

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: 978-83-289-2897-8

© 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_ebook

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)

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

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

Dla Terri, mojej opoki. Dziękuję Ci.

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 o) | 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 przepływy 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> |179 |
| 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> | 219 |
| 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> |254 |
| 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. | <i>Optymalizacja</i> | 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. <i>Parametr MAXDOP</i> | 349 |
| | 11.8.2. <i>Opcja SORT_IN_TEMPDB</i> | 350 |
| | 11.8.3. <i>Opcja OPTIMIZE_FOR_SEQUENTIAL_KEY</i> | 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. <i>Weryfikacja poprawności wykonania kopii zapasowych</i> | 372 |
| | 12.4.2. <i>Weryfikacja integralności kopii zapasowej</i> | 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 |

Wstęp

SQL Server to niezwykle bogaty i potężny zestaw narzędzi, który można wykorzystać do przechowywania, pobierania, przetwarzania i przekształcania danych. Na przestrzeni lat do tego produktu dodano wiele ulepszeń i dodatkowych funkcji, co sprawia, że jest to bardzo obszerny temat do nauczenia się i opanowania.

SQL Server zaprojektowano tak, aby był łatwy w użyciu, ale ta kombinacja prostoty i elastyczności sprawia, że bardzo łatwo jest go źle wykorzystać, a trudno zrobić to dobrze. Ponieważ dane są niezwykle istotne dla każdej organizacji, niewłaściwe korzystanie z SQL Servera może narazić firmę na wiele zagrożeń. Ryzyko to obejmuje zarówno spadek produktywności spowodowany słabą wydajnością, jak i problemy z bezpieczeństwem. Organizacje mogą nawet naruszyć przepisy dotyczące zgodności, jeśli dane zostaną utracone, staną się niedostępne lub wpadną w niepowołane ręce.

Pracuję z SQL Serverem od dwóch dekad. W tym czasie miałem szczęście uczestniczyć w niektórych z największych i najbardziej złożonych projektów SQL Server w Londynie. Pełniłem różne funkcje: programisty T-SQL, specjalisty ds. analizy danych, administratora baz danych, inżyniera platformy i architekta. Szczególnie ceniłem sobie pracę jako instruktor SQL Servera, co dało mi możliwość prowadzenia szkoleń w całej Europie i poznania wielu niezwykle ciekawych ludzi.

Przez ostatnie 10 lat miałem zaszczyt pisać książki techniczne o SQL Serverze. Obejmowały one różnorodne tematy, takie jak administracja, automatyzacja, bezpieczeństwo, AlwaysOn oraz zaawansowane typy danych. W rzeczywistości *SQL Server. 100 najczęstszych błędów i jak ich skutecznie unikać* to mój jedyny projekt książkowy.

Ponieważ pracowałem w różnych obszarach związanych z SQL Serverem dla wielu organizacji z listy FTSE 100, miałem okazję zaobserwować mnóstwo błędów popełnianych przez osoby, które nieoczekiwanie znalazły się w roli administratorów baz danych, często bez formalnego przeszkolenia. To właśnie było inspiracją dla mnie do napisania tej książki. Chciałem zebrać w jednym tomie najczęstsze błędy, z którymi się spotkałem lub które sam czasem popełniałem, aby pomóc innym uniknąć wpadania w te same pułapki.

Wiele książek technicznych przedstawia tylko „pomyślne scenariusze”. Uczą one czytelników, jak działa SQL Server, „kiedy wszystko idzie dobrze”, lub pokazują, jak napisać kod spełniający konkretne wymagania. Ta książka jest inna. Zwraca uwagę na powszechne błędne przekonania i nieprawidłowe konfiguracje, wyjaśniając, dlaczego są one problematyczne. Następnie opisuje, co można zrobić inaczej, aby uniknąć tych problemów.

Podziękowania

Pisanie książek to trudne zadanie. Wymaga ogromnej ilości czasu. Może być stresujące i prowadzić do izolacji społecznej. Z tego powodu pisanie ma wpływ nie tylko na autora, ale także na jego rodzinę. Dlatego chciałbym szczególnie podziękować mojej partnerce, Terri, zarówno za jej nieskończoną cierpliwość, jak i za niekończące się filiżanki kawy, które pojawiały się na moim biurku.

Pisanie jest również sportem zespołowym, a ta książka nie byłaby tak dobra, gdyby nie wysiłek Connora O'Briena, redaktora prowadzącego tego projektu, oraz Granta Fritcheya, redaktora technicznego, który ma również tytuły Microsoft Data Platform MVP i AWS Community Builder, a obecnie pracuje jako Product Advocate w Red Gate Software. Ogromne podziękowania dla Was obu za wkład i zaangażowanie.

Ponadto chciałbym podziękować wszystkim recenzentom, którzy podzielili się swoimi cennymi uwagami. Byli to: Adam Wan, Alexander Makeev, Amol Gote, Andrew Briers, Andrew Judd, Ankit Virmani, Ben McNamara, Christian Leverenz, Dave Corun, Edward Pollack, Eli Rabinovitz, Esref Durna, Evan Benjamin, Grant Colley, Ian Stirk, Ivan A. Fernandez, João Marcelo Borovina Josko, Jonathan Reeves, José Alberto Reyes Quevedo, Josephine Bush, Mary Anne Thygesen, Meghal Gandhi, Mick Wilson, Mihaela Barbu, Nadir Doctor, Naga Santhosh Reddy Vootukuri, Peter A Schott, Praveen Raju, Prithvi Shivashankar, Regina Obe, Richard Jepps, Ruben Vandeginste, Scott Ling, Stephen Viljoen, Tania Lincoln, Wenyu Shi i Wondi Wolde — Wasze sugestie pomogły ulepszyć tę książkę.

O książce

Książka *SQL Server. 100 najczęstszych błędów i jak ich skutecznie unikać* prowadzi okazjonalnych administratorów baz danych przez pole minowe systemu SQL Server, pomagając im uniknąć typowych pułapek, które czyhają na specjalistów od baz danych. Obejmuje ona szeroki zakres tematów, w tym administrację, programowanie i bezpieczeństwo.

Kto powinien przeczytać tę książkę?

Ta książka jest przeznaczona dla okazjonalnych administratorów baz danych, początkujących specjalistów ds. SQL Servera, programistów aplikacji, którzy muszą pracować z SQL Serverem, a nawet doświadczonych profesjonalistów, którzy nie mieli jeszcze do czynienia z najnowszymi wersjami tego produktu.

Układ książki: przewodnik po treści

Książka składa się z 14 rozdziałów:

- W rozdziale 1. przedstawiono SQL Server oraz metody nauki stosowane w tej książce. Omówiono również „pomyłkę numer o”, która jest źródłem wszystkich innych pomyłek — błędne przekonanie, że SQL Server to „tylko baza danych”.
- W rozdziale 2. omówiono standardy kodowania. Przeanalizowano w nim wpływ wyborów stylistycznych, a także problemy techniczne wynikające z zastosowania niewłaściwych standardów. Przyjrzymy się tu również konwencjom nazewniczym i ich wpływowi na łatwość utrzymania kodu.

- W rozdziale 3. omówiono typy danych SQL Servera. W tym rozdziale przyjrzymy się skutkom stosowania niewłaściwych typów danych. Zbadamy również zaawansowane typy danych i ocenimy konsekwencje ich nieużywania.
- Rozdział 4. poświęcony jest projektowaniu baz danych. Skupię się w nim głównie na normalizacji i wyjaśnię, dlaczego dobry projekt bazy danych jest tak istotny. Przyjrzymy się również zastosowaniu kluczy oraz skutkom podejmowania niewłaściwych decyzji w tym obszarze.
- Rozdział 5. zagłębia się w szczegóły T-SQL. Przeanalizujemy w nim szereg błędów, które mogą prowadzić do nieprzewidywalnych rezultatów i niskiej wydajności. Przyjrzymy się również pętlom w T-SQL i omówimy alternatywne rozwiązania.
- W rozdziale 6. omówiono typowe błędy popełniane podczas korzystania z usługi SQL Server Integration Services (SSIS). Przyjrzymy się w nim częstym pomyłkom, takim jak utrata nieprawidłowych danych, brak optymalizacji wczytywania danych oraz wykorzystywanie SSIS wyłącznie jako narzędzia do koordynacji zadań.
- W rozdziale 7. omówiono obsługę błędów, testowanie, kontrolę wersji i wdrażanie. W tym rozdziale dowiesz się, jak obsługiwać błędy w T-SQL i rozwiązywać problemy z kodem. Omówimy także korzyści płynące z testowania oraz stosowania nowoczesnych mechanizmów wdrażania.
- W rozdziale 8. przejdziemy do administracji i przeanalizujemy błędy popełniane podczas instalacji SQL Servera. Wśród tych błędów znajdują się wybór nieodpowiedniego systemu operacyjnego oraz niewłaściwej edycji SQL Servera. Omówimy również automatyzację procesu instalacji oraz sytuacje, w których warto rozważyć wybór rozwiązań chmurowych.
- W rozdziale 9. omówiono zarządzanie instancjami i bazami danych. Przeanalizujemy w nim skutki lekceważenia planowania wydajności oraz zaniedbywania aktualizacji systemu. Przyjrzymy się również uszkodzeniom baz danych i skryptom administracyjnym.
- Rozdział 10. koncentruje się na optymalizacji. W tym rozdziale przeanalizujemy optymalizacje na poziomie instancji, tabel, zapytań i transakcji. Omówimy również błąd polegający na próbach rozwiązania problemu poprzez rozbudowywanie sprzętu.
- W rozdziale 11. przeanalizowano błędy i nieporozumienia związane z indeksami. Przyjrzymy się w nim fragmentacji indeksów, ich konserwacji oraz ich interakcjom z procesami ETL (ekstrakcji, transformacji i wczytywania danych).

- Rozdział 12. poświęcony jest zapasowym kopiom baz danych. W tym rozdziale omawiamy szereg błędów związanych z tym niezwykle istotnym tematem. Błędy te obejmują zarówno niedostosowanie strategii backupu do wymagań organizacji w zakresie odzyskiwania danych, jak i nieuwzględnienie aspektów bezpieczeństwa przy planowaniu strategii backupu.
- W rozdziale 13. omówiono wysoką dostępność. Przyjrzymy się różnicom między wysoką dostępnością a usuwaniem skutków awarii, aby upewnić się, że nasza strategia dostępności spełnia wymagania. Rozważymy też kwestie techniczne, takie jak przeciążone klastry oraz sytuacje, w których grupy dostępności nie są właściwym wyborem technologicznym.
- Rozdział 14. to dogłębna analiza błędów związanych z bezpieczeństwem. Przyjrzymy się różnym problemom, od ziarnistości kont usługowych po `xp_cmdshell`. Omówimy również typowe zagrożenia, takie jak ataki polegające na podstawianiu całych wartości czy wstrzykiwanie kodu SQL, a także sposoby ochrony przed nimi.

O kodzie

Ta książka zawiera wiele przykładów kodu, zarówno w numerowanych listingach, jak i w tekście. We wszystkich przypadkach kod jest zapisany specjalną czcionką, aby łatwo go było zidentyfikować. Niektóre listingi zawierają adnotacje, które pomagają zrozumieć działanie bardziej złożonych fragmentów kodu.

Pełny kod źródłowy wszystkich przykładów z książki możesz pobrać pod adresem <https://ftp.helion.pl/przyklady/ss100b.zip>.

O autorze



Peter A. Carter ma dwudziestoletnie doświadczenie jako programista, administrator, architekt, instruktor i autor książek o SQL Serverze. Obecnie kieruje zespołem inżynierów platformy dla znanej marki w Londynie. Posiada szeroką i głęboką wiedzę na temat SQL Servera. Napisał wiele książek poświęconych różnym aspektom SQL Servera, od administracji i bezpieczeństwa po automatyzację i zaawansowane typy danych.

Wprowadzenie do SQL Servera



W tym rozdziale:

- Co powinieneś wiedzieć o tej książce
- Pomyłka indeksowa związana z SQL Serverem (pomyłka numer 0)
- Model C4
- Komponenty, protokoły i platformy SQL Servera
- Znaczenie prawidłowego korzystania z SQL Servera

W ostatnich latach termin DBA (ang. *database administrator*) stał się ogólnym określeniem dla każdego, kto pracuje z bazami danych. Uważam, że takie podejście nie oddaje sprawiedliwości technologii ani też specjalistom, którzy się nią zajmują. SQL Server to rozbudowany produkt, a złożone aplikacje, które w pełni wykorzystują jego możliwości, wymagają od zespołów szerokiego zakresu kompetencji. Dlatego w tej książce, mówiąc o DBA, będę odnosił się konkretnie do administratorów baz danych.

Omówimy zagadnienia istotne dla osób pełniących następujące role:

- Administratorzy baz danych
- Programiści baz danych
- Inżynierowie ETL (ekstrakcji, transformacji i wczytywania danych)

Książka może być również przydatna dla osób z następujących grup, jeśli praca z bazami danych wchodzi w zakres ich obowiązków:

- Architekci baz danych
- Twórcy hurtowni danych
- Testerzy
- Specjaliści data science
- Inżynierowie cyberbezpieczeństwa
- Programiści specjalizujący się w analityce biznesowej

1.1. Pomyłka indeksowa związana z SQL Serverem (pomyłka numer 0)

Współczesny ekosystem SQL Servera jest rozległy i złożony, co prowadzi mnie do pomyłki numer zero — terminu zapożyczonego z wirusologii, gdzie przypadek indeksowy (lub zerowy) opisuje pierwszego pacjenta, który zaraził się wirusem i przekazał go innym. W tym kontekście jest to źródło wszystkich innych pomyłek związanych z SQL Serverem. Błędu tego zwykle nie popełniają specjaliści od baz danych. Popołniają go zazwyczaj architekci rozwiązań, kierownicy projektów, menedżerowie programów i analitycy biznesowi. Można podsumować go jednym zdaniem, które zdarza mi się usłyszeć aż nazbyt często: „Ale to przecież tylko baza danych, prawda?”.

To założenie o prostocie prowadzi do niezliczonej liczby projektów, w których nie uwzględnia się odpowiednich zasobów potrzebnych do zbudowania solidnej warstwy dostępu do danych lub nie przewiduje się wystarczająco dużo czasu na jej właściwe opracowanie. W rezultacie nie poświęca się należytej uwagi późniejszemu utrzymaniu środowiska, a termin *DBA* zaczyna być używany w odniesieniu do każdego, kto ma do czynienia z bazami danych, niezależnie od jego rzeczywistych kompetencji.

Jak rozwiązuje się te problemy? Oto pojawia się „przypadkowy administrator” — osoba z ograniczonym doświadczeniem w SQL Serverze, często programista aplikacji, który w przeszłości tworzył niewielkie bazy danych jako zaplecze dla swoich programów. Nagle ta osoba staje się odpowiedzialna za rozwój, optymalizację, administrację i bezpieczeństwo wszystkiego, co ma związek z SQL Serverem.

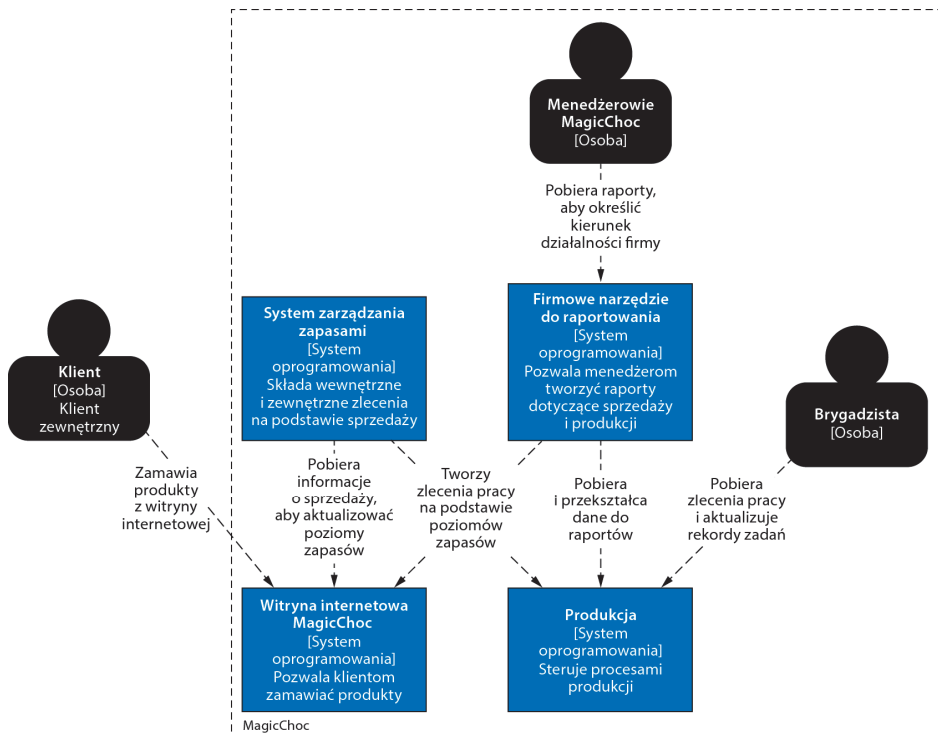
Czy ten scenariusz brzmi znajomo? Jeśli tak, czytaj dalej, ponieważ w tej książce omówię typowe pomyłki popełniane przez specjalistów od baz danych, którzy dopiero zaczynają swoją przygodę w tej dziedzinie. Wielu z nich to właśnie „przypadkowi administratorzy”.

W tej książce omówimy błędy popełniane podczas wykonywania zadań związanych z tworzeniem oprogramowania, administracją, wysoką dostępnością (ang. *high availability*, HA), usuwaniem skutków awarii (ang. *disaster recovery*, DR) oraz bezpieczeństwem.

1.2. Przegląd SQL Servera

SQL Server to popularny system zarządzania relacyjnymi bazami danych (ang. *relational database management system*, RDBMS) opracowany przez Microsoft. W najprostszej formie zapewnia platformę do hostingu baz danych i zarządzania nimi. Ma również wiele innych, bardziej zaawansowanych funkcji, takich jak raportowanie, transformacja danych czy zarządzanie danymi głównymi.

Aby lepiej zrozumieć możliwości SQL Servera, wyobraźmy sobie przykład firmy cukierniczej o nazwie MagicChoc. Firma ta ma stronę internetową hostowaną w chmurze publicznej, za pośrednictwem której sprzedaje swoje czekoladowe przysmaki. W jej fabryce działa też małe centrum danych, w którym znajdują się aplikacje do zarządzania produkcją, kontroli zapasów oraz system raportowania. Ilustruje to schemat architektury systemu przedstawiony na rysunku 1.1.



Rysunek 1.1. Diagram krajobrazu systemu MagicChoc

Model C4

W tej książce oprócz diagramów technicznych będziemy w odpowiednich przypadkach używać również modelu C4. Model C4 to zestaw standardowych diagramów architektonicznych. Obejmuje cztery podstawowe diagramy: diagram kontekstu systemu, diagram kontenerowy, diagram komponentów i diagram kodu. Diagram kontekstu systemu znajduje się na najwyższym poziomie i obrazuje interfejsy aplikacji z użytkownikami i innymi aplikacjami. Każdy kolejny poziom zagłębia się w konkretny obszar aplikacji, dostarczając coraz bardziej szczegółowych informacji. Najniższy poziom szczegółowości reprezentuje diagram kodu.

Model zawiera również dodatkowe schematy: diagram krajobrazu systemu, który przedstawia interakcje między różnymi aplikacjami w portfolio; diagram dynamiczny, ilustrujący współdziałanie elementów statycznych w czasie wykonywania aplikacji w celu realizacji określonej funkcji; oraz diagram wdrożeniowy, pokazujący platformę, na której aplikacja jest uruchamiana.

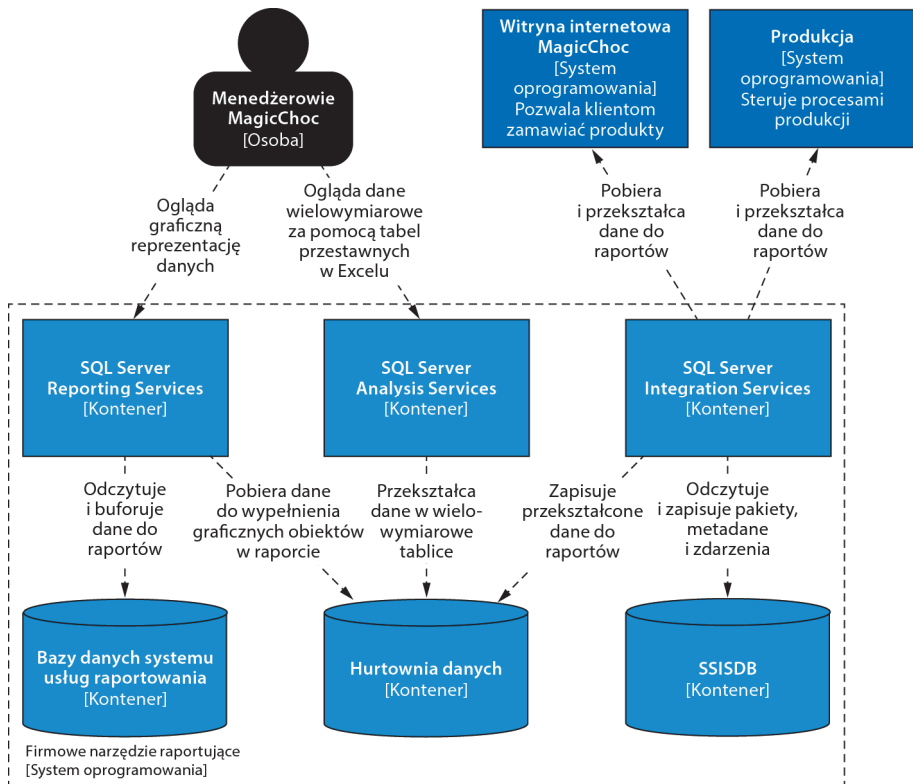
WSKAZÓWKA Choć szczegółowe omówienie modelu C4 wykracza poza ramy niniejszej książki warto wiedzieć, że pełny jego opis można znaleźć pod adresem <https://c4model.com>.

Ciekawym aspektem tej architektury systemowej jest to, że wszystkie cztery aplikacje korzystają z komponentu SQL Server. Strona internetowa MagicChoc jest hostowana w Azure i wykorzystuje bazę danych Azure SQL do przechowywania informacji. System zarządzania zapasami oraz aplikacja produkcyjna mają swoje bazy danych SQL Server, które są hostowane w tej samej instancji SQL Servera działającej na maszynie wirtualnej w centrum danych. Narzędzie raportowe firmy używa hurtowni danych, która jest hostowana w centrum danych, ale wykorzystuje również usługi SQL Server Reporting Services (SSRS) jako interfejs użytkownika oraz usługi SQL Server Analysis Services (SSAS) do tworzenia wielowymiarowych modeli danych. Ponadto korzysta z usług SQL Server Integration Services (SSIS) do pobierania danych z innych aplikacji i przekształcania ich w zdenormalizowaną strukturę, zoptymalizowaną pod kątem raportowania.

Przyjrzyjmy się teraz bliżej diagramowi kontenerowemu skupionemu na aplikacji raportującej. Diagram ten, przedstawiony na rysunku 1.2, pozwala dostrzec szeroki zakres możliwości oferowanych przez stos technologii SQL Servera.

UWAGA Diagramy C4 powinny być zawsze stosowane odpowiednio do sytuacji. Należy wybierać i łączyć najbardziej odpowiednie diagramy dla danego scenariusza. W tym rozdziale diagram krajobrazu systemu oraz diagram kontenerowy najlepiej ilustrują omawiany scenariusz, jednak w dalszej części książki będziemy korzystać również z innych diagramów C4.

Jak widać, aplikacja raportująca wykorzystuje wiele komponentów SQL Servera do realizacji wymagań biznesowych. Oprócz tego korzysta z baz danych, które współpracują z hurtownią danych i umożliwiają działanie tych komponentów. Pełna lista głównych komponentów SQL Servera 2022 znajduje się w tabeli 1.1.

**Rysunek 1.2.** Diagram kontenerowy aplikacji raportującej**Tabela 1.1.** Główne komponenty SQL Servera

| Komponent | Opis |
|--------------------------|---|
| Analysis Services (SSAS) | Umożliwia tworzenie i hostowanie wielowymiarowych i tabelarycznych modeli danych, które można wykorzystać do zaawansowanego raportowania. |
| Azure Connected Services | Zapewnia ścisłą integrację między Azure a SQL Serverem hostowanym lokalnie. Obejmuje to możliwość łatwego integrowania lokalnych instancji SQL Servera z funkcjami Azure, takimi jak Synapse, Purview i Microsoft Defender. Dodatkowo podczas instalacji instancji można skonfigurować proste połączenie Azure Arc. Zapewnia to ujednolicony widok instalacji SQL Servera w środowiskach wielochmurowych i lokalnych i pozwala na w pełni zautomatyzowane oceny techniczne. |
| Silnik bazy danych | Podstawowa usługa zarządzania bazami danych, która umożliwia budowanie i hostowanie relacyjnych baz danych. Zawiera komponenty systemu operacyjnego drugiego poziomu do zarządzania pamięcią i zasobami procesora. Obejmuje również zabezpieczenia danych, technologie wysokiej dostępności, replikację danych, integrację z heterogenicznymi źródłami danych oraz obsługę danych częściowo ustrukturyzowanych, takich jak XML i JSON. |

Tabela 1.1. Główne komponenty SQL Servera – ciąg dalszy

| Komponent | Opis |
|--|---|
| Data Quality Services (DQS) | Rozwiązanie do zapewniania wysokiej jakości danych, które oferuje bazę wiedzy wspierającą kluczowe zadania związane z jakością danych, takie jak standaryzacja i deduplikacja. Komponent serwera DQS obejmuje funkcje kontroli jakości i przechowywania danych, podczas gdy komponent klienta DQS zapewnia graficzny interfejs użytkownika dla ekspertów dziedzinowych. |
| Wirtualizacja danych z PolyBase | Pozwala programistom używać T-SQL do odpytywania zewnętrznych źródeł danych, takich jak Azure Blob, tabele Delta (domyślny format tabel w Azure Databricks), Hadoop, MongoDB, Oracle, S3 i Teradata. |
| Integration Services (SSIS) | Usługi zapewniające wszechstronne operacje ETL. Często używane do pobierania danych z heterogenicznych źródeł, denormalizacji danych i wypełniania hurtowni danych. Służy również do integrowania danych z zewnętrznych źródeł, takich jak usługi internetowe i serwery FTP, z bazami danych SQL Server. |
| Machine Learning Services (w bazie danych) | Umożliwia programistom używanie skryptów R i Python wewnątrz baz danych. Skrypty te mogą być wykorzystywane do przygotowywania danych do uczenia maszynowego lub do trenowania, oceny i wdrażania modeli uczenia maszynowego. |
| Master Data Services (MDS) | Rozwiązanie do zarządzania danymi głównymi (ang. <i>Master Data Management</i> , MDM), które umożliwia zarządcom danych zarządzanie głównymi danymi firmy. Zarządcy mogą tworzyć modele danych i reguły. Dane główne można również eksportować, co ułatwia ich udostępnianie interesariuszom biznesowym. |
| Reporting Services (SSRS) | Usługi zapewniające graficzne narzędzie do tworzenia raportów. Raporty mogą zawierać dane tabelaryczne, a także wykresy, mapy i inne elementy graficzne. Raporty mogą być złożone, a system obsługuje parametry, zmienne, powiązane raporty oraz buforowanie raportów. |

Jak widać, SQL Server to rozbudowany i złożony produkt, o którym można napisać (i napisano) wiele tomów. Dlatego w tej książce skupimy się głównie na silniku bazy danych, choć poruszymy również temat integracji z chmurą i SSIS. Zatem zagłębmy się nieco bardziej w temat silnika bazy danych.

1.2.1. Przegląd silnika bazy danych

Aby wyjaśnić działanie silnika bazy danych, przyjrzyjmy się drodze, jaką przebywa kwerenda wprowadzona przez użytkownika. Wyobraźmy sobie, że klient MagicChoc przegląda stronę internetową i postanawia bliżej przyjrzeć się batonikowi LushBar. Strona wykonuje następującą kwerendę, która odczytuje tabelę `dbo.Products` i zwraca nazwę oraz opis produktu wraz z jego zdjęciem. Instrukcja `CASE` jest używana do zwrócenia odpowiedniej informacji w zależności od ilości produktu w magazynie. Jeśli jest więcej niż 10 sztuk produktu, zwraca tekst „W magazynie”. Jeśli jest o sztuk, zwraca „Brak w magazynie”. Jeśli jest

od 1 do 10 sztuk, tworzy ciąg znaków informujący użytkownika o dokładnej liczbie dostępnych sztuk:

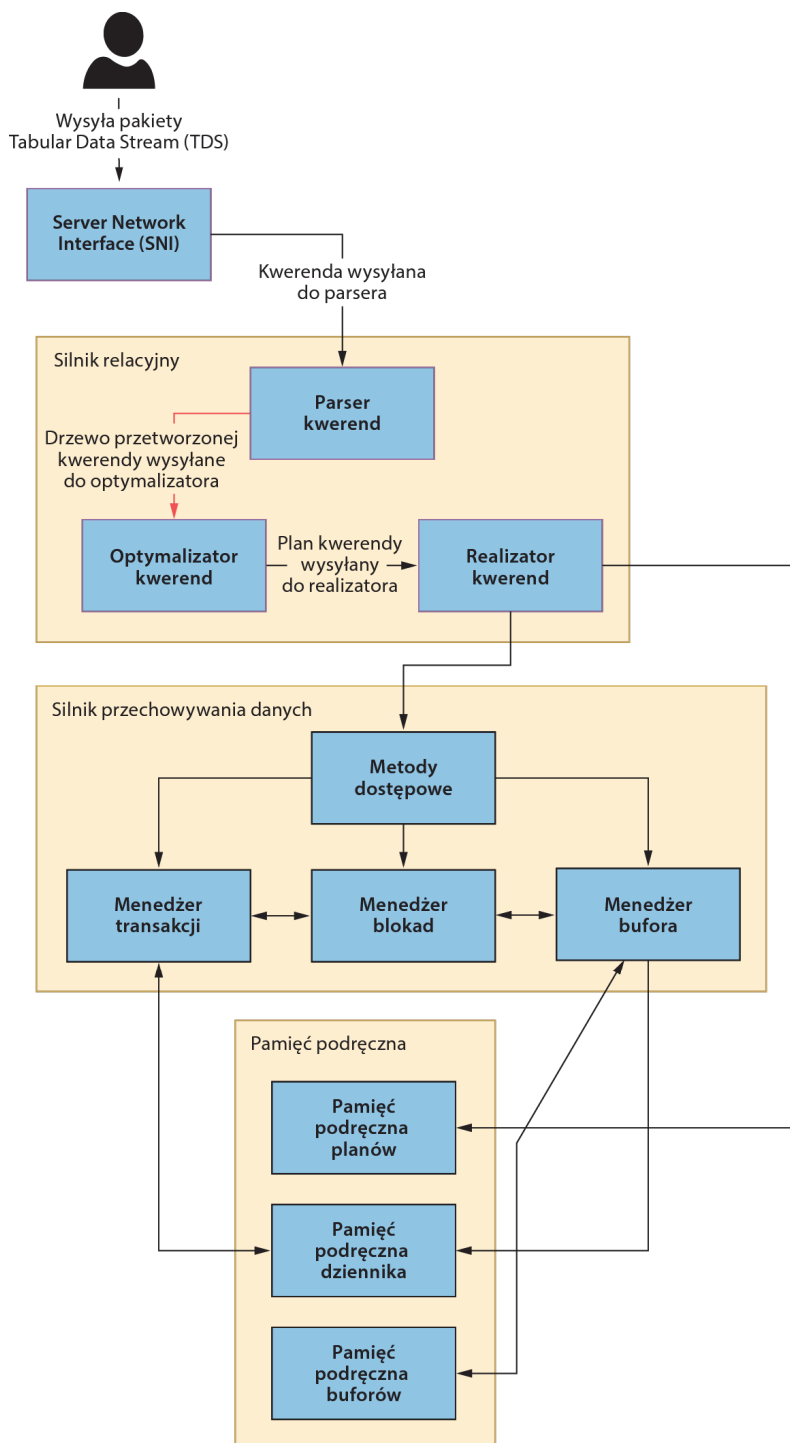
```
SELECT
    ProductName
    , ProductDescription AS Description
    , CASE
        WHEN StockQty >= 10
            THEN N'W magazynie'
        WHEN StockQty > 0 AND StockQty < 10
            THEN CAST(StockQty AS NVARCHAR) + ' nadal w magazynie'
        ELSE 'Brak w magazynie'
    END
    , ProductImage
FROM dbo.Products WITH (NOLOCK)
WHERE ProductID = @ProductID
```

UWAGA Użycie NOLOCK w tym kontekście jest pomyłką, którą omówimy w rozdziale 5.

Podróż tej kwerendy przedstawiono na rysunku 1.3. Na diagramie widać, że użytkownik komunikuje się z SQL Serverem za pomocą protokołu aplikacyjnego Tabular Data Stream. SQL Server odbiera te dane przez interfejs SNI, który jest częścią warstwy protokołów SQL Servera. Następnie warstwa protokołów przekazuje żądanie do silnika relacyjnego.

Tutaj następuje parsowanie kwerendy. Jeśli popełniłeś błąd w składni kwerendy, to właśnie parser zgłosi jego wystąpienie. Proces ten nie tylko sprawdza poprawność składni kwerendy, ale obejmuje również algebraizację, która przekształca nazwy obiektów na ich identyfikatory. W rezultacie powstaje wysoce znormalizowane drzewo kwerendy, które jest następnie przekazywane do optymalizatora zapytań.

Optymalizator zapytań to zaawansowany proces, który znajduje się w samym sercu SQL Servera. Różnica między SQL a wieloma innymi językami, takimi jak C czy BASIC, polega na tym, że SQL jest językiem opisowym, a nie nakazowym. W językach nakazowych programista dokładnie określa, co język ma zrobić. Natomiast w języku opisowym, takim jak SQL, programista po prostu opisuje wyniki, które chce uzyskać. Optymalizator zapytań odpowiada za znalezienie najbardziej efektywnego sposobu zwrócenia pożądanego zbioru wyników. Jak można sobie wyobrazić, ma to ogromny wpływ na wydajność silnika bazy danych. Dlatego optymalizator wykorzystuje typy obiektów, typy danych i statystyki, a także inne metadane, aby opisać dane w kolumnach i indeksach oraz ocenić koszt różnych planów wykonania. Choć optymalizator jest imponującym komponentem, to nie zastąpi dobrze napisanego kodu. Możesz mu pomóc, unikając typowych pułapek w pisaniu kodu T-SQL, takich jak używanie kursorów, o czym będziemy mówić w rozdziałach 5. i 9. Innym błędem, który może utrudnić pracę optymalizatora, jest zaniedbywanie aktualizacji statystyk i indeksów. Temu zagadnieniu przyjrzymy się bliżej w rozdziale 11.

**Rysunek 1.3.** Przepływ kwerendy przez silnik bazy danych

WSKAZÓWKA SQL Server 2022 może również korzystać z mechanizmu zwanego *sprężeniem zwrotnym optymalizatora*. Wykorzystuje on magazyn kwerend oraz inteligentne przetwarzanie kwerend do optymalizacji różnych aspektów planu kwerendy, takich jak przydzielanie pamięci czy maksymalny stopień paralelizmu, przez analizę wydajności danej kwerendy z biegiem czasu. Temat ten zostanie szerzej omówiony w rozdziale 10.

Ustalony plan jest przekazywany do *realizatora kwerend*. Ten komponent współpracuje z silnikiem bazy danych, aby odczytać (lub zapisać) wymagane dane. W proces ten zaangażowany jest *menedżer transakcji*, odpowiedzialny za organizowanie i dystrybucję wyników pojedynczej transakcji. Choć nasza kwerenda nie została uruchomiona w kontekście transakcji jawnej, nadal zostanie wykonana w ramach transakcji niejawnej. Wydajność transakcji może być obniżona, jeśli wybrano nieodpowiedni poziom izolacji transakcji, co omówimy szerzej w rozdziale 10. *Menedżer blokad* odpowiada za blokowanie obiektów w celu zapewnienia spójności transakcyjnej. W przypadku naszej kwerendy możliwe jest odczytanie wierszy, które nigdy nie zostały zatwierdzone, ponieważ użyliśmy wskazówki NOLOCK. Wrócimy do tego zagadnienia w rozdziale 5. *Menedżer buforów* to komponent, który zarządza danymi przechowywanymi w pamięci podręcznej.

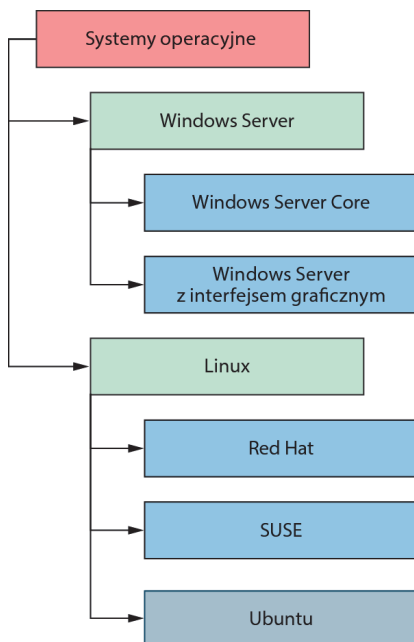
Trzy obszary pamięci podręcznej przedstawiono na rysunku 1.3. *Pamięć podręczna planów* przechowuje złożone plany kwerend, co oznacza, że kolejne wywołania tej samej kwerendy mogą pominąć proces optymalizacji. *Pamięć podręczna dziennika* buforuje rekordy dziennika transakcji przed zapisaniem ich na dysku.

WSKAZÓWKA Wpisy dziennika są zawsze zapisywane na dysku przed zatwierdzeniem transakcji, chyba że stosowana jest opóźniona trwałość.

Pamięć podręczna buforów przechowuje strony danych, które zostały odczytane z dysku. Ważne jest, aby pamiętać, że kwerenda zawsze jest realizowana z użyciem pamięci podręcznej, a nigdy bezpośrednio z danych przechowywanych na dysku. Nawet jeśli wymagane strony danych nie znajdują się w pamięci podręcznej, zostają odczytane z dysku i umieszczone w tej pamięci, po czym kwerenda jest realizowana z wykorzystaniem tych buforowanych danych. Częstym błędem jest pobieranie w kwerendzie większej ilości danych niż to konieczne. Jeśli tak się dzieje, pamięć podręczna buforów zapełnia się szybciej niż powinna. W rezultacie starsze dane są usuwane wcześniej z pamięci podręcznej. To z kolei może prowadzić do pogorszenia wydajności, ponieważ trzeba częściej odczytywać dane z dysku. Omówimy to szerzej w rozdziale 4.

1.2.2. Platformy heterogeniczne

Warto pamiętać, że SQL Server to już nie tylko „baza danych dla Windows”. Obecnie jest on obsługiwany na wielu różnych platformach. Przyjrzyjmy się najpierw systemom operacyjnym, w których działa SQL Server 2022. Systemy te przedstawiono na rysunku 1.4.



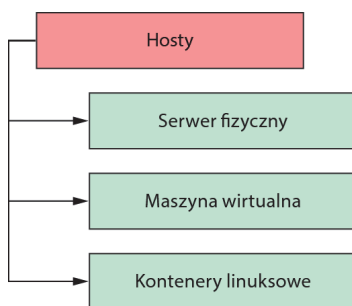
Rysunek 1.4. Systemy operacyjne obsługiwane przez SQL Server

UWAGA Edycje SQL Server Express i Standard można również zainstalować w systemach Windows 10 i Windows 11.

Jak widać, SQL Server można zainstalować nie tylko w *Windows Server Core*, czyli wersji Windows obsługiwanej wyłącznie przez PowerShell, bez interfejsu graficznego, ale także w trzech różnych dystrybucjach Linuksa. Oznacza to, że specjaliści od baz danych mogą pracować nie tylko z interfejsem graficznym, ale również z PowerShellem i Bashem. To otwiera nowe możliwości dla administratorów baz danych, którzy dotychczas byli ograniczeni do jednego środowiska.

Warto również omówić hosty, na których można zainstalować SQL Server 2022, co przedstawiono na rysunku 1.5.

Od pewnego czasu SQL Server jest oficjalnie obsługiwany przez platformę VMware. Bardziej zaskakujące może być to, że najnowsze wersje SQL Servera mogą również działać w kontenerach. Oznacza to, że niektórzy specjaliści od baz danych będą musieli zaznajomić się z takimi technologiami, jak Docker czy Kubernetes.



Rysunek 1.5. Hosty obsługiwane przez SQL Server

UWAGA Kiedy pisałem tę książkę, SQL Server działał tylko w kontenerach linuksowych. Wersja SQL Servera przeznaczona dla kontenerów Windows była dostępna w wersji beta, ale program ten został już zakończony. Nic nie stoi na przeszkodzie, aby stworzyć własne kontenery Windows z SQL Serverem w oparciu o podstawowy obraz kontenera Windows Core. Należy jednak pamiętać, że ponieważ jest to rozwiązanie nieoficjalne, nigdy nie powinno być stosowane w środowisku produkcyjnym.

Warto również rozważyć chmurowe instalacje SQL Servera. SQL Server może działać w chmurze na maszynach wirtualnych w modelu „infrastruktura jako usługa” (ang. *Infrastructure as a Service*, IaaS). W Azure nazywane są one Azure Virtual Machine, a w GCP lub AWS — instancjami EC2. Jednakże dostawcy ci oferują również rozwiązania w modelu „platforma jako usługa” (ang. *Platform as a Service*, PaaS).

W AWS dostępne są obrazy maszyn aplikacyjnych (ang. *Application Machine Image*, AMI) z zainstalowanym systemem SQL Server. W zależności od wybranego obrazu AMI można zakupić SQL Server wraz z instancją EC2 na podstawie umowy licencyjnej z dostawcą usług (ang. *Service Provider License Agreement*, SPLA). W modelu tym licencja jest wliczona w godzinową opłatę za instancję EC2. Można też wykorzystać własną licencję (o ile jest to zgodne z warunkami umowy licencyjnej z firmą Microsoft).

Istnieje również możliwość skorzystania z usługi RDS, która jest rozwiązaniem typu „baza danych jako usługa” (ang. *Database as a Service*, DBaaS). W tym przypadku serwer i instancja SQL Servera są zarządzane przez AWS, a Ty odpowiadasz jedynie za zarządzanie hostowanymi bazami danych.

W Azure możesz tworzyć maszyny wirtualne SQL Servera, korzystając z obrazów Azure VM z zainstalowanym SQL Serverem. Podobnie jak w AWS, dostępny jest model z wliczoną licencją, gdzie koszt licencji SQL Servera jest wliczony w cenę maszyny wirtualnej, co pozwala na płacenie za faktyczne użycie. Możesz również skorzystać z modelu licencjonowania Azure Hybrid Benefit (AHB), który umożliwia wykorzystanie posiadanej już licencji SQL Servera. Dodatkowo w Azure dostępna jest licencja HA/DR, umożliwiająca hostowanie repliki SQL Servera używanej wyłącznie do zapewniania wysokiej dostępności lub usuwania skutków awarii.

Azure oferuje usługę Azure SQL Database, która jest rozwiązaniem typu DBaaS, podobnym do opcji hostingowej AWS RDS. Ponadto dostępna jest usługa SQL Instance as a Service o nazwie Azure SQL Managed Instance. Ta opcja zapewnia równowagę między maszyną wirtualną a rozwiązaniem DBaaS, umożliwiając użytkownikom zarządzanie własną instancją SQL Servera, podczas gdy podstawowym systemem operacyjnym i maszyną wirtualną zarządza Azure.

Wreszcie Azure oferuje rozwiązanie Azure SQL Edge, które zapewnia bazę danych dla internetu rzeczy (IoT) z funkcjami strumieniowania i przetwarzania danych.

1.3. Dlaczego SQL Server jest wciąż wart zachodu?

W ostatnich latach sporo osób powątpiewa, czy SQL Server jest jeszcze wart zachodu. Zazwyczaj wynika to z jednego z dwóch powodów. Pierwszy to założenie, że bazy relacyjne tak naprawdę nie są już potrzebne, ponieważ wszystko korzysta z NoSQL.

To założenie nie jest do końca trafne. Choć w ostatniej dekadzie bazy NoSQL znacząco zyskały na popularności i są dobrym wyborem w wielu przypadkach związanych z analizą danych, to wciąż istnieje miejsce dla baz relacyjnych. Sprowadza się to do starego powiedzenia: „Nie wciskaj kwadratowego kołka w okrągły otwór!”. Po prostu zawsze należy używać odpowiedniego narzędzia do danego zadania. Próba wtłoczenia danych, które naturalnie pasują do bazy relacyjnej, w środowisko nieustrukturyzowane jest równie niewłaściwa jak próba umieszczenia danych nieustrukturyzowanych w bazie SQL Servera.

Drugi powód wynika z założenia, że korzystanie z rozwiązań DBaaS oznacza, iż nie ma już potrzeby samodzielnego uruchamiania serwerów SQL. W rezultacie może się wydawać, że umiejętności związane z SQL Serverem nie są już potrzebne.

Ponownie, to założenie nie jest do końca poprawne. Usługi DBaaS i zarządzane instancje to bardzo przydatne narzędzia w arsenale każdego specjalisty od baz danych. Nie eliminują one jednak potrzeby posiadania umiejętności związanych z SQL Serverem. Wynika to z dwóch powodów. Po pierwsze, nawet bazy danych hostowane w usługach DBaaS wymagają napisania i zoptymalizowania kodu w celu uniknięcia niskiej wydajności. Po drugie, DBaaS lub nawet zarządzane instancje nie zawsze są dobrym wyborem. Czasami wręcz nie da się ich zastosować w danej sytuacji. Przyczyn może być kilka, a jedną z nich jest koszt. Płacisz dostawcy usług chmurowych za hostowanie, aktualizowanie, zabezpieczanie i optymalizację wszystkich warstw stosu poniżej bazy danych. Oznacza to dodatkowe koszty. Jeśli przeniesiesz każdą bazę danych do RDS lub

Azure SQL Database, możesz zrekompensować te koszty, rezygnując z zatrudniania administratorów baz danych. Jeśli jednak z jakichś powodów nie możesz przenieść wszystkich swoich baz danych do DBaaS, skończysz płacąc podwójnie.

Fizyczni i logiczni administratorzy baz danych

Fizyczny administrator bazy danych to termin opisujący DBA, który dobrze zna się na zarządzaniu instancjami SQL Servera, ale nie ma zaawansowanych umiejętności w zakresie warstwy bazy danych. Stanowi to przeciwieństwo logicznych administratorów baz danych, którzy specjalizują się w tworzeniu i optymalizacji baz danych, ale nie mają doświadczenia w zarządzaniu instancjami.

Spotkałem się z taką strukturą zespołów bazodanowych w firmach korzystających z usług zewnętrznego dostawcy infrastruktury, który buduje i utrzymuje instancje SQL Servera, dbając o ich dostępność. Odpowiedzialność za bazy danych jest często przekazywana wewnętrznemu zespołowi wsparcia aplikacji, który zajmuje się obsługą i optymalizacją elementów na poziomie bazy danych.

Z mojego doświadczenia wynika, że prawie nigdy nie da się przenieść wszystkich baz danych do DBaaS. Jednym z powodów jest wsparcie ze strony dostawców oprogramowania. Jeśli korzystasz z gotowego, komercyjnego produktu opartego na bazie danych SQL Server, to jesteś na łasce dostawcy, który musi wyrazić zgodę na wspieranie produktu po przeniesieniu bazy do chmury.

Kolejnym powodem jest to, że wiele firm posiada duże, złożone aplikacje bazodanowe, czasami zawierające setki tysięcy wierszy kodu wbudowanego w komponenty takie jak SQL Server Integration Services. Często nie można ich po prostu „dźwignąć i przesunąć”; pod określeniem tym kryje się migracja aplikacji bazodanowej w niezmienionej formie, bez żadnych przekształceń ani modernizacji. Zwykle wymaga to przebudowy architektury, co wiąże się ze znacznymi nakładami czasu i środków, które często trudno jest uzasadnić.

1.4. Dlaczego właściwe korzystanie z SQL Servera jest ważne?

W młodości grywałem w grę planszową o nazwie Othello. Towarzyszący jej slogan brzmiał: „Minuta, żeby się nauczyć, całe życie, żeby opanować”. SQL Server zawsze mi o niej przypomina. Jedną z zalet SQL Servera w porównaniu z konkurencyjnymi rozwiązaniami jest to, że bardzo łatwo go wdrożyć. Trudno jednak zrobić to dobrze.

Ma to znaczenie, ponieważ źle wdrożona instancja SQL Servera może prowadzić do różnych problemów, takich jak:

- niska wydajność,
- kradzież danych,
- ataki ransomware,
- niezgodność z przepisami,
- utrata danych w przypadku awarii lub katastrofy,
- długotrwały przestój w przypadku katastrofy.

Niewłaściwie zaimplementowany kod bazy danych może prowadzić do:

- niskiej wydajności,
- naruszeń bezpieczeństwa,
- nieobsłużonych awarii,
- kodu, którego nie da się rozwijać.

Problemy te mogą prowadzić do ogromnych wydatków, utraty przychodów, nadszarpnięcia reputacji, zwiększonej rotacji pracowników, a nawet postępowań sądowych. Dlatego właściwe wdrożenie i obsługa SQL Servera mają kluczowe znaczenie.

Podsumowanie

- Źródłem większości pomyłek popełnianych przez użytkowników SQL Servera jest błędne założenie, że bazy danych są proste i łatwe w obsłudze.
- SQL Server to popularny system zarządzania relacyjnymi bazami danych.
- SQL Server to rozbudowany i złożony ekosystem składający się z wielu komponentów.
- Głównym komponentem SQL Servera jest silnik bazy danych.
- Silnik bazy danych składa się z takich elementów, jak silnik relacyjny oraz silnik przechowywania danych
- SQL Server może działać zarówno w systemie Windows, jak i Linux.
- SQL Server może działać na serwerach fizycznych, maszynach wirtualnych oraz w kontenerach linuxowych.
- SQL Server jest dostępny w chmurze publicznej jako usługa IaaS lub PaaS.
- Prawidłowe wdrożenie SQL Servera pozwala uniknąć takich problemów, jak niska wydajność, utrata danych czy naruszenia bezpieczeństwa.

Standardy programowania



W tym rozdziale:

- Wprowadzenie do standardów programowania
- Konwencje nazewnicze
- Standardy kodowania

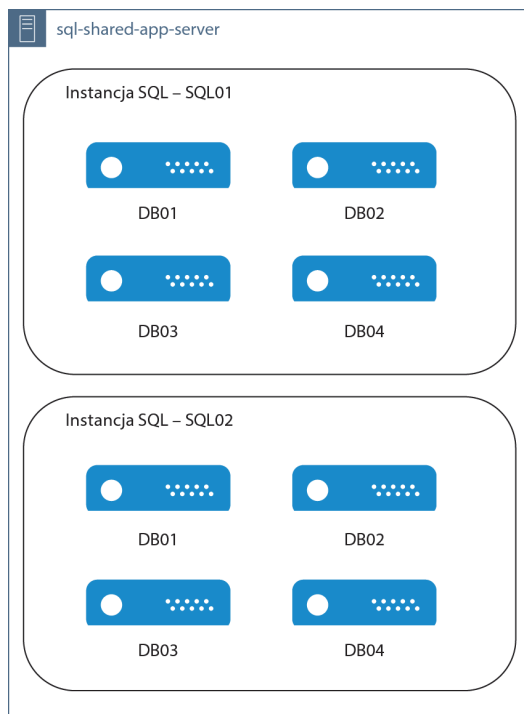
W tym rozdziale zbadamy, dlaczego określenie (i przestrzeganie!) standardów programowania jest tak istotne. Zaczniemy od zdefiniowania, co dokładnie mamy na myśli, mówiąc o standardach programowania w kontekście tworzenia aplikacji bazodanowych z wykorzystaniem SQL Servera. Standardy programowania obejmują następujące obszary:

- konwencje nazewnicze,
- standardy kodowania,
 - stylistyczne,
 - techniczne.

W szczególności przyjrzymy się typowym błędom popełnianym przez specjalistów od baz danych i porozmawiamy o tym, jak ich unikać.

