka numer 0, 22

porządkowa pozycja kolumny, 54

porządkowanie danych, 136, 151

postaci normalne, 100

potok CI/CD, 215, 218

PowerShell, 226

poziom izolacji, 325
 nadmiernie restrykcyjny, 316
 optymistyczny, 320
 Read Uncommitted, 315

poziomy dostępności, 403

prefiks sp_, 48

prefiksy
 obiektowe, 44
 usuwanie, 46

procedura xp_cmdshell, 422, 443

procedury składowane
 pobieranie definicji, 39
 systemowe, 50
 tworzenie, 181
 wywoływanie, 49
 zwracanie listy, 38

proces
 CI/CD, 215, 216
 ETL, 264, 377–381
 zarządzania konfiguracją, 243

programowanie nowoczesne, 209

projekt bazy danych, 90

przechwytywanie zmian danych, CDC, 230

przeciążanie klastra, 410

przepływ
 danych, 153, 156, 157, 169, 172
 dostosowanie ustawień, 177
 konfiguracja, 176
 niepowodzenie, 159

kwerendy, 28
 sterowania, 153, 156

przesyłanie dzienników, log shipping, 409

przetwarzanie transakcyjne, 92

przywracanie po awarii, DR, 397

punkt przerywania, 203

Q

Query Store, 139

R

RDBMS, relational database management system, 23

realizator kwerend, 29

redundancja danych, 92

rejestrwanie błędów, 192

rekonstruowanie XML, 80

rekurencyjne wyrażenie tabelaryczne, 67

relacyjne bazy danych, 93

replikacja, 409

REST API, 85

rola bazodanowa, 418

rozszerzenie Integration Services, 152

RPO, recovery point objective, 364, 365, 395

RTO, recovery time objective, 364, 365, 395

ruch boczny, 422

S

scalenie zmian, pull request, 214

Schema Compare, 203

server jako usługa bazodanowa, DBaaS, 31, 231, 252

silniki bazy danych, 25–28
 przechowywania danych, 28
 relacyjny, 27

skanowanie indeksu klastrowego, 117

słownik danych, 101

słowo kluczowe
 DISTINCT, 140, 142, 143, 151
 NONCLUSTERED, 134, 140
 ROOT, 81
 TYPE, 80

sortowanie, 139

specyfikacje audytu, 429

spójność, consistent, 183

Sprint, 41

SQL Server, 22, 23, 128
 Agent, 196
 bezpieczeństwo, 414
 dokładanie sprzętu, 322
 funkcje, 235, 252
 główne komponenty, 25, 26
 instalacje, 219
 chmurowe, 31
 nazwy, 220
 skrypty, 236, 240
 tworzenie, 251
 instalowanie wszystkich funkcji, 233
 kontenery, 224
 obrazy
 chmurowe, 248, 253
 złote, 237
 obsługiwane
 hosty, 31
 systemy operacyjne, 30, 221
 optymalizacja, 287
 tryby uwierzytelniania, 419
 typy danych, 56
 SSAS, SQL Server Analysis Services, 24, 25
 SSIS, SQL Server Integration Services, 24, 26, 152
 koordynacja T-SQL, 165
 obsługa błędów, 165
 odrzućanie wierszy, 156
 ograniczenia, 153
 operacje ETL, 152, 174
 rejestrwanie zdarzeń, 165
 transformacje blokujące, 173
 wydajność, 165
 wyrażenia, 171
 SSMS, SQL Server Management Studio, 39, 196
 SSRS, SQL Server Reporting Services, 24, 26
 standardy kodowania, 52
 sarta, 328
 stopień kompresji, 325
 strategia
 drobnoziarnista, 421
 gruboziarnista, 421
 styl kodowania, 53
 sumy kontrolne stron, 376
 system
 kontroli wersji, 211
 zarządzania relacyjnymi bazami danych, RDBMS, 23

szacowanie licznosci, CE, 301
 szatkowanie
 dokumentu JSON, 87
 dokumentu XML, 73, 75
 szyfrowanie
 hierarchia, 431
 na poziomie komórek, 430, 444

Ś

średnik, 67
 środowisko programistyczne, 227
 Desktop Experience, 226
 Visual Studio, 226