2.1. Przykładowa pomyłka

Wyobraź sobie topologię przedstawioną na rysunku 2.1. Załóżmy, że odbierasz telefon od osób odpowiedzialnych za aplikację o nazwie TimeChewer. Zgłaszają one, że niektóre kwerendy działają powoli, i proszą Cię, żebyś to sprawdził. Informują, że baza danych aplikacji jest hostowana na serwerze o nazwie sql-shared-app-server.



Rysunek 2.1. Topologia SQL Server ze złymi konwencjami nazewniczymi

Łączysz się z serwerem i czujesz, jak serce Ci się ściska. Instancje noszą nazwy SQL01 i SQL02. Nie daje Ci to żadnej wskazówki, która z nich może zawierać bazę danych aplikacji. Postanawiasz więc połączyć się z każdą po kolei, mając nadzieję na znalezienie bazy o nazwie przypominającej „TimeChewer”. Niestety, witają Cię bazy danych o nazwach DB01, DB02, DB03 i DB04.

Teraz musisz skontaktować się z zespołem odpowiedzialnym za aplikację i poprosić o znalezienie łańcucha połączenia oraz ustalenie, z którą bazą danych i z którą instancją łączy się aplikacja. W końcu otrzymujesz odpowiedź, że TimeChewer łączy się z bazą DB03 na serwerze SQL01. Postanawiasz więc przyjrzeć się jednej z procedur składowanych, które podobno działają niewydajnie, i przeanalizować jej definicję. Definicja tej procedury wygląda tak:


```
ALTER PROCEDURE dbo.proc01
AS
BEGIN
;with t3 as (select col1, col2, col3 from tbl03) select t1.col1, t2.col2, t1.col2, t1.col3,
t3.col1, t3.col2 from dbo.tbl01 t1 inner join tbl02 t2 on t1.col1 = t2.col1 and t1.col3 < 55
inner join t3 on t3.col1 = t1.col1 union select col1, col2, col3, NULL, NULL, '0' from
dbo.tbl04 ;
END
```

Jak przypuszczam, wpatrujesz się teraz w tę procedurę składowaną, próbując zrozumieć, co ona właściwie robi. W skrócie, po spędzeniu 30 minut na ustalaniu, która baza danych ma problem, będziesz musiał poświęcić kolejne 15 minut na przeformatowanie kodu, żeby był bardziej czytelny, i próbę rozszyfrowania, jak działa ta bardzo prosta procedura składowana. Zmarnujesz 45 minut, zanim w ogóle zaczniesz badać problem.

Choć jest to skrajny przykład, dobrze ilustruje, dlaczego konwencje nazewnictwa i standardy kodowania są tak istotne i czemu lekceważenie ich prowadzi prostą drogą do frustracji.

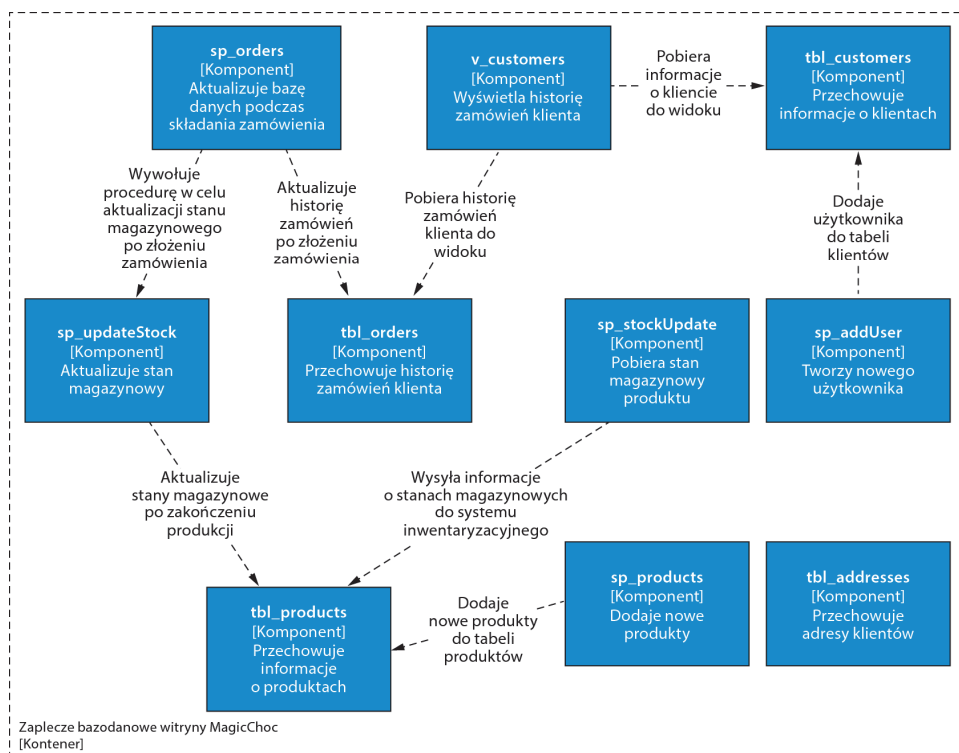
2.2. Numer 1 — nieopisowe nazwy obiektów

Choć przykład nieopisowych nazw baz danych przedstawiony we wstępie do tego rozdziału był dość skrajny, zdarzało mi się spotykać przypadki, gdy ktoś używał SQL01, SQL02 itp. jako nazw instancji, co naprawdę może przyprawić o ból głowy. Nieopisowe nazewnictwo jest jednak bardziej rozpowszechnione na poziomie kodu i to właśnie na tym aspekcie skupimy się w tym rozdziale.

WSKAZÓWKA Więcej informacji na temat problemów wynikających z niewłaściwego nazywania instancji znajdziesz w rozdziale 8.

Aby omówić to zagadnienie, wróćmy do przykładowej firmy MagicChoc z rozdziału 1. W tym rozdziale zajmujemy się bazą danych SalesDB, która stanowi zaplecze dla strony internetowej. Rysunek 2.2 to diagram komponentów tej bazy danych, na którym przedstawiono obiekty bazy danych (tabele, procedury i widoki) znajdujące się w jej strukturze.

Zauważ, że nazwy obiektów w tej bazie danych nie zostały dobrze przemyślane. Zawsze powinieneś dążyć do tego, aby kod sam się dokumentował. Wbrew powszechnemu przekonaniu nie oznacza to konieczności zaśmiecania kodu komentarzami. Komentarze mogą być pomocne w wyjaśnianiu skomplikowanej logiki, ale o wiele ważniejsze jest, aby kod był napisany i ustrukturyzowany w taki sposób, żeby każdy, kto go czyta (przy założeniu, że zna język programowania), mógł zrozumieć, co ten kod ma osiągnąć. Opisowe nazwy obiektów są kluczowym elementem tego procesu.



Rysunek 2.2. Diagram komponentów bazy danych SalesDB

Wyobraź sobie, że poproszono Cię o naprawienie błędu w procedurze aktualizującej system inwentaryzacyjny po złożeniu zamówienia. Ponieważ nie znasz dokładnie tego procesu, Twoim pierwszym krokiem może być uruchomienie kwerendy zwracającej listę procedur składowanych w bazie danych. Poniżej (listing 2.1) przedstawiono trzy sposoby wykonania tej czynności. Wszystkie trzy kwerendy zwrócą te same wyniki.

Listing 2.1. Zwracanie listy procedur składowanych

```

SELECT
    name
FROM sys.procedures ; ← Pobiera listę z systemowego widoku .procedures.

SELECT
    name
FROM sys.objects
WHERE type_desc = 'SQL_STORED_PROCEDURE' ; ← Pobiera listę z systemowego widoku
                                              sys.objects, filtrując ją według opisu typu.

SELECT
    name
FROM sys.objects
WHERE type = 'P' ; ← Pobiera listę z systemowego widoku sys.objects, filtrując ją według typu.
  
```

Oczywiście możesz również skorzystać z SQL Server Management Studio (SSMS) i przejrzeć procedury za pomocą interfejsu graficznego. Niezależnie od tego, jaką metodę wybierzesz do wyświetlenia obiektów, zauważysz, że istnieją procedury o nazwach `updateStock` i `stockUpdate`. Która z nich jest tą, którą musisz debugować? Trudno powiedzieć. Zanim rozpoczniesz swoje zadanie, będziesz musiał przejrzeć definicje obu procedur składowanych albo sprawdzić, gdzie dana procedura jest wywoływana, aby ustalić nazwę właściwej procedury.

W tym przypadku ustalenie, skąd wywoływana jest procedura składowana, będzie nieco skomplikowane, ponieważ jest ona faktycznie wywoływana przez inną procedurę składowaną — `sp_orders`. Podobnie jak poprzednio, ta procedura ma ogólną, mało znaczącą nazwę, która nie wyjaśnia jej celu. Dlatego będziesz musiał cofnąć się do aplikacji, która zainicjowała cały proces, aby odkryć pierwszą procedurę w łańcuchu wywołań.

Wyobraźmy sobie teraz, że udało Ci się ustalić, iż procedura `orders` jest pierwszą procedurą składowaną w łańcuchu. Definicję tej procedury możesz znaleźć za pomocą SSMS lub przez wykonanie kwerendy przedstawionej na listingu 2.2.

Listing 2.2. Pobieranie definicji procedury składowanej

```
SELECT s.definition
FROM sys.objects o
INNER JOIN sys.sql_modules s
    ON o.object_id = s.object_id
WHERE o.name = 'sp_orders' ;
```

Łączy widok `sys.sql_modules`, który przechowuje definicje procedur, z widokiem `sys.objects` zawierającym nazwy obiektów. Połączenie to odbywa się na podstawie kolumny `object_id`, która występuje w obu widokach katalogowych.

WSKAZÓWKA Osobiście wolę szukać definicji procedury składowanej w eksploratorze obiektów w SSMS, zamiast pobierać ją z widoku katalogowego `sys.sql_modules`. Wynika to stąd, że pobieranie z `sys.sql_modules` nie zachowuje formatowania.

Definicję procedury składowanej `sp_orders` można znaleźć na listingu 2.3. Parametr `@AddressID` jest niejasny — czy odnosi się do adresu rozliczeniowego, czy adresu dostawy? Parametr `@Address` jest zarówno niejasny, jak i mylący. Jego nazwa sugeruje rzeczywisty adres, podczas gdy ma on przechowywać identyfikator adresu. Ponadto nie wiadomo, o który adres chodzi. Kolejnym niejasnym parametrem jest `@date`. Czy to data zamówienia? A może data dostawy? Mógłby to nawet być znacznik czasowy wstawienia rekordu do tabeli! W treści procedury występują również mylące nazwy zmiennych. Zmienna `@stock` przechowuje zamówioną ilość produktu, a `@product` przechowuje identyfikator produktu. Te nazwy po prostu nie są klarowne i trzeba przeanalizować kod, aby zrozumieć ich przeznaczenie.

Listing 2.3. Definicja procedury `sp_orders`

```
CREATE PROCEDURE sp_orders
    @CustomerID INT,
    @LineItems XML,
```

```

    @AddressID INT,
    @Address INT,
    @date DATETIME
AS
BEGIN
    DECLARE @Stock INT = 0 ;
    DECLARE @Product INT = 0 ;

    INSERT INTO tbl_orders (
        CustomerID,
        LineItems,
        BillingAddressID,
        DeliveryAddressID,
        Date
    )
    VALUES (
        @CustomerID,
        @LineItems,
        @AddressID,
        @Address,
        @date
    ) ;

    SET @Stock = @LineItems.value('/Product/@qty')[1]', 'int') ;
    SET @Product = @LineItems.value('/Product/@ProductID')[1]', 'int') ;

    EXEC sp_stockUpdate @product, @stock ;
END

```

WSKAZÓWKA Jeśli nie wiesz, jak używać XML w SQL Serverze, nie jesteś w tym odosobniony. W rzeczywistości niewłaściwe wykorzystanie XML to inna częsta pomyłka, której przyjrzymy się bliżej w rozdziale 3.

Podczas analizy definicji tej procedury składowanej już na pierwszy rzut oka widać, że niejednoznaczne nazwy parametrów i zmiennych mogą być źródłem problemów w przyszłości. Jeśli konieczne będzie ulepszenie lub zdiagnozowanie funkcji związanych z adresami lub czasem, trzeba będzie dodatkowej pracy, by zrozumieć znaczenie danych. Pokazuje to, jak ważne jest stosowanie czytelnych i opisowych nazw w kodzie.

Na razie musimy jednak naprawić problem z aktualizacją stanów magazynowych. Definicja tej procedury składowanej pozwoliła nam ustalić, że kod, który wymaga debugowania, prawdopodobnie znajduje się w procedurze `sp_stock ↵Update`. Procedurę tę przedstawiono na listingu 2.4.

Listing 2.4. Definicja procedury `sp_stockUpdate`

```

CREATE PROCEDURE sp_stockUpdate
    @ProductStockLevel INT,
    @StockID INT
AS
BEGIN
    UPDATE tbl_products
    SET StockQty = StockQty @productStockLevel

```

Nazwy parametrów są niespójne zarówno z procedurą `sp_orders`, jak i z nazwami kolumn w tabeli.

```
WHERE ProductID = @StockID ;

UPDATE [DCSVR01\Inventory].InventoryDB.dbo.productStock
SET StockQty = StockQty @productStockLevel
WHERE ProductID = @StockID ;
END
```

Jak widać, `sp_stockUpdate` to prosta procedura składowana, która aktualizuje stan magazynowy w bazie `SalesDB`, a następnie wykorzystuje połączenie z serwerem, aby zaktualizować stan magazynowy w systemie inwentaryzacyjnym. Widać też, że przyczyną błędu są niespójne nazwy, które sprawiły, że programista się pomylił i przekazał identyfikator produktu do obliczenia, które aktualizuje pozostały stan magazynowy, jednocześnie filtrując produkty na podstawie ilości zamówionego towaru zamiast rzeczywistego identyfikatora produktu.

Przykład z życia wzięty

Ten przykład jest dość prosty i ma charakter poglądowy, ale możesz sobie wyobrazić, że w pracy ze skomplikowanym kodem takie problemy szybko stają się koszmarem. Przykład w tym podrozdziale luźno opiera się na rzeczywistym przypadku, z którym się spotkałem. Pewien programista stworzył bardzo złożony proces składający się z pięciu warstw procedur składowanych i wielu punktów wejścia z zewnętrznych aplikacji. Łącznie wszystkie procedury liczyły około 2000 wierszy kodu. Po odejściu programisty odkryto błąd, a próba jego rozwiązania ujawniła, że programista używał zamiennie nazw zmiennych, parametrów i kolumn. Sam błąd był prosty i powinien zostać naprawiony w ciągu kilku godzin, ale ze względu na splątaną sieć nazewnictwa rozwiązanie problemu zajęło trzy dni. Następnie potrzeba było kolejnych dwóch tygodni, aby ujednolicić wszystkie nazwy i przeprowadzić testy regresyjne.

WSKAZÓWKA Przykładem dobrego nazewnictwa na listingu 2.4 są nazwy kolumn kluczy głównych w tabelach. Zauważ, że tabela `tbl_products` ma kolumnę `ProductID`, tabela `tbl_orders` ma kolumnę `OrderID`, a tabela `tbl_addresses` ma kolumnę `AddressID`. Częstym błędem jest używanie samego ID jako nazwy kolumny klucza głównego we wszystkich tabelach, co oznacza, że kolumna klucza głównego w każdej tabeli ma tę samą nazwę. Łatwo sobie wyobrazić, jak mylące może to być w kontekście złożonej procedury składowanej.

Jak się przekonałeś, niepoświęcenie odpowiedniej uwagi nazewnictwu może wprowadzać błędy, powodować konieczność wykonywania pracy związanej z próbami naprawy problemów i utrudniać rozszerzanie funkcjonalności. Dlatego warto poświęcić trochę czasu na wymyślenie spójnych i znaczących nazw.

Może się to wydawać proste i zazwyczaj tak jest w przypadku małych projektów bazodanowych. Jednak przy dużych projektach, tworzonych przez dłuższy czas przez wielu programistów, sprawy mogą się znacznie skomplikować. Jeszcze trudniej jest, gdy projekt realizowany jest zgodnie z metodyką *zwinną*. W przeciwieństwie do tradycyjnego modelu kaskadowego projekty zwinne wykorzystują metody takie jak *Sprint* czy *Kanban*, które dzielą zaległe zadania na małe fragmenty, co pozwala dostarczać przyrostową funkcjonalność w szybszym tempie.

Podejście zwinne może działać bardzo dobrze i przynosić wiele korzyści w porównaniu z projektami kaskadowymi. Kiedy jednak jest źle realizowane, może pojawić się błędne założenie, że projektowanie, architektura i planowanie nie są potrzebne. W projektach bazodanowych architektura zawsze ma kluczowe znaczenie, ponieważ należy uwzględnić istotne elementy projektowe, takie jak schemat tabel. Zaniedbanie tego etapu prawdopodobnie doprowadzi do chaotycznego zbioru nienormalizowanych tabel, powielania danych, problemów z wydajnością oraz zwiększonej złożoności. Skoro i tak poświęcamy czas na wstępne rozważania nad projektem tabel, to prostym i stosunkowo szybkim zadaniem jest również przyjrzenie się innym elementom projektowym, takim jak nazewnictwo obiektów.

W dobrze zarządzanym projekcie bazodanowym wykorzystującym metodykę zwinną na początku projektu przeprowadza się kilka sprintów projektowych lub okresowe sprinty, podczas których projektowana jest część bazy danych (zwykle jeden schemat).

Co zatem powinien był zrobić programista, który napisał procedury składowane MagicChoc? Przyjrzyjmy się nowej definicji procedury `sp_orders` przedstawionej na listingu 2.5. Po pierwsze, parametry adresowe są teraz wyraźnie oznaczone jako identyfikatory. Co ważniejsze, jest teraz jasne, który parametr reprezentuje adres rozliczeniowy, a który adres dostawy. Parametr `@date` jest teraz jednoznacznie datą zamówienia. Nie tylko zmieniliśmy nazwę parametru, ale również zaktualizowaliśmy nazwę kolumny w tabeli, co usuwa niejednoznaczność na tym poziomie. Zmienna `@product` została zmieniona na `@ProductID`, a `@stock` na `@OrderQty`. Dodatkowo zaktualizowaliśmy definicję XML, aby pasowała do nowych nazw, co sprawia, że dane są bardziej czytelne na wszystkich poziomach. Na koniec poprawiliśmy wywołanie procedury `sp_updateProductStockLevel`, aby przekazywać zmienne w odpowiedniej kolejności. Ta ostatnia zmiana naprawia błąd.

Listing 2.5. Nowa definicja procedury `sp_orders`

```
ALTER PROCEDURE sp_orders
    @CustomerID INT,
    @LineItems XML,
    @BillingAddressID INT,
    @DeliveryAddressID INT,
    @OrderDate DATETIME
AS
BEGIN
    DECLARE @OrderQty INT = 0 ;
    DECLARE @ProductID INT = 0 ;

    INSERT INTO tbl_orders (CustomerID, LineItems, BillingAddressID, DeliveryAddressID,
        ↳OrderDate)
    VALUES (@CustomerID, @LineItems, @BillingAddressID, @DeliveryAddressID, @OrderDate) ;
```

```

SET @OrderQty = @LineItems.value('/Product/@OrderQty')[1]', 'int') ;
SET @ProductID = @LineItems.value('/Product/@ProductID')[1]', 'int') ;

EXEC sp_updateProductStockLevel @OrderQty, @ProductID ;
END

```

WSKAZÓWKA Jak powiedziała by mój ulubiony prywatny detektyw z lat 80.: „Wiem, o czym myślisz, i masz rację. Pewnie powinniśmy zmienić również nazwę procedury?”. Jednak na razie zostawiłem ją bez zmian, ponieważ omówimy tę kwestię dokładniej, kiedy zajmiemy się pomyłką numer 2.

Rozważ też nową definicję procedury `sp_stockUpdate` na listingu 2.6. Zauważysz, że nazwa procedury składowanej została zmieniona na znacznie bardziej opisową — `sp_updateProductStockLevel`. Dzięki takiej nazwie nie mielibyśmy problemu ze znalezieniem procedury na początku tego przykładu.

Listing 2.6. Definicja procedury `sp_updateProductStockLevel`

```

CREATE PROCEDURE sp_updateProductStockLevel
    @OrderQty INT,
    @ProductID INT ← Nazwy parametrów są teraz zgodne z nazwami zmiennych
                     przekazywanych do funkcji.
AS
BEGIN
    UPDATE tbl_products
    SET StockQty = StockQty @OrderQty
    WHERE ProductID = @ProductID ;

    UPDATE [DCSVR01\Inventory].InventoryDB.dbo.productStock
    SET StockQty = StockQty @OrderQty
    WHERE ProductID = @ProductID ;
END

```

Po przeanalizowaniu definicji dwóch nowych procedur składowanych zauważysz, że znaczące i spójne nazewnictwo w istotny sposób pomaga w tworzeniu samodokumentującego się kodu. Choć moglibyśmy dodać komentarze do tych procedur, przyniosłoby to raptem niewielką korzyść. Każdy, kto zna T-SQL, może teraz przeczytać te definicje i odtąd będzie dla niego jasne, co robią te procedury i jakie dane są przekazywane. Nie oznacza to, że komentarze nie mają swojego zastosowania. Są one przydatne głównie do wyjaśnienia koncepcyjnej logiki biznesowej w bardzo długich procedurach składowanych lub do określania zewnętrznych danych wejściowych i wyjściowych, na przykład w kontrakcie API. Jeśli Twój kod sam się dokumentuje, komentarze nie powinny być konieczne do wyjaśnienia niskopoziomych definicji.

2.3. Numer 2 — używanie prefiksów obiektowych

Niektórzy programiści lubią stosować prefiksy obiektowe. Praktyka ta polega na dodawaniu prefiksu `tbl_` na początku nazw wszystkich tabel, `sp_` — na początku nazw procedur składowanych, `fn_` — na początku nazw funkcji, a `v_` — na początku nazw widoków.

Wyobraź sobie, że masz bazę danych zawierającą 250 tabel. Szukasz definicji kolumny w jednej z tych tabel, więc szybko przeglądasz listę tabel w eksploratorze obiektów. Jednak wszystkie nazwy tabel mają prefiks `tbl_`. To automatycznie wydłuża czas analizy podczas przeglądania listy.

Dzieje się tak dlatego, że Twój mózg jest zaprogramowany do rozpoznawania typowych wzorców. Weźmy na przykład alfabet — znasz go całkiem dobrze, prawda? A teraz spróbuj wymienić litery od końca, od Z do A. To zaskakująco trudne. Dlaczego? Bo Twój mózg nie jest zaprogramowany na taki wzorec. Nawet jeśli nauczysz się go na pamięć, nie oznacza to, że dla innych będzie to równie łatwe.

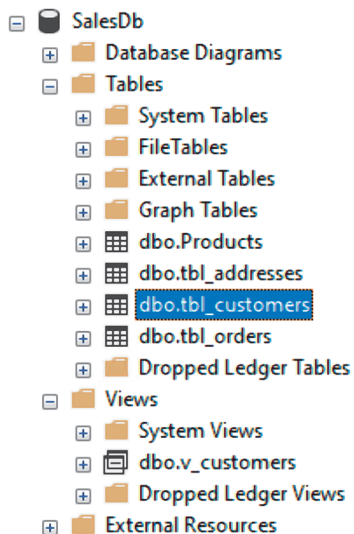
To samo dotyczy przeglądania obiektów z prefiksami. Nawet jeśli Ty nauczysz się czytać łańcuch od piątego znaku, inni programiści, analitycy biznesowi czy administratorzy mogą nie mieć takiej umiejętności. W ten sposób nieświadomie utrudniasz im pracę i obniżasz ich wydajność.

Gdyby istniał dobry powód, aby to robić — a w niektórych językach, takich jak ARM w Azure, faktycznie istnieje — rozważyłbym taki kompromis. Jednak w przypadku SQL Servera nie ma żadnego uzasadnienia dla takiego podejścia. Argument, który zwykle się przytacza, brzmi: „Bo potrzebuję łatwego sposobu na identyfikowanie typów obiektów”.

Wykorzystajmy więc przykład `MagicChoc` do przeanalizowania tego zagadnienia z kilku perspektyw. W bazie `SalesDB` mamy tabelę `tbl_Orders` i procedurę składowaną `sp_orders`. Dodatkowo istnieją tabela `tbl_customers` oraz widok `v_customers`. Pierwszą perspektywą jest prosta identyfikacja typu obiektu. Rozważmy to na przykładzie obiektów związanych z klientami. Możemy zbadać te obiekty nad dwa sposoby — za pośrednictwem SSMS albo z poziomu kodu. Na rysunku 2.3 pokazano oba obiekty w eksploratorze obiektów.

Jak widać, łatwo odróżnić tabelę od widoku. Tabela jest wyświetlana w węźle *Tables*, natomiast widok znajduje się w węźle *Views*.

A gdybyś chciał przeglądać obiekty programowo? Kwerenda z listingu 2.7 pokazuje, jak za pomocą T-SQL pobrać szczegółowe informacje o obu obiektach. Zauważ, że łatwo jest odróżnić dzięki kolumnie `type_desc` w tabeli `sys.objects`.



Rysunek 2.3. Przeglądanie obiektów w eksploratorze

Listing 2.7. Pobieranie obiektów związanych z klientami

```
SELECT
    name
    , type_desc
FROM sys.objects
WHERE name LIKE '%customers%' ;
```

A oto wyniki tej kwerendy:

| Name | type_desc |
|---------------|------------|
| tbl_customers | USER_TABLE |
| v_customers | VIEW |

Ów legendarny detektyw z lat 80. powiedziałby na to: „Wiem, co myślisz, i masz rację. Gdybyśmy usunęli prefiks, tabela i widok customers miałyby identyczną nazwę, a to jest niedozwolone”. I w ten sposób płynnie przechodzimy do drugiej perspektywy.

Kiedy omawialiśmy pierwszą pomyłkę, wspomnieliśmy o nadawaniu obiektom znaczących nazw, które sprawiają, że kod sam się dokumentuje. Zastanówmy się nad tym zagadnieniem z perspektywy obiektów związanych z klientami. Tabela i widok mają dokładnie taką samą nazwę. Jeśli nie pełnią dokładnie tej samej funkcji, nie może być to prawidłowe rozwiązanie, chyba że widok zwraca po prostu wszystkie dane z jednej tabeli. A jeśli widok zwraca te same dane co tabela, to po co w ogóle istnieje?

Powody tworzenia widoków odwzorowujących dokładnie jedną tabelę

Widoki zwracające wszystkie kolumny z jednej tabeli tworzy się z dwóch powodów. Pierwszym z nich jest zablokowanie tabel w celu uniknięcia przypadkowych zmian w ich strukturze. Jeśli utworzysz widok z opcją `WITH SCHEMABINDING`, nie będzie możliwe zmodyfikowanie definicji tabeli bez uprzedniego usunięcia opcji `SCHEMABINDING` z widoku. Nie polecam jednak tego podejścia. Znacznie lepszym i bardziej zrozumiałym rozwiązaniem jest po prostu ograniczenie dostępu do modyfikacji struktury bazy danych poprzez odpowiednią strategię bezpieczeństwa. Istnieje zasada projektowa zwana „zasadą najmniejszego zaskoczenia”, której jestem zwolennikiem.

Drugim powodem tworzenia widoku zwracającego wszystkie kolumny z jednej tabeli są rygorystyczne zasady, które wymagają, aby wszystkie aplikacje korzystały z danych poprzez warstwę abstrakcji. Zasadniczo popieram ideę stojącą za tym podejściem, które polega na ukryciu złożoności na poziomie serwera SQL i umożliwieniu programistom aplikacji prostego pobierania danych z widoków i procedur składowanych. Kiedy jednak jest ono posunięte do skrajności, prowadzi to do tworzenia dodatkowych obiektów bez realnych korzyści. W rezultacie masz po prostu więcej obiektów do zarządzania.

W bazie danych SalesDB tabela `tbl_customers` przechowuje informacje o klientach, więc jej nazwa jest adekwatna. Natomiast widok `v_customers` zwraca zarówno dane klientów, jak i dane zamówień, dlatego jego nazwa nie jest odpowiednia. Bardziej trafna nazwa mogłaby brzmieć `customerOrders`.

Tabela `tbl_addresses` nie ma podobnie nazwanych widoków, procedur ani funkcji, więc możemy po prostu usunąć jej prefiks. Tabela `tbl_products` ma podobnie nazwaną procedurę składowaną, ale procedura ma obecnie prefiks `sp_`, więc również możemy zmienić nazwę tabeli. Powinniśmy jednak zmienić nazwę procedury składowanej na `sp_addOrder`, aby była bardziej znacząca.

WSKAZÓWKĄ Jak powiedziałby detektyw Magnum: „Wiem, co myślisz, i masz rację. Czy nie powinniśmy usunąć prefiksu `sp_` z nazw procedur składowanych?”. Owszem, powinniśmy, ale omówimy to dokładniej przy okazji pomyłki numer 3. Na razie zostawimy je w obecnej formie.

Skrypt przedstawiony na listingu 2.8 usunie prefiks z tabel i widoku.

Listing 2.8. Usuwanie prefiksu

```
EXEC sp_rename 'tbl_addresses', 'addresses' ;

EXEC sp_rename 'tbl_orders', 'orders' ;

EXEC sp_rename 'tbl_customers', 'customers' ; ← sp_rename automatycznie aktualizuje
DROP VIEW dbo.v_customers ;                 ograniczenia klucza obcego.
GO

CREATE VIEW dbo.customerOrders ← Usuwa widok i tworzy go pod nową nazwą. Pozwala to zmienić
AS                                          również definicję tak, aby używała nowych nazw tabel.
SELECT
```

```

        c.FirstName
    , c.LastName
    , c.email
    , o.LineItems.value('/Product/@ProductName)[1]', 'int') AS ProductName
    , o.LineItems.value('/Product/@OrderQty)[1]', 'int') AS OrderQty
    , o.OrderDate
FROM dbo.customers c
INNER JOIN dbo.orders o
    ON c.CustomerID = o.CustomerID ;
GO

DROP PROCEDURE dbo.sp_orders ;
GO

CREATE PROCEDURE dbo.sp_addOrder ←
    @CustomerID INT,
    @LineItems XML,
    @BillingAddressID INT,
    @DeliveryAddressID INT,
    @OrderDate DATETIME    @OrderQty INT,
    @ProductID INT
AS
BEGIN
    DECLARE @OrderQty INT = 0 ;
    DECLARE @ProductID INT = 0 ;

    INSERT INTO orders (
        CustomerID,
        LineItems,
        BillingAddressID,
        DeliveryAddressID,
        OrderDate
    )
    VALUES (
        @CustomerID,
        @LineItems,
        @BillingAddressID,
        @DeliveryAddressID,
        @OrderDate
    ) ;

    SET @OrderQty = @LineItems.value('/Product/@OrderQty)[1]', 'int') ;
    SET @ProductID = @LineItems.value('/Product/@ProductID)[1]', 'int') ;

    EXEC sp_updateProductStockLevel @OrderQty, @ProductID ;
END
GO

DROP PROCEDURE dbo.sp_updateProductStockLevel ;
GO

CREATE PROCEDURE dbo.sp_updateProductStockLevel ←
    @OrderQty INT,
    @ProductID INT
AS
BEGIN
    UPDATE products
    SET StockQty = StockQty @OrderQty

```

Choć można używać `sp_rename` na procedurach składowanych, musimy również zmienić definicje tak, aby używały nowych nazw tabel.

Choć można używać `sp_rename` na procedurach składowanych, musimy również zmienić definicje tak, aby używały nowych nazw tabel.

```

WHERE ProductID = @ProductID ;

UPDATE [DCSVR01\Inventory].InventoryDB.dbo.productStock
SET StockQty = StockQty @OrderQty
WHERE ProductID = @ProductID ;
END

```

2.4. Numer 3 — niesławny prefiks sp_

W pomyłce numer 2 wyjaśniłem, dlaczego prefiksy nie są dobrym pomysłem, więc z jakiego powodu poświęcam osobny podrozdział prefiksowi sp_? Najlepiej wyjaśnić to na przykładzie. Rozważmy definicję procedury składowanej sp_AddUser w bazie SalesDB, którą to definicję znajdziesz na listingu 2.9. Procedura ta po prostu dodaje nowego użytkownika do aplikacji, gdy rejestruje się on przez stronę internetową MagicChoc.

Listing 2.9. Definicja procedury sp_AddUser

```

CREATE PROCEDURE sp_addUser
    @UserDetails XML
AS
BEGIN
    OPEN SYMMETRIC KEY MagicChocKey
        DECRYPTION BY CERTIFICATE MagicChocCertificate ;

    INSERT INTO dbo.customers (
        FirstName
        , LastName
        , email
        , UserPassword
    )
    VALUES (
        @UserDetails.value('/User/FirstName')[1], 'nvarchar(128)')
        , @UserDetails.value('/User/LastName')[1], 'nvarchar(128)')
        , @UserDetails.value('/User/email')[1], 'nvarchar(512)')
        , ENCRYPTBYKEY(KEY_GUID('MagicChocKey'), @UserDetails.
            ↳value('/User/UserPassword')[1], 'nvarchar (128)'))
    ) ;

    CLOSE SYMMETRIC KEY MagicChocKey ;
END

```

Informacje o nowym użytkowniku są przekazywane z aplikacji jako fragment kodu XML.

Do wyodrębnienia wartości z fragmentu XML i wstawienia ich do tabeli customers używana jest metoda value języka xQuery. Hasło jest szyfrowane przed wstawieniem do tabeli.

Aplikacja podobno zgłasza dziwny błąd. Aby zdiagnozować problem, możemy zasymulować wywołanie procedury składowanej przez aplikację, używając skryptu przedstawionego na listingu 2.10.

Listing 2.10. Wywoływanie procedury składowanej sp_AddUser

```

DECLARE @UserDetails XML ; ← Deklaruje zmienną typu XML.

SET @UserDetails = N'<User>
  <FirstName>Peter</FirstName>
  <LastName>Carter</LastName>
  <email>peter@carter.com</email>
  <UserPassword>myPaSSwOrd</UserPassword> ← Zapisuje fragment kodu XML w zmiennej @UserDetails.
</User>' ;

EXEC sp_adduser @UserDetails ; ← Przekazuje fragment kodu XML do procedury składowanej sp_AddUser.

```

Użycie tego skryptu do wywołania procedury składowanej powoduje zgłoszenie następującego błędu:

```

Msg 257, Level 16, State 3, Procedure sp_adduser, Line 0 [Batch Start Line 94]
Implicit conversion from data type xml to nvarchar is not allowed. Use the CONVERT
function to run this query.

```

No cóż, to dziwne! Błąd występuje w wierszu o. procedury składowanej, co wskazuje na problem z przekazywaniem zmiennej @UserDetails. Jednak nie zachodzi tu żadna konwersja z NVARCHAR na XML. Zmienna jest typu XML, a parametr procedury również jest zdefiniowany jako typ XML. Więc co się właściwie dzieje?

Przeprowadźmy eksperyment. Wywołajmy procedurę bez żadnych parametrów, używając polecenia EXEC sp_adduser. Otrzymamy następujący komunikat o błędzie:

```

Msg 201, Level 16, State 4, Procedure sp_adduser, Line 0 [Batch Start Line 104]
Procedure or function 'sp_adduser' expects parameter '@loginame', which was not supplied.

```

Teraz błąd jest inny, ale to jeszcze dziwniejsze. System zgłasza, że nie przekazujemy parametru @loginame. Jednak nasza procedura składowana nie ma parametru @loginame. To prawie tak, jakbyśmy wykonywali niewłaściwą procedurę składowaną.

Musimy dotrzeć do sedna tej sprawy. W tym celu wykonajmy kwerendę z listingu 2.11, która zwraca wyniki z widoku systemowego sys.all_objects. Obiekt ten łączy informacje o obiektach użytkownika i obiektach systemowych.

Listing 2.11. Pobieranie informacji z sys.all_objects

```

SELECT
    name
    , SCHEMA_NAME(schema_id) AS SchemaName ← SCHEMA_NAME() to funkcja systemowa, która przekształca identyfikatory schematów w nazwy schematów.
    , type_desc
    , is_ms_shipped
FROM sys.all_objects
WHERE name = 'sp_adduser' ;

```

Oto wyniki tego zapytania:

| name | SchemaName | type_desc | is_ms_shipped |
|------------|------------|----------------------|---------------|
| sp_adduser | dbo | SQL_STORED_PROCEDURE | 0 |
| sp_adduser | sys | SQL_STORED_PROCEDURE | 1 |

Wyniki pokazują, że istnieją dwie procedury o tej samej nazwie. Pierwszy wynik to nasza procedura składowana. Możemy to stwierdzić, ponieważ znajduje się ona w schemacie dbo, a jej flaga `is_ms_shipped` ma wartość `false`. Drugi wynik to systemowa procedura składowana o tej samej nazwie. Wiemy, że jest to procedura systemowa, ponieważ znajduje się w schemacie sys, a jej flaga `is_ms_shipped` ma wartość `true`.

WSKAZÓWKA Flaga `is_ms_shipped` oznacza obiekty, które zostały utworzone wewnętrznie przez SQL Server.

Systemowe procedury składowane są przechowywane w „ukrytej” bazie danych tylko do odczytu o nazwie `resource`, która to baza zawiera wszystkie obiekty systemowe. Obiekty te są widoczne we wszystkich bazach danych, co pozwala na łatwy dostęp do nich z dowolnej bazy. Wszystkie systemowe procedury składowane mają prefiks `sp_`, który powinien być zarezerwowany wyłącznie dla procedur systemowych.

Podczas wykonywania procedury składowanej z prefiksem `sp_` SQL Server najpierw szuka tej procedury w bazie danych `master`. Dopiero jeśli jej tam nie znajdzie, próbuje odnaleźć ją w lokalnej bazie danych, w której procedura została wywołana.

Można to zademonstrować za pomocą skryptu przedstawionego na listingu 2.12. Skrypt ten tworzy procedurę o nazwie `sp_listing_2_12` w bazie danych `master`, a następnie wykonuje ją z poziomu bazy danych `SalesDB`, używając jednoczęściowej nazwy.

Listing 2.12. Dostęp do procedury w bazie master z poziomu bazy użytkownika

```
USE master  ← Zmienia kontekst na master przed utworzeniem procedury.  
GO
```

```
CREATE PROCEDURE sp_listing_2_12  
AS  
BEGIN  
    SELECT 'Witam! Jestem w bazie danych master!' ;  
END  
GO
```

```
USE SalesDB ← Zmienia kontekst na SalesDB przed wywołaniem procedury.  
GO
```

```
EXEC sp_listing_2_12 ;
```

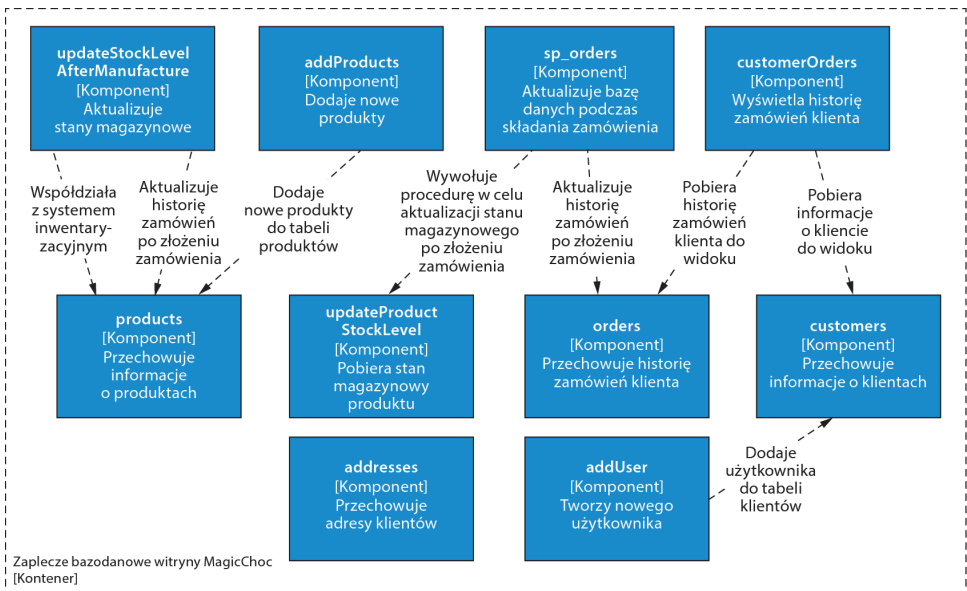
Teraz, kiedy rozumiesz już, dlaczego używanie prefiksu `sp_` nie jest dobrym pomysłem, zajmijmy się uporządkowaniem nazw naszych procedur. Możemy to zrobić za pomocą skryptu przedstawionego na listingu 2.13. Ponieważ nie musimy aktualizować definicji procedur, do wykonania tego zadania możemy użyć procedury `sp_rename`.

Listing 2.13. Porządkowanie nazw procedur

```
EXEC sp_rename 'sp_addOrder', 'addOrder' ;
EXEC sp_rename 'sp_addUser', 'addUser' ;
EXEC sp_rename 'sp_updateProductStockLevel', 'updateProductStockLevel' ;
EXEC sp_rename 'sp_products', 'addProduct' ;
EXEC sp_rename 'sp_updateStock', 'updateStockLevelAfterManufacture' ;
```

Oprócz usunięcia prefiksu nadaliśmy też obiektom bardziej znaczące nazwy.

Teraz, gdy poprawiliśmy wszystkie pomyłki związane z nazwami obiektów, warto przyrzeć się, jak wpłynęło to na nasz diagram komponentów. Zaktualizowany diagram przedstawiono na rysunku 2.4.



Rysunek 2.4. Zaktualizowany diagram komponentów

2.5. Numer 4 — nieznajdowanie czasu na standardy kodowania

Wyobraź sobie, że należysz do dziesięcioosobowego zespołu programistów pracujących nad dużą aplikacją bazodanową. Wszyscy jesteście doświadczonymi deweloperami i musicie dotrzymać napiętego terminu. Kusi Was, żeby od razu zabrać się do pisania kodu. W końcu żaden z Was nie wypadł sroce spod ogona. Każdy wie, że kod należy wcinać o cztery spacje, prawda? Chwileczkę! Czy to mają być cztery spacje, czy może jednak ma to być znak tabulacji?

Najszybszym sposobem na wywołanie sprzeczki między dwoma programistami jest rozmowa o standardach kodowania. Każdy ma swoje własne standardy, których się trzyma, i każdy uważa, że jego standardy są najlepsze. Jeśli mamy jeden wspólny standard, to nie wszyscy się z nim zgodzą, ale nawet jeśli jest on niedoskonały, to spójność, którą zapewnia, sprawia, że jest lepszy niż brak jakiegokolwiek standardu.

W zależności od rozmiaru i złożoności danego projektu standardy kodowania będą się różnić poziomem szczegółowości. Jednak zawsze będą obejmować zarówno kwestie stylistyczne, jak i techniczne.

W przeciwieństwie do standardów technicznych w przypadku standardów stylistycznych nie ma większego znaczenia, jakie one są. Ważne jest natomiast, żeby je mieć i konsekwentnie stosować w całym projekcie.

Standardy kodowania są związane z architekturą. Omawiając pomyłkę numer 1, mówiliśmy o tym, że projekty SQL Server wymagają właściwej architektury, która pozwala tworzyć optymalnie zaprojektowane i wydajne schematy baz danych. Wspomnieliśmy również, że ta architektura powinna uwzględniać konwencje nazewnnicze. Innym aspektem architektury jest zapewnienie zbioru standardów kodowania, których powinni przestrzegać programiści.

Wybory stylistyczne mogą obejmować następujące kwestie:

- Czy wcięcia w kodzie powinny być wykonywane za pomocą czterech spacji, czy znaku tabulacji?
- Czy każda instrukcja powinna kończyć się średnikiem?
- Czy przy wymienianiu nazw kolumn przecinek powinien się znajdować na końcu wiersza, czy na początku następnego?
- Czy klauzule `ON` powinny być umieszczane w tej samej linii co klauzula `JOIN`?
- Czy słowa kluczowe SQL Servera powinny być pisane wielkimi literami?
- Czy powinna być zastosowana notacja wielbłądzia, czy pascalowa?

Standardy techniczne mogą obejmować następujące zalecenia:

- Nie używaj kursorów.
- Wszędzie, gdzie to możliwe, stosuj UNION ALL zamiast UNION.
- Nie używaj * w klauzuli SELECT.
- Nie używaj NOLOCK.
- Unikaj klauzuli DISTINCT.

UWAGA Te listy mają na celu ogólne przedstawienie kwestii związanych ze standardami kodowania. W żadnym wypadku nie należy ich traktować jako wyczerpujących.

Co zatem się dzieje, jeśli nie masz standardów kodowania? Prosta odpowiedź brzmi: programiści robią wszystko po swojemu. Aby zrozumieć konsekwencje takiego podejścia, warto rozważyć osobno kwestie stylistyczne i techniczne.

Jeśli chodzi o wybory stylistyczne, problem pojawia się podczas konserwacji kodu. W poniższym skrypcie (który widzieliśmy wcześniej w tym rozdziale, ale powtarzamy go tutaj dla wygody) zobaczyliśmy skrajny przykład tego, co może się stać, gdy nie ma ustalonych standardów kodowania:

```
ALTER PROCEDURE [dbo].[proc01]
AS
BEGIN
;with t3 as (select col1, col2, col3 from tbl03) select t1.col1, t2.col2, t1.col2,
t1.col3, t3.col1, t3.col2 from dbo.tbl01 t1 inner join tbl02 t2 on t1.col1 = t2.col1 and
t1.col3 < 55 inner join t3 on t3.col1 = t1.col1 union select col1, col2, col3, NULL, NULL,
'0' from dbo.tbl04 ;
END
```

Choć w rzeczywistości rzadko spotyka się aż tak źle napisany kod, warto przypomnieć sobie z omówienia pomyłki numer 2 informacje dotyczące tego, jak nasze mózgi przyzwyczajają się do pewnych wzorców podczas analizowania list obiektów. Ta sama zasada ma zastosowanie, gdy nasz mózg próbuje zrozumieć kod źródłowy.

Przyjrzyj się kwerendom przedstawionym na listingu 2.14. Wyobraź sobie, że właśnie przeanalizowałeś 50 kwerend napisanych w stylu kwerendy 1. Następnie natrafiasz na kwerendę napisaną w stylu kwerendy 2. W zależności od jej stopnia skomplikowania może to spowolnić Twoją pracę — może o kilka sekund, a może nawet o kilka minut. Obie kwerendy zwracają te same wyniki. Różnią się jedynie sposobem zapisu, który wynika z zastosowania odmiennych standardów.

Listing 2.14. Zmienianie stylu kodowania

```
SELECT ←———— Kwerenda 1
    c.FirstName
    , c.LastName
    , o.LineItems
FROM dbo.customers c
LEFT JOIN dbo.orders o
```

```

ON c.CustomerID = o.CustomerID
WHERE c.CustomerID >= 2 AND c.CustomerID <= 3
AND c.CustomerID <> 2 ;

SELECT ←————— Kwerenda 2
    cust.FirstName,
    cust.LastName,
    ord.LineItems
FROM dbo.customers cust
LEFT JOIN (
    SELECT CustomerID,
    LineItems
    FROM dbo.Orders
) ord ON cust.CustomerID = ord.CustomerID
WHERE cust.CustomerID BETWEEN 2 AND 3 AND cust.CustomerID != 2 ;

```

Brak konsekwentnego stylu kodowania ostatecznie spowalnia rozwiązywanie błędów i rozwój nowych funkcji, w miarę jak aplikacja bazodanowa przechodzi przez kolejne etapy swojego cyklu życia.

Jeśli przyjrzeć się technicznym standardom kodowania, to okaże się, że są one znacznie bardziej restrykcyjne. Nieprzestrzeganie tych standardów przez programistów może prowadzić do znacznego spadku wydajności aplikacji, a nawet do nieoczekiwanych rezultatów. Wszystkie wymienione wcześniej standardy techniczne dotyczą pomyłek często popełnianych przez użytkowników SQL Servera i zostaną omówione szczegółowo w rozdziale 5.

2.6. Numer 5 — używanie porządkowej pozycji kolumny

SQL Server umożliwia używanie numerów porządkowych kolumn w klauzuli ORDER BY. Oznacza to, że w klauzuli SELECT możesz sortować wyniki według pozycji kolumny zamiast według jej nazwy. Rozważmy na przykład kwerendę przedstawioną na listingu 2.15.

Listing 2.15. Sortowanie według porządkowej pozycji kolumny

```

SELECT *
FROM SYS.databases
ORDER BY 54 ; ←————— Sortuje według 54. kolumny na liście SELECT.

```

Według której kolumny została posortowana nasza kwerenda? Według kolumny log_reuse_wait_desc, ale jedynym sposobem, aby to ustalić, byłoby policzenie kolumn w zbiorze wyników aż do 54. kolumny albo uruchomienie z listingu 2.16 kwerendy, która pobiera nazwę kolumny z metadanych przechowywanych w widokach katalogowych.

Listing 2.16. Pobieranie nazwy kolumny z metadanych

```
SELECT
    c.name
FROM SYS.all_columns c
INNER JOIN SYS.all_objects o
    ON o.object_id = c.object_id
WHERE o.name = 'databases'
    AND c.column_id = 54 ;
```

WSKAZÓWKA Gdyby obiekt, którego dotyczy kwerenda, był obiektem użytkownika, a nie obiektem systemowym, kwerenda działałaby również wtedy, gdybyśmy złączyli `sys.objects` z `sys.columns`.

Kolejnym dobrym powodem, aby unikać używania porządkowych pozycji kolumn, jest ich niestabilność w przypadku modyfikacji bazowych tabel. Na przykład gdyby druga kolumna została usunięta z tabeli, musielibyśmy zaktualizować wszystkie kwerendy odwołujące się do kolumn o pozycjach od 3 do n , ponieważ ich numeracja uległaby zmianie. Dlatego zawsze lepiej jest odwoływać się do kolumn według nazw, co zapewnia większą elastyczność i odporność na zmiany w strukturze bazy danych.

Podsumowanie

- Zawsze używaj nazw obiektów, które są znaczące i zrozumiałe, żeby Twój kod sam się dokumentował.
- Zawsze pamiętaj o architekturze bazy danych, czyli o jej projekcie, nawet w przypadku projektów zwinnych.
- Unikaj stosowania prefiksów w nazwach obiektów bazodanowych, ponieważ mogą one w rzeczywistości utrudnić ich odnajdywanie.
- Unikaj zwłaszcza używania prefiksu `sp_` w nazwach procedur składowanych. Taki prefiks wskazuje, że dana procedura jest systemowa, a nie zdefiniowana przez użytkownika.
- Zawsze znajdź czas, aby upewnić się, że Twoja aplikacja bazodanowa ma ustalone standardy kodowania. Powinny one stanowić część Twoich zasad architektonicznych i uwzględniać wybory zarówno stylistyczne, jak i techniczne.
- Unikaj sortowania po numerach porządkowych kolumn — chyba że lubisz rozwiązywać problemy!

3

Typy danych

W tym rozdziale:

- Dlaczego typy danych są ważne
- Konsekwencje użycia niewłaściwego typu danych
- Powody stosowania zaawansowanych typów danych
- Korzyści wynikające z pracy z danymi w formatach XML i JSON

W tym rozdziale przyjrzymy się typom danych i dowiesz się, dlaczego tak ważne jest odpowiednie dobranie typów danych dla kolumn w naszych tabelach. Zaczniemy od omówienia prostych typów danych, które są stosowane powszechnie, ale często nieprawidłowo. Zobaczmy, jaki to może mieć wpływ na koszty i wydajność systemu.

Następnie zbadamy zaawansowane typy danych SQL Servera, które są często niedoceniane. Dowiesz się, w jakich okolicznościach mogą się przydać, i poznasz skutki ich unikania. Warto jednak zauważyć, że choć ten rozdział skupia się na typach HIERARCHYID, XML i JSON, istnieją również inne specjalistyczne typy danych, takie jak GEOGRAPHY i GEOMETRY do danych geoprzestrzennych. Gorąco zachęcam do zapoznania się ze wszystkimi zaawansowanymi typami danych dostępnymi w SQL Serverze.

Firma MagicChoc uznała, że potrzebuje nowej aplikacji kadrowej. Skrypt przedstawiony na listingu 3.1 tworzy bazę danych HumanResources, a następnie pierwszą tabelę: `dbo.employees`. Jak widać, w tabeli tej dla każdej kolumny użyto typu danych `NVARCHAR(MAX)`. Czy to nie wygląda dziwnie? Dlaczego? Przecież

wszystkie dane, które chcemy przechowywać, można wstawić do tego pojemnego typu. Czy to naprawdę ma znaczenie? Przyjrzymy się temu przykładowi w trakcie tego rozdziału.

Listing 3.1. Tworzenie tabeli pracowników

```
CREATE DATABASE HumanResources ;
GO
```

```
USE HumanResources ;
GO
```

```
CREATE TABLE dbo.Employees (
    EmployeeID          NVARCHAR(MAX)    NOT NULL,
    FirstName           NVARCHAR(MAX)    NOT NULL,
    LastName            NVARCHAR(MAX)    NOT NULL,
    DateOfBirth         NVARCHAR(MAX)    NOT NULL,
    EmployeeStartDate   NVARCHAR(MAX)    NOT NULL,
    ManagerID           NVARCHAR(MAX)    NULL,
    Salary              NVARCHAR(MAX)    NOT NULL,
    Department          NVARCHAR(MAX)    NOT NULL,
    DepartmentCode      NVARCHAR(MAX)    NOT NULL,
    Role                NVARCHAR(MAX)    NOT NULL,
    WeeklyContractedHours NVARCHAR(MAX) NOT NULL,
    StaffOrContract     NVARCHAR(MAX)    NOT NULL,
    ContractEndDate     NVARCHAR(MAX)    NULL
);
```

Kolumna ma przechowywać 0 w przypadku pracowników etatowych i 1 w przypadku pracowników kontraktowych.

Dlaczego wybór odpowiedniego typu danych jest tak istotny? Większość osób, które przez pewien czas pracowały z SQL Serverem, rozumie znaczenie ograniczeń. Ograniczenia te występują w różnych formach, takich jak klucze obce, ograniczenia sprawdzające czy ograniczenia NULL.

Ograniczenia są kluczowe dla zapewnienia jakości danych w bazie. Na przykład klucz obcy sprawia, że wartość musi istnieć w innej tabeli, żeby można było ją wstawić lub zaktualizować. Ograniczenie NOT NULL wymusza, aby kolumna zawsze miała wartość i nie była pusta. Z kolei ograniczenie CHECK zapewnia, że wartości w kolumnie spełniają określone kryteria — na przykład że data rozpoczęcia pracy przez pracownika nie jest wcześniejsza niż jego 16. urodziny.

Wielu osobom umyka jednak fakt, że typ danych również stanowi pewne ograniczenie — ograniczenie, które dotyczy każdej kolumny w każdej tabeli w całej bazie danych. Jest to fundament jakości i funkcjonalności danych. Co więcej, może to pomóc w pisaniu samodokumentującego się kodu. Koncepcja takiego kodu została szerzej omówiona w rozdziale 2.

Rozważając przykład naszej tabeli `Employees`, możemy zauważyć kilka istotnych problemów. Po pierwsze, nie jesteśmy w stanie utworzyć ograniczenia klucza głównego, ponieważ klucze główne nie są obsługiwane na kolumnach typu `NVARCHAR(MAX)`. Po drugie, operacje na kolumnach zawierających daty będą kłopotliwe, ponieważ będą wymagały konwersji. Daty mogą być również wprowadzane w sprzecznych formatach, na przykład `13.01.2023` i `01.13.2023`. Po trzecie, możemy wstawić dowolne dane do dowolnej kolumny. Moglibyśmy umieścić

datę w kolumnie Salary, tekst — w kolumnach przeznaczonych na wartości liczbowe lub daty — w polach tekstowych, takich jak FirstName, LastName lub Department. Wreszcie ponieważ możemy przechowywać do 2 GB danych w każdej kolumnie każdego wiersza, ta tabela mogłaby teoretycznie bardzo szybko urosnąć do ogromnych rozmiarów.

Powinniśmy naprawić to od razu, a skrypt przedstawiony na listingu 3.2 robi to poprzez usunięcie i ponowne utworzenie tabeli. Zmienia on kolumny liczbowe na typ INT, kolumny z datami — na typ DATE oraz redukuje długość kolumn tekstowych do odpowiednich wartości. Dodatkowo tabela zostanie uzupełniona o klucz główny w kolumnie EmployeeID.

Listing 3.2. Aktualizowanie typów kolumn w tabeli pracowników

```
DROP TABLE dbo.Employees ;
GO
```

```
CREATE TABLE dbo.Employees (
    EmployeeID          INT          NOT NULL PRIMARY KEY,
    FirstName           NVARCHAR(32) NOT NULL,
    LastName            NVARCHAR(32) NOT NULL,
    DateOfBirth         DATE         NOT NULL,
    EmployeeStartDate   DATE         NOT NULL,
    ManagerID          INT          NULL,
    Salary              MONEY       NOT NULL,
    Department          NVARCHAR(64) NOT NULL,
    DepartmentCode      NVARCHAR(4) NOT NULL,
    Role                NVARCHAR(64) NOT NULL,
    WeeklyContractedHours INT       NOT NULL,
    StaffOrContract     INT         NOT NULL,
    ContractEndDate     DATE        NULL
);
```

UWAGA Chociaż nowo wybrane typy danych są znacznie bardziej użyteczne niż ogólny NVARCHAR(MAX), wciąż nie są idealne. Problemowi temu przyjrzymy się bliżej w kolejnych podrozdziałach.

3.1. Numer 6 — przechowywanie liczb całkowitych zawsze jako typu INT

Wyobraź sobie, że mamy dużą hurtownię danych. Jedna z tabel faktów zawiera miliard wierszy i jest złączona z pięcioma tabelami wymiarów, z których każda liczy 30 000 wierszy. Wydajność jest niska, a pamięć jest zawsze zapelniona z powodu ilości danych w buforach. Podczas wykonywania kwerend duża ilość danych jest przenoszona do TempDB. Zoptymalizowaliśmy już kwerendy. Przeanalizowaliśmy również strategię indeksowania i upewniliśmy się, że zarówno

indeksy, jak i statystyki są dobrze konserwowane. Wygląda na to, że jedynym rozwiązaniem jest rozbudowa sprzętu, ale podejrzewamy, bazując na trendzie z ostatnich dwóch lat, że dodanie większej ilości pamięci RAM tylko odsuwa problem w czasie. Co powinniśmy zrobić? Dobrym punktem wyjścia byłoby przejście naszych liczbowych typów danych, szczególnie tych używanych w relacjach kluczy głównych i obcych.

Typ INT jest najczęściej używanym, ale też najbardziej nadużywanym typem danych w SQL Serverze. W rzeczywistości mamy cztery typy danych zaprojektowane specjalnie do przechowywania liczb całkowitych. Typy te zostały szczegółowo opisane w tabeli 3.1.

Tabela 3.1. Rodzina typów całkowitych

| Typ danych | Zakres | Rozmiar (bajty) |
|------------|---|-----------------|
| TINYINT | 0 do 255 | 1 |
| SMALLINT | -32 768 do 32 767 | 2 |
| INT | -2 147 483 648 do 2 147 483 647 | 4 |
| BIGINT | -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807 | 8 |

Możemy również przeanalizować to w kodzie, wykonując kwerendę z listingu 3.3. Kwerenda ta wykorzystuje funkcję CAST() do konwersji wartości 1 na każdy z całkowitoliczbowych typów danych. Przekształcona wartość jest następnie przekazywana do funkcji DATALENGTH(), która oblicza rozmiar wartości wejściowej w bajtach.

Listing 3.3. Sprawdzanie rozmiaru typów danych

```
SELECT
    DATALENGTH(CAST(1 AS TINYINT)) AS TinyIntSize
    , DATALENGTH(CAST(1 AS SMALLINT)) AS SmallIntSize
    , DATALENGTH(CAST(1 AS INT)) AS IntSize
    , DATALENGTH(CAST(1 AS BIGINT)) AS BigIntSize ;
```

Oto wyniki tej kwerendy:

| | | | |
|-------------|--------------|---------|------------|
| TinyIntSize | SmallIntSize | IntSize | BigIntSize |
| 1 | 2 | 4 | 8 |

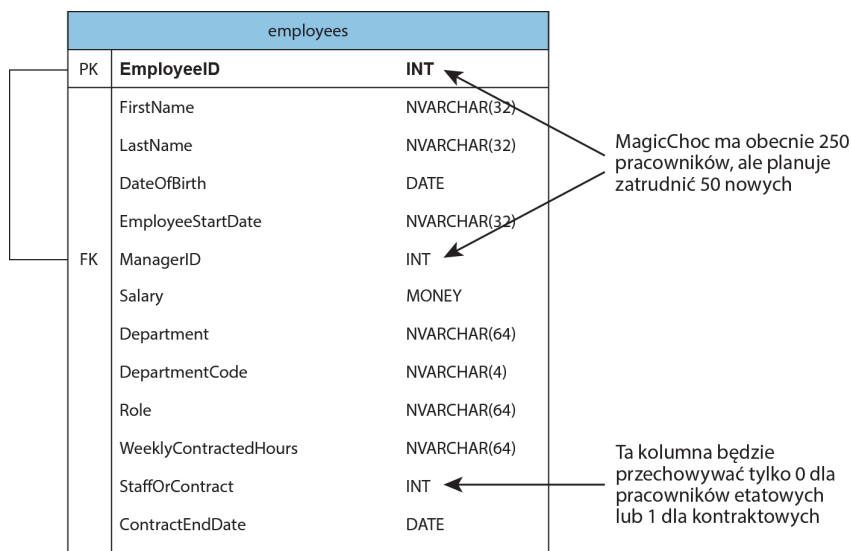
Wyobraźmy sobie, że pięć tabel wymiarów używa typu INT dla kolumny klucza głównego. Każda z tych tabel zawiera 30 000 wierszy, a typ danych SMALLINT może przechowywać ponad 32 000 różnych wartości dodatnich. Oznacza to, że jeśli nie spodziewamy się znaczącego wzrostu liczby wymiarów, możemy zaoszczędzić 2 bajty na każdym wierszu.

WSKAZÓWKA W rzeczywistości zakres SMALLINT obejmuje ponad 64 000 możliwych wartości, ale aby wykorzystać więcej niż 32 000 z nich, musielibyśmy rozpocząć sekwencję numerowania od -32 000. Takie podejście może jednak kłócić się z zasadą najmniejszego zaskoczenia.

W tym momencie zastanawiasz się może: „Po co zawracać sobie głowę oszczędzaniem 2 bajtów?”. Odpowiedź na to pytanie wymaga prostych obliczeń. W każdej z pięciu tabel wymiarów mamy 30 000 wierszy. Przechodząc na typ SMALLINT, zaoszczędzilibyśmy tylko 58 KB na tabelę. Ale nasza tabela faktów ma miliard wierszy. To oznacza, że oszczędzilibyśmy 1,86 GB na każdym kluczu. Pomnóżmy to przez pięć tabel wymiarów, a okaże się, że w przypadku każdej kwerendy tabeli faktów, która przetwarza wszystkie wiersze i wszystkie pięć kluczy, oszczędzamy 9,3 GB. Przeskalujmy to na osiem tabel faktów w hurtowni danych. Powinniśmy również wziąć pod uwagę rozmiar indeksów, które są tworzone na kolumnach kluczy obcych. Teraz wyobraźmy sobie równoległe sesje wykonujące różne kwerendy. Nagle okazuje się, że nasz wybór typu danych ma bezpośredni i namacalny wpływ na zużycie pamięci.

Jak to wszystko ma się do naszej tabeli Employees? Przedstawmy naszą tabelę w formie *diagramu związków encji* (ang. *Entity Relationship Diagram*, ERD). ERD to diagram, który prezentuje encje (tabele) w bazie danych. Pokazuje on relacje między tymi encjami (ograniczenia kluczy głównych/obcych) oraz szczegółowo opisuje atrybuty (kolumny) każdej encji. Opcjonalnie może również zawierać informacje o typie danych każdej kolumny.

Przy tworzeniu diagramów C4 dla aplikacji bazodanowych często wykorzystuje się diagram związków encji (ERD) jako schemat kodu, który dokumentuje strukturę tabel. ERD przedstawiony na rysunku 3.1 pokazuje obecnie tylko encję employees, ale rozbudujemy go w rozdziale 4. Diagram ten zawiera aktualnie zdefiniowane typy danych, ale został dodatkowo opatrzony adnotacjami wskazującymi wartości, jakich spodziewamy się w poszczególnych kolumnach.



Rysunek 3.1. Diagram ERD tabeli Employees

Kolumny EmployeeID i ManagerID będą wymagać do 300 unikatowych wartości. W związku z tym typ danych TINYINT będzie zbyt ograniczony. Możemy jednak użyć typu SMALLINT. Pozwoli to zaoszczędzić 2 bajty na wiersz w porównaniu z obecnie stosowanym typem INT.

Kolumna StaffOrContract to ciekawy przypadek. Będzie przechowywać wartości całkowite, ale tylko w zakresie od 0 do 1. Wprowadza to dodatkowy typ danych, o którym jeszcze nie mówiliśmy: BIT. Typ BIT jest technicznie typem całkowitoliczbowym, ale może przechowywać jedynie wartości 0, 1 oraz NULL. Jest on przeznaczony do przechowywania wartości logicznych, takich jak flagi, i posiada przydatne dodatkowe funkcje.

Jeśli użytkownik wprowadzi TRUE lub FALSE do kolumny typu BIT, SQL Server automatycznie przekształci te wartości, odpowiednio, w 1 i 0. Co więcej, jeśli użytkownik wprowadzi jakąkolwiek wartość liczbową inną niż 0 lub 1, SQL Server automatycznie zamieni ją na 1. Na przykład jeśli wykonamy kwerendę SELECT CAST(86.2 AS BIT), w wyniku otrzymamy 1.

Ponadto, jeśli tabela zawiera wiele kolumn typu BIT, SQL Server optymalizuje ich przechowywanie. Pierwsze osiem kolumn BIT zajmuje tylko 1 bajt przestrzeni. Kolejne osiem kolumn wykorzystuje dodatkowy bajt i tak dalej. Kolumna StaffOrContract jest idealnym kandydatem do zastosowania typu danych BIT, co pozwoli zaoszczędzić 3 bajty na każdy wiersz.

Ponieważ nasza tabela pracowników jest jeszcze pusta, usuńmy ją i utwórzmy na nowo, używając preferowanych typów danych całkowitych. Możemy to zrobić za pomocą skryptu przedstawionego na listingu 3.4.

Listing 3.4. Zmianianie typów danych kolumn całkowitoliczbowych

```
DROP TABLE dbo.Employees
GO
```

```
CREATE TABLE dbo.Employees (
    EmployeeID          SMALLINT          NOT NULL PRIMARY KEY,
    FirstName            NVARCHAR(32)      NOT NULL,
    LastName             NVARCHAR(32)      NOT NULL,
    DateOfBirth          DATE              NOT NULL,
    EmployeeStartDate    DATE              NOT NULL,
    ManagerID            SMALLINT          NULL,
    Salary               MONEY            NOT NULL,
    Department           NVARCHAR(64)     NOT NULL,
    DepartmentCode       NVARCHAR(4)      NOT NULL,
    Role                 NVARCHAR(64)     NOT NULL,
    WeeklyContractedHours INT             NOT NULL,
    StaffOrContract      BIT              NOT NULL,
    ContractEndDate      DATE              NULL
);
```

Zawsze powinniśmy zwracać uwagę na ilość miejsca zajmowanego przez nasze typy danych. Jest to szczególnie istotne w przypadku bardzo dużych tabel lub gdy dana kolumna ma być wykorzystywana w indeksie. Aby ograniczyć zużycie pamięci, powinniśmy używać typów SMALLINT i TINYINT zamiast INT wszędzie tam, gdzie wiadomo, że nie dojdzie do ich przepełnienia.

3.2. Numer 7 — używanie wyłącznie łańcuchów o zmiennej długości

Wyobraźmy sobie, że mamy tabelę przechowującą adresy w Stanach Zjednoczonych. Chcemy zadbać o efektywne wykorzystanie przestrzeni dyskowej. W tym celu używamy kolumn o zmiennej długości dla wszystkich pól, w tym dla poszczególnych wierszy adresu oraz kodu pocztowego. Wiemy, że najdłuższe nazwy miast w Stanach Zjednoczonych to Mooselookmeguntic w Maine i Kleinfeltersville w Pensylwanii, liczące po 17 znaków. Dlatego ustawiamy kolumnę `CityName` jako `VARCHAR(17)`. Wiemy również, że najdłuższa nazwa stanu w USA to „Rhode Island and Providence Plantations”, więc kolumnę przechowującą nazwę stanu ustawiamy na `VARCHAR(48)`. Wiemy, że kod pocztowy będzie miał dokładnie 10 znaków. Skoro znamy jego długość, to czy powinniśmy użyć `VARCHAR(10)`, czy `CHAR(10)`? Czy to w ogóle ma znaczenie? W obu przypadkach dane zajmą 10 znaków, więc powinny zająć 10 bajtów, prawda? A jeśli tak, to po co w ogóle są typy o stałej długości? Okazuje się jednak, że to założenie nie jest poprawne. Aby zrozumieć dlaczego, musisz dowiedzieć się co nieco o sposobie przechowywania danych przez SQL Server.

UWAGA Ciągi znaków o stałej długości, takie jak `CHAR` i `NCHAR`, uzupełniają niewykorzystane miejsce spacjami, jeśli nie zostało ono wypełnione danymi. Jeśli na przykład mamy kolumnę typu `CHAR(8)` i wstawimy do niej wartość `Hello!`, to zajmie ona 8 bajtów pamięci, ponieważ dwa pozostałe znaki zostaną wypełnione spacjami.

SQL Server przechowuje dane w seriach 8-kilobajtowych stron, przy czym każda seria ośmiu stron tworzy 64-kilobajtowy obszar, który zwykle stanowi najmniejszą ilość odczytywanych danych. Każda strona danych ma 96-bajtowy nagłówek, który zawiera informacje dotyczące całej strony, takie jak jej unikatowy identyfikator oraz identyfikator obiektu tabeli (lub indeksu), do którego należy.

Dane tworzące wiersze są następnie przechowywane w slotach na stronie. Sloty przechowują nie tylko dane, ale również niewielką ilość *metadanych*, które sprawiają, że dane stają się użyteczne. Metadane te obejmują informacje o typie rekordu przechowywanego w danym slotcie. Na przykład o tym, czy jest to rekord danych, czy rekord indeksu? Czy zawiera dane-duchy (informacje, które zostały usunięte logicznie, ale jeszcze нефизycznie)?

Inne metadane obejmują długość danych o stałej długości (dotyczy to nie tylko danych znakowych o stałej długości, ale także danych takich jak liczby całkowite), mapę bitową wartości `NULL`, która wskazuje, czy kolumny o zmiennej długości zawierają wartości `NULL`, oraz znacznik wersji. Ten ostatni jest wykorzystywany przez operacje takie jak przebudowa indeksów lub transakcje z optymistycznymi poziomami izolacji. O poziomach izolacji porozmawiamy w rozdziale 10.

Kluczowym elementem metadanych, którym się tutaj zajmiemy, jest *tablica przesunięć kolumn*. Pozwala ona ustalić, gdzie w wierszu zaczynają się poszczególne kolumny o zmiennej długości. Ponieważ dane o zmiennej długości mogą mieć dowolny rozmiar, ta tablica przesunięć jest jedyną możliwością określenia, gdzie kończy się jeden fragment danych, a zaczyna następny.

Każda kolumna o zmiennej długości wymaga 2-bajтового przesunięcia w tabeli, co oznacza, że każda taka kolumna zajmuje o 2 bajty więcej miejsca niż jej odpowiednik o stałej długości. Dotyczy to nawet kolumn przechowujących wartość NULL. W rezultacie, jeśli użyjemy CHAR(10) dla kodu pocztowego, zajmie on 10 bajtów, ale jeśli zastosujemy VARCHAR(10), zajmie 12 bajtów, mimo że rzeczywista długość danych wynosi 10 bajtów.

CHAR i VARCHAR a NCHAR i NVARCHAR

Typy NCHAR i NVARCHAR mogą przechowywać pełne dane Unicode, podczas gdy CHAR i VARCHAR obsługują tylko 8-bitowe strony kodowe. Od wersji SQL Server 2019 możliwe jest korzystanie z sortowań obsługujących UTF-8, dzięki czemu typy CHAR i VARCHAR mogą przechowywać pełen zakres znaków UTF-8. Jednak do pełnej obsługi UTF-16 nadal wymagane są typy danych NCHAR i NVARCHAR.

Typy danych CHAR i VARCHAR używają 1 bajta na znak, natomiast NCHAR i NVARCHAR wykorzystują 2 bajty na znak. Ta dodatkowa przestrzeń jest wymagana dla pełnej 16-bitowej strony kodowej UTF-16. W rezultacie typy danych UTF-16 nie tylko zajmują więcej miejsca, ale także ograniczają liczbę znaków, które można przechowywać na stronie. SQL Server narzuca maksymalną długość 4000 znaków dla danych UTF-16 o stałej długości, w przeciwieństwie do maksymalnej długości 8000 znaków dla danych o stałej długości w 8-bitowej stronie kodowej.

Typy danych VARCHAR(MAX) i NVARCHAR(MAX) umożliwiają przechowywanie danych o zmiennej długości do 2 GB. Jednak pola, które nie mieszczą się na stronie danych, są zapisywane w innym rodzaju jednostki alokacji i nazywane danymi przepełniającymi wiersz.

Ograniczenie rozmiaru dotyczy również danych rozłożonych na wiele kolumn. Niezależnie od sposobu rozmieszczenia kolumn maksymalna długość danych na stronie wynosi 8060 bajtów. Oznacza to, że jeśli mamy tabelę składającą się z kolumny CHAR(5000) i kolumny VARCHAR(5000), a do kolumny o zmiennej długości wstawimy dane o długości 2000 znaków (4000 bajtów), to dane o zmiennej długości zostaną dynamicznie przeniesione na inną stronę i zapisane w jednostce alokacji dla przepełnionego wiersza.

Niektórzy twierdzą, że ze względu na coraz niższe koszty pamięci i przestrzeni dyskowej powinniśmy po prostu używać typów danych NCHAR i NVARCHAR, aby uniknąć potencjalnych problemów z niekompatybilnością stron kodowych między kolumnami. Moje stanowisko w tej kwestii, które zostało wzmocnione przez dodanie sortowań UTF-8, jest takie samo jak w przypadku innych typów danych. Powinniśmy używać najbardziej restrykcyjnego typu danych, który nie spowoduje przepełnienia.

Innymi słowy, jeśli wiemy, że ze względu na charakter naszych danych nigdy nie napotkamy problemów z niezgodnością stron kodowych, powinniśmy używać typów danych dla 8-bitowych stron kodowych. Zapewni to najlepszą wydajność i efektywność wykorzystania zasobów w optymalnej cenie. Jeśli jednak nie możemy tego zagwarantować — na przykład gdy musimy pobierać dane z internetu lub innych niezauważanych źródeł albo wiemy, że będziemy musieli łączyć dane z różnych zestawów znaków — powinniśmy korzystać z typów danych UTF-16.

W kontekście naszej tabeli `Employees` większość kolumn musi mieć zmienną długość. Jednak kolumna `DepartmentCode` zawsze będzie przechowywać dwuznakową wartość, na przykład HR dla kadr, SA dla sprzedaży czy MA dla produkcji. Dlatego powinniśmy zmienić definicję kolumny `DepartmentCode` na `NCHAR(2)`. Pozwoli to zaoszczędzić 2 bajty informacji na każdy wiersz. Używamy Unicode, ponieważ w podrozdziale 3.4 będziemy importować dane od zewnętrznego partnera biznesowego.

Instrukcja w poniższym przykładzie (listing 3.5) odpowiednio aktualizuje tabelę `Employees` z użyciem polecenia `ALTER TABLE` z klauzulą `ALTER COLUMN`.

Listing 3.5. Zmiana kolumny `DepartmentCode` na łańcuch o stałej długości

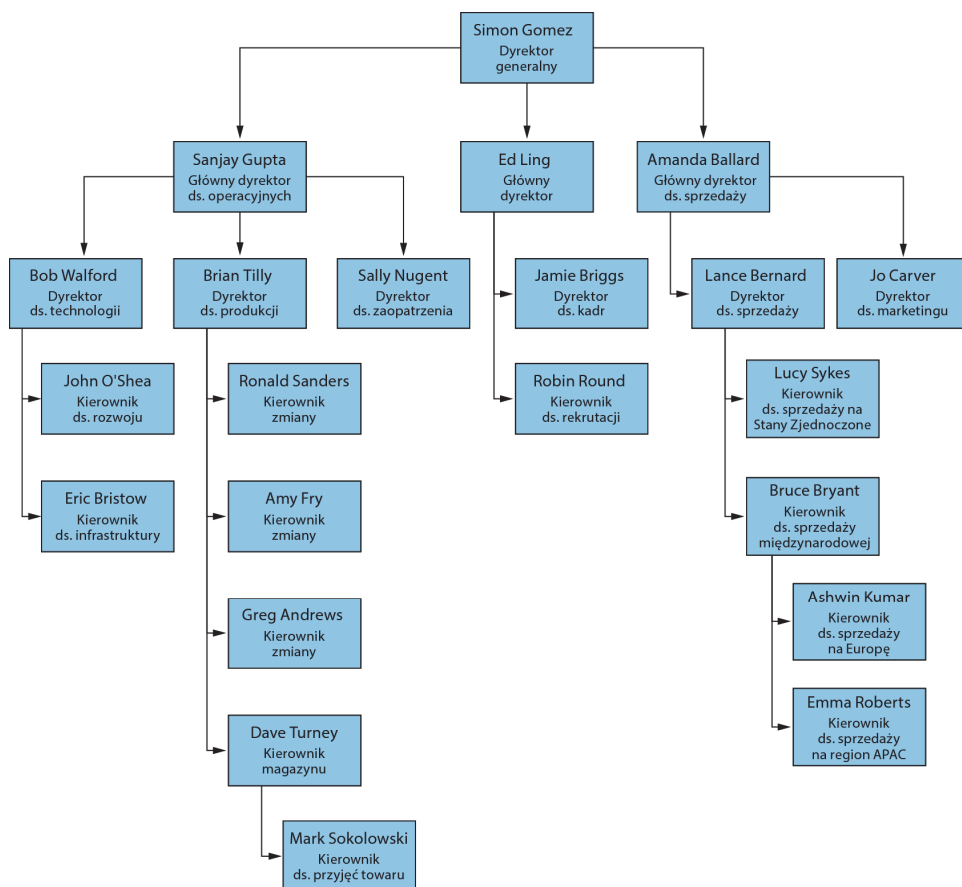
```
ALTER TABLE dbo.Employees  
    ALTER COLUMN DepartmentCode NCHAR(2) ;
```

Stosowanie łańcuchów znaków o zmiennej długości jest właściwym rozwiązaniem, gdy wartości w naszej kolumnie mogą mieć różne długości. Pozwala to uniknąć niepotrzebnego uzupełniania krótszych wartości spacjami i oszczędza miejsce. Jeśli jednak spodziewamy się, że wartości w kolumnie zawsze będą miały tę samą długość, powinniśmy użyć ciągu znaków o stałej długości. W przeciwnym razie dodatkowe 2 bajty na wiersz zostaną wykorzystane na tablicę przesunięć kolumn określającą początek każdej wartości o zmiennej długości.

3.3. Numer 8 — pisanie własnego kodu hierarchii

Przyglądając się naszej tabeli `Employees`, możemy zauważyć, że kolumny `EmployeeID` i `ManagerID` zostały zaprojektowane w celu modelowania hierarchii pracowników. Weźmy pod uwagę schemat organizacyjny przedstawiony na rysunku 3.2, który obrazuje strukturę organizacyjną kadry kierowniczej wyższego szczebla w firmie MagicChoc.

Aby zademonstrować modelowanie tej hierarchii z użyciem tradycyjnego podejścia, które nadal stosuje zaskakująco wielu programistów, zachęcam do uruchomienia skryptu z listingu 3.6, który wstawi rekordy pracowników do tabeli `Employees`. Zwróć uwagę, że kolumna `ManagerID` zawiera `EmployeeID` osoby, której dany pracownik podlega.

**Rysunek 3.2.** Schemat organizacyjny zarządu firmy MagicChoc**Listing 3.6.** Wstawianie rekordów pracowników do tabeli Employees

```

INSERT INTO dbo.Employees (
    EmployeeID
    , FirstName
    , LastName
    , DateOfBirth
    , EmployeeStartDate
    , ManagerID
    , Salary
    , Department
    , DepartmentCode
    , Role
    , WeeklyContractedHours
    , StaffOrContract
    , ContractEndDate
)
VALUES
    (1, 'Simon', 'Gomez', '19691001', '20180101', NULL, 980000, 'C-Suite', 'CS', 'CEO',
    ↳40, 1, NULL),

```

```

(2, 'Sanjay', 'Gupta', '19761001', '20180101', 1, 640000, 'C-Suite', 'CS', 'COO', 40,
↳1, NULL),
(3, 'Ed', 'Ling', '19690403', '20200801', 1, 320000, 'C-Suite', 'CS', 'CPO', 40, 1,
↳NULL),
(4, 'Amanda', 'Ballard', '19830401', '20200301', 1, 350000, 'C-Suite', 'CS', 'Sales
↳Director', 40, 1, NULL),

(5, 'Bob', 'Walford', '19780908', '20191201', 2, 96000, 'Technology', 'TE', 'Head Of
↳Technology', 40, 1, NULL),
(6, 'Brian', 'Tilly', '19710102', '20181001', 2, 89000, 'Manufacturing', 'MA', 'Head
↳Of Manufacturing', 40, 1, NULL),
(7, 'Sally', 'Nugent', '19790302', '20220601', 2, 80000, 'Procurement', 'PR', 'Head Of
↳Procurement', 40, 1, NULL),
(8, 'Jamie', 'Briggs', '19900102', '20190601', 3, 65000, 'Human Resources', 'HR', 'HR
↳Manager', 40, 1, NULL),
(9, 'Lance', 'Bernard', '19910707', '20210601', 4, 98000, 'Sales & Marketing', 'SA',
↳'Head Of Sales', 40, 1, NULL),
(10, 'Jo', 'Carver', '19900810', '20191201', 4, 70000, 'Sales & Marketing', 'SA',
↳'Head Of Marketing', 40, 1, NULL),

(11, 'John', 'O'Shea', '19700609', '20180601', 5, 70000, 'Technology', 'TE',
↳'Development Manager', 40, 1, NULL),
(12, 'Eric', 'Bristow', '20000109', '20221001', 5, 72000, 'Technology', 'TE',
↳'Infrastructure Manager', 40, 1, NULL),
(13, 'Ronald', 'Sanders', '19601209', '20190101', 6, 45000, 'Manufacturing', 'MA',
↳'Shift Manager', 45, 1, NULL),
(14, 'Amy', 'Fry', '19921101', '20190101', 6, 45000, 'Manufacturing', 'MA', 'Shift
↳Manager', 45, 1, NULL),
(15, 'Greg', 'Andrews', '19871212', '20190101', 6, 45000, 'Manufacturing', 'MA',
↳'Shift Manager', 45, 1, NULL),
(16, 'Dave', 'Turney', '19760609', '20190101', 6, 52000, 'Manufacturing', 'MA',
↳'Warehouse Manager', 48, 1, NULL),
(17, 'Mark', 'Sokolowski', '19960209', '20190901', 16, 42000, 'Manufacturing', 'MA',
↳'Goods Inn Manager', 40, 1, NULL),

(18, 'Robin', 'Round', '19940409', '20190601', 3, 60000, 'Human Resources', 'HR',
↳'Recruitment Manager', 40, 1, NULL),
(19, 'Lucy', 'Sykes', '19890201', '20200201', 9, 65000, 'Sales', 'SA', 'US Sales
↳Manager', 40, 1, NULL),
(20, 'Bruce', 'Bryant', '19860304', '20200301', 9, 70000, 'Sales', 'SA',
↳'International Sales Manager', 40, 1, NULL),
(21, 'Ashwin', 'Kumar', '20010212', '20210601', 20, 55000, 'Sales', 'SA', 'Euro Sales
↳Manager', 40, 1, NULL),
(22, 'Emma', 'Roberts', '20000208', '20210601', 20, 55000, 'Sales', 'SA', 'APAC Sales
↳Manager', 40, 1, NULL) ;

```

Teraz wyobraźmy sobie, że poproszono nas o napisanie raportu zawierającego listę wszystkich bezpośrednich podwładnych Sanjaya Gupty. Można to osiągnąć za pomocą następującej prostej kwerendy:

```

SELECT
    FirstName
    , LastName
FROM dbo.Employees
WHERE ManagerID = 2 ;

```

A gdybyśmy musieli napisać kwerendę zwracającą wszystkich bezpośrednich i pośrednich podwładnych Sanjaya Gupty? To zadanie staje się nagle bardziej skomplikowane. Istnieje kilka sposobów utworzenia takiego raportu, w tym niesławne kursory (o których będziemy mówić w rozdziale 5.), ale najlepszą praktyką byłoby użycie rekurencyjnego wspólnego wyrażenia tabelarycznego (ang. *common table expression*, CTE). CTE to tymczasowy zbiór wyników, do którego można się odwoływać wielokrotnie w ramach kwerendy. Może on również odwoływać się do samego siebie, co umożliwia *rekurencję*.

UWAGA Rekurencja może spowodować nieskończoną pętlę w przypadku błędu w kwerendzie. Aby temu zapobiec, istnieje globalne ustawienie kontrolujące maksymalny poziom rekurencji. Domyślnie jest ono ustawione na 100. Jeśli chcemy zmienić tę wartość dla konkretnej kwerendy, możemy użyć wskazówki MAXRECURSION.

Rekurencja jest kluczem do zaimplementowania hierarchii w tym scenariuszu. Na przykład kwerenda z listingu 3.7 wykorzystuje CTE do zwrócenia wszystkich pracowników podlegających bezpośrednio i pośrednio Sanjayowi Gupcie, którego identyfikator pracownika (`EmployeeID`) jest równy 2. W definicji CTE pierwsza kwerenda zwraca rekord pracownika dla samego Sanjaya. Klauzula `UNION ALL` łączy następnie wyniki drugiej kwerendy. Ta druga kwerenda jest rekurencyjna, ponieważ łączy wyniki z tabeli `Employees` z wynikami pierwszej kwerendy w ramach CTE. Ponieważ złączenie odbywa się kolumnach `EmployeeID` w pierwszym zbiorze wyników i `ManagerID` w drugim zbiorze wyników, drugi zbiór wyników zawiera informacje o podległych pracownikach. Końcowa kwerenda `SELECT` znajduje się poza CTE. Zwraca ona wszystkie rekordy z CTE, a następnie łączy się z powrotem z bazową tabelą `Employees`, aby uzupełnić imiona i nazwiska kierowników.

WSKAZÓWKA Zauważ, że średnik kończący instrukcję `SET` znajduje się na początku wiersza rozpoczynającego klauzulę `WITH CTE`. Jest to kwestia stylistyczna. CTE musi zawsze znajdować się na początku wyrażenia. Dlatego powszechną praktyką jest rozpoczynanie wyrażenia od średnika. Zapobiega to błędom kompilacji, nawet jeśli zapomnimy zakończyć poprzednie wyrażenie. Alternatywnie można wprowadzić standard, w którym wszystkie wyrażenia zawsze kończą się średnikiem.

Listing 3.7. Używanie CTE do zwrócenia wszystkich bezpośrednich i pośrednich podwładnych

```
DECLARE @ManagerID INT ;

SET @ManagerID = 2

;WITH EmployeeCTE AS (
    SELECT
        EmployeeID
    ,   FirstName
    ,   LastName
    ,   ManagerID
```

```

FROM dbo.Employees
WHERE EmployeeID = @ManagerID
UNION ALL
SELECT
    Emp.EmployeeID
    , Emp.FirstName
    , Emp.LastName
    , Emp.ManagerId
FROM dbo.Employees AS Emp
INNER JOIN EmployeeCTE AS CTE
    ON CTE.EmployeeID=Emp.ManagerId
)
SELECT
    Emp.EmployeeID
    , Emp.FirstName
    , Emp.LastName
    , Emp.ManagerID
    , Mgr.FirstName AS ManagerFirstName
    , Mgr.LastName AS ManagerLastName
FROM EmployeeCTE Emp
INNER JOIN dbo.Employees Mgr
    ON Emp.ManagerID = Mgr.EmployeeID ;

```

Inna typowa kwerenda, o której napisanie możemy zostać poproszeni, wiąże się z ustaleniem, kto zarządza przełożonym danego pracownika. Często jest to potrzebne przy tworzeniu ścieżek eskalacji. Kwerenda z listingu 3.8 pokazuje, jak można ustalić, kto jest przełożonym kierownika, Emmy Roberts. W tej kwerendzie dodajemy stałą wartość 0 w pierwszej kwerendzie w ramach CTE, z nazwą kolumny `Level` w zbiorze wyników, aby określić, że Emma Roberts znajduje się na poziomie 0. Następnie rekurencyjna kwerenda zwiększa numer poziomu dla każdej warstwy hierarchii. Końcowa kwerenda, poza CTE, filtruje wyniki dla poziomu 2, aby zwrócić encję znajdującą się dwa poziomy powyżej Emmy.

Listing 3.8. Używanie CTE do zwrócenia encji znajdującej się dwa poziomy wyżej w hierarchii

```

DECLARE @EmployeeID INT ;

SET @EmployeeID = 22

; WITH EmployeeCTE AS
(
    SELECT
        employeeid
        , firstname
        , lastname
        , managerid
        , Role
        , 0 as Level
    FROM dbo.Employees
    WHERE EmployeeID = @EmployeeID
    UNION ALL
    SELECT
        emp.EmployeeID

```



```

        , emp.FirstName
        , emp.LastName
        , emp.ManagerID
        , emp.Role
        , Level + 1
    FROM dbo.employees emp
    INNER JOIN EmployeeCTE cte
        ON emp.EmployeeID = cte.ManagerID
)

SELECT
    firstname
    , lastname
    , Role
FROM EmployeeCTE
WHERE Level = 2 ;

```

Problem z tradycyjnym podejściem do implementowania hierarchii polega na tym, że programista musi pisać mnóstwo kodu. Przykłady w tej sekcji są proste i mają charakter ilustracyjny, ale w rzeczywistych scenariuszach rekurencyjne kwerendy CTE mogą być bardzo skomplikowane. Za każdym razem, gdy użytkownicy biznesowi zgłaszają nieco inną prośbę, programiści muszą napisać kod, który będzie odpowiednio przechodził przez hierarchię. Może na przykład pojawić się prośba o zwrócenie listy wszystkich menedżerów na trzecim poziomie struktury organizacyjnej.

Wielu programistów nie zdaje sobie sprawy, że Microsoft wykonał już za nich większość trudnej pracy, implementując typ danych HIERARCHYID. Jest to zaawansowany typ danych napisany w .NET, który pozwala wywoływać metody umożliwiające przemierzanie hierarchii bez pisania skomplikowanych zapytań CTE.

Aby zobaczyć, jak to działa, dodajmy nową kolumnę o nazwie ManagerHierarchyID do tabeli Employees. Kolumna ta będzie miała typ HIERARCHYID. Możemy to zrobić za pomocą skryptu przedstawionego na listingu 3.9.

Listing 3.9. Dodawanie kolumny ManagerHierarchyID

```

ALTER TABLE dbo.Employees ADD
    ManagerHierarchyID HIERARCHYID NULL ;

```

Aby skorzystać z funkcji hierarchicznych SQL Servera, musimy najpierw wymodelować hierarchię. Wyjaśnijmy to na przykładzie fragmentu schematu organizacyjnego. Simon Gomez znajduje się na szczycie hierarchii, który nazwiemy korzeniem.

Hierarchię modeluje się z użyciem formatu z ukośnikami. Korzeń hierarchii jest reprezentowany przez /. Sanjay Gupta i Ed Ling znajdują się na drugim poziomie hierarchii i każdy z nich musi być jednoznacznie zidentyfikowany. Dlatego Sanjay będzie reprezentowany przez /1/, a Ed przez /2/.

Jamie Briggs i Robin Round podlegają Edowi Lingowi, co oznacza, że znajdują się na trzecim poziomie hierarchii. Aby ich jednoznacznie zidentyfikować,

przypisuje się im, odpowiednio, oznaczenia /1/2/1/ oraz /1/2/2/. Struktura ta jest stopniowo rozbudowywana na kolejnych poziomach. Identyfikatory hierarchii są przechowywane w tabeli jako ciągi bitów i wyświetlane w postaci szesnastkowej. Na przykład identyfikator Simona Gomeza na najwyższym poziomie jest zapisany jako 0x, Sanjaya Gupty na drugim poziomie — jako 0x58, a Eda Linga, również na drugim poziomie, jako 0x68. Jamie Briggs i Robin Round mają przypisane, odpowiednio, wartości 0x6AC0 i 0x6B40.

Nie obawiaj się jednak: nie trzeba ręcznie modelować hierarchii. Zamiast tego zastosujemy podejście dwuetapowe, które znacznie ułatwi proces modelowania. Pierwszym krokiem będzie utworzenie tymczasowej tabeli składającej się z trzech kolumn: `EmployeeID`, `ManagerID` oraz numeru wiersza. Ten ostatni wygenerujemy za pomocą funkcji `ROW_NUMBER()`, partycjonując numery według `ManagerID`. Funkcja ta zapewni nam przyrostowe numerowanie, którego potrzebujemy na każdym poziomie gałęzi hierarchii.

Drugim krokiem jest zdefiniowanie wyrażenia CTE opartego na naszej tymczasowej tabeli, które określi korzeń hierarchii. W tym celu użyjemy globalnej metody `GetRoot()` dla typu `HIERARCHYID`, odwołując się do Simona Gomeza, który znajduje się na szczycie hierarchii i dlatego ma wartość `NULL` w polu `ManagerID`.

Kwerenda rekurencyjna będzie budować kolejne poziomy hierarchii poprzez łączenie następnego poziomu z przyrostową wartością generowaną za pomocą funkcji `ROW_NUMBER()`. Następnie możemy użyć instrukcji `UPDATE`, aby pobrać modelowane identyfikatory hierarchii z CTE i zaktualizować tabelę `Employees` poprzez złączenie na kolumnie `EmployeeID`. Pokazano to na listingu 3.10.

Listing 3.10. Generowanie Identyfikatorów hierarchii w tabeli `Employees`

```
SELECT
    EmployeeID
    , ManagerID
    , ROW_NUMBER()
      OVER (PARTITION BY ManagerID ORDER BY ManagerID) AS Incremental
INTO #Hierarchy
FROM dbo.Employees
;WITH HierarchyPathCTE AS (
    SELECT
        hierarchyid::GetRoot() AS ManagerHierarchyID
        , EmployeeID
    FROM #Hierarchy AS C
    WHERE ManagerID IS NULL
    UNION ALL
    SELECT
        CAST(hpc.ManagerHierarchyID.ToString() +
            CAST(h.Incremental AS VARCHAR(30)) +
            '/' AS HIERARCHYID)
        , h.EmployeeID
    FROM #Hierarchy AS h
    JOIN HierarchyPathCTE AS hpc
      ON h.ManagerID = hpc.EmployeeID
)
```

```
UPDATE e
  SET ManagerHierarchyID = hp.ManagerHierarchyID
FROM dbo.Employees e
INNER JOIN HierarchyPathCTE hp
  ON e.EmployeeID = hp.EmployeeID ;
```

Rozróżnianie wielkości liter

Ważna uwaga dotycząca pracy z typem danych HIERARCHYID: metody te są wrażliwe na wielkość liter. Oznacza to, że następująca kwerenda spowoduje błąd:

```
SELECT ManagerHierarchyID.ToString() FROM dbo.Employees ;
```

Natomiast poniższa kwerenda zostałaaby wykonana pomyślnie:

```
SELECT ManagerHierarchyID.ToString() FROM dbo.Employees ;
```

Od tego momentu praca z hierarchiami staje się niezwykle prosta. Przypomnij sobie kwerendę, którą napisaliśmy, aby wygenerować listę wszystkich bezpośrednich i pośrednich podwładnych Sanjaya Gupty. Możemy zastąpić całą tę rekurencyjną kwerendę prostszą instrukcją, którą przedstawiano na listingu 3.11. Wykorzystuje ona metodę `IsDescendantOf()` do filtrowania wyników poprzez identyfikowanie osób znajdujących się w gałęzi hierarchii Sanjaya Gupty.

Listing 3.11. Zwracanie wszystkich bezpośrednich i pośrednich podwładnych Sanjaya Gupty

```
DECLARE @Manager HIERARCHYID ;

SELECT @Manager = ManagerHierarchyID
FROM dbo.Employees
WHERE EmployeeID = 2 ;

SELECT
  EmployeeID
  , FirstName
  , LastName
  , ManagerHierarchyID.ToString()
FROM dbo.Employees
WHERE ManagerHierarchyID.IsDescendantOf(@Manager) = 1 ;
```

W podobny sposób można zastąpić rekurencyjną kwerendę, którą napisaliśmy wcześniej w celu znalezienia przełożonego kierownika, Emmy Roberts, prostym skryptem przedstawionym na listingu 3.12. Wykorzystuje on metodę `GetAncestor()` do poruszania się po hierarchii.

Listing 3.12. Ustalanie przełożonego kierownika, Emmy Roberts

```
DECLARE @EmployeeID INT ;

SET @EmployeeID = 22 ;

SELECT
  FirstName
```

```

        , LastName
        , Role
        , ManagerHierarchyID.ToString() AS ManagerHierarchyID
FROM dbo.Employees
WHERE ManagerHierarchyID = (
    SELECT ManagerHierarchyID.GetAncestor(2)
    FROM dbo.Employees
    WHERE EmployeeID = @EmployeeID
) ;

```

Może pamiętasz, że wspomniałem również o możliwości przeglądania hierarchii na różne sposoby i zasugerowałem, że może pojawić się potrzeba wyszukiwania elementów, które znajdują się na tym samym poziomie hierarchii. Aby przekonać się, jakie to proste, wyobraź sobie, że firma MagicChoc przeprowadza analizę porównawczą wynagrodzeń. Konkretnie, Amy Fry poprosiła o podwyżkę, a dział kadr chce wiedzieć, czy jej pensja jest zbliżona do innych osób na jej poziomie organizacyjnym.

W tym przypadku możemy użyć metody `GetLevel()` do określenia poziomu hierarchii, na którym znajduje się dana osoba, a następnie zwrócić informacje o wszystkich innych pracownikach na tym samym poziomie. Technika ta została zaprezentowana na listingu 3.13.

Listing 3.13. Zwracanie wszystkich pracowników będących na tym samym poziomie organizacyjnym co Amy Fry

```

DECLARE @EmployeeID INT ;

SET @EmployeeID = 14 ;

SELECT
    FirstName
    , LastName
    , Salary
FROM dbo.Employees
WHERE ManagerHierarchyID.GetLevel() = (
    SELECT
        ManagerHierarchyID.GetLevel()
    FROM dbo.Employees
    WHERE EmployeeID = @EmployeeID
) ;

```

3.4. Numer 9 — nieprzechowywanie danych XML w natywnym formacie

Gdy dane XML są przekazywane do bazy danych, programiści często czują się zobowiązani do zapisania tych danych w strukturach relacyjnych. Czasami wynika to z przekonania o dobrych praktykach — skoro używamy bazy danych, to z pewnością lepiej przechowywać dane w formacie relacyjnym. W innych przypadkach wybór ten jest podyktowany obawą przed XML i potencjalną koniecznością pisania skomplikowanych zapytań XQuery w celu uzyskania dostępu do danych. Prawdę mówiąc, istnieje wiele sytuacji, w których przechowywanie danych w formacie relacyjnym jest dobrym pomysłem, ale w niektórych sytuacjach jest to dalekie od prawdy.

Wróćmy do naszej tabeli `Employees`. Do tej pory dodawaliśmy jedynie rekordy dla zespołu zarządzającego, czyli stałych pracowników. Magazyn jednak często zatrudnia pracowników tymczasowych poprzez agencję pracy o nazwie `Total Warehouse Jobs`. `MagicChoc` prowadzi intensywną współpracę z tą agencją, dlatego obie firmy postanowiły zintegrować swoje systemy.

Gdy pracownik tymczasowy kończy pracę w `MagicChoc`, jego dane są usuwane z systemu. Jednak firma `Total Warehouse Jobs` zachowuje informacje o jego poprzednich umowach. Jeśli dana osoba podpisze nową umowę z `MagicChoc`, `Total Warehouse Jobs` ma obowiązek dostarczyć listę jej wcześniejszych umów wraz z pełnionymi stanowiskami. Pomaga to `MagicChoc` przydzielić pracownikowi odpowiednie zadania, w których ma on już doświadczenie.

Natura formatu XML sprawia, że jest on rozszerzalny i czytelny dla człowieka, obsługuje walidację schematów i może być uniwersalnie przetwarzany. To czyni go popularnym wyborem w integracji systemów. W naszym scenariuszu uzgodniono, że po podpisaniu umowy firma `Total Warehouse Jobs` prześle szczegóły dotyczące poprzednich umów pracownika w dokumencie XML. Te informacje muszą zostać zapisane w bazie danych `HumanResources`.

Aplikacja magazynowa wykorzystuje te dane na co dzień do obliczania, które osoby zostaną przypisane do jakich zadań, w zależności od bieżących priorytetów produkcyjnych. W procesie ETL dane muszą być przesyłane do aplikacji magazynowej w formacie XML.

3.4.1. Szatkowanie XML

Sposób, w jaki niektórzy programiści podchodzą do tego scenariusza, polega na szatkowaniu otrzymanych danych XML przed zapisaniem ich w tabeli relacyjnej, a następnie rekonstruowaniu dokumentu XML w celu przesłania go do odbiorcy.

WSKAZÓWKA Szatkowanie XML (ang. *shredding*) to proces usuwania znaczników z danych i organizowania ich w postaci relacyjnej.

Gdybyśmy chcieli zastosować to podejście w naszym przykładzie MagicChoc, pierwszym krokiem byłoby stworzenie schematu do przechowywania historycznych danych kontraktowych. Ze względu na charakter tych danych będziemy potrzebować trzech tabel, aby uniknąć duplikowania informacji. Jest to kluczowa koncepcja normalizacji, którą omówimy szerzej w rozdziale 4. Przykładową strukturę tabel przedstawiono na listingu 3.14.

Listing 3.14. Tworzenie struktury tabel na historyczne dane kontraktowe

```
CREATE TABLE dbo.ContractHistory (
    ContractHistoryID SMALLINT NOT NULL PRIMARY KEY IDENTITY,
    EmployeeID SMALLINT NOT NULL
    REFERENCES dbo.Employees(EmployeeID),
    ContractStartDate DATE NOT NULL,
    ContractEndDate DATE NOT NULL
);

CREATE TABLE dbo.Skills (
    SkillsID SMALLINT NOT NULL PRIMARY KEY IDENTITY,
    Skill VARCHAR(30) NOT NULL
);

CREATE TABLE dbo.ContractSkills (
    ContractSkillsID INT NOT NULL PRIMARY KEY IDENTITY,
    ContractHistoryID SMALLINT NOT NULL
    REFERENCES dbo.ContractHistory(ContractHistoryID),
    SkillID SMALLINT NOT NULL
    REFERENCES dbo.Skills(SkillsID)
);
```

Ponieważ właśnie utworzona tabela `Skills` jest tabelą referencyjną, zanim przejdziemy dalej, dodajmy do niej trochę przykładowych danych. W tym celu użyjemy skryptu z listingu 3.15.

Listing 3.15. Dodawanie danych do tabeli `Skills`

```
INSERT INTO dbo.Skills (Skill)
VALUES
    ('Picker'),
    ('Packer'),
    ('Stock Take'),
    ('Forklift Driver'),
    ('Machine 1 operator'),
    ('Machine 2 operator'),
    ('Machine 3 operator'),
    ('Machine 4 operator');
```

Historyczne dane kontraktowe są przechowywane w dokumencie XML zorientowanym na elementy, z głównym elementem o nazwie `<EmployeeContracts>`.

Odwzorowania zorientowane na elementy i na atrybuty

W zorientowanym na elementy dokumencie XML element zawiera elementy podrzędne, które przechowują jego właściwości. Inaczej jest w dokumentach XML zorientowanych na atrybuty — w dokumentach tych właściwości elementu są przechowywane w jego atrybutach. Na przykład w poniższym fragmencie:

```
<Employee ID="23"></Employee>
```

ID jest atrybutem elementu `<Employee>` i ma wartość 23. W przypadku prostych dokumentów wybór między elementami a atrybutami jest w dużej mierze kwestią stylu. Jednak gdy mamy do czynienia ze złożonymi węzłami, które będą się powtarzać lub muszą występować w określonej kolejności, konieczne jest użycie elementów, ponieważ nie da się tego osiągnąć za pomocą atrybutów.