T

tabele
 bardzo duże, 149
 kompresowanie, 311
 krawędziowe, 149
 krawędziowe XML, 76
 partycjonowane, 303, 307
 temporalne, 149
 TDE, Transparent Data Encryption, 430
 TempDB, 275
 testowanie strategii DR, 405
 testy
 jednostkowe, 206, 218
 tworzenie, 209
 wydajności, 164, 165
 transakcje, 183
 poziomy izolacji, 315, 320
 usuwanie wielu wierszy, 148
 transformacje
 agregacyjne, 173
 blokujące, 173
 nieblokujące, 173
 trwałość, durable, 183
 tryb
 PATH, 86
 FOR XML, 80
 trzecia postać normalna, 3NF, 108
 T-SQL, 85, 128
 tworzenie
 alarmu, 197
 hierarchii szyfrowania, 432
 indeksów nieklastrowych, 119
 kontenera, 225

tworzenie
 migawki, 369
 obiektu audytu, 427
 potoku CI/CD, 215
 repozytorium, 212
 roli serwerowej, 418
 skryptu do instalowania, 240
 tabeli partycjonowanej, 306
 testu jednostkowego, 209
 wyzwalacza, 425
 zaszyfrowanej kopii zapasowej, 394

typ danych
 BIT, 61
 CHAR, 62, 63
 DATE, 58
 HIERARCHYID, 69, 71, 89
 INT, 58, 59
 MONEY, 58
 NCHAR, 63
 NVARCHAR, 56, 57, 63
 SMALLINT, 59–61
 TINYINT, 61
 VARCHAR, 62, 63
 XML, 73

typy
 danych
 sprawdzanie rozmiaru, 59
 zmienianie, 61
 zdarzeń audytu, 428

U

uprawnienie
 CONTROL SERVER, 423
 EXEC, 423

usługa
 Azure SQL Database, 32
 DBaaS, 32
 GitHub, 211
 RDS, 31
 SSAS, 24
 SSIS, 24, 152
 SSRS, 24
 Volume Shadow Copy Service, 371

usuwanie
 prefiksu, 46
 wierszy partiami, 150, 151

UTF-8, 63
 UTF-16, 63

uwierzytelnianie, 419
 mieszane, 443
 używanie kursorów, 145

V

Visual Studio, 156, 226
 analiza kodu, 199
 funkcje debugowania, 198
 okno
 Compare, 205
 Error List, 199
 Immediate, 201
 Locals, 203
 punkt przzerwania, 202
 rejestracja DAC, 215
 VLF, virtual log file, 268
 VMware, 30

W

warstwa abstrakcji, 46
 wartość
 atomowa, 99
 NULL, 62, 129, 151
 zliczanie, 130

widok dynamicznego zarządzania, DMV,
 46, 230, 285, 361

wielkość liter, 71

Windows, 30

Windows Server Core, 226

wirtualizacja danych, 26

własny
 kod hierarchii, 64
 schemat bazy danych, 93

wskazówka
 NOLOCK, 131, 151
 Query Store, 298

wskaźnik
 RPO, 395
 RTO, 395

wydajność, 131, 151, 164, 165
 planowanie, 271
 usług SSIS, 165

wrażenia SSIS, 171

wysoka dostępność, HA, 397

wyzwalacz, 425

wzorzec
 wzrostu liniowego, 272
 wzrostu wykładniczego, 273

X

XML, 40, 73, 84, 89
 atrybuty, 75
 elementy podrzędne, 75
 rekonstruowanie, 80
 szatkowanie, shredding, 74, 75
XQuery, 73, 76

Z

zadania
 Execute T-SQL Statement, 166, 169, 178
zarządzanie
 bazą danych, 254
 instancją, 254
 konfiguracją, 242

zasada najmniejszych przywilejów, 416, 443
zbiór, 138
ziarnistość konta usługowego, 421
zintegrowane środowisko programistyczne,
 IDE, 227
złączenie typu Hash Join, 298
znak
 @, 82
 gwiazdki, 147
 średnika, 67

Ż

źródło
 danych, 175
 pliku płaskiego, 157

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

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