Pod elementem głównym znajduje się złożony element o nazwie `<Employee>`, który zawiera elementy podrzędne przechowywane w naszej tabeli `Employees`. Zawiera on również zagnieżdżony element złożony o nazwie `<Contracts>`, który z kolei zawiera powtarzający się element `<Contract>` z informacjami o każdej umowie zawartej z pracownikiem. W elemencie `<Contract>` zagnieżdżony jest kolejny element złożony o nazwie `<Skills>` zawierający powtarzający się element `<Skill>`. Poniżej przedstawiono przykład danych, jakie otrzymujemy od firmy Total Warehouse Jobs:

```
<EmployeeContracts>
  <Employee>
    <EmployeeID>23</EmployeeID>
    <FirstName>Robert</FirstName>
    <LastName>Blake</LastName>
    <DateOfBirth>19781212</DateOfBirth>
    <Contracts>
      <Contract>
        <StartDate>20200101</StartDate>
        <EndDate>20203006</EndDate>
        <Skills>
          <Skill>Forklift driver</Skill>
          <Skill>Picker</Skill>
          <Skill>Packer</Skill>
        </Skills>
      </Contract>
      <Contract>
        <StartDate>20210101</StartDate>
        <EndDate>20211212</EndDate>
        <Skills>
          <Skill>Picker</Skill>
          <Skill>Stock Take</Skill>
        </Skills>
      </Contract>
    </Contracts>
  </Employee>
</EmployeeContracts>
```

Jeśli chcemy przechowywać ten zbiór danych w tradycyjnej strukturze relacyjnej, naszym pierwszym zadaniem będzie poszatkowanie otrzymanych danych.

W tym celu możemy użyć funkcji `OPENXML()`, która zwróci zestaw wierszy z dokumentu XML, albo kombinacji metod XQuery: `nodes()` i `value()`. W tym przykładzie użyjemy `OPENXML()`, ponieważ jest to metoda, którą najczęściej spotykam w praktyce.

Pierwszą trudnością związaną z funkcją `OPENXML()` jest to, że nie ma ona wbudowanego parsera XML. Dlatego zanim prześlemy dokument XML do funkcji, musimy go przetworzyć za pomocą parsera MSXML. Aby odpowiednio przygotować dokument, możemy skorzystać z procedury składowanej `sp_xml_preparedocument`. Procedura ta przetworzy dokument, a obiekt wynikowy będzie zawierał uchwyt do przechowywanego w pamięci drzewa zawierającego węzły dokumentu. Uchwyt ten można następnie przekazać do funkcji `OPENXML()`.

UWAGA Podczas korzystania z procedury `sp_xml_preparedocument` ważne jest, aby po użyciu funkcji `OPENXML()` wywołać procedurę `sp_xml_removedocument`. Pozwoli to zwolnić zajmowaną pamięć.

Musimy również przekazać do funkcji `OPENXML()` wyrażenie XPath, które identyfikuje wiersze do przetworzenia. W wyrażeniu tym wskażemy najniższy poziom naszej hierarchii, a następnie w naszych odwzorowaniach użyjemy operatora `../` do poruszania się w kierunku wyższych poziomów.

WSKAZÓWKA XPath to zdefiniowany przez W3C język służący do nawigacji po węzłach w dokumencie XML. Wyrażeń XPath używa się do wybierania konkretnych węzłów w strukturze dokumentu XML.

Ostatni parametr jest opcjonalny i określa sposób wypełnienia kolumny nadmiarowej. Możliwe wartości są szczegółowo opisane w tabeli 3.2.

Tabela 3.2. Opcje wypełniania kolumny nadmiarowej

| Wartość | Opis |
|---------|---|
| 0 | Mapowanie zorientowane na atrybuty. |
| 1 | Stosuje mapowanie zorientowane na atrybuty, a następnie mapowanie zorientowane na elementy. |
| 2 | Stosuje mapowanie zorientowane na elementy, a następnie mapowanie zorientowane na atrybuty. |
| 8 | Nie kopiuje danych do właściwości przepełnienia. |

Klauzuli `WITH` używa się w połączeniu z funkcją `OPENXML()` do określenia typu danych i odwzorowania węzłów w dokumencie. Jeśli klauzula `WITH` zostanie pominięta, SQL Server zwróci *krawędziową tabelę XML*, która szczegółowo opisuje strukturę dokumentu i zawiera takie informacje, jak URI przestrzeni nazw, prefiks przestrzeni nazw oraz wskaźniki do następnych i poprzednich elementów równorzędnych.

Skrypt przedstawiony na listingu 3.16 pokazuje, jak może wyglądać proces szatkowania danych i wstawiania ich do tabel. Na początku deklarujemy zmienne

do przechowywania surowego dokumentu XML oraz uchwytu do przechowywanego w pamięci drzewa dokumentu. Następnie tworzymy dwie zmienne tabelaryczne. Pierwsza będzie przechowywać wyniki funkcji `OPENXML()`, a druga symuluje wewnętrzne dane pracowników, które normalnie pobieralibyśmy z systemu kadrowego i które są potrzebne do wypełnienia tabeli `Employees`. Kolejnym krokiem jest analiza składniowa danych XML przed wywołaniem funkcji `OPENXML()` i wstawienie wyników do zmiennej tabelarycznej. Następnie wykonujemy kwerendy, aby powielić dane i wstawić je do właściwych tabel. Te instrukcje `INSERT` są zawarte w transakcji (o czym będzie mowa w rozdziale 10.), co oznacza, że jeśli jedna z operacji wstawiania się nie powiedzie, wycofane zostaną wszystkie. Zapobiega to powstaniu niespójności między tabelami, które musielibyśmy ręcznie naprawiać w przypadku awarii. Przed rozpoczęciem transakcji włączamy opcję `XACT_ABORT`. Powoduje to zatrzymanie transakcji nawet w przypadku drobnego błędu, takiego jak naruszenie ograniczenia klucza obcego.

WSKAZÓWKA Zgadza się! Masz absolutną rację! Powinniśmy dodać do tego kodu obsługę błędów, zwłaszcza że dane XML pochodzą z zewnętrznego źródła. Obsługę błędów omówimy w rozdziale 7.

Listing 3.16. Szatkowanie danych za pomocą funkcji `OPENXML()`

```

DECLARE @RawContractDetails XML ; <----- Deklaruje zmienną do przechowywania surowego XML.
DECLARE @ParsedContractDetails INT ; <-----
DECLARE @ShreddedData TABLE (
    EmployeeID          INT
    , FirstName          NVARCHAR(32)
    , LastName           NVARCHAR(32)
    , DateOfBirth        DATE
    , ContractStartDate  DATE
    , ContractEndDate    DATE
    , Skill              NVARCHAR(30)
) ;
-- Deklaruje zmienną przechowującą wskaźnik do
-- przechowywanego w pamięci drzewa parsowania.
DECLARE @InternalEmployeeData TABLE (
    EmployeeStartDate    DATE
    , ManagerID          SMALLINT
    , Salary              MONEY
    , Department         NVARCHAR(64)
    , DepartmentCode     NCHAR(2)
    , Role               NVARCHAR(64)
    , WeeklyContractedHours INT
    , StaffOrContract    BIT
    , ContractEndDate    DATE
    , ManagerHierarchyID HIERARCHYID
) ;

INSERT INTO @InternalEmployeeData
VALUES (
    '20230101',
    14,
    39000,

```

```

'Manufacturing',
'MA',
'Warehouse Operative',
40,
0,
'20231231',
'/1/2/2/1/'
) ;

SET @RawContractDetails = N'<EmployeeContracts>
  <Employee>
    <EmployeeID>23</EmployeeID>
    <FirstName>Robert</FirstName>
    <LastName>Blake</LastName>
    <DateOfBirth>19781212</DateOfBirth>
    <Contracts>
      <Contract>
        <StartDate>20200101</StartDate>
        <EndDate>20200603</EndDate>
        <Skills>
          <Skill>Forklift driver</Skill>
          <Skill>Picker</Skill>
          <Skill>Packer</Skill>
        </Skills>
      </Contract>
      <Contract>
        <StartDate>20210101</StartDate>
        <EndDate>20211231</EndDate>
        <Skills>
          <Skill>Picker</Skill>
          <Skill>Stock Take</Skill>
        </Skills>
      </Contract>
    </Contracts>
  </Employee>
</EmployeeContracts>' ;

EXEC sp_xml_preparedocument @ParsedContractDetails OUTPUT, @RawContractDetails ;

```

Parsuje dokument XML.

```

INSERT INTO @ShreddedData
SELECT *
FROM OPENXML(@ParsedContractDetails,
'/EmployeeContracts/Employee/Contracts/Contract/Skills/Skill', 2)
WITH (
  EmployeeID          SMALLINT          '.../EmployeeID',
  FirstName           NVARCHAR(32)      '.../FirstName',
  LastName            NVARCHAR(32)      '.../LastName',
  DateOfBirth         DATE              '.../DateOfBirth',
  ContractStartDate   DATE              '.../StartDate',
  ContractEndDate     DATE              '.../EndDate',
  Skill               NVARCHAR(30)      'text()'
) ;

```

← Szatkuje dane XML i wstawia wartości relacyjne do zmiennej tabelarycznej.

```

SET XACT_ABORT ON ;

```

← Włącza opcję XACT_ABORT, więc jeśli którakolwiek instrukcja w transakcji zakończy się niepowodzeniem, zostanie wycofana cała transakcja.

```

BEGIN TRANSACTION
  INSERT INTO dbo.Employees
  SELECT

```

← Zaczyna transakcję w celu zaktualizowania tabeli, więc jeśli jedna instrukcja się nie powiedzie, transakcja zostanie wycofana.

```

        s.EmployeeID
    , s.FirstName
    , s.LastName
    , s.DateOfBirth
    , i.EmployeeStartDate
    , i.ManagerID
    , i.Salary
    , i.Department
    , i.DepartmentCode
    , i.Role
    , i.WeeklyContractedHours
    , i.StaffOrContract
    , i.ContractEndDate
    , i.ManagerHierarchyID
FROM @ShreddedData s
INNER JOIN @InternalEmployeeData i
    ON 1=1
GROUP BY
    s.EmployeeID
    , s.FirstName
    , s.LastName
    , s.DateOfBirth
    , i.EmployeeStartDate
    , i.ManagerID
    , i.Salary
    , i.Department
    , i.DepartmentCode
    , i.Role
    , i.WeeklyContractedHours
    , i.StaffOrContract
    , i.ContractEndDate
    , i.ManagerHierarchyID ;

INSERT INTO dbo.ContractHistory(
    EmployeeID,
    ContractStartDate,
    ContractEndDate
)
SELECT
    EmployeeID
    , ContractStartDate
    , ContractEndDate
FROM @ShreddedData
GROUP BY
    EmployeeID
    , ContractStartDate
    , ContractEndDate ;

INSERT INTO dbo.ContractSkills(ContractHistoryID, SkillID)
SELECT
    ch.ContractHistoryID
    , s.SkillsID
FROM @ShreddedData sd
INNER JOIN dbo.Skills s
    ON TRIM(s.Skill) = TRIM(sd.Skill)
INNER JOIN dbo.ContractHistory ch
    ON sd.EmployeeID = ch.EmployeeID
    AND sd.ContractStartDate = ch.ContractStartDate

```

```
AND sd.ContractEndDate = ch.ContractEndDate ;
```

```
COMMIT
```

```
EXEC sp_xml_removedocument @ParsedContractDetails ; ← Usuuwa z pamięci drzewo parsowania.
```

No cóż, to było pracochłonne zadanie, prawda? Dla SQL Servera również! W moim środowisku testowym, które jest instancją EC2 typu t2.large uruchomioną wyłącznie na potrzeby tego skryptu, wykonanie zadania zajęło 31 ms. To 31 ms na przetworzenie pięciu wierszy danych wstawianych do trzech tabel. Na razie tylko poszatkowaliśmy dane. W następnym punkcie przyjrzymy się procesowi rekonstruowania dokumentu XML w celu wysłania go do klienta. Prawdopodobnie zaczynasz rozumieć, dlaczego nie polecam tego podejścia w naszym konkretnym przypadku.

3.4.2. Rekonstruowanie XML

Teraz, gdy poszatkowaliśmy dane XML i wstawiliśmy je do tabel, musimy utworzyć proces, którego mechanizm ETL będzie używał do przesyłania danych do aplikacji hurtowni danych. Oznacza to odtworzenie dokumentu XML z danych przechowywanych w tabelach. Możemy to osiągnąć za pomocą kwerendy SELECT z klauzulą FOR XML. Klauzula FOR XML oferuje cztery tryby działania, które zostały szczegółowo opisane w tabeli 3.3.

Tabela 3.3. Tryby FOR XML

| Tryb | Opis |
|----------|---|
| RAW | Najbardziej podstawowy tryb tworzenia XML. Generuje płaski dokument XML, z jednym elementem na wiersz. |
| AUTO | Generuje dokumenty XML z zagnieżdżonymi elementami. Zagnieżdżenie jest kontrolowane przez warunki złączeń w kwerendzie. Automatyczne formatowanie oznacza, że mamy minimalną kontrolę nad formatem XML. |
| PATH | Umożliwia zaawansowaną kontrolę nad formatem dokumentu XML poprzez odwzorowywanie kolumn w kwerendzie na węzły XML w określonym miejscu hierarchii. |
| EXPLICIT | Oferuje podobną kontrolę nad formatowaniem jak tryb PATH, ale jest bardzo skomplikowany. Zazwyczaj nie ma potrzeby używania tego trybu. |

Aby utworzyć odpowiednią strukturę naszego dokumentu XML, musimy użyć klauzuli FOR XML PATH wraz z podkwerendami, które również korzystają z tej klauzuli. Podkwerendy są niezbędne do obsługi powtarzających się elementów na potomnym poziomie dokumentu.

Kwerenda z listingu 3.17 pokazuje, jak możemy osiągnąć ten cel, używając dwóch poziomów podkwerend. Najbardziej wewnętrzna kwerenda zwraca umiejętności powiązane z danym kontraktem. Wyniki te są przekształcane w XML z użyciem klauzuli FOR XML PATH. Klauzula ta wykorzystuje słowa kluczowe TYPE,

aby określić, że wyniki będą poprawnie uformowanym dokumentem XML, oraz ROOT — do zdefiniowania nazwy elementu głównego. W podkwerendzie zewnętrznej stosuje się tę samą metodę do pobrania szczegółów z powtarzającego się elementu Contracts. Na koniec kwerenda zewnętrzna zwraca dane pracownika, które zostaną umieszczone na szczycie hierarchii.

Listing 3.17. Rekonstruowanie dokumentu XML

```
SELECT
    e.EmployeeID 'EmployeeID'
  , e.FirstName 'FirstName'
  , e.LastName 'LastName'
  , e.DateOfBirth 'DateOfBirth'
  , (
      SELECT
        ch.ContractStartDate 'StartDate'
      , ch.ContractEndDate 'EndDate'
      , (
          SELECT
            s.Skill 'Skill'
          FROM dbo.Skills s
          INNER JOIN dbo.ContractSkills cs
            ON s.SkillsID = cs.SkillID
          WHERE cs.ContractHistoryID = ch.ContractHistoryID
          FOR XML PATH(''), TYPE, ROOT('Skills')
        )
      FROM dbo.ContractHistory ch
      WHERE EmployeeID = e.EmployeeID
      FOR XML PATH('Contract'), TYPE, ROOT('Contracts')
    )
FROM dbo.Employees e
WHERE EmployeeID = 23
FOR XML PATH('Employee'), ROOT('EmployeeContracts') ;
```

W moim środowisku testowym ta kwerenda zajęła 52 ms. Wyobraź sobie teraz środowisko produkcyjne, gdzie takie procesy są uruchamiane nieustannie przez wielu użytkowników. Pomyśl też o sytuacjach, gdy te operacje muszą być wykonywane na dużych, złożonych dokumentach XML. Łatwo zrozumieć, dlaczego może to szybko stać się bardzo kosztowne.

3.4.3. Unikanie dodatkowych kosztów przez przechowywanie danych w formacie XML

Konwersja danych z formatu XML na relacyjny i z powrotem wymagała sporo zachodu i zajęła łącznie 83 ms. Moglibyśmy zaoszczędzić czas przeznaczony na programowanie oraz kompilację i wykonanie, gdybyśmy przechowywali dane w formacie XML. Przyjrzyjmy się zatem, jaki wpływ na pisanie kodu i przetwarzanie danych miałoby przechowywanie danych w ich natywnym formacie.

Ponieważ wszystkie informacje dotyczące umiejętności i umów pracownika są przechowywane w jednym dokumencie XML, nie ma potrzeby tworzenia osobnych tabel `EmployeeContracts` i `EmployeeSkills`. Zamiast tego możemy po prostu wstawić dokument XML do tabeli `Employees`. Zanim zaczniemy, zaktualizujemy tabelę `Employees`, dodając kolumnę o nazwie `PreviousContracts`, która będzie przechowywać te dane. Odpowiednie polecenie pokazano na listingu 3.18.

Listing 3.18. Dodawanie kolumny `PreviousContracts` do tabeli `Employees`

```
ALTER TABLE dbo.Employees
    ADD PreviousContracts XML NULL ;
```

W naszym konkretnym przypadku nie unikniemy użycia niewielkiej ilości kodu XQuery. Wynika to stąd, że dane będą indeksowane na podstawie identyfikatora pracownika (`EmployeeID`), który jest przekazywany przez firmę Total Warehouse Jobs wewnątrz dokumentu XML. W związku z tym będziemy musieli wyodrębnić tę wartość wraz z imieniem i nazwiskiem pracownika oraz jego datą urodzenia.

Listing 3.19 przedstawia skrypt, którego możemy użyć do dodania nowego użytkownika do tabeli `Employees`. Podobnie jak w poprzednim przykładzie, symulujemy nasz system kadrowy, aby dostarczyć niektóre wartości, które wypełnią tabelę `Employees`. Używamy w tym celu zmiennej tabelarycznej o nazwie `@InternalEmployeeData`. Następnie wykorzystujemy metodę `value()` z XQuery, aby wyodrębnić węzły `EmployeeID`, `FirstName`, `LastName` i `DateOfBirth` z dokumentu XML. Metoda `value()` przyjmuje ścieżkę XPath do węzła, który chcemy wyodrębnić, a także typ danych SQL Servera, na który węzeł zostanie odwzorowany. Metoda `value()` wymaga pojedynczej wartości, dlatego indeks żadanego węzła jest obowiązkowy, nawet jeśli węzeł się nie powtarza. Warto zauważyć, że `EmployeeID` w dokumencie XML zostało zwiększone, aby uniknąć niepowodzenia związanego ze wstawianiem z powodu naruszenia ograniczenia klucza głównego.

WSKAZÓWKA Wyodrębniamy wartość elementu, ale gdybyśmy chcieli wyodrębnić atrybut, musielibyśmy poprzedzić jego nazwę symbolem @.

Listing 3.19. Dodawanie pracownika do tabeli `Employees`

```
DECLARE @RawContractDetails XML ;
DECLARE @InternalEmployeeData TABLE (
    EmployeeStartDate    DATE
    , ManagerID          SMALLINT
    , Salary              MONEY
    , Department          NVARCHAR(64)
    , DepartmentCode     NCHAR(2)
    , Role                NVARCHAR(64)
    , WeeklyContractedHours INT
    , StaffOrContract    BIT
    , ContractEndDate    DATE
    , ManagerHierarchyID HIERARCHYID
```

```
) ;
```

```
INSERT INTO @InternalEmployeeData
VALUES (
```

```
    '20230101',
    14,
    39000,
    'Manufacturing',
    'MA',
    'Warehouse Operative',
    40,
    0,
    '20231231',
    '/1/2/2/1/'
);
```

```
SET @RawContractDetails = N'<EmployeeContracts>  Definiuje dokument XML.
```

```
    <Employee>
      <EmployeeID>25</EmployeeID>
      <FirstName>Robert</FirstName>
      <LastName>Blake</LastName>
      <DateOfBirth>19781212</DateOfBirth>
      <Contracts>
        <Contract>
          <StartDate>20200101</StartDate>
          <EndDate>20200603</EndDate>
          <Skills>
            <Skill>Forklift driver</Skill>
            <Skill>Picker</Skill>
            <Skill>Packer</Skill>
          </Skills>
        </Contract>
        <Contract>
          <StartDate>20210101</StartDate>
          <EndDate>20211231</EndDate>
          <Skills>
            <Skill>Picker</Skill>
            <Skill>Stock Take</Skill>
          </Skills>
        </Contract>
      </Contracts>
    </Employee>
  </EmployeeContracts>' ;
```

```
INSERT INTO dbo.Employees
```

```
SELECT 
    @RawContractDetails.value('/EmployeeContracts/Employee/EmployeeID')[1]', 'SMALLINT')
    AS EmployeeID
, @RawContractDetails.value('/EmployeeContracts/Employee/FirstName')[1]',
    'NVARCHAR(32)') AS FirstName
, @RawContractDetails.value('/EmployeeContracts/Employee/LastName')[1]', 'NVARCHAR(32)')
    AS LastName
, @RawContractDetails.value('/EmployeeContracts/Employee/DateOfBirth')[1]', 'DATE') AS
    DateOfBirth
, EmployeeStartDate
, ManagerID
, Salary
, Department
```

Lista SELECT składa się z wywołań metody value() w celu wyodrębnienia węzłów z dokumentu XML.

```
, DepartmentCode  
, Role  
, WeeklyContractedHours  
, StaffOrContract  
, ContractEndDate  
, ManagerHierarchyID  
, @RawContractDetails AS PreviousContracts  
FROM @InternalEmployeeData ;
```

Wykonanie tego skryptu zajęło 33 ms w moim środowisku testowym. Teraz możemy pobrać dokument XML dla aplikacji magazynowej za pomocą prostej kwerendy SELECT z listingu 3.20.

Listing 3.20. Pobieranie dokumentu XML dla aplikacji magazynowej

```
SELECT  
    PreviousContracts  
FROM dbo.Employees  
WHERE EmployeeID = 25 ;
```

W moim środowisku testowym ta kwerenda SELECT wykonała się w czasie krótszym niż 1 ms. Przy zaokrągleniu tego czasu do 1 ms oznacza to, że wstawienie danych i zwrócenie ich w formacie XML zajęło łącznie 34 ms. W rezultacie całkowity czas przetwarzania jest ponad dwukrotnie krótszy, kiedy unikamy szatkowania danych.

Czy to oznacza, że nigdy nie powinniśmy szatkować danych XML? Absolutnie nie! Jest wiele dobrych powodów, żeby to robić. Ogólna zasada, którą stosuję, mówi, że T-SQL jest znacznie wydajniejszy niż XQuery. Jeśli więc nasz scenariusz wymaga częstego odczytywania danych, ich poszatkowanie jest dobrym pomysłem. Z drugiej strony, jeśli często zapisujemy dane, ale rzadko je odczytujemy, najlepszym podejściem jest przechowywanie ich w natywnym formacie XML. Błędem, którego powinniśmy unikać, jest szatkowanie danych w każdej sytuacji (lub przechowywanie ich zawsze w formacie XML). Decyzję projektową należy podjąć w oparciu o wymagania aplikacji.

3.5. Numer 10 — ignorowanie JSON

Programiści SQL mają tendencję nie tylko do unikania danych XML, często unikają również danych w formacie JSON. Jak już widzieliśmy przy okazji omawiania XML, może to prowadzić do nieefektywnego projektowania tabel i obniżenia wydajności. Wyobraźmy sobie, że poproszono nas o rozszerzenie schematu danych, aby umożliwić przechowywanie domowych adresów pracowników.

Typowym sposobem modelowania adresów w SQL Serverze jest stworzenie tabeli `Adresses` z kluczem głównym `AdressID`. Ta sama nazwa kolumny występuje w tabeli `Employees`, a także potencjalnie w innych tabelach, gdzie adres jest

istotny, jak na przykład `Offices` czy `Sites` w przypadku bazy danych kadrowych. Kolumna w tabeli `Employees` miałaby ograniczenie klucza obcego, dzięki czemu można by przechowywać w niej tylko odpowiednie wartości z tabeli `Addresses`.

Taki model jest w porządku, ale tabela adresów będzie szeroka i rozrzedzona, ponieważ nie wszystkie adresy mają taką samą liczbę wierszy, a my musimy być przygotowani na każdą ewentualność. Sytuację pogarsza fakt, że MagicChoc jest firmą międzynarodową. Weźmy na przykład kody pocztowe. W Wielkiej Brytanii znane są jako „post codes”, w Tajlandii jako „yóu dì qū hào”, a w Bahrajnie jako numery bloków, by wymienić tylko kilka przykładów. Ponieważ mają różne formaty w różnych krajach, musielibyśmy albo przechowywać je w bardzo elastycznym typie danych, albo użyć osobnych, rzadko wypełnionych kolumn dla każdego kraju. Warto również zauważyć, że wiele krajów, takich jak Bahamy, Dominika, Fidzi i wiele państw afrykańskich, nie ma żadnego odpowiednika kodu pocztowego, co jeszcze bardziej zwiększa rozrzedzenie danych w tabeli.

W zależności od wymagań aplikacji lepszym rozwiązaniem może być przechowywanie danych w formacie JSON. Podobnie jak XML, JSON ma częściowo ustrukturyzowany format, który pozwala na pomijanie nieistotnych informacji. Staje się coraz bardziej popularny, ponieważ jest bardzo lekkim formatem, zawierającym znacznie mniej znaczników niż XML.

Ze względu na efektywność języka T-SQL w wyszukiwaniu danych stosuję podczas modelowania tę samą zasadę zarówno dla JSON, jak i dla XML. Mianowicie, jeśli dane będą często zapisywane, a rzadko odczytywane, to JSON jest dobrym wyborem. Jeśli jednak dane są znacznie częściej odczytywane niż zapisywane, wolę przechowywać je w formie relacyjnej.

JSON ma również inne zastosowania, takie jak modelowanie domen danych, które w innym przypadku wymagałyby podziału między bazę relacyjną a NoSQL. Dzięki SQL Serverowi oba rodzaje danych można przechowywać w tym samym schemacie, co ułatwia komunikację z interfejsami REST API lub innymi systemami, które wymieniają dane w formacie JSON. Obecnie możliwe jest także przechowywanie i analizowanie w SQL Serverze danych dzienników opartych na JSON.

Zobaczmy zatem, jak uniknąć pomyłki polegającej na ignorowaniu JSON i wymodelować dane adresowe pracowników właśnie w tym formacie. Jeśliby nasze adresy miały się odnosić do wielu encji, takich jak biura czy lokalizacje, moglibyśmy i tak utworzyć osobną tabelę `Addresses`, żeby adresy w formacie JSON mogły być używane przez różne encje. Jesteśmy jednak przekonani, że tylko encja `Employees` będzie wymagać interakcji z danymi adresowymi, dlatego dodamy dodatkową kolumnę do naszej tabeli `Employees`, aby przechowywać te dane. Listing 3.21 przedstawia polecenie, które utworzy tę kolumnę.

Listing 3.21. Dodawanie kolumny `Address` do tabeli `Employees`

```
ALTER TABLE dbo.Employees
ADD Address NVARCHAR(MAX) ;
```

UWAGA Chwileczkę! Zamierzamy modelować dane w formacie JSON, więc dlaczego właśnie utworzyliśmy kolumnę typu NVARCHAR(MAX)? Odpowiedź jest taka, że w momencie pisania tej książki typ danych JSON jest dostępny do testów w Azure SQL Database, ale w przypadku SQL Servera działającego lokalnie lub w usłudze IaaS JSON nie ma własnego typu danych. Jest przechowywany jako VARCHAR lub NVARCHAR i indeksowany jak tekst. Dzięki temu JSON jest w pełni kompatybilny z każdą funkcją SQL Servera, która przetwarza tekst.

Najpierw przyjrzyjmy się strukturze dokumentu JSON, którego użyjemy do przechowywania adresów. Format JSON wykorzystuje proste pary nazwa:wartość ujęte w cudzysłowy. Zagnieżdżanie oznacza się nawiasami klamrowymi {}. Nawiasy kwadratowe [] oznaczają tablicę. Nasz dokument adresowy ma element główny EmployeeAddress i zawiera węzły EmployeeID oraz Address. Węzeł Address ma dalsze, zagnieżdżone węzły, które przechowują poszczególne wiersze adresu. W zależności od wymagań dla każdego adresu można dodawać kolejne wiersze lub zastępować istniejące węzły, co rozwiązuje problem rozrzedzenia danych. Oto przykładowy dokument:

```
{
  "EmployeeAddress": [
    {
      "EmployeeID": 1,
      "Address": {
        "Line1": "5331 Rexford Court",
        "City": "Montgomery",
        "State": "AL",
        "ZipCode": "36116"
      }
    }
  ]
}
```

Dokument ten można wygenerować za pomocą instrukcji SELECT z klauzulą FOR JSON. Dostępne są dwie opcje przetwarzania FOR JSON: AUTO i PATH. Tryb AUTO automatycznie formatuje dokument na podstawie kolejności kolumn i sekwencji łączenia tabel. Tryb PATH daje bardziej szczegółową kontrolę nad formatowaniem.

WSKAZÓWKA Klauzuli FOR JSON AUTO można używać tylko w odniesieniu do tabeli, więc teoretycznie nie da się zastosować jej w sposób, jaki planujemy. Można jednak łatwo obejść to ograniczenie, dodając klauzulę FROM do dowolnej tabeli. Osobiście często używam w tym celu sys.tables, ale nada się też każda inna tabela. Jeśli zastosujemy tę technikę, musimy dodać TOP 1 do klauzuli SELECT, aby uniknąć zwracania węzła dla każdego rekordu w tabeli.

Kwerenda przedstawiona na listingu 3.22 wygeneruje nasz przykładowy dokument JSON. Warto zauważyć, że używamy trybu PATH, więc każda kolumna ma alias określający nazwę węzła i jego pozycję w hierarchii JSON.

Listing 3.22. Generowanie dokumentu JSON

```

SELECT
    1 AS 'EmployeeID'
    , '5331 Rexford Court' AS 'Address.Line1'
    , 'Montgomery'        AS 'Address.City'
    , 'AL'                 AS 'Address.State'
    , '36116'              AS 'Address.ZipCode'
FOR JSON PATH, ROOT ('EmployeeAddress') ;

```

Skrypt z listingu 3.23 pokazuje, jak zaktualizować tabelę `Employees`, aby dodać adres pracownika, jeśli mamy dane adresowe w formacie JSON. Klauzula `WHERE` w instrukcji `UPDATE` wykorzystuje funkcję `JSON_VALUE()` do pobrania identyfikatora pracownika (`EmployeeID`) z dokumentu JSON. Pierwszym parametrem tej funkcji jest dokument JSON, w którym ma być przeprowadzone wyszukiwanie. Drugi parametr określa ścieżkę do węzła. Ścieżka JSON zawsze zaczyna się od `$.`, co reprezentuje cały dokument. Węzeł główny, `EmployeeAddress`, jest tablicą (oznaczoną nawiasami kwadratowymi), co oznacza, że musimy podać indeks tablicy, aby określić, dla którego rekordu chcemy zwrócić wartość. Jeśli nie podamy tego indeksu, otrzymamy wynik `NULL`, nawet jeśli istnieje tylko jeden rekord, jak w naszym przypadku.

Listing 3.23. Dodawanie danych adresowych do tabeli Employees

```

DECLARE @EmployeeAddress NVARCHAR(MAX) ;

SET @EmployeeAddress = (
    SELECT
        1 AS 'EmployeeID'
        , '5331 Rexford Court' AS 'Address.Line1'
        , 'Montgomery'        AS 'Address.City'
        , 'AL'                 AS 'Address.State'
        , '36116'              AS 'Address.ZipCode'
    FOR JSON PATH, ROOT ('EmployeeAddress')
) ;

UPDATE dbo.Employees
SET Address = @EmployeeAddress
WHERE EmployeeID =
    JSON_VALUE(@EmployeeAddress, '$.EmployeeAddress[0].EmployeeID') ;

```

Jeśli kiedykolwiek zajdzie potrzeba poszatkowania dokumentu JSON na format relacyjny, można skorzystać z funkcji `OPENJSON()`. Format i działanie `OPENJSON()` będą znane czytelnikom, którzy wcześniej zapoznali się z funkcją `OPENXML()` opisaną w tym rozdziale. Zaletą funkcji `OPENJSON()` jest jednak to, że nie trzeba przygotowywać dokumentu przed jej użyciem. Pozwala to zaoszczędzić zasoby systemowe.

Na listingu 3.24 używamy funkcji `OPENJSON()` do przetworzenia dokumentu JSON z adresami pracowników. Pierwszym parametrem przekazywanym do funkcji jest sam dokument JSON. Drugi parametr określa ścieżkę do najwyższego poziomu hierarchii, który chcemy przeanalizować. Pamiętaj, że `$` reprezentuje

cały dokument, a podobnie jak w przypadku funkcji `JSON_VALUE()`, podana ścieżka musi uwzględniać format tablicowy poprzez określenie indeksu tablicy. Klauzula `WITH` określa kolumny, które chcemy zwrócić z dokumentu, wraz z typami danych SQL, na które zostaną odwzorowane, oraz ścieżką do odpowiedniego węzła, gdy węzeł nie znajduje się na tym samym poziomie co wyrażenie ścieżki.

Listing 3.24. Szatkowanie danych JSON na wartości relacyjne

```
DECLARE @EmployeeAddress NVARCHAR(MAX) ;

SET @EmployeeAddress = (
    SELECT
        1 AS 'EmployeeID'
        , '5331 Rexford Court' AS 'Address.Line1'
        , 'Montgomery' AS 'Address.City'
        , 'AL' AS 'Address.State'
        , '36116' AS 'Address.ZipCode'
    FOR JSON PATH, ROOT ('EmployeeAddress')
) ;

SELECT *
FROM OPENJSON(@EmployeeAddress, '$.EmployeeAddress[0]')
WITH (
    EmployeeID SMALLINT,
    Line1 NVARCHAR(64) '$.Address.Line1',
    City NVARCHAR(64) '$.Address.City',
    [State] NCHAR(2) '$.Address.State',
    ZipCode NVARCHAR(10) '$.Address.ZipCode'
) ;
```

Jak widać, JSON może odgrywać istotną rolę w SQL Serverze, jeśli jest odpowiednio wykorzystywany. Obsługa formatu JSON w SQL Serverze pozwala na denormalizację danych i uproszczenie złożonych modeli, integrację z bazami NoSQL, a także uniknięcie przetwarzania danych przesyłanych do interfejsów API.

Podsumowanie

- Zawsze używaj najbardziej restrykcyjnego typu danych, który umożliwi przechowywanie wszystkich potencjalnie wymaganych wartości. Jest to szczególnie istotne w przypadku wartości całkowitych używanych w ograniczeniach kluczy lub wartości, które są indeksowane.
- Zawsze używaj łańcuchów znaków o stałej długości, aby zmniejszyć zużycie pamięci, gdy długość łańcuchów w kolumnie jest jednakowa. Natomiast gdy długość łańcuchów może się różnić, stosuj łańcuchy o zmiennej długości.
- Unikaj pisania własnego kodu i tworzenia samoreferencyjnych tabel podczas modelowania i budowania hierarchii. Zamiast tego wykorzystaj

typ danych HIERARCHYID, który oferuje wiele wbudowanych metod, znacznie ułatwiających pracę programisty.

- Gdy mamy do czynienia z danymi XML, nie powinniśmy czuć się zobowiązani do ich szatkowania na model relacyjny. Warto zawsze rozważyć wymagania aplikacji i odpowiednio wymodelować schemat danych. Może to oznaczać przechowywanie danych w natywnym formacie XML, ale nie musi.
- Nie obawiaj się formatu JSON. Istnieją przypadki, w których wykorzystanie danych JSON jest dobrym wyborem, pozwalającym uprościć nasze modele danych.

4 Projekt bazy danych

W tym rozdziale:

- Pomyłki projektowe użytkowników SQL Servera i dlaczego ważne jest ich unikanie
- Pomyłka polegająca na braku normalizacji bazy danych
- Pomyłki popełniane przy projektowaniu i tworzeniu kluczy

W tym rozdziale omówimy typowe pomyłki popełniane podczas projektowania baz danych. Błędy te mogą prowadzić do wielu problemów, w tym do niskiej wydajności kodu.

Błędy projektowe pojawiają się na najwcześniejszym etapie cyklu rozwoju oprogramowania, zanim jeszcze napisany zostanie jakikolwiek kod. Aby to zilustrować, przyjrzyjmy się firmie MagicChoc. Firma uważa, że jej procesy są zbyt rozdrobnione, dlatego zleciła napisanie nowej aplikacji, która połączy funkcje tradycyjnej sprzedaży i zaopatrzenia w jeden interfejs ze wspólnym zapleczem. W tym celu zarząd MagicChoc określił, że chce przechowywać następujące elementy danych:

- Sales Order Date (Data zamówienia sprzedaży),
- Sales Order Number (Numer zamówienia sprzedaży),
- Sales Person Name (Nazwisko sprzedawcy),
- Sales Person Email (Adres e-mail sprzedawcy),

- Sales Area Name (Nazwa obszaru sprzedaży),
- Sales Area Manager (Kierownik obszaru sprzedaży),
- Customer Company Name (Nazwa firmy klienta),
- Customer Contact Name (Nazwisko kontaktu u klienta),
- Customer Contact Email (Adres e-mail kontaktu u klienta),
- Customer Invoice Address (Adres klienta do faktury),
- Customer Delivery Addresses (Adresy dostawy),
- Sales Order Delivery Due Date (Termin dostawy zamówienia sprzedaży),
- Sales Order Delivery Actual Date (Rzeczywista data dostawy zamówienia sprzedaży),
- Sales Order Item (Pozycja zamówienia sprzedaży),
- Sales Order Quantities (Ilości w zamówieniu sprzedaży),
- Currier Used for Delivery (Kurier dostarczający zamówienie),
- Product Name (Nazwa produktu),
- Product Stock Level (Stan magazynowy produktu),
- Product Next Manufacture Date (Data następnej produkcji),
- Product Next Manufacture Quantity (Ilość do wyprodukowania),
- Back Order Manufacturing ID (Identyfikator zamówienia oczekującego),
- Product Type Name (Nazwa typu produktu),
- Product Type Description (Opis typu produktu),
- Product Category Name (Nazwa kategorii produktu),
- Product Category Description (Opis kategorii produktu),
- Product Subcategory Name (Nazwa podkategorii produktu),
- Product Subcategory Description (Opis podkategorii produktu),
- Supplier Name (Nazwa dostawcy),
- Supplier Contact Name (Nazwisko kontaktu u dostawcy),
- Supplier Contact Email (Adres e-mail kontaktu u dostawcy),
- Supplier Address (Adres dostawcy),
- Purchase Order Date (Data zamówienia kupna),
- Purchase Order Number (Numer zamówienia kupna),
- Purchase Order Items (Pozycje zamówienia kupna),
- Purchase Order Quantities (Ilości w zamówieniu kupna).

W tym rozdziale zaprojektujemy strukturę tabel niezbędnych do przechowywania tych danych. W trakcie tego procesu przyjrzymy się, jak unikać typowych błędów projektowych. Szczególną uwagę poświęcimy problemom, które mogą wynikać z braku normalizacji bazy danych. Następnie omówimy kwestie wydajnościowe związane z niewłaściwym doбором klucza głównego. Na koniec przeanalizujemy konsekwencje pominięcia ograniczeń kluczy obcych w projekcie bazy danych.

4.1. Numer 11 — brak normalizacji

Wielokrotnie pytałem programistów: „Czy baza danych jest znormalizowana?”. Odpowiedź niezmiennie brzmiała: „Tak!”. Niestety w wielu przypadkach programiści byli stosunkowo niedoświadczeni w projektowaniu baz danych i błędnie interpretowali moje pytanie, myśląc, że pytam o to, czy zaprojektowali bazę danych do przetwarzania transakcyjnego (OLTP), a nie do działania w charakterze hurtowni danych. W rzeczywistości, zamiast normalizować swoją bazę danych, po prostu kierowali się własnym osądem przy tworzeniu jej schematu.

Aby zaprojektować schemat naszej bazy danych, stworzymy diagram związków encji (ang. *Entity Relationship Diagram*, ERD). Jest to graficzna reprezentacja, która opisuje *encje* (obiekty w modelu danych) oraz ich *atrybuty* (cechy opisujące te obiekty). ERD pokazuje również, jak poszczególne obiekty są ze sobą powiązane w ramach modelu.

WSKAZÓWKA ERD często jest diagramem na poziomie kodu w modelu C4, gdy używa się C4 do dokumentowania projektu bazy danych. W związku z tym w tym podrozdziale będziemy pośrednio tworzyć diagram kodu C4.

W kolejnych punktach przyjrzymy się najpierw niewłaściwemu sposobowi projektowania schematu bazy danych, czyli podejściu „intuicyjnemu”. W tym podejściu będziemy polegać wyłącznie na naszym doświadczeniu w organizacji danych. Ponieważ każdy ma nie tylko inny poziom, ale również po prostu rodzaj doświadczenia, stosowanie tej nieustrukturyzowanej metody może prowadzić do wielu problemów. Przeanalizujemy niektóre z najczęstszych trudności, jakie mogą się pojawić, takie jak wartości nieatomowe i duplikacja danych.

Na koniec przyjrzymy się, jak unikać błędów w projektowaniu baz danych dzięki technice normalizacji. Omówimy to ustrukturyzowane podejście do modelowania baz danych, które zostało opracowane przez Edgara Codd’a w latach 70. XX wieku i do dziś nie straciło na aktualności. Jest to zestaw metodycznych kroków mających wyeliminować redundancję danych. Stosując to podejście, będziemy przestrzegać ściśle określonych reguł, których celem jest uniknięcie duplikacji danych i maksymalizacja efektywności schematu bazy danych. Poruszymy również temat weryfikacji poprawności normalizacji przy użyciu diagramu związków encji oraz przyjrzymy się koncepcji generalizacji danych.

4.1.1. Projektowanie schematu na podstawie własnego osądu

Zaprojektujmy schemat nowej bazy danych sprzedaży i zaopatrzenia, o którą poprosiła firma MagicChoc. Firma chce modelować 35 elementów danych, więc dla ułatwienia podzielimy je na sekcje. Zaczniemy od danych dotyczących zamówień sprzedaży, a konkretnie od następujących elementów:

- Sales Order Date (Data zamówienia sprzedaży),
- Sales Order Number (Numer zamówienia sprzedaży),
- Sales Person Name (Nazwisko sprzedawcy),
- Sales Person Email (Adres e-mail sprzedawcy),
- Sales Area Name (Nazwa obszaru sprzedaży),
- Sales Area Manager (Kierownik obszaru sprzedaży),
- Sales Order Delivery Due Date (Termin dostawy zamówienia sprzedaży),
- Sales Order Delivery Actual Date (Rzeczywista data dostawy zamówienia sprzedaży),
- Sales Order Item (Pozycja zamówienia sprzedaży),
- Sales Order Quantities (Ilości w zamówieniu sprzedaży),
- Currier Used for Delivery (Kurier dostarczający zamówienie).

Atrybut Sales Order Number (Numer zamówienia sprzedaży) będzie unikatowy, więc wydaje się dobrym kandydatem na klucz dla naszej encji. Problematiczna jest relacja między Sales Order Item (pozycją zamówienia sprzedaży) a Sales Order Quantities (ilościami w zamówieniu sprzedaży). Zauważamy tu wyraźny związek „jeden do wielu”. Jest to relacja między encjami, w której jeden rekord w jednej encji może być powiązany z wieloma rekordami w drugiej. W tym przypadku w danym zamówieniu może być więcej niż jeden zamówiony produkt, dlatego decydujemy się na wydzielenie pozycji zamówienia do osobnej encji. Biorąc pod uwagę wymóg przechowywania ogólnych informacji o produktach, uznajemy również za sensowne pominięcie szczegółów produktu i utworzenie jedynie klucza do encji Products, którą zaprojektujemy wkrótce.

Klucze

W relacyjnych bazach danych tabele łączy się ze sobą za pomocą kluczy. Istnieją dwa podstawowe rodzaje kluczy — klucz główny i klucz obcy. Klucz główny służy do jednoznacznej identyfikacji wiersza w tabeli. Klucz obcy natomiast jest używany do wskazywania na wartość klucza głównego lub unikatowe ograniczenie w innej tabeli.

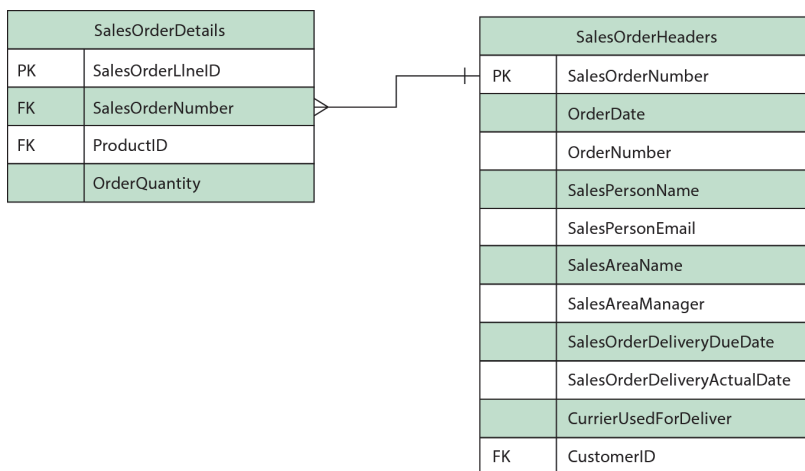
Wyobraźmy sobie tabelę z danymi pracowników. Musimy zapisać informację o biurze, w którym pracuje każdy pracownik. Moglibyśmy dodać kolumnę Office do tabeli Employees, ale oznaczałoby to wielokrotne powtarzanie tych samych nazw biur

(a potencjalnie także adresów, numerów telefonów itp.). Dlatego lepszym rozwiązaniem jest przeniesienie informacji dotyczących biur do osobnej tabeli `Offices`. W tej tabeli możemy użyć klucza głównego, nazwijmy go `OfficeID`, do jednoznacznej identyfikacji każdego biura. Następnie w tabeli `Employees` możemy zastosować klucz obcy (często, choć nie zawsze, o tej samej nazwie co klucz główny). Ten klucz obcy będzie odnosił się do wartości w kolumnie klucza głównego tabeli `Offices`, gwarantując, że w kolumnie klucza obcego będą wprowadzane tylko istniejące wartości klucza głównego. To zapewnia tzw. *referencyjną integralność* danych.

Jeśli klucz ma znaczenie biznesowe, jak na przykład `SocialSecurityNumber`, nazywamy go *kluczem naturalnym*. Natomiast jeśli klucz nie ma znaczenia biznesowego, jak w przypadku arbitralnie przydzielanego, rosnącego numeru (na przykład `EmployeeID`), mówimy o *kluczu sztucznym*.

Klucz naturalny może się składać z wielu kolumn. Jeśli klucz obejmuje kilka kolumn, nazywamy go *kluczem złożonym*. Warto ograniczać stosowanie kluczy złożonych, jeśli to możliwe.

Podobnie, przyjmując, że klienci mogą składać wiele zamówień, a zgodnie z wymaganiami musimy przechowywać dane klientów, decydujemy się utworzyć w tabeli `Orders` klucz, który połączy ją z tabelą `Customers`, gdy ta zostanie utworzona. Na koniec zauważamy, że nie ma dobrego kandydata na klucz główny, więc tworzymy klucz sztuczny. W rezultacie otrzymujemy początek diagramu ERD, który można zobaczyć na rysunku 4.1.



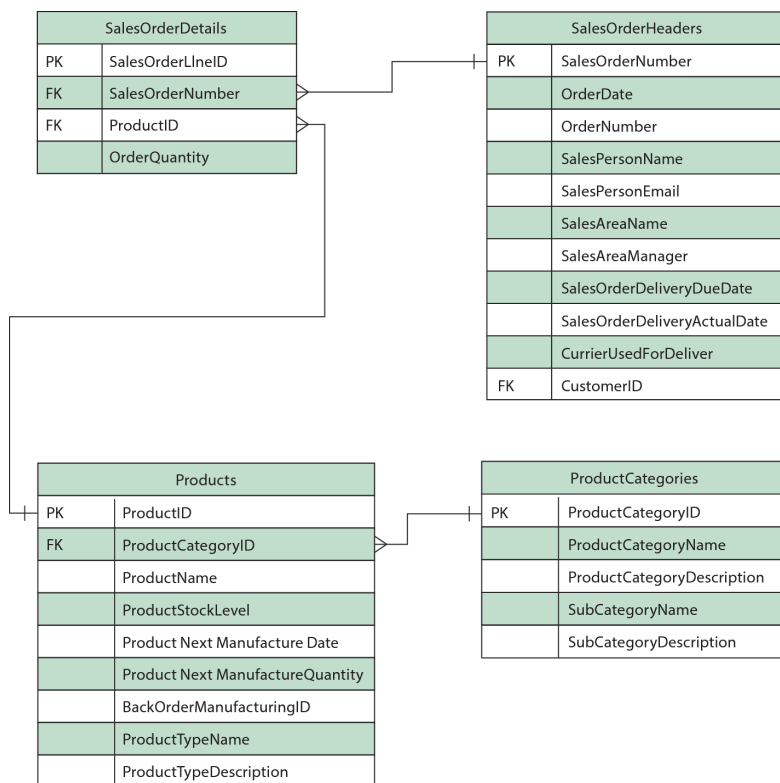
Rysunek 4.1. Diagram ERD dla zamówień sprzedaży

Następnie przyjrzymy się wymaganiom dotyczącym danych produktów. Musimy przechowywać następujące informacje:

- Product Name (Nazwa produktu),
- Product Stock Level (Stan magazynowy produktu),
- Product Next Manufacture Date (Data następnej produkcji),

- Product Next Manufacture Quantity (Ilość do wyprodukowania),
- Back Order Manufacturing ID (Identyfikator zamówienia oczekującego),
- Product Type Name (Nazwa typu produktu),
- Product Type Description (Opis typu produktu),
- Product Category Name (Nazwa kategorii produktu),
- Product Category Description (Opis kategorii produktu),
- Product Subcategory Name (Nazwa podkategorii produktu),
- Product Subcategory Description (Opis podkategorii produktu).

Postanawiamy utworzyć tabelę Products, a ponieważ nie mamy oczywistego klucza naturalnego, tworzymy klucz sztuczny, który będzie powiązany z atrybutem ProductID w encji SalesOrderDetails (Szczegóły zamówienia). Zauważamy również, że może istnieć relacja „jeden do wielu” między produktami a kategoriami produktów, dlatego postanawiamy utworzyć osobną encję ProductCategories (Kategorie produktów), którą połączymy z encją Products za pomocą kolejnego klucza sztucznego. Po połączeniu tych encji z encjami zamówień sprzedaży otrzymamy diagram ERD przedstawiony na rysunku 4.2.



Rysunek 4.2. Diagram ERD z dodaną encją produktów

Następnie zbudujemy encję Customers, która będzie zawierać następujące atrybuty:

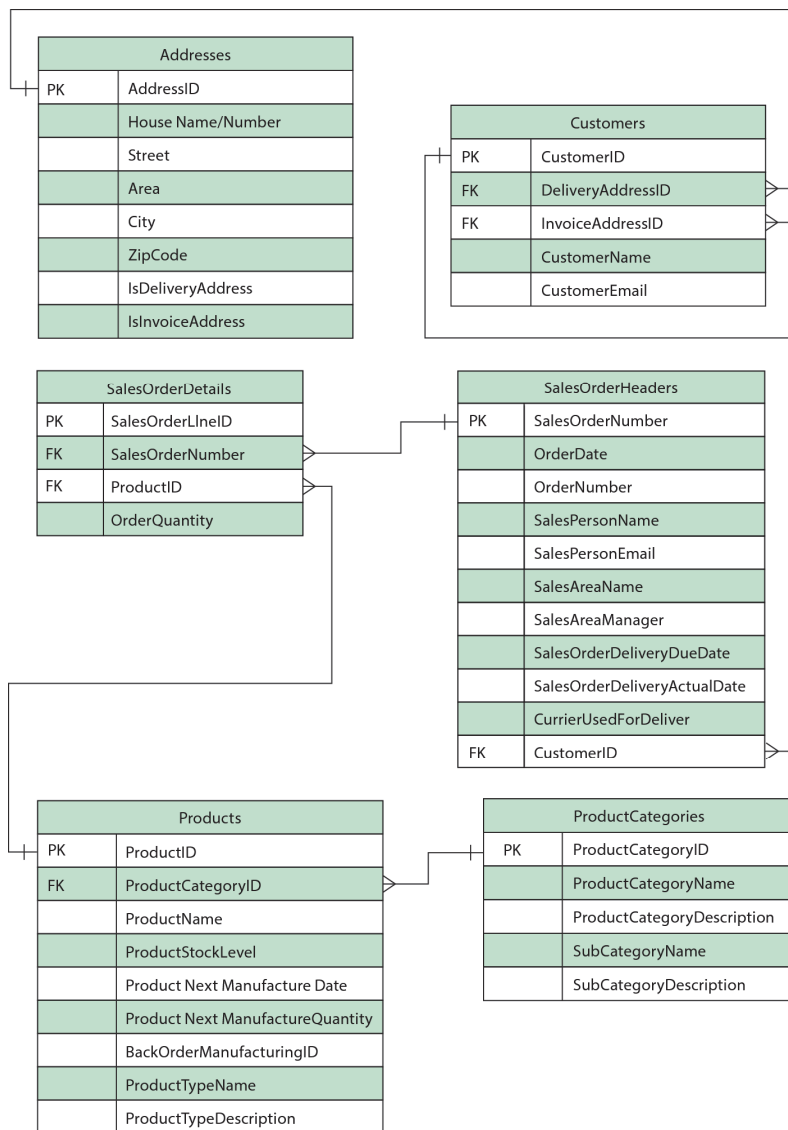
- Customer Company Name (Nazwa firmy klienta),
- Customer Contact Name (Nazwisko kontaktu u klienta),
- Customer Contact Email (Adres e-mail kontaktu u klienta),
- Customer Invoice Address (Adres klienta do faktury),
- Customer Delivery Addresses (Adresy dostawy).

Aby uniknąć szerokiej, rozrzedzonej tabeli, postanawiamy przenieść adresy do osobnej encji. Zauważamy też, że możemy utworzyć jedną encję Addresses, która będzie powiązana zarówno z adresem do faktury, jak i adresem dostawy do klienta. Pozwala to uniknąć duplikowania danych, gdy adres do faktury jest taki sam jak adres dostawy. Osiągamy to, dodając flagi dla każdego typu adresu. Dla obu encji utworzymy klucze sztuczne. Po dodaniu tych encji do naszego istniejącego projektu otrzymujemy diagram ERD przedstawiony na rysunku 4.3.

Na koniec dodajmy encje dla dostawców i zamówień kupna. Atrybuty danych, które musimy wymodelować dla tych encji, to:

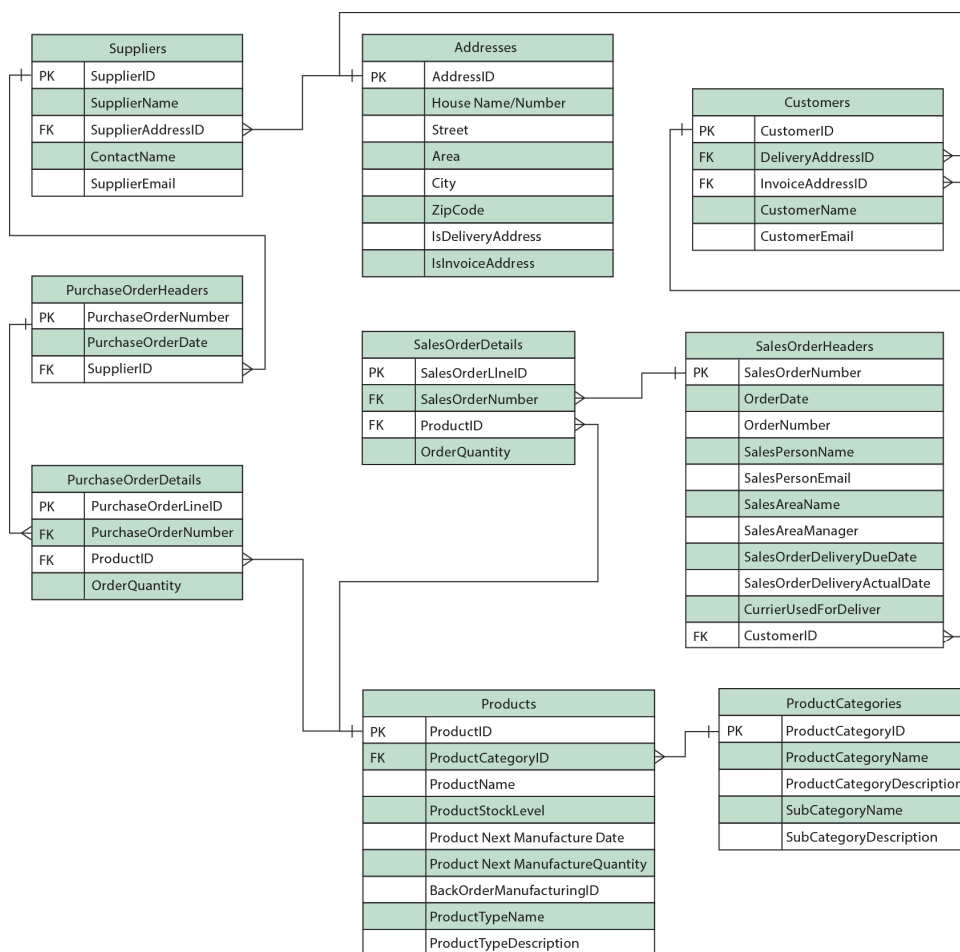
- Supplier Name (Nazwa dostawcy),
- Supplier Contact Name (Nazwisko kontaktu u dostawcy),
- Supplier Contact Email (Adres e-mail kontaktu u dostawcy),
- Supplier Address (Adres dostawcy),
- Purchase Order Date (Data zamówienia kupna),
- Purchase Order Number (Numer zamówienia kupna),
- Purchase Order Items (Pozycje zamówienia kupna),
- Purchase Order Quantities (Ilości w zamówieniu kupna).

Postanawiamy wymodelować dostawców i zamówienia kupna w sposób podobny do tego, jak wymodelowaliśmy klientów i zamówienia sprzedaży. Stworzymy encję Suppliers, którą powiążemy z istniejącą encją Addresses w celu przechowywania adresów dostawców. Podzielimy zamówienia zakupu na dwie encje: jedną dla nagłówek zamówień (PurchaseOrderHeaders) i jedną dla szczegółów zamówień (PurchaseOrderDetails). Pozwoli to na zamawianie wielu pozycji w ramach jednego zamówienia bez konieczności powielania danych zamówienia dla każdej pozycji. Decydujemy się na utworzenie klucza sztucznego w encjach Suppliers i PurchaseOrderDetails, ale w encji PurchaseOrderHeaders jako klucz naturalny wykorzystamy numer zamówienia (SalesOrderNumber).



Rysunek 4.3. Diagram ERD z dodaną encją klientów

Jeśli dodamy te pozostałe elementy do naszego istniejącego projektu, otrzymamy diagram ERD przedstawiony na rysunku 4.4. W ten sposób mamy gotowy projekt schematu bazy danych. Nie było to takie trudne, prawda? Ale na czym polega problem z takim podejściem? Otóż w schemacie znajduje się kilka błędów, które sprawią, że korzystanie z naszej bazy danych będzie bardzo utrudnione i może powodować problemy z wydajnością. W następnej sekcji omówimy niektóre z tych pomysłów i ich konsekwencje.



Rysunek 4.4. Diagram ERD z encjami dostawców i zamówień zakupu

4.1.2. Problemy z naszym schematem bazy danych

Na pierwszy rzut oka nasz schemat bazy danych może wydawać się całkiem sensowny, ale jeśli przyjrzymy się bliżej niektórym decyzjom projektowym, zauważymy potencjalne problemy. Zaczniemy od analizy nazw klientów i dostawców, co jest jedną z najbardziej zauważalnych kwestii.

Zarówno atrybut `SupplierName` encji `Suppliers`, jak i atrybut `CustomerName` encji `Customers` będą przechowywać imiona i nazwiska. Może to powodować różne problemy. Wyobraź sobie, że nasza aplikacja sprzedażowa ma funkcję korespondencji seryjnej i chcemy rozpoczynać listy do klientów zwrotem „Dear Robert”, gdzie Robert jest imieniem klienta. Aby pobrać to imię, musielibyśmy wykonać kwerendę w rodzaju:

```
SELECT SUBSTRING(CustomerName, 0, CHARINDEX(' ', CustomerName, 1))
FROM Customers ;
```

Utrudnia to pisanie i czytanie kodu, a w dodatku jest mniej wydajne niż zwykle wybranie kolumny `FirstName` z tabeli. Dlatego zawsze powinniśmy się upewnić, że nasze wartości są atomowe. *Wartość atomowa* to taka, której nie da się dalej podzielić bez utraty znaczenia. Na przykład imię i nazwisko Peter A. Carter można podzielić na trzy wartości atomowe — imię, inicjał drugiego imienia i nazwisko. Argument, że można je podzielić na 12 znaków, nie ma sensu, ponieważ te pojedyncze znaki nie miałyby żadnego kontekstowego znaczenia.

WSKAZÓWKA Wyjątkiem od tej reguły jest sytuacja, w której po znormalizowaniu schematu celowo podejmujesz decyzję o denormalizacji danych. Możesz na przykład zdecydować się na przechowywanie danych adresowych w formacie JSON, co omówiliśmy w rozdziale 3. Inne powody unikania normalizacji to m.in. praca z danymi analitycznymi lub tworzenie tabel przygotowawczych.

Kolejny problem tkwi w naszej encji `ProductCategories`. Encja ta zawiera zarówno kategorie, jak i podkategorie produktów. Jednak w ramach jednej kategorii może istnieć wiele podkategorii. Oznacza to, że będziemy musieli powielać nazwy i opisy kategorii produktów dla każdej podkategorii. Duplikowanie danych marnuje miejsce, co sprawia, że tabela staje się większa, a przez to wolniejsza w odczycie, zajmuje więcej pamięci i zmniejsza wydajność operacji na indeksach. Większy problem pojawia się jednak przy wstawianiu i aktualizacji danych. Te operacje staną się znacznie mniej wydajne, ponieważ będziemy musieli aktualizować informacje o kategoriach produktów w wielu wierszach. Moglibyśmy uniknąć tego problemu, rozbijając dane na więcej encji i łącząc je za pomocą relacji klucz główny-obcy.

Postanowiliśmy przechowywać nazwę obszaru sprzedaży (`SalesAreaName`) w encji `SalesOrderHeader`. Oznacza to, że nazwa ta będzie powielana wielokrotnie, ponieważ może dotyczyć wielu wierszy. Wiąże się to również z koniecznością powtarzania informacji o kierowniku obszaru sprzedaży (`SalesAreaManager`) w każdym rekordzie. Gdyby szczegóły obszaru sprzedaży zostały wyodrębnione do osobnej encji i połączone z powrotem za pomocą relacji klucz główny-obcy, można by tego uniknąć.

Podobny problem mamy z nazwiskiem kontaktu u dostawcy (`SupplierContact ↪ Name`). Nie dość, że wartość ta nie jest atomowa, to w zależności od reguł biznesowych może sprawić, że będziemy duplikować dane. Co, jeśli dwóch naszych dostawców należy do tej samej osoby? A co, jeśli w przyszłości pojawi się potrzeba przechowywania dodatkowych informacji kontaktowych, takich jak numer telefonu konkretnej osoby? To tylko pogłębi problem i sprawi, że nasza encja dostawców stanie się bardzo szeroka.

Rodzi to wiele pytań. Jak moglibyśmy uniknąć tych problemów projektowych? Czy są jeszcze jakieś inne kwestie związane z naszym projektem? W jaki sposób

możemy to w ogóle przetestować, nie budując całego schematu i nie wypełniając go danymi?

Odpowiedzią na wszystkie te pytania jest zastosowanie procesu normalizacji do projektowania schematu naszej bazy danych. W kolejnym punkcie wyjaśnię, jak wykorzystać ten proces w praktyce.

UWAGA Dodatkowy problem, którego nie zauważylibyśmy bez zrozumienia danych i reguł biznesowych, polega na tym, że numery zamówień sprzedaży będą miały format ABC1234D-E12. To spowoduje, że klucz podstawowy będzie bardzo szeroki. Temat ten zostanie omówiony dalej w tym rozdziale.

4.1.3. Projektowanie schematu bazy danych z wykorzystaniem normalizacji

Jak wspomniano wcześniej w tym rozdziale, normalizacja to formalny proces modelowania danych, który został opracowany przez Edgara Codda w latach 70. XX wieku i dobrze przeszedł próbę czasu. Istnieje dziesięć postaci normalnych, czyli etapów w tym procesie, które z biegiem lat były dodawane do tej metody. Najnowsza postać normalna, zwana „Essential Tuple Normal Form”, została wprowadzona stosunkowo niedawno, bo w 2012 roku. Mówiąc ściślej, postaci normalne to:

- pierwsza postać normalna (1NF),
- druga postać normalna (2NF),
- trzecia postać normalna (3NF),
- postać normalna z kluczem elementarnym (EKNF),
- postać normalna Boyce’a-Codda (BCNF lub postać 3.5),
- czwarta postać normalna (4NF),
- postać normalna z krotką istotną (ETNF),
- piąta postać normalna (5NF),
- postać normalna z kluczem domenowym (DKNF),
- szósta postać normalna (6PN).

W zdecydowanej większości przypadków nie ma potrzeby wykraczania poza trzecią postać normalną (3NF). Warto również pamiętać, że nadmierna normalizacja schematu może prowadzić do innych problemów. Dlatego w tym rozdziale skupimy się na przekształceniu naszego schematu danych do postaci 3NF.

Aby przekształcić dane do pierwszej postaci normalnej (1NF), należy zadbać o to, by wszystkie atrybuty były atomowe i miały unikatowe nazwy, a ich kolejność nie miała znaczenia. Trzeba też przenieść powtarzające się grupy atrybutów do osobnej encji oraz upewnić się, że kolejność wierszy nie jest istotna.

Podczas przekształcania danych do drugiej postaci normalnej (2NF) upewniamy się, że wszystkie atrybuty są zależne od wszystkich składników złożonego klucza głównego. Jeśli tak nie jest, przenosimy je do nowej encji. Następnie, przekształcając dane do trzeciej postaci normalnej (3NF), dbamy o to, by atrybuty nie były zależne od atrybutów niebędących częścią klucza głównego. Jeśli taka zależność występuje, ponownie wskazuje to, że należy je przenieść do nowej encji.

Wskazówki dotyczące normalizacji

Chociaż istnieją różne narzędzia do normalizacji danych, uważam, że najlepszym narzędziem do tego procesu jest zwykły arkusz kalkulacyjny. W przypadku mniejszych zbiorów danych, takich jak ten omawiany tutaj, stosuję metodę jednej kolumny na każdą postać normalną, zostawiając puste wiersze między encjami. Pozwala mi to porównywać modele jeden obok drugiego i analizować logikę, którą stosuję.

Kolejną ważną wskazówką dotyczącą normalizacji jest zrozumienie danych. Oznacza to współpracę z analitykiem biznesowym lub osobą odpowiedzialną za aplikację w celu poznania znaczenia biznesowego i reguł biznesowych stojących za każdym atrybutem. Zaniedbanie tego etapu nieuchronnie doprowadzi do błędnych założeń, które wpłyną na model.

Zrozumienie tych zagadnień pomoże Tobie lub Twojemu analitykowi biznesowemu opracować kolejny ważny dokument, zwany *słownikiem danych*. Jest on zazwyczaj tworzony w arkuszu kalkulacyjnym i zawiera szczegółowe informacje, takie jak:

- nazwa atrybutu,
- opis/znaczenie atrybutu,
- reguły biznesowe, takie jak oczekiwane wartości, unikatowość,
- przykładowe dane,
- typ danych,
- zależności nadrzędne,
- pochodzenie danych.

Pochodzenie danych jest istotne w złożonych aplikacjach bazodanowych. Śledzi ono ścieżkę, jaką przebył element danych, aby dotrzeć do tabeli, w której obecnie się znajduje. Na przykład, gdybyśmy mieli bazę danych używaną do marketingu internetowego, moglibyśmy przechowywać w niej identyfikator pliku cookie. Pochodzenie w tym przypadku mogłoby być takie:

1. Identyfikator śledzenia (w pliku CSV z danymi o wyświetleniach dostarczonym przez dostawcę ciasteczek).
2. Identyfikator śledzenia (w tabeli Cookies w schemacie przygotowawczym).
3. Identyfikator pliku cookie (w tabeli Cookies w schemacie marketingowym).

Na koniec — nie bój się popełniać błędów. Zbuduj swój model, przetestuj go, a jeśli coś poszło nie tak, po prostu napraw to i spróbuj ponownie.

Zanim zaczniemy, podzielimy nasze atrybuty na trzy ogólne typy danych: Sales (sprzedaż), Purchasing (zaopatrzenie) oraz Products (produkty) (które odnoszą się zarówno do sprzedaży, jak i zaopatrzenia). W ten sposób otrzymamy nasze dane wyjściowe w postaci nieznormalizowanej, co przedstawiono poniżej:

Sales

Sales Order Date
Sales Order Number
Sales Person Name
Sales Person Email
Sales Area Name
Sales Area Manager
Customer Company Name
Customer Contact Name
Customer Contact Email
Customer Invoice Address
Customer Delivery Addresses
Sales Order Delivery Due Date
Sales Order Delivery Actual Date
Sales Order Item
Sales Order Quantities
Carrier Used for Delivery

Purchasing

Supplier Name
Supplier Contact Name
Supplier Contact Email
Supplier Address
Purchase Order Date
Purchase Order Number
Purchase Order Items
Purchase Order Quantities

Products

Product Name
Product Stock Level
Product Next Manufacture Date
Product Next Manufacture Quantity
Back Order Manufacturing ID
Product Type Name
Product Type Description
Product Category Name
Product Category Description
Product Subcategory Name
Product Subcategory Description

PIERWSZA POSTAĆ NORMALNA

Aby relacja była w pierwszej postaci normalnej (1NF), musi spełniać następujące warunki:

- Wszystkie atrybuty są atomowe.
- Każdy atrybut ma unikatową nazwę.
- Kolejność atrybutów nie ma znaczenia.
- Powtarzające się grupy są usuwane.
- Wszystkie wiersze muszą być unikatowe (musi istnieć klucz).

UWAGA W procesie normalizacji relacja jest odpowiednikiem encji w diagramie ERD lub tabeli w bazie danych.

W naszym przykładzie każdy atrybut ma już unikatową nazwę, a kolejność atrybutów nie ma znaczenia, więc mamy dobry punkt wyjścia. Musimy jednak rozbić niektóre atrybuty na mniejsze, atomowe części. Musimy też sprawdzić dane pod kątem powtarzających się grup, czyli zestawów kolumn, które się powtarzają. Ponadto musimy znaleźć *kandydata na klucz*, czyli minimalny zestaw kolumn tworzący *nadklucz*. Nadklucz to klucz jednoznacznie identyfikujący każdy wiersz, a minimalny jest wtedy, kiedy usunięcie którejkolwiek kolumny z tego klucza uniemożliwiłoby jednoznaczną identyfikację wszystkich wierszy.

Najpierw zidentyfikujmy klucze, które będą służyć do jednoznacznego identyfikowania wierszy w każdej relacji. Dane dotyczące sprzedaży można jednoznacznie zidentyfikować za pomocą numeru zamówienia sprzedaży. W związku z tym nie ma potrzeby stosowania *klucza złożonego*, czyli klucza składającego się z wielu atrybutów. Jeśli klucz składa się z pojedynczego atrybutu, nazywamy go *atrybutem podstawowym*.

Dane dotyczące zaopatrzenia można jednoznacznie zidentyfikować za pomocą numeru zamówienia kupna. W związku z tym numer zamówienia sam w sobie stanowi nadklucz i możemy go wykorzystać jako kandydata na pojedynczy klucz.

Danych dotyczących produktów nie można jednoznacznie zidentyfikować wyłącznie na podstawie nazwy produktu, ponieważ niektóre części zamawiane od dostawców mają takie same nazwy jak produkty sprzedawane klientom. Dlatego do unikatowej identyfikacji każdego rekordu musimy użyć zarówno nazwy produktu, jak i nazwy typu produktu. Te dwa atrybuty będą zatem tworzyć nasz kandydujący klucz.

Proces ten pomógł nam wykryć pierwszy problem z naszym modelem danych. Nasze dane dotyczące zarówno sprzedaży, jak i kupna zawierają nazwy produktów (pozycje zamówienia sprzedaży oraz pozycje zamówienia kupna). Problem w tym, że do jednoznacznej identyfikacji produktu potrzebujemy nazwy produktu i nazwy typu produktu. Dodajmy zatem nazwę typu produktu do każdej z tych encji. Powinniśmy również dodać zależności do każdej relacji, która łączy się z encją produktów. Przy okazji zmieńmy nazwy atrybutów Sales Order Items oraz Purchase Order Items na Product Name, aby wyraźniej pokazać zależność.

Następnie powinniśmy przyjrzeć się wartościom nieatomowym. W naszych nieuporządkowanych danych znajdują się wartości nieatomowe, które musimy rozbić na mniejsze części:

- **Sales Person Name** — rozdzielimy ją na Sales Person First Name i Sales Person Last Name.
- **Sales Area Manager** — rozdzielimy ją na Sales Area Manager First Name i Sales Area Manager Last Name.
- **Customer Contact Name** — rozdzielimy ją na Customer Contact First Name i Customer Contact Last Name.

- **Customer Invoice Address** — rozdzielimy ją na następujące atrybuty:
 - Invoice Address Street (Adres do faktury — ulica),
 - Invoice Address Area (Adres do faktury — obszar),
 - Invoice Address City (Adres do faktury — miasto),
 - Invoice Address Zip Code (Adres do faktury — kod pocztowy),
 - Invoice Address Country (Adres do faktury — kraj).
- **Customer Delivery Address** — rozdzielimy ją na następujące atrybuty:
 - Delivery Address Street (Adres dostawy — ulica),
 - Delivery Address Area (Adres dostawy — obszar),
 - Delivery Address City (Adres dostawy — miasto),
 - Delivery Address Zip Code (Adres dostawy — kod pocztowy),
 - Delivery Address Country (Adres dostawy — kraj).
- **Product Name i Product Type Name (w relacji Sales)**. To bardziej skomplikowana kwestia. Ten atrybut będzie przechowywać wiele produktów zamówionych w ramach jednego zamówienia, ale nie możemy go rozbić na wiele osobnych atrybutów, ponieważ nie ma określonej liczby pozycji, które można umieścić w pojedynczym zamówieniu. Dlatego wyodrębnimy te informacje do osobnej relacji i przeniesiemy do niej numer zamówienia sprzedaży (Sales Order Number), który wybraliśmy jako klucz. Dzięki temu nowa relacja będzie mogła łączyć się z pierwotną relacją. Przeniesiemy do niej również ilości w zamówieniu sprzedaży (Sales Order Quantities, które skrócimy do Quantity), z którymi w przeciwnym razie byłby ten sam problem.
- **Supplier Contact Name** — rozdzielimy ją na Supplier Contact First Name i Supplier Contact Last Name.
- **Supplier Address** — rozdzielimy ją na następujące atrybuty:
 - Supplier Address Street (Adres dostawcy — ulica),
 - Supplier Address Area (Adres dostawcy — obszar),
 - Supplier Address City (Adres dostawcy — miasto),
 - Supplier Address Zip Code (Adres dostawcy — kod pocztowy),
 - Supplier Address Country (Adres dostawcy — kraj).
- **Product Name i Product Type Name (w relacji Purchasing)**. Podobnie jak w przypadku relacji Sales, musimy przenieść te atrybuty do nowej relacji i powiązać je z numerem zamówienia kupna, aby połączyć encje. Dodatkowo przeniesiemy atrybut ilości w zamówieniu kupna (Purchase Order Quantities), skracając jego nazwę do Quantity.

Na koniec musimy się przyjrzeć powtarzającym się grupom. Być może zauważyłeś już na liście punktowanej, że atrybuty tworzące adres do faktury są dokładnym powtórzeniem atrybutów tworzących adres dostawy w relacji Sales. Dlatego powinniśmy przenieść oba te adresy do nowej relacji, którą nazwiemy

Addresses. Do nowej relacji dodamy dodatkowy atrybut, dla którego kluczem kandydującym będą ulica i kod pocztowy. Następnie wprowadzimy te informacje z powrotem do relacji Sales dwukrotnie — raz dla adresu do faktury i ponownie dla adresu dostawy.

WSKAZÓWKA Chociaż nie jest to wyraźnie określone w regułach normalizacji, w tym momencie sensowne byłoby dodanie adresu dostawcy (Supplier Address) do relacji Addresses. W ten sposób mielibyśmy jedną tabelę przechowującą wszystkie adresy w naszej bazie danych. Minimalizuje to duplikację danych i, w przypadku adresów, oznacza, że pojedyncza walidacja adresu sprawia, że jest on gotowy do wykorzystania w dowolnej encji.

Generalizacja danych

Poprzez dodanie wszystkich trzech typów adresów do jednej relacji zastosowaliśmy technikę modelowania danych zwaną generalizacją. Polega ona na zebraniu atrybutów z grupy powiązanych relacji lub encji i utworzeniu nowej, bardziej ogólnej encji.

Bardziej zaawansowanym rodzajem *generalizacji* jest tworzenie typów nadrzędnych i podrzędnych. Stosując tę technikę, możemy na przykład zdecydować się na uogólnienie sprzedawców, kontaktów u klienta, kontaktów u dostawcy i kierowników obszarów sprzedaży. Wszystkie ich wspólne atrybuty, takie jak imię, nazwisko i adres e-mail, zostałyby przeniesione do encji People (Osoby).

Niektóre typy osób mogą mieć unikatowe atrybuty. Na przykład dla sprzedawcy konieczne może być przechowywanie dodatkowych informacji, takich jak cel sprzedaży czy numer pracownika. W takim przypadku możemy utworzyć osobną encję dla sprzedawców, która będzie zawierała te specyficzne atrybuty i będzie powiązana z główną encją People.

W tym przypadku encja People staje się typem nadrzędnym. Sales Person to podtyp encji People. Dobry przykład takiego modelowania można znaleźć w bazie danych AdventureWorks, która jest przykładową bazą danych SQL Server. W tej bazie tabela Employees jest podtypem nadrzędnego typu Person.

Bazę danych AdventureWorks można pobrać pod adresem <https://mng.bz/oOod>.

Poniższy schemat ilustruje, w jaki sposób nasze dane zostały przekształcone do postaci 1NF, zgodnie z wcześniej omówionymi zagadnieniami:

```

Sales
Sales Order Date
PK Sales Order Number
Sales Person First Name
Sales Person Last Name
Sales Person Email
Sales Area Name
Sales Area Manager First Name
Sales Area Manager Last Name
Customer Company Name
Customer Contact First Name
Customer Contact Last Name
Customer Contact Email
FK Invoice Address Street
  
```

FK Invoice Address Zip Code
FK Delivery Address Street
FK Delivery Address Zip Code
Sales Order Delivery Due Date
Sales Order Delivery Actual Date
Currier Used for Delivery

Sales Order Details

PK FK Product Name
PK FK Product Type Name
Quantity
PK FK Sales Order Number

Address

PK Street
Area
City
PK Zip Code

Purchasing

Supplier Name
Supplier Contact First Name
Supplier Contact Last Name
Supplier Contact Email
FK Supplier Address Street
FK Supplier Address Zip Code
Purchase Order Date
PK Purchase Order Number

Purchase Order Details

PK FK Product Name
PK FK Product Type Name
Quantity
PK FK Purchase Order Number

Products

PK Product Name
Product Stock Level
Product Next Manufacture Date
Product Next Manufacture Quantity
Back Order Manufacturing ID
PK Product Type Name
Product Type Description
Product Category Name
Product Category Description
Product Subcategory Name
Product Subcategory Description

Znakomicie. Nasze relacje zaczynają coraz bardziej przypominać schemat bazy danych. Jednak modelowanie bazy danych tylko do 1NF jest zwykle uważane za złe rozwiązanie. Dlatego przejdźmy dalej i przyjrzymy się, jak możemy przekształcić naszą bazę danych do 2NF.

DRUGA POSTAĆ NORMALNA

Aby relacje spełniały warunki 2NF, muszą być zgodne z następującymi regułami:

- Relacje muszą być w postaci 1NF.
- Wszystkie atrybuty muszą być funkcjonalnie zależne od całego klucza.

Wiemy już, że nasze relacje są w 1NF, ale jak zapewnić zależność funkcjonalną od całego klucza? Warto zauważyć, że zasada ta dotyczy tylko relacji ze złożonym kluczem głównym. Z definicji, jeśli klucz naturalny składa się z jednego atrybutu podstawowego, to wszystkie pozostałe atrybuty muszą być od niego zależne.

W naszym przykładzie mamy tylko jeden atrybut, który nie jest zależny od wszystkich atrybutów w kluczu złożonym. Jest to atrybut Product Type Description (Opis typu produktu) w relacji Product. Atrybut ten zależy wyłącznie od atrybutu Product Type Name (Nazwa typu produktu). Wyodrębnijmy zatem atrybuty Product Type Description i Product Type Name do osobnej relacji, w której Product Type Name stanie się atrybutem podstawowym. Jednocześnie dodamy Product Type Name z powrotem do relacji Product jako klucz obcy.

W związku z tym nasze dane w postaci 2NF przedstawiają się następująco:

Sales

| | |
|----|----------------------------------|
| | Sales Order Date |
| PK | Sales Order Number |
| | Sales Person First Name |
| | Sales Person Last Name |
| | Sales Person Email |
| | Sales Area Name |
| | Sales Area Manager First Name |
| | Sales Area Manager Last Name |
| | Customer Company Name |
| | Customer Contact First Name |
| | Customer Contact Last Name |
| | Customer Contact Email |
| FK | Invoice Address Street |
| FK | Invoice Address Zip Code |
| FK | Delivery Address Street |
| FK | Delivery Address Zip Code |
| | Sales Order Delivery Due Date |
| | Sales Order Delivery Actual Date |
| | Currier Used for Delivery |

Sales Order Details

| | |
|-------|--------------------|
| PK FK | Product Name |
| PK FK | Product Type Name |
| | Quantity |
| PK FK | Sales Order Number |

Address

| | |
|----|----------|
| PK | Street |
| | Area |
| | City |
| PK | Zip Code |

Purchasing

Supplier Name
 Supplier Contact First Name
 Supplier Contact Last Name
 Supplier Contact Email
 FK Supplier Address Street
 FK Supplier Address Zip Code
 Purchase Order Date
 PK Purchase Order Number

Purchase Order Details

PK FK Product Name
 PK FK Product Type Name
 Quantity
 PK FK Purchase Order Number

Products

PK Product Name
 Product Stock Level
 Product Next Manufacture Date
 Product Next Manufacture Quantity
 Back Order Manufacturing ID
 PK FK Product Type Name
 Product Category Name
 Product Category Description
 Product Subcategory Name
 Product Subcategory Description

Product Types

PK Product Type Name
 Product Type Description

TRZECIA POSTAĆ NORMALNA

Na koniec musimy przekształcić nasze relacje do postaci 3NF. Zasady trzeciej postaci normalnej określają:

- Relacje muszą być w drugiej postaci normalnej (2NF).
- Wszystkie atrybuty muszą być nieprzechodnio zależne od klucza.

Nasze relacje są już w 2NF, co jest dobrym punktem wyjścia. Teraz musimy zadbać o *nieprzechodniość zależności*, co oznacza, że atrybuty nie powinny zależeć od żadnych atrybutów niebędących kluczami w ramach danej relacji.

Analizując nasze dane, zauważymy wiele przykładów atrybutów, które są bardziej zależne od atrybutów niepodstawowych niż od klucza kandydującego. Pierwszy przykład znajduje się w relacji Sales: są to atrybuty Sales Person First Name i Sales Person Last Name (Imię sprzedawcy i Nazwisko sprzedawcy). Można założyć, że wartość Sales Person Email (Adres e-mail sprzedawcy) będzie unikatowa, co oznacza, że mogłaby służyć do identyfikacji każdej unikatowej kombinacji tych atrybutów. W związku z tym przeniesiemy te trzy atrybuty do nowej relacji Sales Person (Sprzedawca), gdzie Sales Person Email będzie atrybutem

podstawowym, a następnie wprowadzimy z powrotem Sales Person Email do relacji Sales jako klucz obcy.

Relacja Sales zawiera również atrybuty Sales Area Manager First Name i Sales Area Manager Last Name (Imię kierownika obszaru sprzedaży i Nazwisko kierownika obszaru sprzedaży). Oba te atrybuty można jednoznacznie zidentyfikować za pomocą atrybutu Sales Area Name (Nazwa obszaru sprzedaży). Dlatego przenieśliśmy je do osobnej relacji Sales Areas (Obszary sprzedaży), zachowując atrybut Sales Area Name (Nazwa obszaru sprzedaży) jako klucz obcy w relacji Sales.

Atrybuty Customer Contact First Name, Customer Contact Last Name, Customer Email, wraz z kluczami adresu dostawy i kluczami adresu do faktury, mogą być unikatowo identyfikowane przez atrybut Customer Company Name. Dlatego przenieśliśmy te dane do osobnej relacji Customers, którą połączymy z powrotem za pomocą klucza Customer Company Name (Nazwa firmy klienta). Jest to ciekawy przykład, ponieważ po utworzeniu tej nowej relacji można zauważyć, że imię i nazwisko kontaktu u klienta można również identyfikować za pomocą adresu e-mail klienta. W związku z tym powinniśmy przenieść te atrybuty do jeszcze jednej relacji o nazwie Customer Contact (Kontakt u klienta), która będzie łączona z relacją Customers za pomocą atrybutu Customer Contact Email jako klucza.

W podobny sposób, analizując relację Purchasing, zauważymy, że klucze Supplier Contact First Name, Supplier Contact Last Name i Supplier Address mogą być jednoznacznie identyfikowane za pomocą atrybutu Supplier Name (Nazwa dostawcy), dlatego powinny zostać przeniesione do relacji Suppliers (Dostawcy). Ale atrybuty Supplier Contact First Name, Supplier Contact Last Name i Supplier Contact Email powinny zostać wówczas przeniesione do relacji Supplier Contact (Kontakt u dostawcy) z kluczem w postaci atrybutu Supplier Contact Email (E-mail kontaktu u dostawcy).

WSKAZÓWKA Encje Supplier Contacts, Customer Contacts i Sales Person są dobrymi kandydatami do generalizacji. Gdybyśmy chcieli uogólnić te dane, moglibyśmy utworzyć pojedynczą encję Contacts, która zawierałaby wszystkie typy kontaktów. Aby ułatwić filtrowanie, moglibyśmy dodać do tej encji atrybut Contact Type (Typ kontaktu). Ta encja łączyłaby się następnie z encjami Suppliers, Customers i Sales.

W relacji Products atrybuty Product Category Name, Product Category Description i Product Subcategory Description mogą być jednoznacznie identyfikowane przez Product Subcategory Name. Dlatego przeniesiemy te atrybuty do osobnej relacji Product Subcategories. Ponadto atrybut Product Category Description powinien zostać przeniesiony do relacji Product Categories, w której kluczem będzie atrybut Product Category Name.

Wreszcie atrybuty Product Next Manufacture Date i Product Next Manufacture Quantity można powiązać z atrybutem Back Order Manufacturing ID, dlatego przeniesiemy te atrybuty do nowej relacji Back Orders (Zamówienia oczekujące).

Nasze dane powinny teraz być w trzeciej postaci normalnej, jak pokazano niżej:

Sales

Sales Order Date
PK Sales Order Number
FK Sales Person Email
FK Sales Area Name
FK Customer Company Name
Sales Order Delivery Due Date
Sales Order Delivery Actual Date
Currier Used for Delivery

Sales Person

Sales Person First Name
Sales Person Last Name
PK Sales Person Email

Sales Areas

PK Sales Area Name
Sales Area Manager First Name
Sales Area Manager Last Name

Customer

PK Customer Company Name
FK Customer Contact Email
FK Invoice Address Street
FK Invoice Address Zip Code
FK Delivery Address Street
FK Delivery Address Zip Code

Customer Contact

Customer Contact First Name
Customer Contact Last Name
PK Customer Contact Email

Sales Order Details

PK FK Product Name
PK FK Product Type Name
Quantity
PK FK Sales Order Number

Address

PK Street
Area
City
PK Zip Code

Purchasing

FK Supplier Name
Purchase Order Date
PK Purchase Order Number

Suppliers

PK Supplier Name
FK Supplier Contact Email
FK Supplier Address Street
FK Supplier Address Zip Code

Supplier Contact

Supplier Contact First Name
 Supplier Contact Last Name
 PK Supplier Contact Email

Purchase Order Details

PK FK Product Name
 PK FK Product Type Name
 Quantity
 PK FK Purchase Order Number

Products

PK Product Name
 Product Stock Level
 FK Back Order Manufacturing ID
 PK FK Product Type Name
 FK Product Subcategory Name

Product Types

PK Product Type Name
 Product Type Description

Product SubCategories

FK Product Category Name
 PK Product Subcategory Name
 Product Subcategory Description

Product Categories

PK Product Category Name
 Product Category Description

Back Orders

Product Next Manufacture Date
 Product Next Manufacture Quantity
 PK Back Order Manufacturing ID

Teraz powinniśmy poświęcić chwilę na uporządkowanie kilku kwestii. Pierwszą są nazwy naszych relacji. Po przeprowadzonym modelowaniu nazwa relacji Sales nie oddaje już właściwie reprezentowanych przez nią danych. Zmieńmy ją na Sales Order Header (Nagłówek zamówienia sprzedaży). Podobnie, Purchasing nie odzwierciedla już odpowiednio swoich atrybutów, więc powinniśmy zmienić tę nazwę na Purchase Order Header (Nagłówek zamówienia kupna).

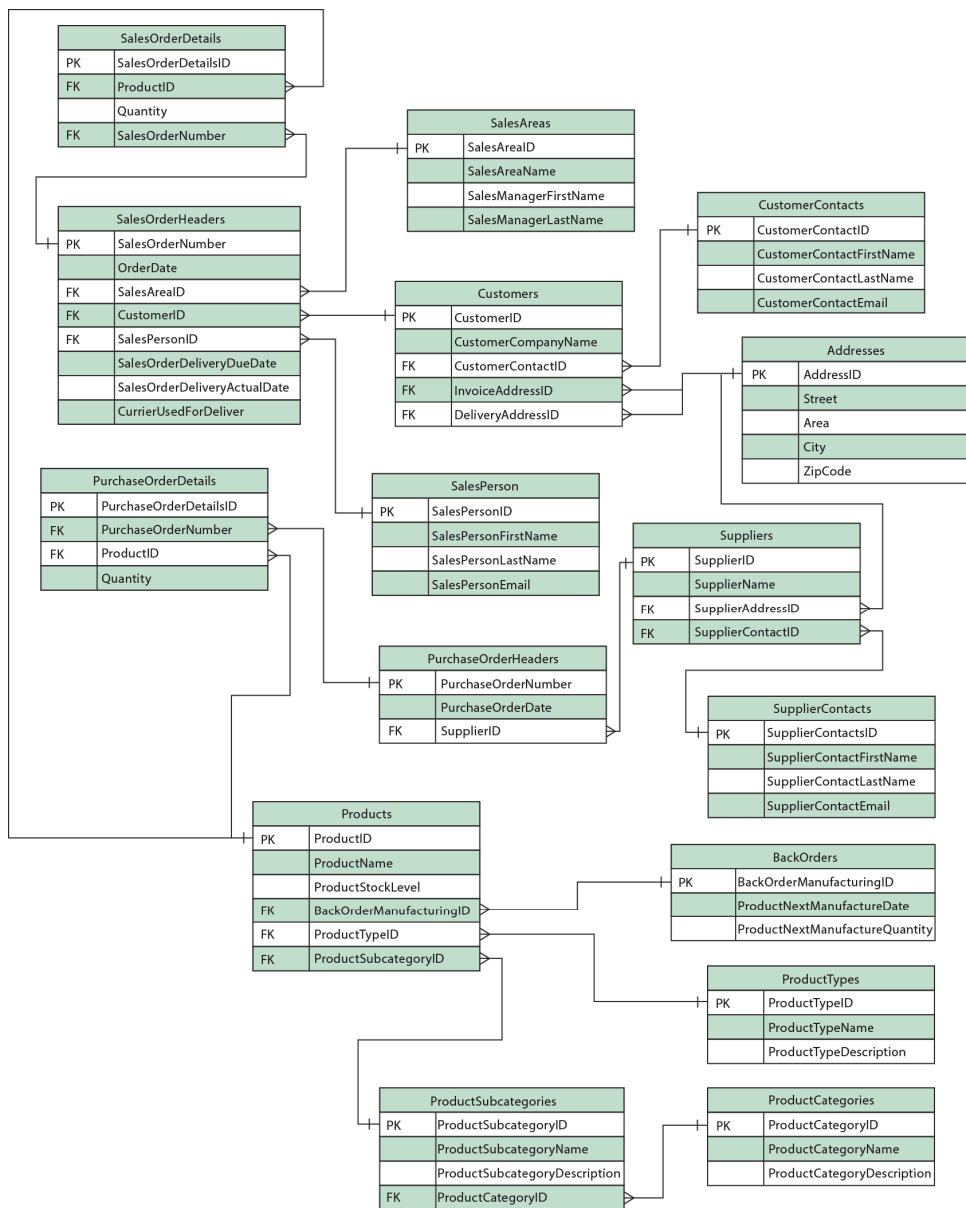
Szerokie klucze główne są niekorzystne. Nie będę tu wchodzić w szczegóły, ponieważ zajmiemy się tym podczas omawiania następnej pomyłki, ale obecnie wszystkie nasze klucze są *kluczami naturalnymi*, czyli takimi, które mają znaczenie biznesowe. Jeśli mamy klucze złożone, powinniśmy zastąpić je *kluczami sztucznymi*, czyli takimi, które przechowują arbitralną wartość bez znaczenia biznesowego — zwykle inkrementowaną liczbę. Powinniśmy również zastąpić klucze oparte na tekście kluczami sztucznymi. To również sprawia, że klucze stają się węższe.

Mając to na uwadze, wprowadzimy następujące zmiany:

- Encja Sales Person będzie mieć nowy klucz o nazwie Sales Person ID.
- Encja Sales Areas będzie mieć nowy klucz o nazwie Sales Area ID.
- Encja Customers będzie mieć nowy klucz o nazwie Customer ID.
- Encja Customer Contacts będzie mieć nowy klucz o nazwie Customer Contact ID.
- Encja Sales Order Details będzie mieć nowy klucz o nazwie Sales Order Details ID.
- Encja Addresses będzie mieć nowy klucz o nazwie Address ID.
- Encja Suppliers będzie mieć nowy klucz o nazwie Supplier ID.
- Encja Supplier Contacts będzie mieć nowy klucz o nazwie Supplier Contact ID.
- Encja Purchase Order Details będzie mieć nowy klucz o nazwie Purchase Order Details ID.
- Encja Products będzie mieć nowy klucz o nazwie Product ID.
- Encja Product Types będzie mieć nowy klucz o nazwie Product Type ID.
- Encja Product Subcategories będzie mieć nowy klucz o nazwie Product Subcategory ID.
- Encja Product Categories będzie mieć nowy klucz o nazwie Product Category ID.

Na koniec powinniśmy usunąć wszystkie spacje z nazw relacji i atrybutów. Przyda się to, kiedy będziemy tworzyć nazwy tabel i kolumn w bazie danych. Gdyby w nazwach relacji i atrybutów występowały jakieś znaki specjalne lub słowa zastrzeżone (czyli słowa używane przez SQL Server, takie jak „SELECT” czy „table”), również powinniśmy rozważyć ich zmianę. Pominięcie tego kroku oznaczałoby konieczność ograniczania identyfikatorów podczas odwoływania się do obiektów SQL Servera. Polega to na umieszczaniu nazw w nawiasach kwadratowych. Na przykład zamiast `SELECT MyColumn FROM MyTable` musielibyśmy napisać `SELECT [MyColumn] FROM [MyTable]`. Stosowanie ograniczonych identyfikatorów jest uważane za złą praktykę, ponieważ zaśmieca kod i świadczy o nieprzestrzeganiu standardowych zasad i konwencji nazewnictwa.

Nasz projekt jest gotowy, ale jak możemy go przetestować? Nie ma jednoznacznej odpowiedzi na to pytanie, ale dobrym punktem wyjścia jest utworzenie diagramu związków encji (ang. *Entity Relationship Diagram*, ERD). Analizując ERD, powinniśmy zwrócić uwagę, czy wszystkie encje, które muszą być połączone, są powiązane relacją „jeden do wielu”. Jeśli znajdziemy jakiegokolwiek połączenia „wiele do wielu” lub „jeden do jednego”, prawdopodobnie będzie to oznaczać błąd w modelu. Przyjrzyjmy się zatem, jak wygląda nasz model po zbudowaniu ERD. Diagram ten przedstawiono na rysunku 4.5.



Rysunek 4.5. Diagram ERD znormalizowanego projektu

Ostatnim zadaniem jest napisanie skryptu, który utworzy obiekty w naszej bazie danych. Skrypt przedstawiony na listingu 4.1 wygeneruje obiekty, z których część wykorzystamy w późniejszych przykładach.

UWAGA W skrypcie celowo pozostawiono pewien błąd. Tabela `PurchaseOrderDetails` zawiera kolumnę `PurchaseOrderNumber`, ale nie utworzono klucza obcego łączącego ją z tabelą `PurchaseOrderHeaders`. Konsekwencje tej pomyłki omówimy dalej w tym rozdziale.

Listing 4.1. Tworzenie obiektów bazy danych

```

CREATE DATABASE MagicChoc ;
GO

USE MagicChoc ;
GO

CREATE TABLE dbo.BackOrders (
    BackOrderManufacturingID      INT      NOT NULL
        IDENTITY PRIMARY KEY,
    ProductNextManufactureDate    DATE     NOT NULL,
    ProductNextManufactureQuantity INT     NOT NULL
) ;

CREATE TABLE dbo.ProductCategories (
    ProductCategoryID             SMALLINT      NOT NULL
        IDENTITY PRIMARY KEY,
    ProductCategoryName           NVARCHAR(64)   NOT NULL,
    ProductCategoryDescription    NVARCHAR(256)  NULL
) ;

CREATE TABLE dbo.ProductSubcategories (
    ProductSubcategoryID          SMALLINT      NOT NULL
        IDENTITY PRIMARY KEY,
    ProductCategoryID             SMALLINT      NOT NULL
        REFERENCES dbo.ProductCategories(ProductCategoryID),
    ProductSubcategoryName        NVARCHAR(64)   NOT NULL,
    ProductSubcategoryDescription NVARCHAR(256)  NULL
) ;

CREATE TABLE dbo.ProductTypes (
    ProductTypeID                 SMALLINT      NOT NULL
        IDENTITY PRIMARY KEY,
    ProductTypeName               NVARCHAR(64)   NOT NULL,
    ProductTypeDescription        NVARCHAR(256)  NOT NULL
) ;

CREATE TABLE dbo.Products (
    ProductID                     INT            NOT NULL
        IDENTITY PRIMARY KEY,
    ProductName                   NVARCHAR(64)   NOT NULL,
    ProductStockLevel             INT            NOT NULL,
    BackOrderManufacturingID      INT            NULL
        REFERENCES dbo.BackOrders(BackOrderManufacturingID),
    ProductTypeID                 SMALLINT      NOT NULL
        REFERENCES dbo.ProductTypes(ProductTypeID),
    ProductSubcategoryID          SMALLINT      NOT NULL
        REFERENCES dbo.ProductSubcategories(ProductSubcategoryID)
) ;

CREATE TABLE dbo.Addresses (
    AddressID                     INT            NOT NULL
        IDENTITY PRIMARY KEY,
    Street                        NVARCHAR(128)  NOT NULL,
    Area                          NVARCHAR(64)   NULL,

```

```

        City          NVARCHAR(64)    NOT NULL,
        ZipCode       NVARCHAR(10)    NOT NULL
    ) ;
CREATE TABLE dbo.SupplierContacts (
    SupplierContactID INT              NOT NULL
        IDENTITY PRIMARY KEY,
    SupplierContactFirstName NVARCHAR(32) NOT NULL,
    SupplierContactLastName NVARCHAR(32) NOT NULL,
    SupplierContactEmail    NVARCHAR(256) NOT NULL
) ;

CREATE TABLE dbo.Suppliers (
    SupplierID INT              NOT NULL
        IDENTITY PRIMARY KEY,
    SupplierName NVARCHAR(32)   NOT NULL,
    SupplierContactID INT       NOT NULL
        REFERENCES dbo.SupplierContacts(SupplierContactID),
    SupplierAddressID INT       NOT NULL
        REFERENCES dbo.Addresses(AddressID)
) ;

CREATE TABLE dbo.PurchaseOrderHeaders (
    PurchaseOrderNumber INT NOT NULL PRIMARY KEY,
    SupplierID INT NOT NULL
        REFERENCES dbo.Suppliers(SupplierID),
    PurchaseOrderDate DATE NOT NULL
) ;

CREATE TABLE dbo.PurchaseOrderDetails (
    PurchaseOrderDetailsID INT NOT NULL
        IDENTITY PRIMARY KEY,
    ProductID INT NOT NULL
        REFERENCES dbo.Products(ProductID),
    Quantity INT NOT NULL,
    PurchaseOrderNumber INT NOT NULL
) ;

```

W tej kolumnie brakuje ograniczenia klucza obcego, który powinien wiązać ją z tabelą PurchaseOrderHeader.

```

CREATE TABLE dbo.CustomerContacts (
    CustomerContactID INT              NOT NULL
        IDENTITY PRIMARY KEY,
    CustomerContactFirstName NVARCHAR(32) NOT NULL,
    CustomerContactLastName NVARCHAR(32) NOT NULL,
    CustomerContactEmail    NVARCHAR(256) NOT NULL
) ;

CREATE TABLE dbo.Customers (
    CustomerID INT              NOT NULL
        IDENTITY PRIMARY KEY,
    CustomerCompanyName NVARCHAR(32) NOT NULL,
    CustomerContactID INT NOT NULL
        REFERENCES dbo.CustomerContacts(CustomerContactID),
    InvoiceAddressID INT NOT NULL
        REFERENCES dbo.Addresses(AddressID),
    DeliveryAddressID INT NOT NULL
        REFERENCES dbo.Addresses(AddressID)
) ;

```

```

CREATE TABLE dbo.SalesAreas (
    SalesAreaID          SMALLINT          NOT NULL
        IDENTITY        PRIMARY KEY,
    SalesAreaName         NVARCHAR(32)      NOT NULL,
    SalesAreaManagerFirstName NVARCHAR(32)  NOT NULL,
    SalesAreaManagerLastName NVARCHAR(32)  NOT NULL
) ;

CREATE TABLE dbo.SalesPersons (
    SalesPersonID        SMALLINT          NOT NULL
        IDENTITY        PRIMARY KEY,
    SalesPersonFirstName NVARCHAR(32)      NOT NULL,
    SalesPersonLastName  NVARCHAR(32)      NOT NULL,
    SalesPersonEmail     NVARCHAR(256)     NOT NULL
) ;

CREATE TABLE dbo.SalesOrderHeaders (
    SalesOrderNumber      NCHAR(12)        NOT NULL    PRIMARY KEY,
    SalesOrderDate        DATE              NOT NULL,
    SalesPersonID         SMALLINT          NOT NULL
        REFERENCES dbo.SalesPersons(SalesPersonID),
    SalesAreaID           SMALLINT          NOT NULL
        REFERENCES dbo.SalesAreas(SalesAreaID),
    CustomerID            INT               NOT NULL
        REFERENCES dbo.Customers(CustomerID),
    SalesOrderDeliveryDueDate DATE          NOT NULL,
    SalesOrderDeliveryActualDate DATE       NULL,
    CarrierUsedforDelivery NVARCHAR(32)     NOT NULL
) ;

CREATE TABLE dbo.SalesOrderDetails (
    SalesOrderDetailsID   INT              NOT NULL    IDENTITY    PRIMARY KEY,
    ProductID             INT              NOT NULL
        REFERENCES dbo.Products(ProductID),
    Quantity              INT              NOT NULL,
    SalesOrderNumber      NCHAR(12)        NOT NULL
        REFERENCES dbo.SalesOrderHeaders(SalesOrderNumber)
) ;
GO

```

Modelowanie danych z wykorzystaniem normalizacji może początkowo wydawać się skomplikowane, ale po kilku próbach zaczyna stawać się naturalne. Z pewnością ma wiele zalet w porównaniu z podejściem opartym na intuicji i jest znacznie mniej podatne na błędy. Pozwala nam zastosować metodę, która sprawdziła się w praktyce i wytrzymała próbę czasu.

4.2. Numer 12 — używanie szerokiego klucza podstawowego

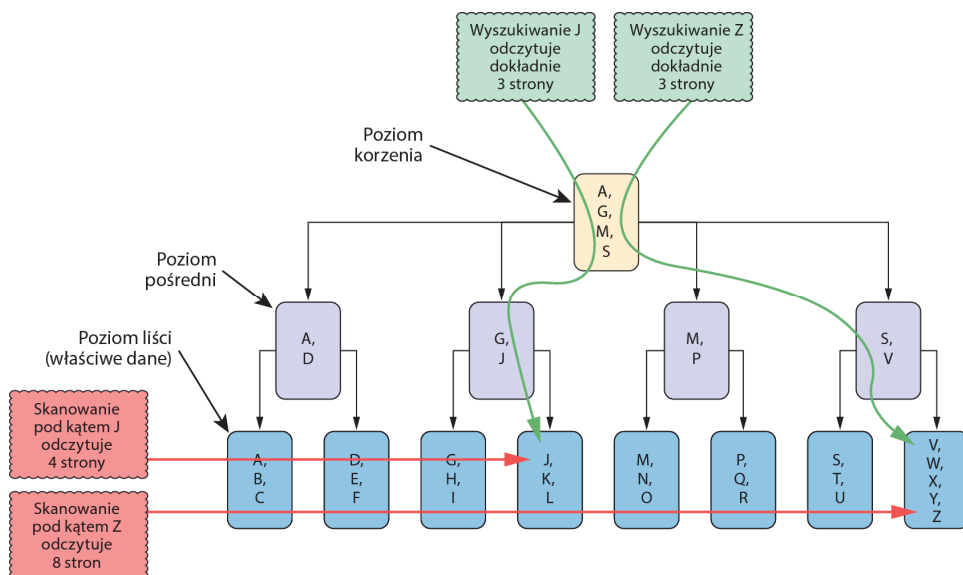
Jak wspomniano w poprzednim podrozdziale, wybraliśmy `SalesOrderNumber` jako klucz główny tabeli `SalesOrderHeaders`, ale formatem numerów zamówień jest `ABC1234D-E12`. Oznacza to, że nasza kolumna ma typ danych `NCHAR(12)`. W rezultacie dla każdego wiersza kolumna zajmuje 24 bajty pamięci, w przeciwieństwie do 4 bajtów, które zajmowałby typ `INT`. Nawet `BIGINT` zużywałby tylko 8 bajtów pamięci.

W rozdziale 3. omówiliśmy korzyści płynące z używania jak najmniejszych typów danych, ale te dane zawierają litery. W zależności od reguł biznesowych moglibyśmy zastosować typ `CHAR(12)`, ale nawet to zajmie 12 bajtów pamięci na każdy wiersz — co więc możemy zrobić? I jakie znaczenie ma fakt, że jest to kolumna klucza głównego?

Aby odpowiedzieć na te pytania, zastanówmy się nad strategią indeksowania tabeli `SalesOrderHeaders`. Kolumna klucza głównego domyślnie będzie miała *indeks klastrowy*. Indeks klastrowy organizuje dane w strukturę *B-drzewa*. Składa się ona z wielu poziomów stron indeksowych, które zawierają wskaźniki do stron indeksowych na niższych poziomach struktury. Zawsze istnieje jeden poziom główny, składający się z pojedynczej strony, znanej jako mapa alokacji indeksu (ang. *Index Allocation Map*, IAM). W zależności od rozmiaru tabeli może występować zero lub więcej poziomów pośrednich. Najniższy poziom (liście) indeksu klastrowego to właściwe strony danych tabeli.

WSKAZÓWKA Pełne omówienie indeksów znajdziesz w rozdziale 11.

Ponieważ dane w indeksie są uporządkowane, rekordy można odnaleźć bardzo szybko za pomocą operacji zwanej wyszukiwaniem w *indeksie klastrowym*. W ramach tej operacji przemierzane są poziomy indeksu, aby zlokalizować rekord. Jest to przeciwieństwo *skanowania indeksu klastrowego*, czyli operacji, w której trzeba przeszukać każdą stronę poziomu liści indeksu aż do znalezienia wymaganego rekordu. Operacje te, wraz z przykładową strukturą B-drzewa, zostały zilustrowane na rysunku 4.6. Warto zauważyć, że podczas wyszukiwania pojedynczego wiersza maksymalna liczba stron odczytanych w operacji wyszukiwania jest równa liczbie poziomów B-drzewa. Natomiast maksymalna liczba stron odczytanych podczas skanowania indeksu jest równa liczbie wszystkich stron na poziomie liści indeksu, a tym samym tabeli. Jeśli operacja może zwrócić wiele wierszy, wyszukiwania używa się w celu znalezienia punktu początkowego do odczytu poziomu liści.



Rysunek 4.6. Wyszukiwanie i skanowanie indeksu klastrowego w postaci B-drzewa

WSKAZÓWKA Ponieważ indeks klastrowy porządkuje strony danych samej tabeli, może istnieć tylko jeden taki indeks dla danej tabeli. Tabela bez indeksu klastrowego nazywana jest *stertą*. Na stercie rekordy są przechowywane w nieokreślonej kolejności. Sterty często wykorzystuje się w tabelach przygotowawczych, do których wstawia się duże ilości tymczasowych danych, gdzie kolejność nie ma znaczenia, ponieważ może to zoptymalizować wydajność.

Pracownicy firmy poinformowali nas, że często wyszukują zamówienia sprzedaży, filtrując je według daty. Dlatego chcemy zoptymalizować te kwerendy, tworząc indeks nieklastrowy na kolumnie `SalesOrderDate`. Indeks nieklastrowy tworzy strukturę B-drzewa bardzo podobną do indeksu klastrowego. Różnica polega na tym, że zamiast zawierać rzeczywiste dane na poziomie liści, indeks nieklastrowy zawiera wskaźniki do danych w indeksie klastrowym lub na stercie. Ponieważ poziom liści to tylko wskaźniki, oznacza to, że rzeczywiste dane nie są uporządkowane. Dzięki temu możemy mieć wiele indeksów nieklastrowych w jednej tabeli. Wskaźniki do indeksu klastrowego składają się z klucza indeksu klastrowego. Oznacza to, że szeroki klucz klastrowy jest powielany we wszystkich indeksach nieklastrowych.

UWAGA Jeśli indeks nieklastrowy zawiera kolumny typu `INCLUDE`, to oprócz wskaźnika do indeksu lub sterty w indeksie będą przechowywane rzeczywiste wartości tych kolumn.

Ponadto firma poinformowała nas, że kwerendy dotyczące zamówień sprzedaży zazwyczaj będą zawierać informacje o obszarze sprzedaży i dane klienta. Te informacje są przechowywane w różnych tabelach, dlatego warto utworzyć

indeksy nieklastrowe dla kolumn SalesAreaID i CustomerID, które są kluczami obcymi tych tabel. Tworzenie indeksów nieklastrowych na kluczach obcych powoduje, że klucze te są uporządkowane w taki sam sposób jak ich odpowiedniki (klucze główne) w tabelach, z którymi łączymy dane. Dzięki temu operacje łączenia mogą być wykonywane bardziej efektywnie.

Aby poznać konsekwencje posiadania szerokiego klucza głównego, najpierw dodajmy trochę danych do odpowiednich tabel. Skrypt z listingu 4.2 dodaje niewielką ilość danych do tych tabel, a następnie tworzy indeksy nieklastrowe w tabeli SalesOrderHeaders. Nie ma potrzeby tworzenia indeksu klastrowego, ponieważ został on automatycznie utworzony podczas definiowania klucza głównego dla tej tabeli.

Listing 4.2. Dodawanie danych i tworzenie indeksów nieklastrowych

```
INSERT INTO dbo.SalesPersons (
    SalesPersonFirstName,
    SalesPersonLastName,
    SalesPersonEmail
)
VALUES
    ('Robin', 'Wells', 'robin.wells@magicchoc.com'),
    ('Jack', 'Jones', 'jack.jones@magicchoc.com'),
    ('Jane', 'Smith', 'jane.smith@magicchoc.com') ;

INSERT INTO dbo.SalesAreas (
    SalesAreaName,
    SalesAreaManagerFirstName,
    SalesAreaManagerLastName
)
VALUES
    ('US', 'Lucy', 'Sykes'),
    ('Euro', 'Ashwin', 'Kumar'),
    ('APAC', 'Emma', 'Roberts') ;

INSERT INTO dbo.Addresses (Street, Area, City, ZipCode)
VALUES
    ('744 Saxon Rd', NULL, 'Crawfordsville', '47933'),
    ('267 Old York Ave.', NULL, 'Reno', '89523'),
    ('923 Taylor Ave.', NULL, 'Charlotte', '28205'),
    ('942 Cactus Street', NULL, 'Albany', '12203'),
    ('32 Selby Drive', NULL, 'Pittsfield', '01201'),
    ('65 New Street', 'Landford', 'Salisbury', 'SP5 2QP') ;

INSERT INTO dbo.CustomerContacts (
    CustomerContactFirstName,
    CustomerContactLastName,
    CustomerContactEmail
)
VALUES
    ('Ralphie', 'Buchanan', 'Ralphie.Buchanan@Pitt.com'),
    ('Bettie', 'Peters', 'bpeters@cookingschmooking.co.uk'),
    ('Zackery', 'McEachern', 'Zackery.McEachern@wilsonindustries.com') ;
```

```

INSERT INTO dbo.Customers (
    CustomerCompanyName,
    CustomerContactID,
    InvoiceAddressID,
    DeliveryAddressID
)
VALUES
    ('Pitt and Co', 1, 1, 2),
    ('Cooking Schmooking', 2, 3, 4),
    ('Wilson Industries', 3, 6, 6) ;

INSERT INTO dbo.SalesOrderHeaders (
    SalesOrderNumber,
    SalesOrderDate,
    SalesPersonID,
    SalesAreaID,
    CustomerID,
    SalesOrderDeliveryDueDate,
    SalesOrderDeliveryActualDate,
    CurrierUsedforDelivery
)
VALUES
    ('C001634D-U06', '20230501', 1, 1, 1, '20230503', '20230503', 'LHD'),
    ('WIL1635D-E16', '20230616', 2, 2, 3, '20230630', NULL, 'GoodSpeed International'),
    ('PIT1636D-U04', '20230616', 3, 1, 1, '20230706', NULL, 'LHD') ;

GO

CREATE NONCLUSTERED INDEX NI_SalesOrderHeaders_OrderDate
    ON SalesOrderHeaders(SalesOrderDate) ;
CREATE NONCLUSTERED INDEX NI_SalesOrderHeaders_SalesAreaID
    ON SalesOrderHeaders(SalesAreaID) ;
CREATE NONCLUSTERED INDEX NI_SalesOrderHeaders_CustomerID
    ON SalesOrderHeaders(CustomerID) ;

```

Następnie, aby zademonstrować problem, musimy zobaczyć, co jest przechowywane na stronach jednego z indeksów, które właśnie utworzyliśmy. Użyjmy w tym celu indeksu NI_SalesOrderHeaders_OrderDate, choć moglibyśmy wybrać dowolny inny indeks.

Aby zajrzeć do wnętrza stron indeksu, należy wykonać trzy czynności. Pierwszym z nich jest ustalenie identyfikatora indeksu. Nie chodzi tu o unikatowy identyfikator obiektu w całej instancji bazy danych, ale o identyfikator indeksu w obrębie tabeli. Możemy wyświetlić szczegóły danego indeksu, odpytując widok `sys.indexes` i filtrując wyniki według nazwy indeksu. Pokazano to na listingu 4.3.

Listing 4.3. Określanie identyfikatora indeksu

```

SELECT
    name
    , index_id
    , type_desc
FROM sys.indexes
WHERE name = 'NI_SalesOrderHeaders_OrderDate' ;

```

U mnie identyfikator indeksu wynosi 2, ale u Ciebie może być inny, jeśli tworzyłeś indeksy w innej kolejności. Teraz, gdy mamy już identyfikator indeksu, kolejnym krokiem jest użycie polecenia `DBCC IND` w celu wyświetlenia listy wszystkich stron, które tworzą ten indeks.

Polecenie to wyświetla łańcuch IAM indeksu, od poziomu korzenia aż do poziomu liści. Zwracany jest jeden wiersz dla każdej strony indeksu, zawierający numery stron, poziom indeksu oraz wskaźniki stron do przemierzania indeksu, a także inne przydatne informacje.

Polecenie przyjmuje trzy parametry: nazwę bazy danych (lub jej identyfikator), nazwę tabeli oraz identyfikator indeksu. Sposób użycia tego polecenia pokazano na listingu 4.4.

Listing 4.4. Użycie polecenia `DBCC IND` w celu ustalenia numerów stron indeksu

```
DBCC IND ('MagicChoc', 'SalesOrderHeaders', 2) ;
```

OSTRZEŻENIE Poniżej przedstawiam moje wyniki, ale pamiętaj, że Twoje numery stron prawie na pewno będą inne. W kolejnych przykładach upewnij się, że używasz własnych numerów stron.

Wyniki przedstawiono na rysunku 4.7. Zauważ, że są tylko dwie strony. Pierwsza strona to strona główna indeksu. Zawsze będzie dokładnie jedna strona główna, niezależnie od rozmiaru indeksu. Nie ma pośrednich poziomów indeksu, ponieważ jest on zbyt mały. Poziom liści indeksu składa się dokładnie z jednej strony, ponieważ tabela zawiera tylko trzy rekordy, więc wszystko mieści się na pojedynczej stronie.

| Results | | Messages | | | | | | | | | | | | | |
|---------|---------|----------|--------|--------|------------|---------|-----------------|-------------------|----------------|----------|------------|-------------|-------------|-------------|-------------|
| | PageFID | PagePID | IAMFID | IAMPID | ObjectID | IndexID | PartitionNumber | PartitionID | iam_chain_type | PageType | IndexLevel | NextPageFID | NextPagePID | PrevPageFID | PrevPagePID |
| 1 | 1 | 579 | NULL | NULL | 1669580986 | 2 | 1 | 72057594046906368 | In-row data | 10 | NULL | 0 | 0 | 0 | 0 |
| 2 | 1 | 600 | 1 | 579 | 1669580986 | 2 | 1 | 72057594046906368 | In-row data | 2 | 0 | 0 | 0 | 0 | 0 |

Rysunek 4.7. Wyniki polecenia `DBCC IND`

Ostatnim krokiem jest użycie kolejnego polecenia `DBCC PAGE` o nazwie `DBCC PAGE`. Służy ono do wyświetlania zawartości strony danych lub indeksu. Przyjmuje ono cztery parametry: nazwę bazy danych, identyfikator pliku, identyfikator strony (uzyskany za pomocą `DBCC IND`) oraz parametr określający opcję formatowania. Ważne jest jednak, aby przed uruchomieniem `DBCC PAGE` upewnić się, że włączona jest flaga śledzenia 3604. Flagi śledzenia służą do włączania i wyłączania różnych funkcji, o czym będziemy mówić szerzej w rozdziale 10. Ta konkretna flaga kieruje wyniki polecenia `DBCC` na konsolę. Sposób użycia polecenia pokazano na listingu 4.5.

Listing 4.5. Użycie polecenia `DBCC PAGE` do przeglądania zawartości strony

```
DBCC TRACEON(3604) ;
DBCC PAGE('MagicChoc', 1, 600, 3) ;
```

Wyniki wykonania tego polecenia w moim systemie są przedstawione na rysunku 4.8. Jak można się było spodziewać, w kolumnie SalesOrderDate (key) widoczna jest wartość klucza indeksu. Warto jednak zwrócić uwagę na sąsiednią kolumnę: SalesOrderNumber (key). Wartość klucza podstawowego jest przechowywana w każdym wierszu. Pozwala ona odwoływać się do danych w indeksie klastrowym.

| Results | | Messages | | | | | | |
|---------|-------|----------|-----|-------|----------------------|------------------------|----------------|----------|
| | Field | PageId | Row | Level | SalesOrderDate (key) | SalesOrderNumber (key) | KeyHashValue | Row Size |
| 1 | 1 | 600 | 0 | 0 | 2023-05-01 | COO1634D-U06 | (#1ae9621c514) | 31 |
| 2 | 1 | 600 | 1 | 0 | 2023-06-16 | PIT1636D-U04 | (efc14f13743) | 31 |
| 3 | 1 | 600 | 2 | 0 | 2023-06-16 | WIL1635D-E16 | (411ea115e4a3) | 31 |

Rysunek 4.8. Wyniki polecenia DBCC PAGE

Klucz podstawowy zajmuje zatem nie tylko 12 bajtów na każdy wiersz w tabeli, ale także do 12 bajtów na każdy wiersz we wszystkich indeksach nieklastrowych tej tabeli. Co gorsza, nasze indeksy nieklastrowe nie są unikatowe. Jeśli indeks nieklastrowy jest unikatowy, wartość klucza klastrowego jest przechowywana tylko na poziomie liści indeksu. Jednak w przypadku nieunikatowych indeksów nieklastrowych wartość klucza klastrowego jest przechowywana w każdym wierszu na wszystkich poziomach indeksu.

Ten szeroki klucz nie tylko zajmuje dodatkową przestrzeń dyskową i pamięć, ale także negatywnie wpływa na wydajność indeksów. Pamiętaj, że strona może przechowywać tylko 8000 bajtów danych. W związku z tym im szerszy klucz, tym więcej stron indeksu jest potrzebnych. W konsekwencji SQL Server musi odczytać więcej stron, aby pobrać wymagane dane.

W tym scenariuszu, gdy mamy szeroką kolumnę używaną jako klucz podstawowy lub gdy klucz podstawowy składa się z wielu kolumn, zalecam utworzenie sztucznego klucza podstawowego i automatyczne wypełnianie go za pomocą mechanizmu IDENTITY. Jeśli musimy zapewnić unikatowość klucza naturalnego, możemy to osiągnąć bez zwiększania rozmiaru indeksów klastrowych i nieklastrowych poprzez zbudowanie unikatowego indeksu nieklastrowego na odpowiedniej kolumnie.

Globalnie unikatowe identyfikatory jako klucze główne

Jednym z przykładów szerokich kluczy głównych jest wykorzystanie globalnie unikatowych identyfikatorów (ang. *globally unique identifier*, GUID).

Identyfikatory GUID można generować za pomocą funkcji NEWID(), która zapewnia mechanizm tworzenia wartości unikatowych w obrębie całego serwera. Czasami musimy jednak używać identyfikatorów GUID jako kluczy podstawowych. Na przykład niektóre gotowe produkty komercyjne wymagają tego w swoich specyfikacjach.

Przy wykorzystaniu identyfikatorów GUID jako kluczy głównych należy wziąć pod uwagę dwa aspekty. Pierwszym z nich jest rozmiar. Zajmują one 16 bajtów, więc warto się zastanowić — czy rzeczywiście potrzebujemy wartości, która jest unikatowa globalnie, czy wystarczy, że będzie unikatowa w obrębie tabeli? Jeśli potrzebujemy unikatowości w całej bazie danych, to mogą istnieć inne, bardziej odpowiednie metody, takie jak mechanizm SEQUENCE, które pozwolą osiągnąć ten cel.

Innym godnym uwagi aspektem jest losowa natura identyfikatorów GUID. Indeks klastrowy z definicji porządkuje dane w tabeli według wartości zawartych w tym indeksie, który zazwyczaj wywodzi się z klucza podstawowego. Jeśli te wartości są losowe, SQL Server będzie musiał ciągle reorganizować strony danych. Oznacza to, że wydajność indeksu może się pogorszyć z powodu problemu znanego jako *podziały stron*.

Gdy dochodzi do podziału strony, dane są przenoszone między stronami, aby zachować ich uporządkowanie. Prowadzi to do nasilenia operacji wejścia-wyjścia i obniżenia wydajności operacji UPDATE i INSERT. Dodatkowo powoduje to fragmentację indeksów, którą omówimy w rozdziale 11. Fragmentacja zmniejsza wydajność operacji SELECT do czasu, aż indeks zostanie przebudowany.

Problem ten można złagodzić, stosując niski współczynnik wypełnienia indeksu klastrowego i przebudowując indeksy przy niskim poziomie fragmentacji. Generalnie jednak zalecam tworzenie nieklastrowego klucza podstawowego, a następnie generowanie indeksu klastrowego na kolumnie typu całkowitego. Następnie można utworzyć klucz podstawowy na kolumnie zawierającej identyfikatory GUID.

4.3. Numer 13 — nieużywanie kluczy obcych

Brak ograniczeń klucza obcego to pomyłka, którą często popełniają mniej doświadczeni programiści. Zwykle tłumaczą to tym, że jeśli potrzebują szybko wstawić dane do tabeli, to ograniczenia zbytnio spowalniają ten proces. Najczęściej spotykałem się z takim podejściem w sytuacjach podobnych do naszej, gdy klucz jest kluczem naturalnym, a więc ma znaczenie biznesowe i jest kontrolowany przez aplikację frontendową.

Problemem w tym rozumowaniu jest to, że aplikacja frontendowa po prostu nie może zagwarantować integralności danych w taki sam sposób, jak robią to ograniczenia w SQL Serverze. Aby dokładniej omówić to zagadnienie, posłużymy się przykładem naszych tabel `PurchaseOrderHeaders` i `PurchaseOrderDetails`.

Być może pamiętasz, że w skrypcie tworzącym bazę danych z listingu 4.1 znajdował się błąd. W rezultacie kolumna `PurchaseOrderNumber` znalazła się w tabeli `PurchaseOrderDetails`, ale ponieważ nie utworzono klucza obcego, nie ma możliwości połączenia jej z tabelą `PurchaseOrderHeaders`.

Zanim przejdziemy dalej, wykonajmy skrypt przedstawiony na listingu 4.6, aby wprowadzić dane do odpowiednich tabel.

Listing 4.6. Wstawianie danych do tabel produktów i zaopatrzenia

```

INSERT INTO dbo.ProductCategories (
    ProductCategoryName,
    ProductCategoryDescription
)
VALUES
    ('Raw Ingridience', NULL),
    ('Machine Parts', 'Parts used by manufacturing for machine maintenance'),
    ('Misc', 'Office supplies and other miscellaneous stock items'),
    ('Services', 'Non-stock purchases, such as transport'),
    ('Confectionary Products', NULL),
    ('Non-confectionary Products', NULL) ;

INSERT INTO dbo.ProductSubcategories (
    ProductCategoryID,
    ProductSubcategoryName,
    ProductSubcategoryDescription
)
VALUES
    (1, 'Chilled Ingredience', 'Ingredience that must be kept between 1C and 5C'),
    (1, 'Frozen Ingredience', 'Ingredience must be kept below -18C'),
    (1, 'Ambient Ingredience', 'Ingredience that should be kept in cool, dry storage'),
    (2, 'Line 1 Components', 'Components required for manufacturing line 1'),
    (2, 'Line 2 Components', 'Components required for manufacturing line 1'),
    (2, 'Line 3 Components', 'Components required for manufacturing line 1'),
    (2, 'Line 4 Components', 'Components required for manufacturing line 1'),
    (3, 'Office Supplies', 'Stationary, etc'),
    (3, 'Misc', NULL),
    (4, 'Curriers', NULL),
    (4, 'Building Maintenance', NULL),
    (5, 'Boxes of chocolates', NULL),
    (5, 'Sweets', NULL),
    (5, 'Chocolate Bars', NULL),
    (6, 'Packaging', 'Product Packaging'),
    (6, 'Merchandise', 'Non-core items which are procured and sold, such as mugs and
    ↳branded gifts') ;

INSERT INTO dbo.ProductTypes (ProductTypeName, ProductTypeDescription)
VALUES
    ('Purchased Product', 'Products which are purchased'),
    ('Sold products', 'Products which are sold'),
    ('Traded Products', 'Products which are both bought and sold') ;

INSERT INTO dbo.SupplierContacts (
    SupplierContactFirstName,
    SupplierContactLastName,
    SupplierContactEmail
)
VALUES
    ('John', 'Smith', 'john.smith@smithfields.com'),
    ('John', 'Doe', 'john.doe@unknownengineering.com'),
    ('Michael', 'Knight', 'mknight@knightridercurriers.com') ;

INSERT INTO dbo.Addresses (
    Street,
    City,
    ZipCode

```



```

)
VALUES
    ('8648 Columbia Street', 'Beachwood', '44122'),
    ('83 Addison Dr.', 'Westerville', '43081'),
    ('508 Mill Pond Street', 'Clinton Township', '48035') ;

INSERT INTO dbo.Suppliers (
    SupplierName,
    SupplierContactID,
    SupplierAddressID
)
VALUES
    ('Smithfeilds', 1, 7),
    ('Unknown Engineering', 2, 8),
    ('Knight Rider Curriers', 3, 9) ;

INSERT INTO dbo.Products (
    ProductName,
    ProductStockLevel,
    ProductTypeID,
    ProductSubcategoryID
)
VALUES
    ('Large head sprocket', 3, 1, 4),
    ('Long weight', 6, 1, 4),
    ('Staples', 8900, 1, 8),
    ('Magic Mug', 38, 3, 16),
    ('Massive Magic Box', 18, 2, 12),
    ('Delivery', -1, 3, 10) ;

```

Teraz zasymulujemy działanie naszej aplikacji frontendowej, która będzie wstawiać zamówienia do tabeli, za pomocą skryptu z listingu 4.7. Aplikacja włącza opcję `XACT_ABORT` i umieszcza obie instrukcje `INSERT` wewnątrz transakcji. To dobre rozwiązanie. Oznacza to, że jeśli jedna z instrukcji się nie powiedzie, druga również zostanie anulowana, co pomaga utrzymać spójność danych.

Listing 4.7. Dodawanie zamówienia zakupowego

```

SET XACT_ABORT ON ;

BEGIN TRANSACTION
    SELECT @@TRANCOUNT ;

    INSERT INTO dbo.PurchaseOrderHeaders (
        PurchaseOrderNumber,
        SupplierID,
        PurchaseOrderDate
    )
    VALUES
        (6826, 2, '20230601'),
        (6827, 2, '20230617') ;

    INSERT INTO dbo.PurchaseOrderDetails (
        ProductID,
        Quantity,
        PurchaseOrderNumber

```

```

)
VALUES
  (4, 3, 6826),
  (5, 4, 6827),
  (4, 1, 6827) ;
COMMIT

```

Zastanówmy się jednak nad powodem rezygnacji z klucza obcego: chodzi o możliwość szybkiego i łatwego aktualizowania tabeli. Wyobraźmy sobie następującą sytuację. Dzwoni do nas szef działu zaopatrzenia i mówi: „Mamy problem z zamówieniem. Potrzebuję, żebyś naprawił to w bazie danych, bo aplikacja nie pozwala mi tego zrobić. Musisz przenieść zamówione pozycje z zamówienia numer 6827 do zamówienia 6828 — natychmiast!”.

Znajdując się pod presją, zupełnie racjonalnie robimy dokładnie to, o co prosił nas kierownik działu zaopatrzenia, i wykonujemy instrukcję przedstawioną na listingu 4.8.

Listing 4.8. Modyfikowanie zamówienia kupna

```

UPDATE dbo.PurchaseOrderDetails
SET PurchaseOrderNumber = 6828 ← Nie byłoby to możliwe, gdyby używano ograniczenia
WHERE PurchaseOrderNumber = 6827 ; klucza obcego.

```

Niestety, kierownik działu zaopatrzenia tak naprawdę chciał, żebyśmy utworzyli nowe zamówienie z numerem 6828, przenieśli zamówione pozycje, a następnie usunęli zamówienie o numerze 6827.

Teraz nasze dane znajdują się w niespójnym stanie. Gdybyśmy wykonali kwerendę, taką jak ta z listingu 4.9, która zwraca szczegóły zamówień z obu tabel, to ani zamówienie o numerze 6827, ani 6828 nie zostałoby uwzględnione w wynikach.

Listing 4.9. Zwracanie informacji o zamówieniu kupna

```

SELECT
  poh.PurchaseOrderNumber
, poh.PurchaseOrderDate
, pod.ProductID
, pod.Quantity
FROM dbo.PurchaseOrderHeaders poh
INNER JOIN dbo.PurchaseOrderDetails pod
  ON poh.PurchaseOrderNumber = pod.PurchaseOrderNumber ;

```

Gdybyśmy wcześniej utworzyli ograniczenie klucza obcego, taka operacja nie byłaby możliwa. Baza danych wymusiłaby integralność referencyjną i nie moglibyśmy ustawić kolumny PurchaseOrderNumber na wartość, która nie istnieje w tabeli PurchaseOrderHeaders.

Aby uniknąć tego problemu, zawsze należy stosować ograniczenia klucza obcego. Błąd w bazie danych MagicChoc można naprawić, uruchamiając skrypt

z listingu 4.10. Przed utworzeniem ograniczenia najpierw poprawiamy wartości danych, w przeciwnym razie tworzenie ograniczenia zakończyłoby się niepowodzeniem.

Listing 4.10. Dodawanie ograniczenia klucza obcego

```
UPDATE dbo.PurchaseOrderDetails
SET PurchaseOrderNumber = 6827
WHERE PurchaseOrderNumber = 6828 ;

ALTER TABLE dbo.PurchaseOrderDetails ADD CONSTRAINT
    FK_PurchaseOrderDetails_PurchaseOrderHeaders
    FOREIGN KEY (PurchaseOrderNumber)
    REFERENCES dbo.PurchaseOrderHeaders (PurchaseOrderNumber) ;
```

Gdy tabele są ze sobą powiązane, dobrą praktyką jest tworzenie relacji między kluczami głównymi i obcymi. Nawet jeśli logika aplikacji zapobiega niespójności danych, to nie poradzi nic na błędy, które mogą wystąpić poza aplikacją.

Podsumowanie

- Aby uniknąć pomyłek, zawsze normalizuj swoje dane, zamiast projektować schemat bazy danych na podstawie własnego osądu.
- W stosownych przypadkach rozważ generalizację danych w celu utworzenia nadtypów i podtypów, ponieważ może to poprawić projekt bazy danych.
- Wykorzystaj diagram związków encji (ERD) do zweryfikowania oraz udokumentowania projektu. Diagram ten pozwoli zwizualizować encje wraz z ich atrybutami, kluczami głównymi i obcymi, a także relacje między nimi.
- Unikaj używania szerokich kolumn lub wielu kolumn jako klucza głównego, ponieważ klucz główny często staje się indeksem klastrowym, a tym samym jest powielany w indeksach nieklastrowych. Może to prowadzić do pogorszenia wydajności bazy danych.
- Zawsze stosuj ograniczenie klucza obcego w przypadku powiązanych ze sobą tabel, aby uniknąć problemów ze spójnością danych.

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 pisanem 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.

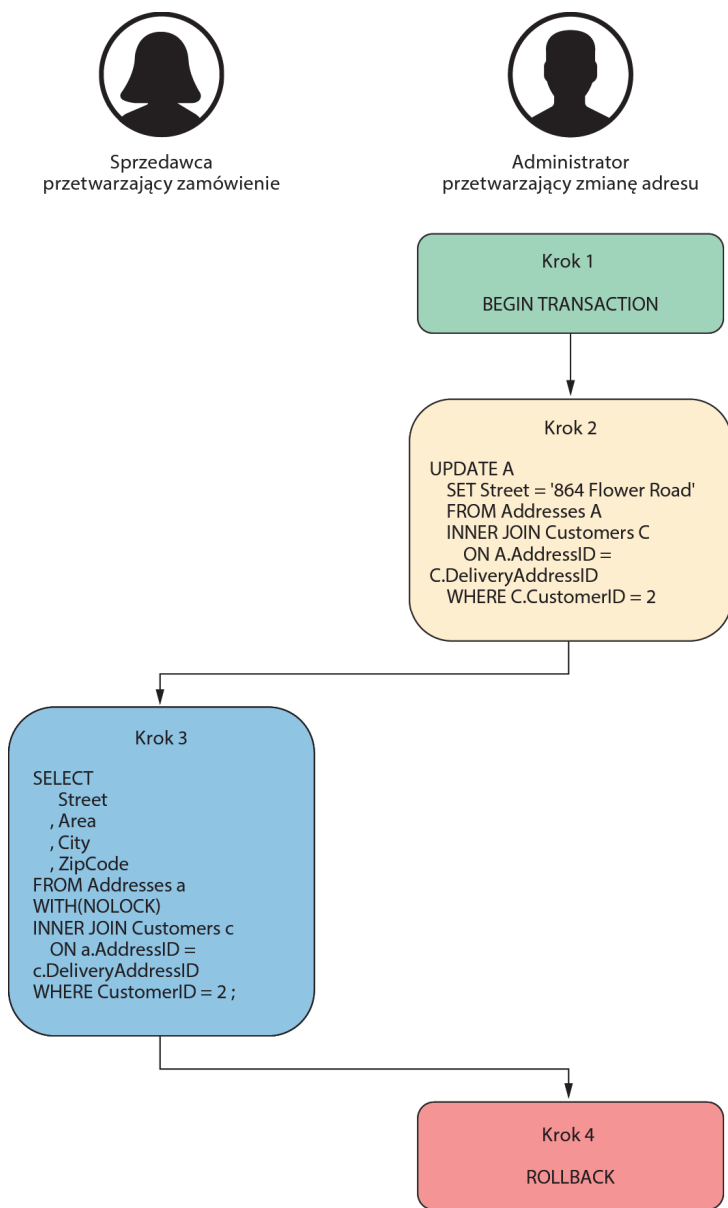
Listing 5.5. Dodawanie wskazówki NOLOCK do kwerendy

```
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 ;

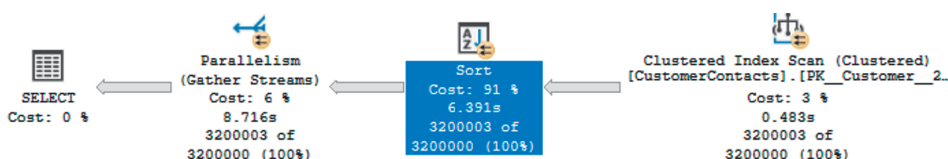
SELECT
    CustomerContactFirstName
    , CustomerContactLastName
    , CustomerContactEmail
FROM dbo.CustomerContacts
ORDER BY CustomerContactEmail ;
```

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.

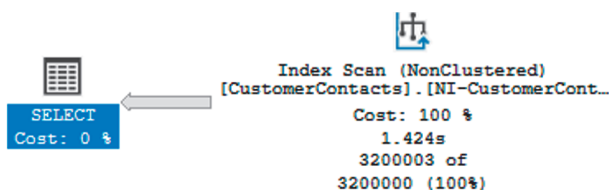
Po drugie, koszt nie jest bezpośrednim miernikiem wydajności. Jest to wartość ważona, obliczana na podstawie zastrzeżonego algorytmu, który szacuje względny koszt operacji procesora i wejścia-wyjścia, a następnie sumuje te wartości, aby uzyskać całkowity koszt operatora, znany jako *koszt poddrzewa*.

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,

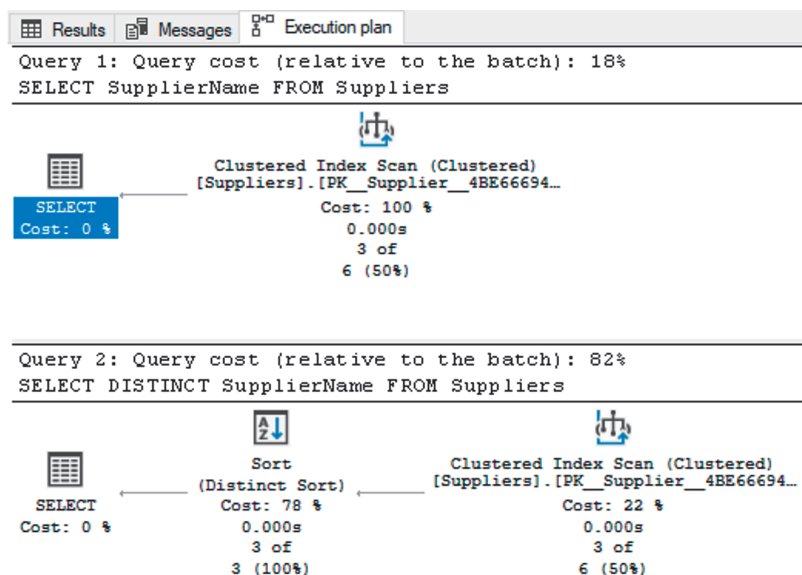
że nie mamy poważnego problemu z jakością danych) ten sam dostawca nie będzie wymieniony dwukrotnie w naszej tabeli, zastosowanie słowa kluczowego *DISTINCT* jest zbędne.

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 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 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.

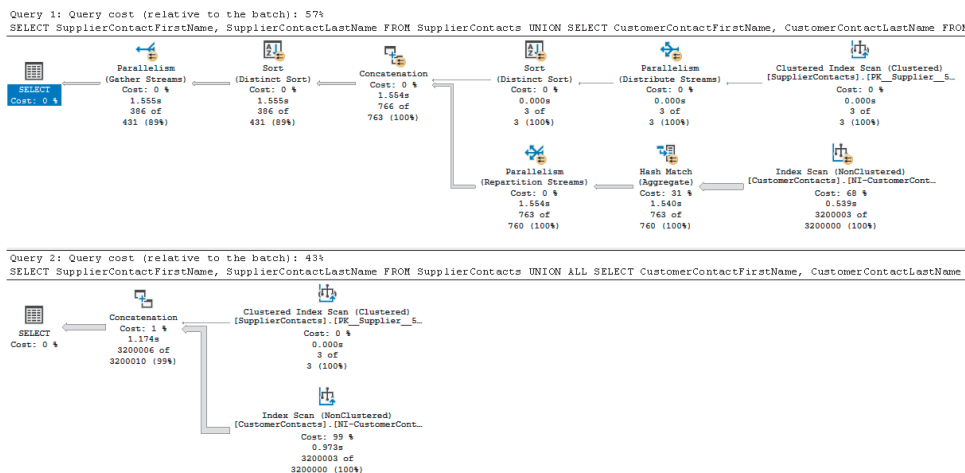
WSKAZÓWKA Istnieją dodatkowe operatory łączenia poziomego o nazwach INTERSECT i EXCEPT. INTERSECT zwraca wyniki z pierwszej kwerendy, które pojawiają się również w wynikach drugiej kwerendy. Z kolei EXCEPT zwraca wyniki z pierwszej kwerendy, których nie ma w wynikach drugiej kwerendy.

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 kursor jest 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 kursor stał się niepotrzebny. 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

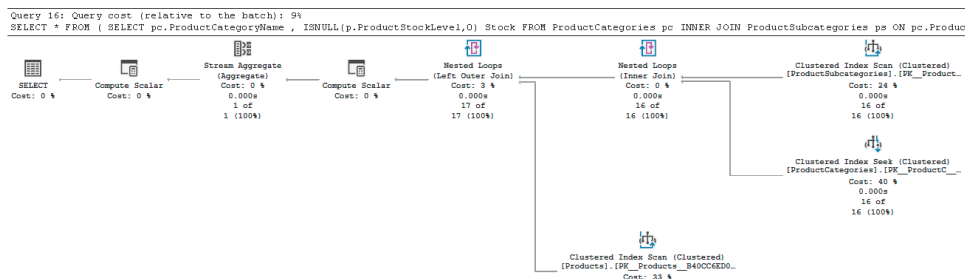
```

```

, 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 ;

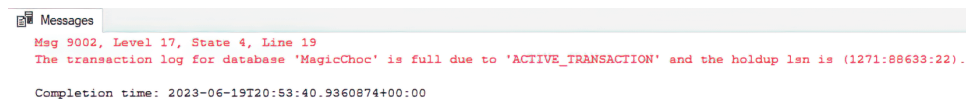
    SET @LoopCounter = @LoopCounter + 1 ;
END
```

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 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 do-
rażnych. 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 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.

6 Programowanie w SSIS

W tym rozdziale:

- Wprowadzenie do SSIS i pomyłki podczas programowania w SSIS
- Utrata niepoprawnych danych
- Nieoptymalizowanie wczytywania danych
- Używanie SSIS jako narzędzia do koordynacji T-SQL
- Pobieranie wszystkich danych z tabeli źródłowej

Usługi SQL Server Integration Services, znane powszechnie jako SSIS, to narzędzie dostarczane z edycjami Enterprise i Standard SQL Servera, choć w wersji Standard występują pewne ograniczenia funkcjonalności. Jest to narzędzie do *wyodrębniania, przekształcania i wczytywania danych* (ang. *extract, transform, load*, ETL) umożliwiające programistom tworzenie procesów przenoszenia i przekształcania danych w intuicyjnym interfejsie graficznym typu „przeciągnij i upuść”.

UWAGA W przykładach w tym rozdziale będzie wykorzystywane narzędzie SQL Server Data Tools, które można pobrać ze strony <https://mng.bz/RNBO>. Konieczne jest również zainstalowanie rozszerzenia Integration Services. Aby to zrobić, przejdź do sekcji *Extensions/Manage Extensions*, a następnie wyszukaj w sklepie „SQL Server Integration Services Projects 2022”.

Pakiet SSIS zawsze składa się z jednego *przepływu sterowania*, który zarządza zadaniami wykonywanymi przez pakiet. W ramach przepływu sterowania można utworzyć *przepływy danych*, które służą do importowania, eksportowania i przekształcania danych w buforach pamięci.

Zadania w przepływie sterowania są połączone ograniczeniami pierwszeństwa. Pozwala to na projektowanie pakietów, w których zadania wykonywane są szeregowo, a nie równolegle. Zadania następujące po ograniczeniu pierwszeństwa zawsze uruchamiane są po zadaniach poprzedzających to ograniczenie.

Ograniczenia

SSIS oferuje trzy rodzaje ograniczeń: sukces, niepowodzenie i ukończenie. Pozwalają one kontrolować przepływ pakietu i tworzyć niestandardową logikę obsługi błędów. Zadania połączone z zagrożeniami działają w następujący sposób:

- Zadanie powiązane z ograniczeniem sukcesu zostanie wykonane po pomyślnym zakończeniu poprzedniego zadania. Jeśli poprzednie zadanie zakończy się niepowodzeniem, to powiązane zadanie nie zostanie uruchomione.
- Zadanie powiązane z ograniczeniem niepowodzenia zostanie wykonane po niepowodzeniu poprzedniego zadania. Jeśli poprzednie zadanie zakończy się sukcesem, to powiązane zadanie nie zostanie uruchomione.
- Zadanie powiązane z ograniczeniem ukończenia zostanie uruchomione po wykonaniu poprzedniego zadania. Zostanie ono wykonane niezależnie od tego, czy poprzednie zadanie zakończyło się sukcesem czy niepowodzeniem.

W przepływie sterowania SSIS ograniczenie sukcesu jest oznaczone kolorem zielonym, ograniczenie niepowodzenia — czerwonym, a ograniczenie zakończenia — niebieskim.

Ograniczeń pierwszeństwa w przepływie sterowania nie należy mylić ze ścieżkami danych w przepływie danych. W przepływie danych niebieska ścieżka oznacza, że wiersze, które pomyślnie przeszły przez poprzedni komponent, zostaną przekazane do następnego. Czerwona ścieżka wskazuje, że wiersze, które nie przeszły przez poprzedni komponent, zostaną przekazane do następnego. Nie ma ścieżki ukończenia.

Gdy zaczynamy tworzyć projekt na podstawie szablonu Integration Services, widzimy pusty przepływ sterowania z oknem narzędziowym SSIS po lewej stronie. Okno to służy do przeciągania zadań na obszar przepływu sterowania lub komponentów na obszar przepływu danych. Zawartość okna narzędziowego zmienia się w zależności od kontekstu — jeśli pracujemy w przepływie danych, zobaczymy komponenty przepływu danych zamiast zadań przepływu sterowania.

Każdy przepływ danych jest w rzeczywistości zadaniem w naszym przepływie sterowania, więc możemy utworzyć przepływ danych, przeciągając zadanie przepływu danych na obszar przepływu sterowania. Następnie możemy wyświetlić przepływ danych, klikając dwukrotnie na zadanie lub przełączając się na kartę *Data Flow* u góry naszego obszaru projektowego.

Menedżery połączeń służą do tworzenia połączeń ze źródłami danych, takimi jak bazy danych i pliki płaskie. Standardowo dostępne są menedżery połączeń OLE DB, ADO.NET, plików, plików płaskich i usług analitycznych. Z powodów wydajnościowych przy łączeniu się z instancjami SQL Server w przepływach danych optymalnym wyborem jest często OLE DB. Menedżery połączeń ADO.NET są często używane do łączenia się z SQL Serverem w celu wykonywania zadań *Execute T-SQL Statement* w przepływie sterowania.

Konfiguracja OLE DB

Kilka lat temu firma Microsoft uznała technologię OLE DB za przestarzałą. Jednak w 2018 r. przywrócono jej obsługę i wydano nowy sterownik OLE DB. SQL Native Client 11 nadal jest uznawany za przestarzały i nie jest dostarczany z SQL Serverem 2022. Dlatego, aby korzystać z OLE DB w aplikacjach łączących się z SQL Serverem, w tym w SSIS, należy pobrać i zainstalować sterownik Microsoft OLE DB Driver 19 for SQL Server, który można znaleźć pod adresem <https://mng.bz/AagK>.

Do poprawnego działania tego sterownika wymagany jest pakiet Microsoft Visual C++ Redistributable. Należy go zainstalować w pierwszej kolejności. Pakiet ten można pobrać ze strony <https://mng.bz/ZVXO>.

Po zainstalowaniu tych pakietów odpowiedni sterownik OLE DB pojawi się na liście rozwijanej *Provider*. Domyślnie jednak menedżer połączeń ustawi właściwość *Use Encryption For Data* (użyj szyfrowania danych) na *Mandatory* (obowiązkowo). Jeśli nie masz skonfigurowanych certyfikatów dla swojej instancji SQL Servera, będziesz musiał zmienić to ustawienie na *Optional* (opcjonalnie). Właściwość tę można znaleźć przez kliknięcie przycisku *Data Links* (łącza danych) w oknie dialogowym menedżera połączeń i przejście na kartę *All* (wszystkie).

Programiści mogą popełniać różne pomyłki podczas tworzenia pakietów SSIS. W tym rozdziale przyjrzymy się błędowi, który może prowadzić do utraty danych. Omówimy konsekwencje nieoptymalizowanego wczytywania danych. Przeanalizujemy również skutki wykorzystywania SSIS wyłącznie jako narzędzia do koordynacji, bez korzystania z potoków danych. Na koniec przyjrzymy się błędowi polegającemu na braku filtrowania podczas ekstrakcji danych.

Aby przyjrzeć się niektórym pomyłkom często popełnianym przez użytkowników SSIS, kontynuujmy przykład firmy MagicChoc i rozważmy następujący scenariusz. Dział marketingu planuje reklamować firmę w sieci i chce śledzić odsłony banerów reklamowych wyświetlane potencjalnym klientom. Dane o odsłonach będą otrzymywane w pliku CSV, który będzie przetwarzany przez uruchamiany raz dziennie pakiet SSIS. Pakiet ten umieści dane w tabeli w schemacie staging bazy danych Marketing. Następnie przekształci je i wprowadzi do tabeli w schemacie marketing. Na koniec zagreguje dane i wstawi je do zbiorczej tabeli w schemacie reporting, po czym wyczyści tabelę przygotowawczą.

Jeśli chcesz samodzielnie wykonywać przykłady z tego rozdziału, skorzystaj z przykładowego pliku CSV dostępnego w repozytorium kodu dołączonym do tej książki. Plik nosi nazwę *impressions.csv*. Będziesz także potrzebować bazy danych Marketing, którą utworzysz za pomocą skryptu z listingu 6.1.

Listing 6.1. Tworzenie bazy danych Marketing

```

CREATE DATABASE Marketing ;
GO

USE Marketing ;
GO

CREATE SCHEMA staging ;
GO

CREATE SCHEMA marketing ;
GO

CREATE SCHEMA reporting ;
GO

CREATE TABLE staging.ImpressionsStage (
    ImpressionUID      VARCHAR(MAX)    NULL,
    ReferralURL         VARCHAR(MAX)    NULL,
    CookieID            VARCHAR(MAX)    NULL,
    CampaignID          VARCHAR(MAX)    NULL,
    RenderingID         VARCHAR(MAX)    NULL,
    CountryCode         VARCHAR(MAX)    NULL,
    StateID             VARCHAR(MAX)    NULL,
    BrowserVersion      VARCHAR(MAX)    NULL,
    OperatingSystemID   VARCHAR(MAX)    NULL,
    CostPerMille        VARCHAR(MAX)    NULL,
    EventTime           VARCHAR(MAX)    NULL,
    BidPrice            VARCHAR(MAX)    NULL
) ;

CREATE TABLE marketing.Impressions (
    ImpressionID        BIGINT          NOT NULL    PRIMARY KEY    IDENTITY,
    ImpressionUID       UNIQUEIDENTIFIER NOT NULL,
    ReferralURL         VARCHAR(512)    NOT NULL,
    CookieID            UNIQUEIDENTIFIER NOT NULL,
    CampaignID          BIGINT          NOT NULL,
    RenderingID         BIGINT          NOT NULL,
    CountryCode         TINYINT         NULL,
    StateID             TINYINT         NULL,
    BrowserVersion      BIGINT          NOT NULL,
    OperatingSystemID   BIGINT          NOT NULL,
    BidPrice            MONEY           NOT NULL,
    CostPerMille        MONEY           NOT NULL,
    EventTime           DATETIME        NOT NULL
) ;

CREATE TABLE reporting.ImpressionAggregates (
    ImpressionAggregateID BIGINT    NOT NULL    PRIMARY KEY    IDENTITY,
    CampaignID            BIGINT    NOT NULL,
    CountryCode           TINYINT   NOT NULL,
    EventDate             DATE      NOT NULL,
    AvgBidPrice           MONEY     NOT NULL,
    AvgCostPerMille       MONEY     NOT NULL
) ;

```

W Visual Studio utworzymy nowy projekt o nazwie ImpressionsLoad, wykorzystując typ projektu Integration Services (który musi być zainstalowany). Rozwiązanie powinno mieć tę samą nazwę, a projekt można utworzyć w tym samym katalogu co rozwiązanie.

Terminologia marketingu Internetowego

Dla kontekstu poniżej podano definicje niektórych terminów używanych w tej dziedzinie danych:

- Koszt za tysiąc wyświetleń (CPM) to rzeczywista opłata pobierana za tysiąc odsłon reklamy, ustalana w wyniku automatycznej aukcji powierzchni reklamowej.
- Cena wywoławcza to maksymalna kwota, jaką reklamodawca jest gotów zapłacić za tysiąc wyświetleń w zautomatyzowanej aukcji.
- Odsyłający adres URL to strona internetowa, na której nastąpiła odsłona.
- Cookie ID to unikatowy identyfikator pliku cookie używanego do śledzenia. W związku z tym jest on tożsamy z użytkownikiem.

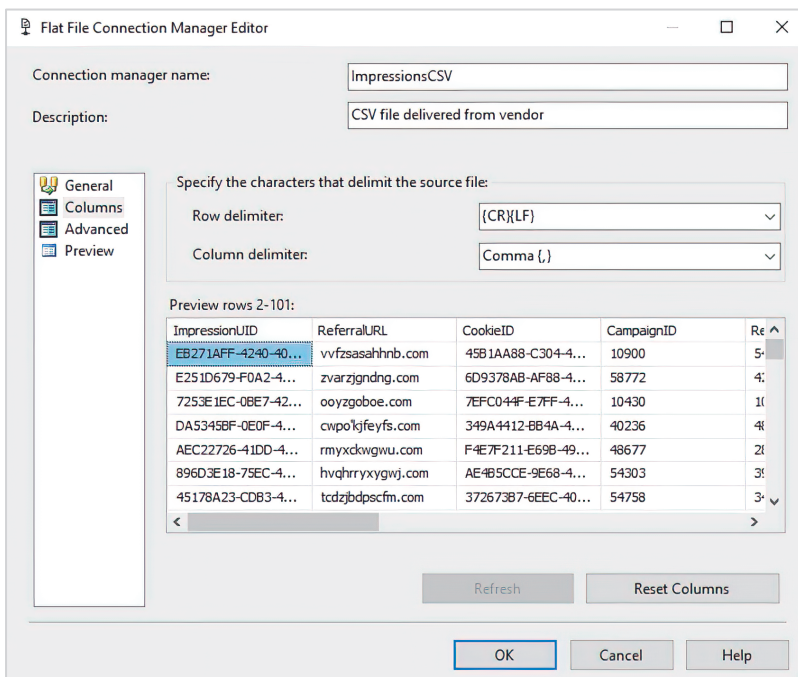
Warto również zauważyć, że „wyświetlenia” są synonimem „odsłon”.

6.1. Numer 22 — odrzucanie niepoprawnych danych

Jedną z najczęstszych pomyłek popełnianych przez programistów SSIS jest odrzucanie wierszy, których nie udaje się wczytać. Często wynika to z błędnych wyobrażeń na temat działania SSIS, przez co programiści nawet nie zdają sobie sprawy, że to robią. Aby to lepiej zrozumieć, stwórzmy prosty pakiet SSIS z przepływem danych, który wczyta dane o odsłonach z naszego pliku CSV (który umieściłem w katalogu głównym dysku C:) do tabeli staging. impressions.

Zacznijmy od przecignięcia zadania przepływu danych na obszar przepływu sterowania i zmiany jego nazwy na *Load Impressions Staging*. Następnie utworzymy dwa menedżery połączeń: jeden dla pliku CSV, a drugi dla instancji SQL Server.

Aby utworzyć menedżer połączeń dla pliku CSV, należy wybrać połączenie typu *Flat File*. Na stronie *General* okna dialogowego można skonfigurować podstawowe parametry połączenia, takie jak nazwa pliku, jego kodowanie i typ (rozdzielany, o stałej szerokości lub z nierównym prawym marginesem). Na stronie *Columns* (rysunek 6.1) należy upewnić się, że jako separator kolumn ustawiono przecinek. Inne możliwe separatory to m.in. tabulator, średnik i kreśka pionowa. Strona *Advanced* pozwala określić właściwości poszczególnych kolumn, na przykład typ danych. Na stronie *Preview* wyświetlanych jest pierwszych 100 wierszy pliku.



Rysunek 6.1. Menedżer połączeń plików płaskich — strona Columns

Na stronie *Advanced*, w oknie dialogowym, powinniśmy ustawić typy danych. SSIS może zasugerować typy danych na podstawie pierwszych 100 wierszy, ale jeśli nie mamy pewności, że te wiersze odpowiednio reprezentują wszystkie możliwe wartości w naszych danych, powinniśmy sami wybrać typy danych. Ponieważ jednak wczytujemy dane do tabeli przygotowawczej o elastycznych typach danych, na razie pozostawimy typy danych jako łańcuchy o długości 50 znaków.

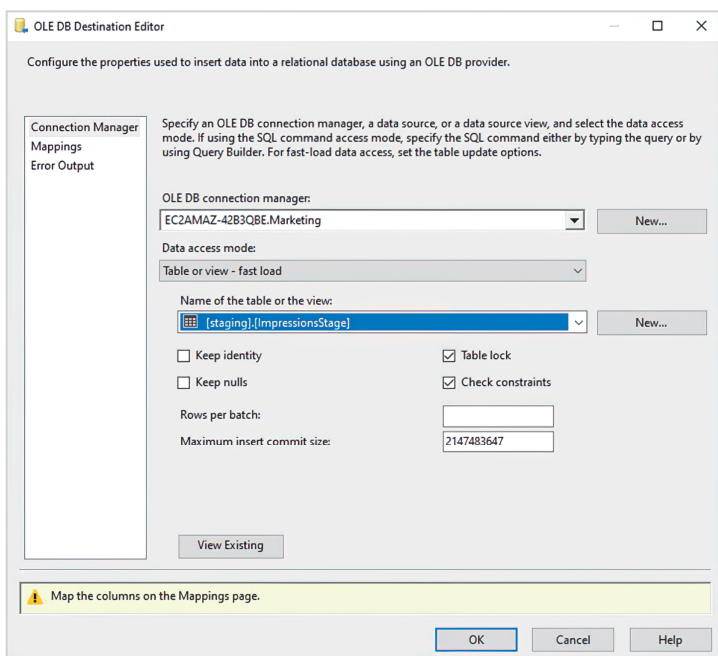
Nasz drugi menedżer połączeń będzie typu OLE DB i będzie wskazywał na instancję SQL Server, w której znajduje się marketingowa baza danych. W tym menedżerze połączeń najpierw wybierzemy z listy rozwijanej dostawcę OLE DB, którego chcemy użyć. W naszym przypadku wybierzemy *Native OLE DB\Microsoft OLE DB Driver 19 for SQL Server*. Następnie określimy nazwę instancji SQL Server oraz nazwę bazy danych (katalog początkowy), z którą będziemy się łączyć. Dodatkowo możemy wybrać sposób uwierzytelniania w instancji. Jeśli użyjemy zintegrowanego uwierzytelniania Windows, pakiet będzie się uwierzytelniał z tożsamością użytkownika, który go uruchamia. Jeśli wybierzemy nazwę użytkownika i hasło do SQL Servera, pakiet użyje uwierzytelniania drugiego poziomu, znanego również jako uwierzytelnianie SQL.

Wracając do przepływu danych, możemy teraz utworzyć źródło pliku płaskiego, które będzie korzystać z naszego menedżera połączeń dla plików płaskich, oraz miejsce docelowe OLE DB, które będzie używać naszego menedżera połączeń OLE DB.

Strona *Columns* źródła przepływu danych pozwala nam mapować *kolumny zewnętrzne*, czyli kolumny pochodzące ze źródła danych, na *kolumny wyjściowe*, które są przekazywane z naszego komponentu do następnego komponentu w przepływie danych. Możemy usunąć kolumny zewnętrzne, usuwając ich zaznaczenie w górnym oknie, a także zmienić nazwy kolumn wyjściowych. W naszym konkretnym przypadku odwzorujemy kolumnę zewnętrzną *ImpressionUID* na kolumnę wyjściową o nazwie *ImpressionID*.

Na stronie *Error Output* źródła pliku płaskiego możemy zdefiniować zachowanie komponentu w przypadku niepowodzenia przy przetwarzaniu dowolnego wiersza. Możemy rozróżnić i określić różne reakcje komponentu w zależności od tego, czy wystąpią błędy ogólne, czy przycięcia danych. Na razie pozostawimy wartości domyślne, co oznacza, że jeśli wiersz nie zostanie pomyślnie przekazany ze źródła do przepływu danych, działanie komponentu zakończy się niepowodzeniem.

Aby dokończyć prosty przepływ danych, będziemy potrzebować również miejsca docelowego OLE DB. W miejscu docelowym możemy wybrać, czy chcemy wczytać dane do nazwanego obiektu, czy do obiektu, który zostanie pobrany ze zmiennej. Dla każdej z tych opcji możemy również zdecydować, czy chcemy skorzystać z opcji szybkiego wczytywania. Opcje te pozwolą nam zoptymalizować wydajność wczytywania; omówimy to dokładniej dalej w tym rozdziale. W naszym przypadku użyjemy trybu szybkiego wczytywania danych do tabeli lub widoku i wybierzemy *staging.ImpressionsStage* jako miejsce docelowe, co pokazano na rysunku 6.2.



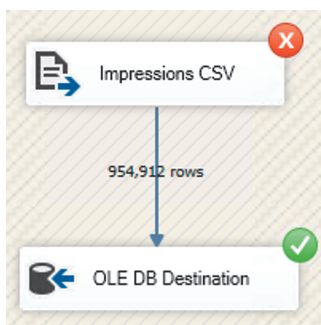
Rysunek 6.2. Edytor miejsca docelowego OLE DB — strona menedżera połączeń

UWAGA Pozostawienie domyślnych ustawień opcji szybkiego wczytywania to pomyłka. Zawsze powinniśmy poświęcić czas na skonfigurowanie tych opcji. Jest to część szerszego problemu nieoptymalizowania procesu wczytywania danych, o czym porozmawiamy w następnym podrozdziale.

Na stronie *Mappings* miejsca docelowego OLE DB wszystkie kolumny wejściowe (pochodzące z poprzedniego komponentu) zostaną automatycznie przypisane do kolumn docelowych (w tabeli docelowej), jeśli ich nazwy są takie same. Kolumna *ImpressionID* nie jest jednak odwzorowana, ponieważ jej nazwa jest inna. Dlatego musimy wybrać ją z listy rozwijanej. W razie potrzeby możemy również zmodyfikować automatyczne odwzorowania.

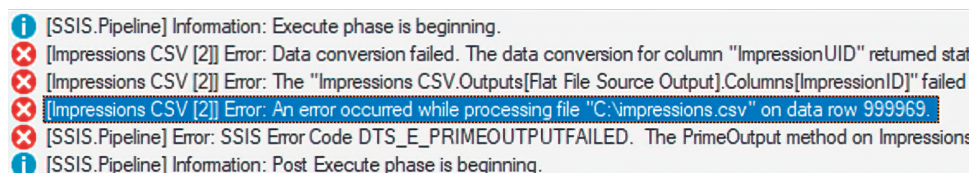
Strona *Error Output* miejsca docelowego określa zachowanie komponentu w przypadku nieudanego wstawienia wiersza. Podobnie jak w przypadku wyjścia błędów w źródle danych, pozostawimy domyślną opcję *Fail Component* (przerwij działanie komponentu). Kolumny w naszej tabeli docelowej są tak elastyczne, że mało prawdopodobne jest wystąpienie błędu wstawiania wiersza, na który można by coś poradzić.

Teraz wykonajmy pakiet w trybie debugowania przez kliknięcie przycisku *Start* na pasku narzędzi i zobaczymy, co się stanie. Wyniki pokazano na rysunku 6.3. Jak widać, udało się wczytać 954 912 wierszy do tabeli przygotowawczej, po czym wystąpił błąd w źródle Impressions CSV. Jest to sygnalizowane czerwonym krzyżykiem.



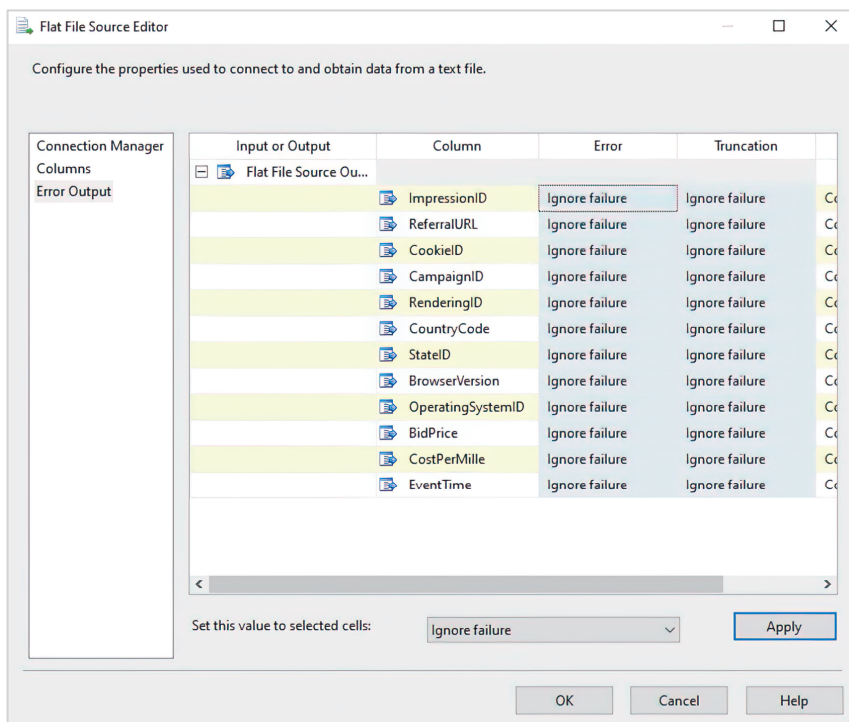
Rysunek 6.3. Nieudany przepływ danych

Jeśli przejdziemy na kartę *Progress*, zobaczymy, że wystąpił błąd przycięcia danych w wierszu 999 969, jak pokazano na rysunku 6.4.



Rysunek 6.4. Błąd na karcie *Progress*

Ależ irytujące! W pliku jest milion wierszy! Nie chcemy przecież stracić całego pliku z powodu jednego błędnego wiersza, prawda? Dlatego wróćmy do strony *Error Output* w edytorze źródła pliku płaskiego i zmienmy działanie komponentu tak, aby w przypadku napotkania błędów ignorował problematyczne wiersze i kontynuował wczytywanie danych. Ilustruje to rysunek 6.5.



Rysunek 6.5. Edytor źródła pliku płaskiego — strona *Error Output* ustawiona na ignorowanie błędów

WSKAZÓWKA Przed modyfikacją pakietu kliknij przycisk *Stop Debugging*.

WSKAZÓWKA Przed ponownym uruchomieniem pakietu opróżnij tabelę przygotowawczą, aby zapobiec podwójnemu wczytaniu danych. Możesz to zrobić za pomocą polecenia `TRUNCATE TABLE Staging.ImpressionsStage`.

Jeśli teraz ponownie uruchomimy pakiet, zobaczymy, że przepływ danych zakończył się sukcesem. Będzie to oznaczone zielonym znacznikiem przy każdym komponencie. Wydaje się, że wszystko działa świetnie, prawda? Niestety, nie do końca. W rzeczywistości popełniliśmy pomyłkę. Co prawda rozwiązaliśmy doraźny problem, ale co się stanie, gdy następny plik będzie zawierał 5 błędnych wierszy? Albo 50? A może nawet 5000? Identyfikator cookie to ważna kolumna, której potrzebujemy, więc nie powinniśmy po prostu odrzucać tych niepoprawnych danych. Z drugiej strony nie chcemy, żeby nasz pakiet zawodził za każdym razem. Jak więc powinniśmy postąpić?

Rozwiązaniem jest przekierowanie nieprawidłowych rekordów do innego miejsca docelowego, aby zespół wsparcia aplikacji mógł je przeanalizować i zdecydować, czy można je bezpiecznie odrzucić, czy też możliwe jest naprawienie danych. Może się nawet okazać, że osoba odpowiedzialna za aplikację będzie musiała skontaktować się z dostawcą i poprosić o nowy zestaw danych. Dane pochodzące z niezaufanych źródeł często są nieoczyszczone, dlatego powinniśmy uwzględnić to w naszym kodzie.

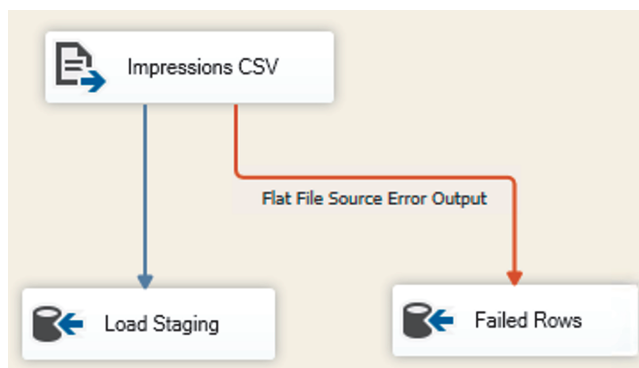
Aby rozwiązać ten problem, najpierw utworzymy nową tabelę na wiersze, w przypadku których zadanie się nie powiodło. Możemy to zrobić za pomocą skryptu przedstawionego na listingu 6.2.

Listing 6.2. Tworzenie tabeli na błędne wiersze

```
CREATE TABLE staging.ImpressionLoadFailures (
  FlatFileSourceErrorOutputColumn VARCHAR(MAX) NOT NULL,
  ErrorCode VARCHAR(MAX) NOT NULL,
  ErrorColumn VARCHAR(MAX) NOT NULL
);
```

Następnie utworzymy nowe miejsce docelowe OLE DB i nazwiemy je *Failed Rows*. Przeciągnij czerwony łącznik ze źródła danych do nowego miejsca docelowego w przepływie danych. Przy obecnej konfiguracji spowoduje to natychmiastowe wyświetlenie okna dialogowego *Configure Error Output*. Okno to jest identyczne jak strona *Error Output* w edytorze źródła danych. Powinniśmy skonfigurować tę stronę tak, aby przekierowywać wiersze zawierające błędy i przycięcia danych.

Skonfigurujmy teraz nowe miejsce docelowe. Na stronie *Connection Manager* wskażemy naszą nową tabelę *ImpressionLoadFailures* jako tabelę docelową. Na stronie *Mappings* będziemy musieli ręcznie odwzorować kolumnę *Flat File Source Error Output*, ponieważ nasza tabela nie zawiera spacji w nazwach kolumn, więc automatyczne odwzorowywanie nie zadziała. Nasz przepływ danych będzie teraz wyglądał jak na rysunku 6.6.



Rysunek 6.6. Przepływ danych z wyjściem błędów

Jeśli teraz ponownie uruchomimy pakiet, niepoprawny wiersz zostanie wstawiony do naszej tabeli `ImpressionLoadFailures`. Możemy go następnie przeanalizować:

```
91C27789-2805-41EB-8E06-A4F5AD147817;jouktjkatujr.com,  
7D522130-5A00-4C4B-83D5-EA0A7E1777CF,  
54879,4433265,220,NULL,128,142,3.3635,1.2668,02/07/2023 07:56
```

Spodziewamy się pliku z danymi rozdzielonymi przecinkami, ale możemy zauważyć, że po pierwszym globalnie unikatowym identyfikatorze (GUID) znajduje się średnik zamiast przecinka. Powoduje to błąd w przetwarzaniu wiersza ze względu na przycięcie danych w kolumnie `Cookie ID`. Wynika to stąd, że użyliśmy sugerowanego typu danych jako ciągu znaków o długości 50, podczas gdy rzeczywista długość kolumny (do pierwszego przecinka) wynosi 53 znaki.

WSKAZÓWKA W niektórych przypadkach warto również dodać ograniczenie niepowodzenia do miejsca docelowego źródła danych. W naszym scenariuszu skonfigurowaliśmy jednak tabelę przygotowawczą tak, że wszystkie kolumny mają typ `VARCHAR(MAX)`, co jest najbardziej pojemnym rozwiązaniem. Nie ma potrzeby stosowania źródła na wypadek niepowodzenia, ponieważ nie moglibyśmy uczynić jego kolumn bardziej pojemnymi.

Powinniśmy zawsze unikać odrzucania nieprawidłowych danych. Ignorowanie błędnych wierszy może być wygodne, ale jeśli dane są na tyle istotne, by je wczytywać, to powinny być również na tyle ważne, by je zachować, nawet jeśli późniejsza analiza wykaże, że nie da się ich naprawić. Jeśli jakaś konkretna kolumna nie jest potrzebna, lepiej w ogóle jej nie wczytywać. Zwiększa to ryzyko niepowodzenia całego procesu i sprawia, że wiersze są większe niż to konieczne, co może również negatywnie wpłynąć na wydajność. Omówimy to dokładniej dalej w tym rozdziale.

6.2. Numer 23 — nieoptymalizowanie wczytywania danych

W poprzednim podrozdziale, podczas tworzenia naszego przepływu danych, mogłeś zauważyć jeszcze jedną pomyłkę, mianowicie brak optymalizacji wczytywania danych. Bez tego nasz pakiet nigdy nie będzie działał optymalnie.

Aby zoptymalizować wczytywanie danych, należy skonfigurować właściwości przepływu danych określające rozmiar buforów, do których dane będą wczytywane podczas odczytu z pliku CSV. Następnie można wykorzystać opcje szybkiego wczytywania w miejscu docelowym OLE DB, aby zoptymalizować masowe wczytywanie danych do SQL Servera.

Optymalizacja pakietów w praktyce

Wielokrotnie zdarzały mi się sytuacje, w których optymalizacja wczytywania danych była absolutnie niezbędna. Typowym przykładem jest wczytywanie kwartalnych i rocznych danych finansowych z systemów księgowych do narzędzi raportowych, które musi zostać ukończone w określonym przedziale czasowym. Innym świetnym przykładem są moje doświadczenia z tą samą dziedziną danych, o której mówimy w tym rozdziale.

Odczytujemy tu pojedynczy plik płaski, który zawiera tylko milion wierszy. Przypominam sobie jednak pracę dla dużej grupy reklamowej. Musieliśmy codziennie wczytywać dane o wyświetleniach, kliknięciach i zdarzeniach, a następnie przeprowadzać złożoną analizę ścieżek kliknięć. Odczytywaliśmy wiele plików od różnych dostawców plików cookie. Każdej nocy trzeba było przetworzyć setki milionów odsłon, dziesiątki milionów kliknięć i miliony zdarzeń.

Niewielki rozmiar naszego przykładowego pliku nie pozwolił nam oszczędzić więcej niż kilka sekund. Jednak w rzeczywistych scenariuszach optymalizacja wczytywania danych pozwalała zaoszczędzić wiele minut. Było to kluczowe, ponieważ pliki często dostarczano dopiero po północy, a musiały być gotowe na początek dnia pracy. Procesy ETL trwały wiele godzin, a bez optymalizacji wczytywania przekraczałyby wyznaczone ramy czasowe, co zakłócałoby codzienną działalność firmy.

Właściwości przepływu danych, które kontrolują jego rozmiar, to `Default BufferMaxRows`, określająca maksymalną liczbę wierszy w buforze, oraz `Default BufferSize`, określająca maksymalny rozmiar bufora danych. Domyślnie maksymalna liczba wierszy wynosi 10 000, a maksymalny rozmiar to 1 MB. Rozmiar bufora będzie ograniczony przez limit, który zostanie osiągnięty jako pierwszy.

Może to być mylące, dlatego zalecam ustawienie właściwości `AutoAdjust BufferSize` na `True`. Dzięki temu nie będziesz musiał martwić się o właściwość `BufferSize`, ponieważ zostanie ona automatycznie skonfigurowana tak, aby odpowiadała wybranej maksymalnej liczbie wierszy w buforze.

Opcje szybkiego wczytywania w miejscu docelowym OLE DB pozwalają nam określić, czy dla kolumny powinna być wyłączona specyfikacja tożsamości, czy powinny być zachowane wartości NULL, czy powinny być wyłączone ograniczenia sprawdzające oraz czy powinna być używana blokada tabeli. Umożliwiają one również określenie maksymalnej liczby wierszy w partii oraz maksymalnego rozmiaru zatwierdzania dla operacji wstawiania.

Ponieważ ładujemy dane do tabeli przejściowej, która jest płaską strukturą bez kluczy, ograniczeń czy wartości NULL, możemy pominąć te ustawienia w naszym scenariuszu. Gdybyśmy jednak mieli takie ograniczenia, ich włączenie wiązałoby się z pewnym kosztem wydajnościowym. Zadbamy natomiast o zaznaczenie opcji blokady tabeli, co pozwoli uniknąć konfliktów i zmniejszy narzut związany z eskalacją blokad.

Konfiguracja maksymalnego rozmiaru partii i rozmiaru zatwierdzania to kluczowe parametry. Domyślnie dane są wczytywane do tabeli w ramach jednej transakcji. Może to powodować problemy z wydajnością ze względu na nadmierne

wykorzystanie bazy TempDB i zapewnianie dziennika transakcji. Zakładamy, że odczyt danych z pliku płaskiego będzie wolniejszy niż wstawianie danych do tabeli tymczasowej, dlatego ustawimy maksymalny rozmiar partii i zatwierdzania na poziomie odpowiednim do liczby wierszy w naszym źródle danych.

Niestety nie ma „magicznej” liczby wierszy, która powinna znajdować się w każdym buforze czy w każdej partii danych. Optymalne wartości będą ściśle zależeć od specyfiki Twojego środowiska, dostępnych zasobów, charakterystyki obciążenia oraz profilu przetwarzanych danych. Aby znaleźć najlepsze wartości, należy przetestować przepływ danych z różnymi opcjami rozmiaru.

WSKAZÓWKA Gdy wykonuję tę operację, zazwyczaj zaczynam od małej wartości i stopniowo ją zwiększam. Zwykle prowadzi to do stopniowej poprawy wydajności, która z czasem zaczyna się stabilizować. Kiedy czasy wykonania zaczynają rosnąć, wiem, że osiągnąłem optymalny rozmiar.

Tabela 6.1 zawiera wyniki różnych testów, które przeprowadziłem na przepływie danych. Pierwsza kolumna przedstawia maksymalną liczbę wierszy w buforze, która odpowiada maksymalnej liczbie wierszy na partię. Druga kolumna pokazuje czas wykonania pakietu w sekundach. Te czasy wykonania można odczytać z karty *Progress* przed zatrzymaniem wykonywania lub z karty *Execution Results* po zakończeniu procesu.

Tabela 6.1. Testy wydajności dla różnych wielkości bufora

| Rozmiar bufora/partii | Czas wykonania (sekundy) |
|-----------------------|--------------------------|
| 1000 | 49,032 |
| 5000 | 36,828 |
| 10 000 | 35,813 |
| 20 000 | 34,875 |
| 50 000 | 34,328 |
| 75 000 | 35,125 |

WSKAZÓWKA Jeśli chcesz przeprowadzić testy we własnym środowisku, pamiętaj, aby za każdym razem opróżnić tabelę przygotowawczą, aby zachować rzetelność testu. Pamiętaj też, że wyniki mogą się różnić w zależności od specyfikacji Twojego komputera i innych uruchomionych procesów.

W moim środowisku wyraźnie widać, że optymalny rozmiar bufora to 50 000 wierszy. Warto jednak sprawdzić hipotezę, że to źródło danych w postaci płaskiego pliku może być czynnikiem ograniczającym wydajność. Dlatego przeprowadźmy dodatkowe testy. Tym razem zachowamy stałą wartość `MaximumRowsPerBuffer` równą 50 000 dla każdego testu, ale będziemy zmieniać rozmiar partii. Wyniki tych testów znajdują się w tabeli 6.2. Pamiętaj jednak, że rezultaty są specyficzne dla konkretnego środowiska, więc w Twoim przypadku mogą być inne.

Tabela 6.2. Testy wydajności dla różnych rozmiarów partii

| Rozmiar partii | Czas wykonania (sekundy) |
|----------------|--------------------------|
| 25 000 | 35,125 |
| 75 000 | 34,962 |

Zgodnie z oczekiwaniami wyniki pokazują, że odczyt pliku płaskiego jest czynnikiem ograniczającym i nie możemy uzyskać poprawy wydajności poprzez usunięcie dopasowania rozmiaru partii do rozmiaru bufora.

Zawsze powinniśmy optymalizować wydajność wczytywania danych. Nieoptymalna wydajność może spowodować, że pakiety przekroczą okna czasowe ETL. Może to mieć wpływ na działalność biznesową lub procesy konserwacji, takie jak tworzenie kopii zapasowych.

6.3. Numer 24 — używanie SSIS jako narzędzia do koordynacji T-SQL

Częstą pomyłką popełnianą przez początkujących programistów SSIS jest unikanie przepływów danych i wczytywanie lub przetwarzanie danych między tabelami z użyciem zadań *Execute T-SQL* w przepływie sterowania.

Takie podejście sprawia, że programiści nie wykorzystują w pełni możliwości SSIS. Oznacza to, że wykonujemy jedynie standardowe kwerendy T-SQL. W rezultacie proces ETL nie będzie luźno powiązany ze źródłami danych, przyszli programiści ETL, którzy przejmą nasze zadania, nie będą mieli graficznej reprezentacji logiki, a my nie będziemy mogli skorzystać z obsługi błędów i rejestrowania zdarzeń oferowanych przez SSIS. Błędny wiersz spowoduje niepowodzenie całego zapytania, zamiast umożliwić przekierowanie pojedynczego problematycznego rekordu. Ponadto stracimy możliwość śledzenia danych w potoku, co utrudni diagnozowanie problemów z wydajnością.

Wydajność usług SSIS

Warto zaznaczyć, że jeśli tabele źródłowe i docelowe znajdują się w tym samym serwerze, wykorzystanie natywnego T-SQL będzie prawdopodobnie wydajniejsze niż transformacja danych przy użyciu potoków przepływu danych. Jest to spowodowane tym, że T-SQL lepiej radzi sobie z operacjami takimi jak łączenie i scalanie danych. Dlatego w przypadku niektórych dużych wczytywań, gdzie mamy do czynienia z pojedynczym serwerem bazy danych, natywny T-SQL może być właściwym podejściem, szczególnie gdy mamy ograniczone okno czasowe na proces ETL. Powinniśmy zawsze dobierać odpowiednie narzędzie do konkretnego zadania.

Można argumentować, że w takim scenariuszu korzyści z używania usług SSIS mogą być niewielkie w porównaniu z narzutem związanym z ich uruchamianiem. Lepszym rozwiązaniem może być wykorzystanie zadania SQL Server Agent i użycie etapów zadania do koordynowania przepływu pracy. Jednak gdy chodzi o wczytywanie i przekształcanie danych między serwerami, usługi SSIS zwykle zapewniają lepszą wydajność niż połączony serwer.

Warto również rozważyć kompromis między wydajnością a funkcjonalnością. Jeśli nie mamy ograniczeń czasowych dotyczących okna ETL, możemy uznać, że narzędzia do obsługi błędów, rejestrowania zdarzeń i debugowania dostępne w SSIS są ważniejsze niż sama szybkość przetwarzania.

Aby zbadać efekty wykorzystania SSIS wyłącznie jako narzędzia do koordynacji, rozbudujemy nasz pakiet o transformację danych z tabeli przygotowawczej do głównej tabeli marketingowej, a następnie do tabeli raportowej. Na koniec nasz pakiet powinien opróżnić tabelę przygotowawczą. W kolejnym punkcie wyjaśnię, jak utworzyć te zadania z użyciem przepływu sterowania jako narzędzia do koordynacji zadań *Execute T-SQL Statement*.

6.3.1. Koordynowanie zadań *Execute T-SQL Statement*

Pierwszym krokiem w ulepszaniu naszego pakietu będzie utworzenie nowego menedżera połączeń ADO. Ten menedżer połączeń zostanie wykorzystany przez nasze zadania *Execute T-SQL Statement* do łączenia się z instancją SQL Servera.

Gdy mamy już menedżer połączeń, możemy utworzyć zadania wykonujące nasze zapytania SQL. Najpierw powinniśmy skonfigurować nowe zadanie *Execute T-SQL Statement* tak, aby wykonywało kwerendę z listingu 6.3. Kwerenda ta wykonuje instrukcję `INSERT`, konwertując dane w przypadkach, gdy SQL Server nie może wykonać niejawnej konwersji. Instrukcja `SET DATEFORMAT` na początku jest niezbędna, aby SQL Server poprawnie rozpoznawał daty w kolumnie `EventTime`. Bez niej zadanie zakończyłoby się niepowodzeniem z powodu błędów uzupełnienia typu danych.

Listing 6.3. Wczytywanie danych do tabeli Impressions

```
SET DATEFORMAT DMY
INSERT INTO marketing.Impressions (
    ImpressionUID,
    ReferralURL,
    CookieID,
    CampaignID,
    RenderingID,
    CountryCode,
    StateID,
    BrowserVersion,
    OperatingSystemID,
    BidPrice,
```

```

        CostPerMille,
        EventTime
    )
SELECT
    ImpressionUID
    , ReferralURL
    , CookieID
    , CampaignID
    , RenderingID
    , CASE
        WHEN CountryCode = 'NULL' THEN NULL
        ELSE CountryCode
    END CountryCode
    , CASE
        WHEN StateID = 'NULL' THEN NULL
        ELSE StateID
    END StateID
    , BrowserVersion
    , OperatingSystemID
    , CAST(BidPrice AS MONEY)
    , CAST(CostPerMille AS MONEY)
    , EventTime
FROM staging.ImpressionsStage ;

```

Kolejne zadanie *Execute T-SQL Statement*, które utworzymy, nosi nazwę *Merge Aggregates* i służy do wypełnienia tabeli `reporting.ImpressionAggregates`. W tym zadaniu powinniśmy użyć kwerendy z listingu 6.4. Kwerenda ta wykorzystuje instrukcję `MERGE` do agregowania danych według `CampaignID`, `CountryCode` i `Event` ↪ `Date` przed wczytaniem ich do tabeli zbiorczej. Jeśli wiersz dla danej kampanii, kraju i daty już istnieje, zostaną zaktualizowane średnia cena oferty i średni CPM. W przeciwnym razie zostanie wstawiony nowy wiersz.

Listing 6.4. Wypełnianie tabeli Aggregates

```

MERGE INTO reporting.ImpressionAggregates AS Target
USING (
    SELECT
        CampaignID
        , CountryCode
        , AVG(BidPrice) AS AvgBidPrice
        , AVG(CostPerMille) AS AvgCostPerMille
        , CAST(EventTime as DATE) AS EventDate
    FROM marketing.Impressions
    GROUP BY
        CampaignID
        , CountryCode
        , CAST(EventTime as DATE)
) AS source
ON (
    Source.CampaignID = Target.CampaignID
    AND Source.CountryCode = Target.CountryCode
    AND Source.EventDate = Target.EventDate
)
WHEN MATCHED THEN
    UPDATE SET

```

```

        AvgBidPrice = Source.AvgBidPrice
        , AvgCostPerMille = Source.AvgCostPerMille
WHEN NOT MATCHED THEN
    INSERT (
        CampaignID,
        CountryCode,
        EventDate,
        AvgBidPrice,
        AvgCostPerMille
    )
VALUES (
    Source.CampaignID,
    Source.CountryCode,
    Source.EventDate,
    Source.AvgBidPrice,
    Source.AvgCostPerMille
) ;

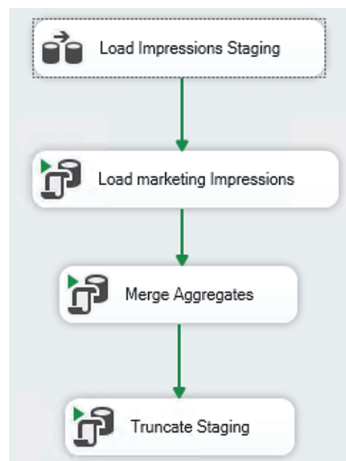
```

Wykonanie naszego ostatniego zadania spowoduje opróżnienie tabeli przygotowawczej po zakończeniu wczytywania, z wykorzystaniem w tym celu instrukcji z listingu 6.5. Opróżnianie tabeli jest całkowicie poprawnym zastosowaniem zadania *Execute T-SQL Statement*. Operacji tej nie można wykonać w ramach przepływu danych. Zachowamy to zadanie nawet po zoptymalizowaniu pakietu i zastąpieniu zadań *Execute T-SQL Statement* przepływami danych.

Listing 6.5. Opróżnianie tabeli pośredniej

```
TRUNCATE TABLE staging.ImpressionsStage ;
```

Wszystkie nasze zadania powinny być połączone ograniczeniami powodzenia. Wymusi to sekwencyjne wykonywanie zadań. Oznacza to również, że jeśli jedno z zadań zakończy się niepowodzeniem, kolejne zadania nie zostaną uruchomione. Rysunek 6.7 ilustruje, jak będzie wyglądał nasz przepływ sterowania na tym etapie.



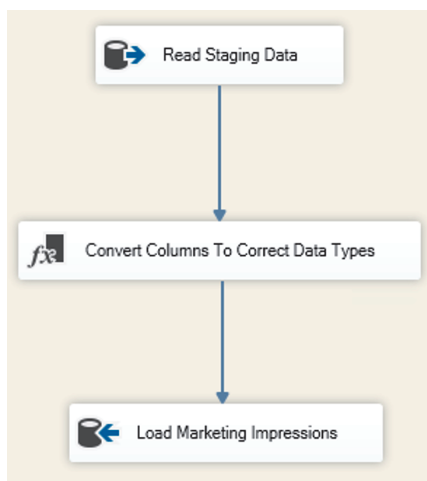
Rysunek 6.7. Pakiet używany do koordynowania zadań T-SQL

Pakiet jest teraz kompletny, ale nie jest najlepszym rozwiązaniem do wczytywania danych, ponieważ w tym przypadku dane pochodzą z niezaufanego źródła i prawdopodobnie są zanieczyszczone. Wyobraźmy sobie, że wczytaliśmy niepoprawny wiersz do naszej tabeli przejściowej. Dzięki elastycznym kolumnom w tej tabeli udało nam się doprowadzić dane do tego etapu, ale gdy spróbujemy zapisać je w tabeli `marketing.Impressions`, operacja wstawiania wiersza zakończy się niepowodzeniem z powodu błędu konwersji typów danych. W takiej sytuacji niepowodzeniem zakończy się całe zadanie. Trudno będzie też znaleźć przyczynę błędu, ponieważ SQL Server nie wskaże konkretnego wiersza, który spowodował problem.

Jak zatem utworzyć lepszy pakiet? Otóż należy użyć zadań przepływu danych. Aby to zrobić, usuńmy zadania *Load Marketing Impressions* i *Merge Aggregate*. W kolejnym punkcie wyjaśnię, jak zastąpić je przepływami danych.

6.3.2. Przekształcanie zadań Execute T-SQL Statement w przepływy danych

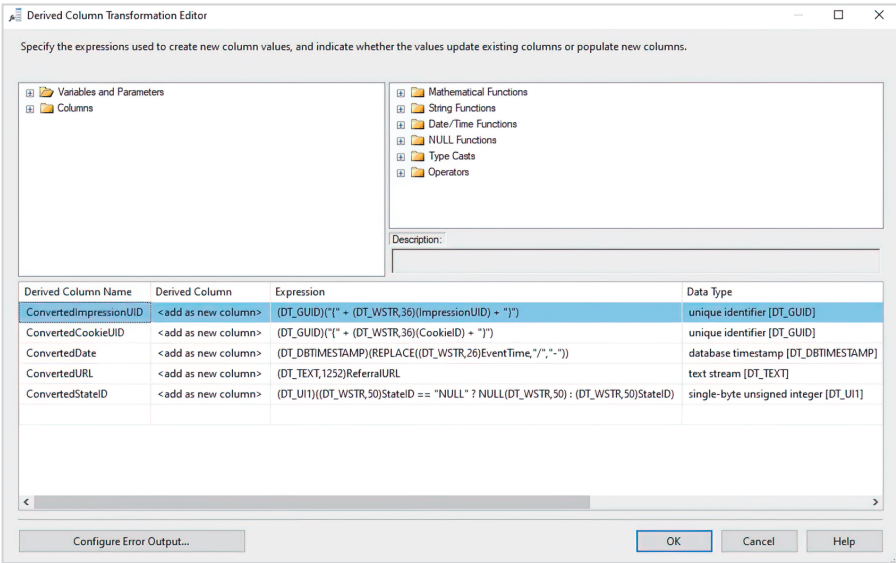
Przepływ danych *Load Marketing Impressions* będzie wyglądał podobnie do przepływu zilustrowanego na rysunku 6.8. Mamy tu źródło, które odczytuje dane z tabeli przygotowawczej. Następnie znajduje się transformacja kolumn pochodnych, która przekształca kolumny w odpowiednie typy danych. Na koniec miejsce docelowe przepływu danych wstawia wiersze do tabeli `marketing.Impressions`.



Rysunek 6.8. Przepływ danych *Load Marketing Impressions*

Źródło i miejsce docelowe to proste komponenty; cała magia dzieje się w transformacji kolumny pochodnej. Na rysunku 6.9 zauważysz, że użyliśmy *wyrażeń*

SSIS, które są kombinacją kolumn, funkcji, operatorów i literalów, aby utworzyć nowe kolumny wyjściowe zawierające przekształcone wartości. *Derived Column Name* to pole tekstowe, w którym możemy określić nazwę kolumny wyjściowej. Pole *Derived Column* to lista rozwijana, która pozwala nam wybrać nazwę kolumny do zastąpienia lub dodać wynik jako nową kolumnę. Pole *Expression* można wypełnić całkowicie dowolnie. Alternatywnie, nazwy kolumn, zmiennych i parametrów można przeciągać z obszaru w lewym górnym rogu okna dialogowego. Obszar w prawym górnym rogu umożliwia przeciąganie funkcji do pola *Expression*. Pole *Data Type* jest wypełniane automatycznie na podstawie wyrażenia. Po prawej stronie znajdują się również inne pola, które są automatycznie wypełniane w zależności od typu danych wyjściowych. Są to pola takie jak *Length*, *Precision*, *Scale* i *Code Page*. Jeśli w wyrażeniu wystąpi błąd, zostanie ono podświetlone na czerwono, a wskazanie wyrażenia myszą spowoduje wyświetlenie szczegółów błędu parsowania.



Rysunek 6.9. Transformacja kolumny pochodnej

Pierwsze dwa wyrażenia służą do konwersji wartości GUID. Aby SSIS poprawnie rozpoznał te wartości jako identyfikatory GUID, muszą one być ujęte w nawiasy klamrowe. W tym celu jawnie konwertujemy kolumnę wejściową na 36-znakowy łańcuch Unicode i dodajemy nawiasy klamrowe na początku i końcu. Następnie przekształcamy całą wartość na typ danych DT_GUID.

Wyrażenie konwertujące kolumnę EventTime najpierw przekształca dane wejściowe na 26-znakowy ciąg tekstowy, zamieniając znaki / na -. Jest to konieczne, ponieważ SSIS nie rozpoznaje formatu daty używanego w tabeli przygotowawczej. Na końcu wynik jest rzutowany na typ danych DB_TIMESTAMP.

Wyrażenie służące do konwersji adresu URL wykonuje po prostu jawne rzutowanie na łańcuch znaków przy użyciu strony kodowej 1252. Ta strona kodowa jest synonimem strony kodowej Windows 1252.

Najbardziej interesujące jest prawdopodobnie wyrażenie do konwersji kolumny StateID. Używamy go nie tylko do zmiany typu danych, ale także do zastąpienia ciągu znaków "NULL" wartością NULL. Dla osób z doświadczeniem w .NET składnia może wydawać się znajoma, ale aby zrozumieć to wyrażenie, rozbijmy je na części. Pierwsza część znajduje się po prawej stronie znaku : (dwukropka) i przekazuje StateID jako kolumnę wejściową po jawnym rzutowaniu na 50-znakowy łańcuch. Po lewej stronie znaku : sprawdzamy, czy wartość w StateID (która znowu musi być jawnie rzutowana) jest równa "NULL". Jeśli tak, zastępujemy tę wartość wartością NULL. Występuje tu kolejne rzutowanie, ponieważ w SSIS istnieje inny typ wartości NULL dla każdego typu danych. Na przykład NULL(DT_WSTR,50) to inny typ niż NULL(DT_14). Wreszcie, poza nawiasami po lewej stronie, konwertujemy ostateczną wartość na DT_UI1, czyli jednobajtową liczbę całkowitą bez znaku.

Wyrażenia użyte w tej transformacji zostały szczegółowo opisane na listingu 6.6.

Listing 6.6. Wyrażenia SSIS

```
(DT_GUID)("{ " + (DT_WSTR,36)(ImpressionUID) + "}") ← ConvertedImpressionUID
(DT_GUID)("{ " + (DT_WSTR,36)(CookieID) + "}") ← ConvertedCookieUID
(DT_DBTIMESTAMP)(REPLACE((DT_WSTR,26)EventTime,"/","-")) ← ConvertedDate
(DT_TEXT,1252)ReferralURL ← ConvertedURL
(DT_UI1)((DT_WSTR,50)StateID == "NULL" ? NULL(DT_WSTR,50) :
(DT_WSTR,50)StateID) ← ConvertedStateID
```

Największą zaletą tego przepływu danych w porównaniu z odpowiadającym mu zadaniem *Execute T-SQL Statement* jest możliwość obsługi błędów. Jak omówiono wcześniej w tym rozdziale, moglibyśmy ulepszyć ten przepływ danych, dodając ścieżki obsługi błędów, dzięki czemu niepoprawne wiersze mogłyby być przekierowywane do tabeli błędów danych. Pozwoliłoby to zespołowi wsparcia aplikacji na bieżąco zajmować się problematycznymi wierszami, których nie można przekonwertować na bardziej restrykcyjne typy danych. W przypadku zadania *Execute T-SQL Statement* cała operacja wstawiania zakończyłaby się niepowodzeniem i wymagałaby debugowania oraz naprawy danych, zanim jakiegokolwiek dane byłyby dostępne dla biznesu.

Przepływ danych *Merge Aggregates* będzie wyglądał mniej więcej tak, jak na rysunku 6.10. Przepływ ten zaczyna się od źródła danych, które odczytuje dane o odsłonach z tabeli marketing.Impressions. Dalej następuje transformacja kolumny pochodnej, która agreguje dane dotyczące daty i czasu do zaokrąglonych dat. Transformacja agregująca oblicza następnie średnią cenę oferty i CPM. Na końcu znajduje się miejsce docelowe typu upsert, które wstawia wiersze do tabeli reporting.ImpressionAggregates, jeśli dana kombinacja kluczy jeszcze nie istnieje, lub aktualizuje zagregowane wartości, jeśli taka kombinacja już istnieje.

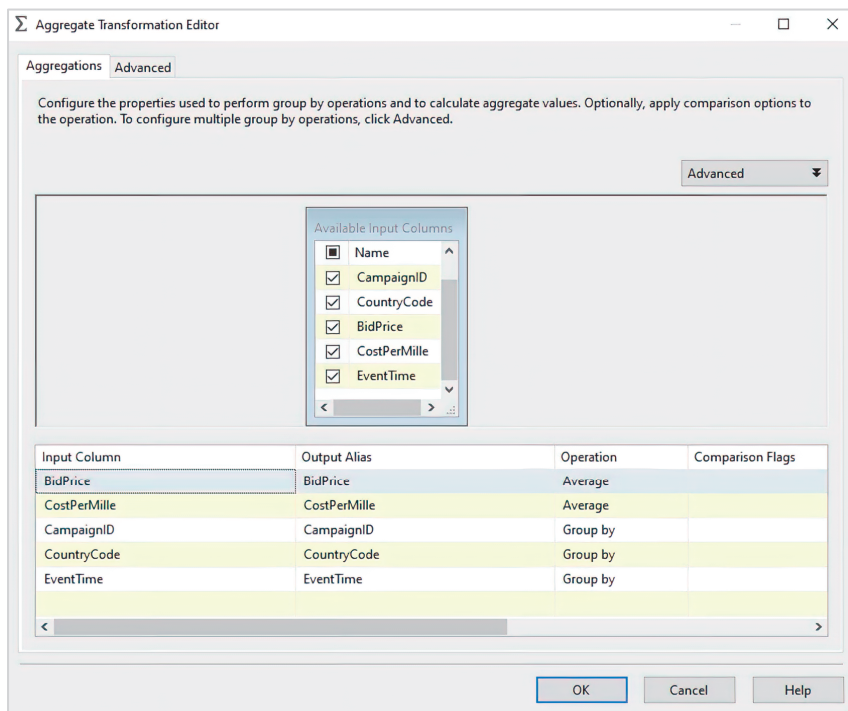


Rysunek 6.10. Przepływ danych Merge Aggregates

Przepływy danych z transformacjami są często znacznie bardziej złożone niż w tym przykładzie. W przypadku skomplikowanych przekształceń przepływ danych może mieć wiele zalet w porównaniu z zadaniem *Execute T-SQL Statement*. Nie tylko możemy przekierować pojedyncze wiersze w przypadku błędu, ale zyskujemy też graficzny widok naszej logiki. Może to pomóc w wyjaśnieniu logiki biznesowej innym programistom. Możemy też obserwować wykonanie naszego przepływu w czasie rzeczywistym, a dzięki temu zrozumieć, gdzie występują błędy lub gdzie dochodzi do spadku wydajności. Może to pomóc w procesie tworzenia oprogramowania.

Źródło *Read Marketing Impressions* to prosty odczyt z tabeli bazowej. Transformacja *Rollup DateTime To Date* to transformacja kolumny pochodnej, która generuje nową kolumnę wyjściową z użyciem wyrażenia SSIS (`DT_DBTIMESTAMP`) \hookrightarrow (`DT_DBDATE`) `EventTime`. Wyrażenie to najpierw przekształca wartość w typ `DT_DBDATE`, co usuwa składnik godzinowy. Dla ułatwienia dopasowania typów danych wartość jest następnie natychmiast przekształcana z powrotem na typ `DT_DBTIMESTAMP`, co dodaje składnik godzinowy, ale ponieważ informacja o godzinie została wcześniej usunięta, zostanie ona ustawiona na 00:00:00.000.

Transformacja *Aggregate Impression Costs* wykorzystuje operację agregacji do obliczenia średnich wartości `BidPrice` i `CostPerMille` pogrupowanych według kolumn `CampaignID`, `CountryCode` oraz `EventTime`. Na rysunku 6.11 pokazano stronę *Aggregations* w edytorze transformacji agregacyjnych. Jak widać, w górnej części okna dialogowego wybraliśmy odpowiednie kolumny wejściowe do transformacji, a w dolnej części określiliśmy rodzaj agregacji, którą chcemy zastosować. Dla miar wybraliśmy średnią (*Average*), a dla kolumn kluczowych opcję grupowania (*Group by*). Spowoduje to wygenerowanie różnych średnich danych kosztowych dla każdej unikatowej kombinacji kolumn `CampaignID`, `CountryCode` i `EventTime`.



Rysunek 6.11. Edytor transformacji agregacyjnych

WSKAZÓWKA Inne dostępne funkcje agregujące to *Count* (zliczanie), *Count Distinct* (zliczanie unikatowych wartości), *Sum* (sumowanie), *Minimum* (znajdowanie wartości minimalnej) i *Maximum* (znajdowanie wartości maksymalnej).

Transformacje blokujące

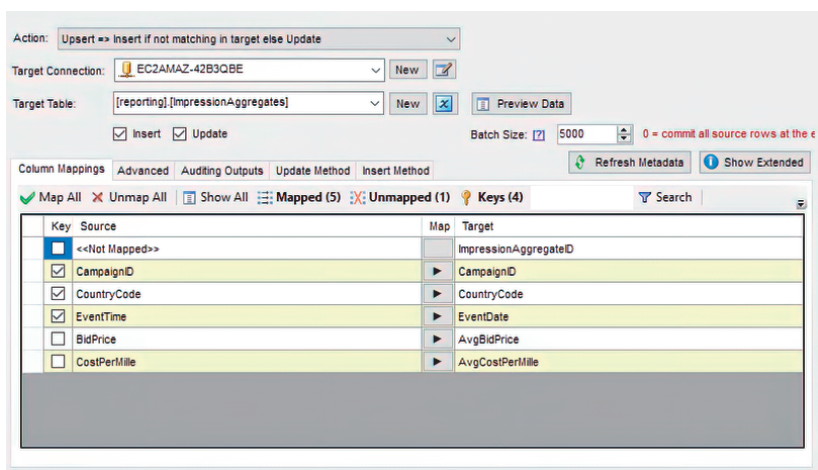
Warto zaznaczyć, że transformacje w SSIS mogą być blokujące lub nieblokujące. Transformacje *nieblokujące*, takie jak transformacja kolumny pochodnej, będą wyprowadzać wiersze należące do partii zaraz po ich przetworzeniu. Inne transformacje natomiast nie wyprowadzają żadnych danych wyjściowych, dopóki wszystkie wiersze nie zostaną przetworzone. Są one znane jako transformacje *blokujące*, ponieważ komponenty znajdujące się dalej w strumieniu danych nie mogą rozpocząć pracy, dopóki transformacja blokująca nie zakończy przetwarzania. Transformacja agregacyjna jest transformacją blokującą, ponieważ nie może potwierdzić ostatecznych wartości zagregowanych, dopóki nie otrzyma wszystkich wierszy. W przeciwnym razie mogłaby przetwarzać i zwracać niepoprawne agregacje, gdyż wartości te prawdopodobnie zmieniłyby się po otrzymaniu kolejnych wierszy z tymi samymi kluczami grupującymi.

Miejsce docelowe typu upsert nie jest standardowym komponentem SSIS. Chociaż istnieją różne metody osiągnięcia podobnego rezultatu przy użyciu natywnych komponentów SSIS, to nie są zalecane, ponieważ często cechują się bardzo niską wydajnością. Dostępne są różne rozwiązania do operacji upsert,

w tym oferty firm COZYROC i SentryOne, ale w tym rozdziale skupimy się na komponencie firmy ZappySys, który jest częścią pakietu SSIS PowerPack. Można go nabyć pod adresem <https://mng.bz/2gVd>.

Na stronie *Settings*, w oknie dialogowym *Upsert Destination*, wybieramy operację, którą chcemy wykonać. W naszym przypadku jest to operacja upsert, ale komponent obsługuje również synchronizację (upsert + usuwanie), hurtową aktualizację i hurtowe usuwanie. Następnie wybieramy źródło danych, którego chcemy użyć, oraz tabelę docelową. Możemy także dostosować rozmiar partii w celu optymalizacji wydajności.

Na karcie *Column Mappings* okna dialogowego *Upsert Destination*, pokazanej na rysunku 6.12, odwzorowujemy wszystkie kolumny źródłowe i zaznaczamy opcję *Key* dla kolumn *CampaignID*, *CountryCode* i *EventTime*. Kolumny kluczowe określają klucze biznesowe, które będą używane do dopasowywania wierszy podczas decydowania, czy należy wstawić wiersz, czy dokonać aktualizacji.



Rysunek 6.12. Okno dialogowe *Upsert Destination* — karta *Column Mappings*

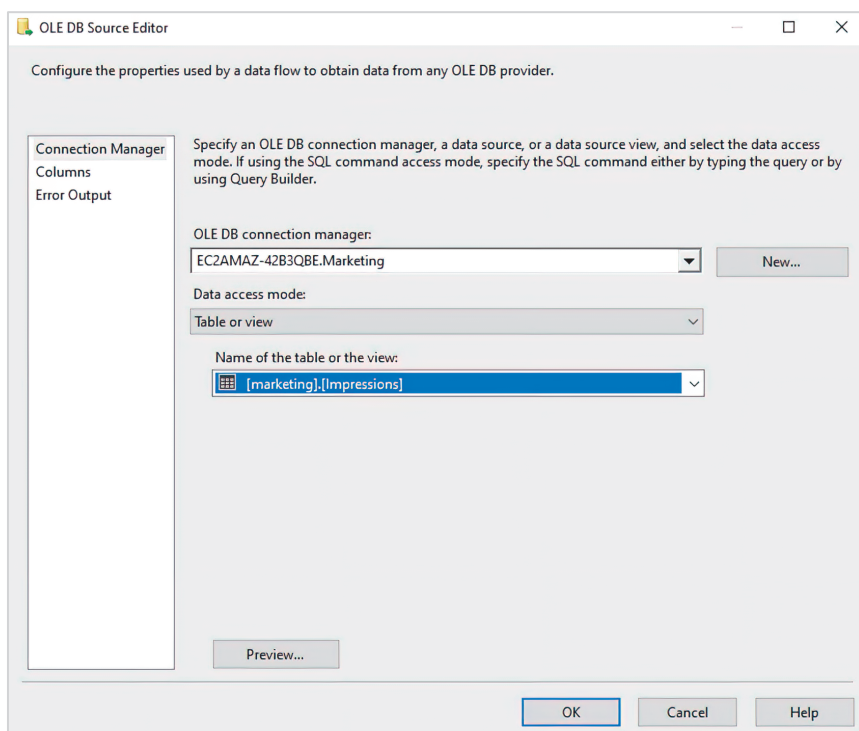
Na stronie *Advanced* okna dialogowego możemy skonfigurować polecenia, które zostaną wykonane przed operacją wstawiania danych i po niej. Może to być przydatne do tworzenia i usuwania indeksów wspierających wczytywanie danych. Można tu również określić wskazówki dla bazy danych, takie jak nałożenie blokady na tabelę.

Przy wykonywaniu operacji ETL między serwerami korzystanie z przepływów danych jest zazwyczaj lepszym rozwiązaniem niż używanie SSIS jako narzędzia do koordynacji skryptów T-SQL. Gdy wykonujemy operacje ETL między tabelami w tym samym serwerze, SSIS może działać wolniej niż skrypt T-SQL. Może to jednak nadal być lepsza opcja, jeśli potrzebujemy gotowych funkcji rejestrowania zdarzeń, obsługi błędów i debugowania, a także możliwości odrzucania pojedynczych wierszy z powodu występujących w nich błędów.

6.4. Numer 25 — pobieranie wszystkich danych, kiedy potrzebny jest tylko ich podzbiór

Podczas budowania przepływu danych kuszące może być skonfigurowanie źródła danych tak, żeby odczytywało wszystkie dane z tabeli źródłowej. Pobieranie całej tabeli jest opcją domyślną. Wystarczy wybrać naszą tabelę z rozwijanej listy i gotowe.

W niektórych przypadkach rzeczywiście musimy pobrać i przetworzyć wszystkie dane z tabeli. Jeśli tak jest, nie ma nic złego w użyciu tej opcji. Jednak jeśli potrzebujemy tylko części danych z tabeli, takie podejście może drastycznie obniżyć wydajność. Weźmy na przykład nasz przepływ danych *Merge Aggregates*. W poprzednim podrozdziale skonfigurowaliśmy źródło danych tak, aby po prostu pobierało wszystkie dane z tabeli *marketing.Impressions*, co pokazano na rysunku 6.13.



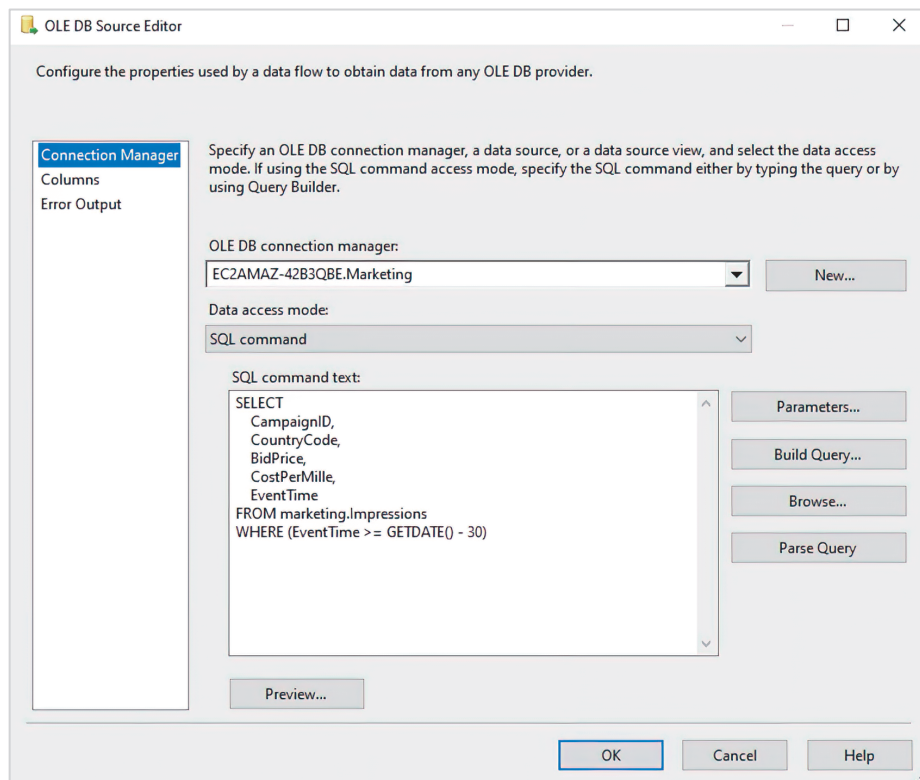
Rysunek 6.13. Pierwotna konfiguracja źródła danych

Wykonanie tego przepływu danych na moim stanowisku testowym zajmuje 16,403 sekundy.

Jeśli jednak przyjrzymy się przepływowi danych, zauważymy, że jedyne kolumny potrzebne w tym procesie to CampaignID, CountryCode, EventTime, BidPrice i CostPerMille. Gdybyśmy usunęli pozostałe, zbędne kolumny, buforby byłyby mniejsze i bardziej wydajne.

Wyobraźmy sobie również, że istnieje reguła biznesowa, zgodnie z którą dane o przeszłych odsłonach są dostarczane tylko za ostatnie 30 dni. Oznacza to, że dane starsze niż 30 dni nie ulegną zmianie, więc nie ma potrzeby ponownego przetwarzania zagregowanych wartości. Możemy zmniejszyć liczbę buforów danych, które przepływ danych musi przetworzyć, filtrując dane u źródła.

Aby to osiągnąć, zmodyfikujemy nasze źródło przepływu danych tak, aby czytywało dane za pomocą polecenia, zamiast pobierać wszystkie dane z tabeli. Zmodyfikowany przepływ danych przedstawiono na rysunku 6.14.



Rysunek 6.14. Nowa konfiguracja przepływu danych

Kwerendę wykorzystaną w tym źródle przepływu danych można znaleźć na listingu 6.7. Kwerenda pobiera tylko wymagane kolumny i filtruje dane tak, aby do przepływu trafiły jedynie odsłony z czasem zdarzenia z ostatnich 30 dni.

Listing 6.7. Wczytywanie danych dotyczących odsłon

```
SELECT
    CampaignID,
    CountryCode,
    BidPrice,
    CostPerMille,
    EventTime
FROM marketing.Impressions
WHERE (EventTime >= DATEADD(day, -30, GETDATE()));
```

Proces, po uruchomieniu przepływu danych na moim stanowisku testowym z nową konfiguracją, zakończył się w ciągu 2,937 sekundy. Jeśli mamy ograniczone okno czasowe na ETL, takie usprawnienia wydajności, zastosowane w całym naszym systemie, mogą sprawić ogromną różnicę.

WSKAZÓWKA Ponieważ filtr opiera się na dacie, to jeśli wykonujesz ten przykład w przyszłości, prawdopodobnie nie otrzymasz żadnych wyników. Możesz to rozwiązać na dwa sposoby: zaktualizować czasy zdarzeń w swojej tabeli bazowej lub zmodyfikować warunek WHERE w źródłowej kwerendzie przepływu danych.

Jeśli nie potrzebujemy wszystkich danych ze źródła, powinniśmy je przefiltrować u źródła, aby zmniejszyć rozmiar i liczbę buforów danych wymaganych w naszym przepływie.

Podsumowanie

- Podczas wczytywanie danych z zewnętrznego źródła powinniśmy starać się zachować jak najwięcej informacji. Warto rozważyć użycie tabel przygotowawczych z elastycznymi typami kolumn oraz wykorzystać funkcję przekierowania wierszy, aby przenieść nieprawidłowe dane do tabeli obsługi błędów.
- Dostosowanie ustawień przepływu danych, takich jak `MaximumRowsPerBuffer`, może mieć duży wpływ na wydajność systemu. Jest to szczególnie istotne w przypadku, gdy mamy ograniczone okno czasowe na wykonanie zadań ETL.
- Podczas wczytywania danych do SQL Servera w miarę możliwości używaj źródeł i miejsc docelowych OLE DB w celu zwiększenia wydajności. Zawsze dostosowuj opcje szybkiego wczytywania OLE DB, aby zoptymalizować wydajność w swoim środowisku.
- Nie ma „magicznej”, idealnej konfiguracji. Przetestuj różne warianty ustawień, aby sprawdzić, które z nich zapewniają najlepszą wydajność w Twoim środowisku.

- Unikaj używania SSIS jako narzędzia do koordynacji zadań *Execute T-SQL*. Wspomniane podejście powoduje utratę wielu korzyści wynikających z zastosowania SSIS, takich jak wbudowane mechanizmy rejestrowania zdarzeń i obsługi błędów.
- Unikaj pobierania wszystkich danych z tabeli, chyba że jest to naprawdę konieczne. Można znacznie poprawić wydajność pakietu, ograniczając liczbę zwracanych kolumn i wierszy do tych, które są faktycznie potrzebne w przepływie danych. Jeśli nie potrzebujesz wszystkich danych z tabeli, użyj źródła przepływu danych z poleceniem SQL i filtruj dane u źródła.

Obsługa błędów, testowanie, kontrola wersji i wdrażanie

W tym rozdziale:

- Obsługa błędów w T-SQL
- Rozwiązywanie problemów z kodem
- Testowanie wydajności
- Nowoczesne praktyki programistyczne

Gdy myślimy o roli programisty baz danych, łatwo skupić się wyłącznie na kwestii pisania wydajnego kodu T-SQL. W rzeczywistości jednak współczesny programista baz danych musi brać pod uwagę wiele innych aspektów. Jednym z najważniejszych jest obsługa błędów. Jeśli procedura zgłosi błąd w środowisku produkcyjnym, chcemy, aby został on obsłużony w sposób kontrolowany, aby zminimalizować ryzyko niespójności danych. Odpowiednia obsługa błędów może nawet spowodować, że kod podejmie ponowną próbę wykonania operacji, bez interwencji zespołu wsparcia aplikacji. Oczywiście chcemy też być powiadamiani o wystąpieniu błędów, aby móc je zbadać i rozwiązać.

Musimy także umieć efektywnie debugować błędy w kodzie. Pojawienie się błędów w kodzie jest nieuniknione — to fakt, z którym trzeba się pogodzić. Kiedy w naszym kodzie występują usterki, musimy być w stanie je sprawnie

zlokalizować i naprawić. Nie ma sensu pisać dużego fragmentu kodu w ciągu jednego dnia, jeśli potem trzeba spędzić tydzień na jego debugowaniu. Dlatego zrozumienie metodyki debugowania jest kluczowe w pracy ze złożonym kodem.

Częstym błędem popełnianym przez programistów SQL jest brak testowania. Przyjrzymy się testom jednostkowym w ramach szerszego spojrzenia na nowoczesne praktyki programistyczne stosowane (lub nie) przez deweloperów. Omówimy zalety przechowywania kodu w systemie kontroli wersji, pisania testów jednostkowych oraz korzystania z automatycznego procesu budowania i wdrażania. W rozdziale 4. utworzyliśmy bazę danych o nazwie *MagicChoc*, a w rozdziale 6. bazę *Marketing*. W tym rozdziale będziemy korzystać z obu tych baz danych.

Współczesne praktyki programistyczne wymagają przechowywania kodu w systemach kontroli wersji oraz stosowania procesów DevOps do wdrażania oprogramowania. Takie podejście jest standardem od wielu lat, jednak programiści SQL przekonują się do niego dość wolno. Spotkałem wiele zespołów pracujących z SQL Serverem, które nadal trzymają swój „kod źródłowy” w kopiach zapasowych baz danych i ręcznie uruchamiają skomplikowane, pieczołowicie tworzone skrypty do wdrażania aplikacji bazodanowych. W tym rozdziale przyjrzymy się bliżej tym dwóm błędnym praktykom.

Narzędzia programistyczne

Do tej pory w tej książce, z wyjątkiem części poświęconej SQL Server Integration Services (SSIS), korzystałem z SQL Server Management Studio (SSMS) do tworzenia przykładów. Przypuszczam, że Ty również używasz SSMS, aby samodzielnie wykonywać przykłady. To narzędzie jest dobrze znane specjalistom SQL Server i bardzo łatwe w obsłudze.

Istnieje kilka innych narzędzi, które możemy wykorzystać do tworzenia kodu T-SQL, takich jak Visual Studio Code, SQL Server Data Tools (SSDT) czy Azure Data Studio. Niektóre zaawansowane techniki programowania wymagają wykorzystania tych dodatkowych narzędzi deweloperskich. W przypadku pomyłek od numeru 28 do 32 będziemy korzystać z szablonu SQL Server Database Project w Visual Studio. Można go pobrać ze sklepu Visual Studio albo zainstalować SSDT.

7.1. Numer 26 — pisanie kodu bez obsługi błędów

Jeśli nasz kod zawiedzie w środowisku produkcyjnym, ostatnią rzeczą, jakiej chcemy, jest wystąpienie nieobsłużonego błędu. W przypadku awarii kodu zależy nam na tym, aby zawiódł on w sposób kontrolowany i dostarczył użytecznych informacji diagnostycznych zespołowi wsparcia aplikacji, który będzie musiał rozwiązać problem. Aby zgłębić to zagadnienie, przyjrzymy się najpierw procedurze składowanej bez obsługi błędów i zobaczymy, jakie mogą być konsekwencje jej używania.

Procedurę składowaną z listingu 7.1 można utworzyć w bazie danych MagicChoc. Jest to prosta procedura, która przyjmuje parametry niezbędne do utworzenia nowego zamówienia sprzedaży. Generuje ona wymagany prefiks dla numeru zamówienia na podstawie danych klienta, a następnie wstawia odpowiednie rekordy do tabel SalesOrderHeaders i SalesOrderDetails.

Listing 7.1. Tworzenie procedury składowanej dbo.InsertSalesOrder

```
CREATE PROCEDURE dbo.InsertSalesOrder
    @SalesOrderNumber NVARCHAR(12),
    @SalesOrderDate DATE,
    @SalesPersonID INT,
    @SalesAreaID INT,
    @CustomerID INT,
    @SalesOrderDeliveryDueDate DATE,
    @SalesOrderDeliveryActualDate DATE,
    @CurrierUsedForDelivery NVARCHAR(32),
    @ProductID INT,
    @Quantity INT
AS
BEGIN
    BEGIN TRANSACTION

        DECLARE @CustomerPrefix NVARCHAR(12) ;
        SET @CustomerPrefix = (
            SELECT SUBSTRING(CustomerCompanyName,1,3)
            FROM dbo.Customers
            WHERE CustomerID = @CustomerID
        ) ;

        INSERT INTO dbo.SalesOrderHeaders
        VALUES (
            @CustomerPrefix + @SalesOrderNumber
            , @SalesOrderDate
            , @SalesPersonID
            , @SalesAreaID
            , @CustomerID
            , @SalesOrderDeliveryDueDate
            , @SalesOrderDeliveryActualDate
            , @CurrierUsedForDelivery
        ) ;

        INSERT INTO dbo.SalesOrderDetails (
            ProductID
            , Quantity
            , SalesOrderNumber
        )
        VALUES (
            @ProductID
            , @Quantity
            , @CustomerPrefix + @SalesOrderNumber
        ) ;

    COMMIT
END
```

Zobaczmy teraz, jak to działa w praktyce, używając naszej procedury składowanej do utworzenia kilku zamówień sprzedaży. Skrypt przedstawiony na listingu 7.2 wykonuje naszą procedurę składowaną, pomyślnie wstawiając zamówienie zarówno do tabeli `SalesOrderHeaders`, jak i `SalesOrderDetails`.

Listing 7.2. Pomyślne wstawienie zamówienia sprzedaży

```
EXEC dbo.InsertSalesOrder
    '1635D-U06'
    , '2023-08-19'
    , 1
    , 1
    , 2
    , '2023-09-01'
    , NULL
    , 'Get Me There!'
    , 5
    , 1 ;
```

Na razie wszystko idzie zgodnie z planem! Procedura została wykonana pomyślnie i dodano nowe zamówienie sprzedaży. Teraz wykonajmy ją ponownie, używając skryptu z listingu 7.3. Tym razem przed uruchomieniem procedury zmienimy nazwę tabeli `SalesOrderDetails` na `SalesOrderLines`. Oczywiście spowoduje to błąd w transakcji.

Listing 7.3. Próba wstawienia danych do nieistniejącej tabeli

```
EXEC sp_rename 'dbo.SalesOrderDetails', 'SalesOrderLines' ;
GO
```

```
EXEC dbo.InsertSalesOrder
    '1637D-U06'
    , '2023-08-19'
    , 1
    , 1
    , 2
    , '2023-09-01'
    , NULL
    , 'Get Me There!'
    , 5
    , 1 ;
```

Pojawiły się dwa błędy, co widać poniżej:

```
Msg 208, Level 16, State 1, Procedure dbo.InsertSalesOrder, Line 35 [Batch Start Line 2]
Invalid object name 'dbo.SalesOrderDetails'.
Msg 266, Level 16, State 2, Procedure dbo.InsertSalesOrder, Line 35 [Batch Start Line 2]
```

Transaction count after EXECUTE indicates a mismatching number of BEGIN and COMMIT statements. Previous count = 0, current count = 1.

Jak można się było spodziewać, pierwszy błąd wynika z nieprawidłowej nazwy tabeli. Drugi błąd jest jednak bardziej interesujący. Informuje on, że liczba otwartych transakcji na końcu wykonania procedury składowanej jest wyższa niż na

początku. Możemy to sprawdzić przez wykonanie kwerendy `SELECT @@TRANCOUNT`. Przy założeniu, że nie mamy innych otwartych transakcji, wartość ta będzie wynosić 1, podczas gdy powinna wynosić 0. Dzieje się tak, ponieważ transakcja nie została automatycznie wycofana. Musimy ręcznie wydać polecenie `ROLLBACK`, aby zakończyć tę transakcję. Jeśli wykonujesz te kroki, powinieneś zrobić to teraz. Następnie uruchom poniższy kod, aby poprawić nazwę tabeli:

```
EXEC sp_rename 'dbo.SalesOrderLines', 'SalesOrderDetails' ;  
GO
```

Kod w procedurze składowanej jest obudowany *transakcją*, która składa się jednego lub więcej poleceń T-SQL aktualizujących dane i zatwierdzanych lub wycofywanych jako pojedyncza jednostka logiczna. Na najbardziej podstawowym poziomie transakcja musi posiadać cztery kluczowe właściwości, znane jako ACID. *ACID* to akronim oznaczający: atomowość (ang. *atomic*), spójność (ang. *consistent*), izolację (ang. *isolated*) i trwałość (ang. *durable*).

Gdy mówimy, że transakcja jest atomowa, oznacza to, że wszystkie operacje w jej ramach są zatwierdzane albo wycofywane jako jedna całość. Albo wszystkie się powiodą, albo wszystkie zakończą się niepowodzeniem. Spójność transakcji oznacza, że po jej zakończeniu dane w tabelach pozostaną w spójnym stanie. Izolacja transakcji polega na tym, że nie może ona wchodzić w interakcje z innymi transakcjami wykonywanymi równocześnie. Trwałość transakcji gwarantuje, że po jej zatwierdzeniu zaktualizowane dane nie zostaną utracone nawet w przypadku awarii serwera SQL.

SQL Server wykorzystuje wiele mechanizmów, aby zapewnić transakcjom właściwości ACID. Obejmują one blokowanie, które wymusza izolację poprzez uniemożliwienie innym transakcjom jednoczesnej aktualizacji innych wierszy, oraz rejestrowanie zdarzeń w dzienniku transakcji. Dziennik ten jest zapisywany na dysku przed zakończeniem zatwierdzenia transakcji, co gwarantuje trwałość danych.

Należy jednak pamiętać, że SQL Server to duży i złożony produkt, który ma wiele niuansów. Na przykład izolację transakcji można modyfikować, używając różnych poziomów izolacji. Przyjrzymy się im bliżej w rozdziale 10.

Kolejnym niuansiem jest trwałość danych. Funkcja zwana *opóźnionym utrwalaniem* może poprawić wydajność poprzez zmniejszenie konfliktów operacji wejścia-wyjścia w mocno obciążonych systemach, ale kosztem trwałości danych. Innymi słowy, w przypadku nieoczekiwanej awarii systemu istnieje ryzyko utraty zatwierdzonych danych.

W kontekście obsługi błędów bardzo istotna jest kwestia atomowości i spójności danych. Choć transakcja powinna gwarantować, że wszystkie dane zostaną albo zatwierdzone, albo wycofane jako jedna całość, to opcja `XACT_ABORT` określa, jak ściśle ta zasada jest przestrzegana. Jeśli opcja ta jest ustawiona na `OFF` (co jest wartością domyślną), niektóre błędy, takie jak przycięcie danych, nie powstrzymają transakcji przed zatwierdzeniem pozostałych instrukcji.

Przyjrzyjmy się kolejnemu przykładowi. Skrypt z listingu 7.4 ponownie wywołuje naszą procedurę składowaną. Tym razem nazwy obiektów są poprawne, ale próbujemy wstawić nieistniejącą wartość ProductID. Spowoduje to naruszenie ograniczenia klucza głównego.

Listing 7.4. Wstawianie niepoprawnego identyfikatora produktu do zamówienia sprzedaży

```
EXEC dbo.InsertSalesOrder
    '1655D-U06'
    , '2023-08-19'
    , 1
    , 1
    , 2
    , '2023-09-01'
    , NULL
    , 'Get Me There!'
    , 86
    , 1 ;
```

Wynik wykonania tego kodu jest następujący:

```
(1 row affected)
Msg 547, Level 16, State 0, Procedure dbo.InsertSalesOrder, Line 35 [Batch Start Line 0]
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_SalesOrde__Produ__619B8048". The conflict occurred in database "MagicChoc", table
"dbo.Products", column 'ProductID'.
```

Widzimy, że operacja wstawiania do tabeli SalesOrderHeaders powiodła się, ale wstawianie do tabeli SalesOrderDetails zakończyło się niepowodzeniem. Gdybyśmy w tym momencie wykonali zapytanie `SELECT @@TRANCOUNT`, otrzymalibyśmy wartość 0. Oznacza to, że transakcja została zatwierdzona. Konkretnie, zatwierdzono wstawienie danych z pierwszej instrukcji, mimo że druga operacja wstawiania się nie powiodła.

Dzieje się tak dlatego, że przy wyłączonej opcji `XACT_ABORT` (co jest ustawieniem domyślnym) niektóre błędy wykonania, takie jak zakleszczenia lub nieprawidłowe nazwy obiektów, powodują przerwanie transakcji. Jednak inne błędy, na przykład naruszenia ograniczeń kluczy, pozwalają kontynuować transakcję, powodując jedynie niepowodzenie konkretnej instrukcji.

Nasze dane znalazły się zatem w stanie niekompletnym. Zamówienie istnieje w tabeli SalesOrderHeaders, ale nie ma powiązanych z nim pozycji. Co więcej, błąd nie był na tyle poważny, aby został zapisany w dzienniku błędów, co utrudni jego debugowanie.

Pierwszym wnioskiem z tych przykładów jest to, że dobrą praktyką jest włączenie `XACT_ABORT`. Można to zrobić w ramach procedury składowanej lub partii z użyciem polecenia `SET XACT_ABORT ON`. Jednak zalecałbym globalne włączenie tej opcji dla połączeń z instancją. Można to zrobić za pomocą skryptu z listingu 7.5.

Listing 7.5. Globalne włączenie opcji XACT_ABORT dla połączeń z instancją

```
EXEC sp_configure
    'user options'
    , '16384' ;
GO

RECONFIGURE WITH OVERRIDE ;
GO
```

Drugi, ważniejszy wniosek jest taki, że naprawdę warto poprawnie obsługiwać błędy w kodzie. SQL Server oferuje kompleksowe funkcje obsługi błędów, ale programiści często nie korzystają z tych możliwości. To poważna pomyłka, której należy unikać.

Zastanówmy się zatem, jak ulepszyć naszą procedurę składowaną, żeby lepiej obsługiwała błędy. W tym celu warto poznać mechanizm TRY...CATCH, który może być znajomy osobom mającym doświadczenie z platformą .NET.

WSKAZÓWKA Programiści .NET powinni pamiętać, że SQL Server nie obsługuje bloku FINALLY. W językach .NET blok ten stanowi trzeci element konstrukcji i pozwala na wykonanie kodu niezależnie od powodzenia lub niepowodzenia bloku TRY.

W ramach bloku TRY...CATCH SQL Server próbuje wykonać kod zawarty w bloku TRY. Jeśli kod zostanie pomyślnie wykonany, SQL Server opuszcza blok. Jednak w przypadku wystąpienia błędu przerywa wykonywanie bloku TRY w miejscu, w którym wystąpił błąd, i przechodzi do bloku CATCH. Następnie wykonuje kod zawarty w bloku CATCH.

Oznacza to, że możemy spróbować wykonać kod, a jeśli wystąpi błąd, możemy umieścić logikę obsługi błędów wewnątrz bloku CATCH. Logika ta może obejmować wycofanie transakcji i zgłaszanie zrozumiałych komunikatów o błędach.

W SQL Serverze istnieją dwa sposoby zgłaszania zrozumiałych błędów. Pierwszy polega na użyciu funkcji RAISERROR(). Drugi to prostsza metoda obsługi błędów o nazwie THROW. Chociaż oba mechanizmy służą do zgłaszania błędów, istnieją między nimi pewne różnice, które omówimy.

WSKAZÓWKA Czy zauważyłeś błąd w nazwie RAISERROR()? Jeśli nie, przyjrzyj się jeszcze raz. Jest tam błąd ortograficzny. Nie jest to jednak literówka w książce, a w samym SQL Serverze. To prawdopodobnie najbardziej znany i zdecydowanie najbardziej ironiczny błąd pisowni w branży informatycznej!

W ramach bloku CATCH mamy dostęp do funkcji systemowych, które dostarczają informacji o zaistniałym błędzie. Ściślej rzecz biorąc, dostępne są następujące funkcje o intuicyjnych nazwach:

- ERROR_NUMBER() — zwraca numer błędu, który spowodował wykonanie bloku CATCH;
- ERROR_SEVERITY() — zwraca poziom krytyczności błędu;

- `ERROR_STATE()` — zwraca numer stanu błędu, co może pomóc w zidentyfikowaniu jego przyczyny;
- `ERROR_MESSAGE()` — zwraca pełny tekst komunikatu o błędzie;
- `ERROR_PROCEDURE()` — zwraca nazwę procedury składowanej lub wyzwalacza, w którym wystąpił błąd;
- `ERROR_LINE()` — zwraca numer wiersza w procedurze lub wyzwalaczu, w którym wystąpił błąd.

Funkcje te można wykorzystać do rozbudowania logiki obsługi błędów. Można na przykład rozgałęzić kod w bloku `CATCH` w zależności od numeru błędu. Można również zapisywać te dane w tabeli rejestrującej błędy.

Zobaczmy, jak możemy ulepszyć naszą procedurę składowaną, wykorzystując blok `TRY...CATCH`. Po rozpoczęciu transakcji zaczniemy blok `TRY`, który będzie zawierał naszą logikę biznesową. Ostatnia instrukcja w bloku `TRY` zatwierdzi naszą transakcję za pomocą polecenia `COMMIT`.

Po bloku `TRY` następuje bezpośrednio blok `CATCH`, który zawiera logikę obsługi błędów. W tym miejscu wycofujemy transakcję za pomocą instrukcji `ROLLBACK`, a następnie zgłaszamy błąd przy użyciu instrukcji `THROW`.

Istnieją dwa sposoby wykorzystania instrukcji `THROW`. Możemy użyć samego słowa kluczowego `THROW`, co spowoduje zgłoszenie pierwotnego błędu. Alternatywnie możemy określić własny numer błędu, komunikat oraz stan, które mają zostać zgłoszone, dodając je do instrukcji `THROW`. Instrukcja ta nie potrafi za to zgłosić komunikatu o błędzie przechowywanego w tabeli `sys.messages`. Nie możemy też określić poziomu krytyczności błędu. Poziom krytyczności błędu zgłaszanego przez instrukcję `THROW` zawsze wynosi 16, co oznacza, że nie jest on rejestrowany w dzienniku.

Poziomy krytyczności błędów

W SQL Serverze istnieje 25 poziomów krytyczności błędów. Poziomy od 0 do 10 nie są właściwie błędami, lecz komunikatami informacyjnymi. Błędy o poziomie istotności 10 to komunikaty informacyjne, które ze względów kompatybilności są konwertowane do poziomu 0.

Błędy o poziomach krytyczności od 11 do 16 są uważane za błędy, które użytkownik może naprawić samodzielnie. Należą do nich nieprawidłowe nazwy obiektów, błędy składniowe, problemy z uprawnieniami oraz zakleszczenia.

Poziomy od 17 do 19 dotyczą problemów, które zazwyczaj może naprawić tylko administrator. Obejmują one sytuacje takie jak wyczerpanie zasobów, przekroczenie limitów narzuconych przez silnik bazy danych oraz wewnętrzne problemy silnika, które spowodowały niepowodzenie wykonania instrukcji bez zamknięcia połączenia z instancją.

Błędy o poziomie od 20 do 24 są błędami krytycznymi, takimi jak uszkodzenie bazy danych lub nośnika. Błędy o poziomie od 19 do 24 są zapisywane w dzienniku błędów.

Rozważając obsługę błędów, warto pamiętać o kilku kwestiach związanych z poziomami krytyczności. Po pierwsze, tylko błędy o poziomie istotności od 11 do 19 spowodują przeniesienie wykonania kodu do bloku CATCH. Jeśli zostanie zgłoszony błąd o poziomie od 0 do 10, wykonywanie kodu będzie kontynuowane, ponieważ taki błąd ma charakter jedynie informacyjny. Z drugiej strony, jeśli poziom krytyczności wynosi 20 lub więcej, nie ma sensu przenosić wykonania do bloku CATCH, ponieważ błąd jest krytyczny i w wielu przypadkach spowoduje nawet przerwanie połączenia.

Kolejną ważną kwestią jest to, że jak wspomniano wcześniej, błędy zgłaszane przez THROW mają stały poziom krytyczności równy 16. Oznacza to, że nigdy nie zostaną zapisane w dzienniku błędów, co w wielu przypadkach utrudnia diagnozowanie problemów.

Instrukcja THROW może zgłosić jedynie pierwotny błąd systemowy lub błąd doraźny, czyli taki, który nie jest przechowywany w sys.messages. Z tego powodu numer błędu, który zgłaszamy, musi być równy lub większy niż 50 000.

Na koniec trzeba pamiętać, że chociaż każdy poziom krytyczności ma określone znaczenie (na przykład poziom 22 oznacza, że tabela lub indeks zostały uszkodzone w wyniku problemu sprzętowego lub programowego), poziomy te dotyczą tylko błędów systemowych. Możesz utworzyć własny komunikat o błędzie i „przejąć” systemowy poziom do własnych celów, które mogą nie odpowiadać zamierzonemu poziomowi krytyczności. Oczywiście nie byłoby to dobrą praktyką, ponieważ mogłoby zmylić Twoich współpracowników.

Skrypt z listingu 7.6 zawiera zaktualizowaną definicję procedury składowanej, która została umieszczona w bloku TRY...CATCH. Jeśli wykonanie kodu w bloku TRY zakończy się niepowodzeniem, sterowanie zostanie przekazane do bloku CATCH. Tam nastąpi wycofanie transakcji oraz zgłoszenie błędu za pomocą instrukcji THROW.

Listing 7.6. Dodawanie bloku TRY...CATCH do procedury InsertOrders

```
ALTER PROCEDURE dbo.InsertSalesOrder
    @SalesOrderNumber NVARCHAR(12),
    @SalesOrderDate DATE,
    @SalesPersonID INT,
    @SalesAreaID INT,
    @CustomerID INT,
    @SalesOrderDeliveryDueDate DATE,
    @SalesOrderDeliveryActualDate DATE,
    @CurrierUsedForDelivery NVARCHAR(32),
    @ProductID INT,
    @Quantity INT
AS
BEGIN
    BEGIN TRANSACTION
    BEGIN TRY
        DECLARE @CustomerPrefix NVARCHAR(12) ;
        SET @CustomerPrefix = (
            SELECT SUBSTRING(CustomerCompanyName,1,3)
            FROM dbo.Customers
            WHERE CustomerID = @CustomerID
```

```

    ) ;

INSERT INTO dbo.SalesOrderHeaders
VALUES (
    @CustomerPrefix + @SalesOrderNumber
    , @SalesOrderDate
    , @SalesPersonID
    , @SalesAreaID
    , @CustomerID
    , @SalesOrderDeliveryDueDate
    , @SalesOrderDeliveryActualDate
    , @CarrierUsedForDelivery
) ;

INSERT INTO dbo.SalesOrderDetails (
    ProductID
    , Quantity
    , SalesOrderNumber
)
VALUES (
    @ProductID
    , @Quantity
    , @CustomerPrefix + @SalesOrderNumber
) ;

COMMIT
END TRY
BEGIN CATCH
    ROLLBACK ;
    THROW ;
END CATCH
END

```

Nie daje nam to jednak jeszcze większych korzyści. Gdybyśmy ponownie wykonali polecenie z listingu 7.4, otrzymalibyśmy następujący komunikat o błędzie:

```

Msg 2627, Level 14, State 1, Procedure dbo.InsertSalesOrder, Line 23 [Batch Start Line 0]
Violation of PRIMARY KEY constraint 'PK_SalesOrd_CF6C70EEA96DBC48'.
Cannot insert duplicate key in object 'dbo.SalesOrderHeaders'. The duplicate key value is
(Coo1655D-U06).

```

Wprowadźmy zatem kolejne ulepszenia do procedury `InsertSalesOrder`. Skrypt przedstawiony na listingu 7.7 ponownie aktualizuje tę procedurę. Tym razem rozbudowujemy kod w bloku `CATCH`, używając instrukcji `IF` w połączeniu z funkcjami systemowymi, które udostępniają informacje o błędach. Dzięki temu blok `CATCH` będzie działał różnie w zależności od rodzaju zgłoszonego błędu. Jeśli wystąpi naruszenie klucza głównego, zostanie wyświetlony nasz własny komunikat o błędzie. W przypadku zakleszczenia użyjemy polecenia `GOTO`, aby ponownie spróbować wykonać kod w bloku `TRY`. Jeśli pojawi się jakikolwiek inny problem, zgłosimy pierwotny błąd.

Listing 7.7. Rozgałęziony blok CATCH w procedurze InsertSalesOrder

```

ALTER PROCEDURE dbo.InsertSalesOrder
    @SalesOrderNumber NVARCHAR(12),
    @SalesOrderDate DATE,
    @SalesPersonID INT,
    @SalesAreaID INT,
    @CustomerID INT,
    @SalesOrderDeliveryDueDate DATE,
    @SalesOrderDeliveryActualDate DATE,
    @CurrierUsedForDelivery NVARCHAR(32),
    @ProductID INT,
    @Quantity INT
AS
BEGIN
    BEGIN TRANSACTION
    RETRY:
    BEGIN TRY
        DECLARE @CustomerPrefix NVARCHAR(12) ;
        SET @CustomerPrefix = (
            SELECT SUBSTRING(CustomerCompanyName,1,3)
            FROM dbo.Customers
            WHERE CustomerID = @CustomerID
        ) ;

        INSERT INTO dbo.SalesOrderHeaders
        VALUES (
            @CustomerPrefix + @SalesOrderNumber
            , @SalesOrderDate
            , @SalesPersonID
            , @SalesAreaID
            , @CustomerID
            , @SalesOrderDeliveryDueDate
            , @SalesOrderDeliveryActualDate
            , @CurrierUsedForDelivery
        ) ;

        INSERT INTO dbo.SalesOrderDetails (
            ProductID
            , Quantity
            , SalesOrderNumber
        )
        VALUES (
            @ProductID
            , @Quantity
            , @CustomerPrefix + @SalesOrderNumber
        ) ;
    COMMIT
    END TRY
    BEGIN CATCH
        IF ERROR_NUMBER() = 2627 <———— Błąd naruszenia klucza głównego.
        BEGIN
            ROLLBACK ;
            THROW 50001,
                'Wprowadzono zduplikowane zamówienie sprzedaży.', 1 ;
        END
        IF ERROR_NUMBER() = 1205 <———— Błąd zakleszczenia.
    
```

```

BEGIN
    ROLLBACK ;
    GOTO RETRY ;
END
IF ERROR_NUMBER() <> 2627
    AND ERROR_NUMBER() <> 1205  ← Dowolny inny błąd.
BEGIN
    ROLLBACK ;
    THROW ;
END
END CATCH
END

```

Gdybyśmy ponownie wykonali polecenie z listingu 7.4, otrzymalibyśmy następujący komunikat o błędzie:

Msg 50001, Level 16, State 1, Procedure dbo.InsertSalesOrder, Line 52 [Batch Start Line 0]
Wprowadzono zduplikowane zamówienie sprzedaży.

Aby jak najlepiej wykorzystać funkcjonalność obsługi błędów w SQL Serverze, warto łączyć instrukcję `THROW` z funkcją `RAISERROR()`. Na przykład w naszym scenariuszu, jeśli wystąpi błąd ogólny, możemy nadal używać `THROW`, aby przekazywać dalej pierwotny komunikat o błędzie. Natomiast w przypadku naruszenia klucza głównego możemy użyć `RAISERROR()`, aby wysłać informację o błędzie do dziennika zdarzeń.

Aby zastosować to podejście, musimy najpierw utworzyć niestandardowy komunikat o błędzie, który będzie można zgłosić za pomocą funkcji `RAISERROR()`. W tym celu skorzystamy z systemowej procedury składowanej `sp_addmessage`, jak pokazano na listingu 7.8.

Listing 7.8. Tworzenie niestandardowego komunikatu o błędzie

```

EXEC sp_addmessage
    @msgnum = 50001
    , @severity = 16
    , @msgtext = 'Wprowadzono zduplikowane zamówienie sprzedaży.' ;

```

Następnie możemy zaktualizować procedurę `InsertSalesOrder` tak, aby wywoływała funkcję `RAISERROR()`, jak pokazano na listingu 7.9. Użycie składni `WITH LOG` sprawi, że błąd zostanie zapisany w dzienniku błędów SQL Servera, mimo że jego poziom krytyczności wynosi tylko 16.

Listing 7.9. Aktualizowanie procedury `InsertSalesOrder` tak, aby używała funkcji `RAISERROR()`

```

ALTER PROCEDURE dbo.InsertSalesOrder
    @SalesOrderNumber NVARCHAR(12),
    @SalesOrderDate DATE,
    @SalesPersonID INT,
    @SalesAreaID INT,
    @CustomerID INT,
    @SalesOrderDeliveryDueDate DATE,
    @SalesOrderDeliveryActualDate DATE,

```

```

        @CurrierUsedForDelivery NVARCHAR(32),
        @ProductID INT,
        @Quantity INT
AS
BEGIN
    BEGIN TRANSACTION
    RETRY:
    BEGIN TRY
        DECLARE @CustomerPrefix NVARCHAR(12) ;
        SET @CustomerPrefix = (
            SELECT SUBSTRING(CustomerCompanyName,1,3)
            FROM dbo.Customers
            WHERE CustomerID = @CustomerID
        ) ;

        INSERT INTO dbo.SalesOrderHeaders
        VALUES (
            @CustomerPrefix + @SalesOrderNumber
            , @SalesOrderDate
            , @SalesPersonID
            , @SalesAreaID
            , @CustomerID
            , @SalesOrderDeliveryDueDate
            , @SalesOrderDeliveryActualDate
            , @CurrierUsedForDelivery
        ) ;

        INSERT INTO dbo.SalesOrderDetails (
            ProductID
            , Quantity
            , SalesOrderNumber
        )
        VALUES (
            @ProductID
            , @Quantity
            , @CustomerPrefix + @SalesOrderNumber
        ) ;

    COMMIT
    END TRY
    BEGIN CATCH
        IF ERROR_NUMBER() = 2627
        BEGIN
            ROLLBACK ;
            RAISERROR(50001, 16, 1) WITH LOG ;
        END
        IF ERROR_NUMBER() = 1205
        BEGIN
            ROLLBACK ;
            GOTO RETRY ;
        END
        IF ERROR_NUMBER() <> 2627
            AND ERROR_NUMBER() <> 1205
        BEGIN
            ROLLBACK ;
            THROW ;
        END
    END CATCH
END

```

Jeśli ponownie wykonamy polecenie z listingu 7.4, nasz niestandardowy komunikat o błędzie nie tylko zostanie zwrócony do klienta, ale również zapisany w dzienniku błędów SQL Servera, jak pokazano na rysunku 7.1.

| Log file summary: No filter applied | | |
|---|---|--|
| Date ▾ | Source | Message |
| <input type="checkbox"/> 8/29/2023 8:56:40 PM | spid59 | A duplicate sales order has been entered |
| Selected row details: | | |
| Date | 8/29/2023 8:56:40 PM | |
| Log | SQL Server (Current - 8/29/2023 6:38:00 PM) | |
| Source | spid59 | |
| Message | A duplicate sales order has been entered. | |

Rysunek 7.1. Niestandardowy błąd w dzienniku błędów SQL Servera

Alternatywą dla funkcji RAISERROR() z opcją WITH LOG jest wykorzystanie własnej tabeli błędów. Na przykład skrypt przedstawiony na listingu 7.10 najpierw tworzy tabelę do rejestrowania błędów. Następnie aktualizuje procedurę składowaną, przywracając użycie instrukcji THROW. Tym razem jednak modyfikuje blok CATCH tak, aby zapisywał informacje o błędach w nowo utworzonej tabeli.

Listing 7.10. Dodawanie do procedury InsertSalesOrder niestandardowej tabeli do rejestrowania błędów

```
CREATE TABLE dbo.ErrorLog (
    ID          INT      PRIMARY KEY IDENTITY,
    ErrorMessage NVARCHAR(MAX),
    ErrorNumber  INT,
    ErrorSeverity INT
);
GO
```

```
ALTER PROCEDURE dbo.InsertSalesOrder
    @SalesOrderNumber NVARCHAR(12),
    @SalesOrderDate DATE,
    @SalesPersonID INT,
    @SalesAreaID INT,
    @CustomerID INT,
    @SalesOrderDeliveryDueDate DATE,
    @SalesOrderDeliveryActualDate DATE,
    @CurrierUsedForDelivery NVARCHAR(32),
    @ProductID INT,
    @Quantity INT
AS
BEGIN
    BEGIN TRANSACTION
    RETRY:
    BEGIN TRY
        DECLARE @CustomerPrefix NVARCHAR(12) ;
        SET @CustomerPrefix = (
```

```

        SELECT SUBSTRING(CustomerCompanyName,1,3)
        FROM dbo.Customers
        WHERE CustomerID = @CustomerID
    ) ;

    INSERT INTO dbo.SalesOrderHeaders
    VALUES (
        @CustomerPrefix + @SalesOrderNumber
        , @SalesOrderDate
        , @SalesPersonID
        , @SalesAreaID
        , @CustomerID
        , @SalesOrderDeliveryDueDate
        , @SalesOrderDeliveryActualDate
        , @CurrierUsedForDelivery
    ) ;

    INSERT INTO dbo.SalesOrderDetails (
        ProductID
        , Quantity
        , SalesOrderNumber
    )
    VALUES (
        @ProductID
        , @Quantity
        , @CustomerPrefix + @SalesOrderNumber
    ) ;
COMMIT
END TRY
BEGIN CATCH
    IF ERROR_NUMBER() = 2627
    BEGIN
        ROLLBACK;
        INSERT INTO dbo.ErrorLog (
            ErrorMessage,
            ErrorNumber,
            ErrorSeverity
        )
        SELECT
            ERROR_MESSAGE()
            , ERROR_NUMBER()
            , ERROR_SEVERITY();
        THROW 50001,
            'Wprowadzono zduplikowane zamówienie sprzedaży.', 1;
    END
    IF ERROR_NUMBER() = 1205
    BEGIN
        ROLLBACK;
        GOTO RETRY;
    END
    IF ERROR_NUMBER() <> 2627
        AND ERROR_NUMBER() <> 1205
    BEGIN
        ROLLBACK;
        THROW;
    END
END CATCH
END

```

Dodaje instrukcję INSERT, która zapisuje dane w tabeli rejestrowania błędów.

Zastępuje wywołanie RAISERROR() instrukcją THROW.

OSTRZEŻENIE Jeśli zdecydujemy się na niestandardowe podejście do rejestrowania błędów, niezwykle istotna staje się kolejność instrukcji. Instrukcja INSERT musi znajdować się po ROLLBACK, ale przed THROW. Gdyby instrukcja INSERT była umieszczona przed ROLLBACK, stałaby się częścią transakcji i zostałaby wycofana wraz z kodem w bloku TRY. Z kolei gdyby instrukcja INSERT znalazła się po THROW, nie zostałaby w ogóle wykonana.

Warto dodać obsługę błędów do kodu, który modyfikuje dane. Pozwoli to na wycofywanie transakcji i generowanie zrozumiałych komunikatów o błędach. Powinniśmy również rozważyć rejestrowanie komunikatów o błędach tam, gdzie jest to stosowne. W niektórych sytuacjach, takich jak zakleszczenia, może być wskazane zaimplementowanie logiki ponownych prób w naszych procedurach.

7.2. Numer 27 – niealarmowanie o błędach

Po wdrożeniu naszych procedur składowanych w środowisku produkcyjnym istnieją różne sposoby ich wykonywania, w zależności od charakteru kodu. Mogą one być wykonywane przez narzędzie ETL, takie jak SSIS, przez zadanie SQL Server Agent, które jest zorganizowaną serią działań zaplanowanych do uruchomienia o określonej porze, lub wywoływane przez aplikację kliencką.

Jeśli procedura jest wywoływana przez dobrze napisaną aplikację kliencką, błędy zapewne zostaną zauważone przez użytkownika. Jeśli jednak procedura jest uruchamiana przez zautomatyzowany proces, wykrycie problemu może być trudniejsze. W takiej sytuacji osoba odpowiedzialna za wsparcie aplikacji musi aktywnie sprawdzać dzienniki SQL Server Agent, dzienniki SSISDB lub niestandardowe tabele rejestrowania zdarzeń w celu wykrycia awarii.

WSKAZÓWKA Jak omówiono w poprzednim podrozdziale, w zależności od krytyczności błędu i metody jego zgłoszenia może on nie zostać zapisany w dzienniku błędów SQL Servera. Z tego powodu dziennik błędów nie jest wiarygodnym miejscem do sprawdzania błędów aplikacji.

Aktywne kontrole często nie są przeprowadzane, a nawet jeśli są, to są narażone na ludzkie błędy i wiążą się z *kosztem alternatywnym*. Oznacza to, że ktoś poświęca swój czas na wykonywanie rutynowych zadań zamiast pracy o wyższej wartości. W rezultacie bardzo łatwo jest przeoczyć awarię, dopóki użytkownik nie zgłosi, że dane wyglądają nieprawidłowo.

Jak zatem możemy rozwiązać ten problem? Odpowiedzią jest zastosowanie alarmów. Jeśli przyjrzymy się dowolnej warstwie stosu infrastruktury IT, od warstwy sieciowej po system operacyjny, zauważymy, że zespoły wsparcia niezmienne korzystają z systemów alarmowych, które powiadamiają je o błędach

wymagających zbadania lub podjęcia działań naprawczych. Jednak w przypadku aplikacji bazodanowych alarmy często się pomija, co jest pomyłką.

W zależności od skali i priorytetów naszej aplikacji przy odrobinie szczęścia możemy dysponować zaawansowanym oprogramowaniem do monitorowania, takim jak SolarWinds czy LogicMonitor. W takim przypadku warto wykorzystać to oprogramowanie do utworzenia własnego sprawdzianu, który będzie alarmować nas w razie awarii. Jeśli wybierzemy tę opcję, źródło danych będzie zazwyczaj skonfigurowane tak, aby odczytywać dane z tabeli dziennika w regularnych odstępach czasu i generować alarm, jeśli w dzienniku zostanie zapisany błąd. Gdy używamy niestandardowego rejestrowania zdarzeń i tabel dziennika w SSISDB, będzie to zwykle wymagało prostego zapytania SELECT. Jeśli jednak musimy odczytać informacje z dziennika SQL Server Agent, będziemy musieli użyć procedury składowanej `sp_help_jobsteplog`. Jeśli na przykład mamy zadanie SQL Server Agent o nazwie `Populate_Fact_Tables`, możemy użyć następującego polecenia, aby pobrać informacje z dziennika:

```
EXEC msdb.dbo.sp_help_jobhistory  
    @job_name = 'Populate_Fact_Tables'  
    , @mode = 'FULL' ;
```

Jeśli nie mamy do dyspozycji zaawansowanego narzędzia monitorującego, którego moglibyśmy użyć do generowania alarmów, możemy skorzystać z wbudowanych funkcji SQL Servera. W takim przypadku użyjemy funkcji o nazwie `Database Mail` w połączeniu z podsystemem alarmów SQL Server Agent.

Database Mail

Warto w tym miejscu zaznaczyć, że `Database Mail` jest często krytykowany w środowisku jako narzędzie nieprzystosowane do zastosowań korporacyjnych. Chociaż zasadniczo zgadzam się z tą oceną, to szczerze mówiąc, jeśli nasza organizacja nie zainwestowała w zewnętrzne narzędzie, które moglibyśmy wykorzystać do tego celu, musimy pracować z tym, co mamy. Sam kilkakrotnie znalazłem się w podobnym położeniu.

Jedną z głównych wad tej funkcji jest jej mała skalowalność. Funkcja jest zaimplementowana w bazie `msdb`, co w dzisiejszych złożonych środowiskach SQL Server może prowadzić do niepoprawnego działania narzędzia w przypadku przełączenia awaryjnego.

Problem ten częściowo rozwiązano poprzez wprowadzenie grup dostępności, które zawierają własne kopie systemowych baz danych `master` i `msdb`. Kiedy jednak pisałem tę książkę (miejmy nadzieję, że nie wtedy, gdy ją czytasz) w grupach dostępności był błąd, który uniemożliwiał działanie funkcji `Database Mail`.

Warto również zaznaczyć, że do korzystania z tej funkcji potrzebny będzie dostęp do przekątnikowego serwera SMTP. `Database Mail` nie ma wbudowanych usług SMTP. Musi on wysyłać wiadomości e-mail za pośrednictwem istniejącej usługi SMTP.

Szczegółowe omówienie funkcji `Database Mail` wykracza poza ramy niniejszej książki – więcej informacji na ten temat można znaleźć pod adresem <https://mng.bz/PND9>.

Zobaczmy, jak utworzyć alarm, który będzie wyzwalany w reakcji na komunikat o błędzie. Przy założeniu, że usługa Database Mail jest już skonfigurowana, musimy utworzyć dwa elementy. Pierwszym jest operator, który zostanie skonfigurowany do korzystania z profilu Database Mail, a drugim jest sam alarm.

Warto zauważyć, że alarmy SQL Server Agent wykorzystują dziennik zdarzeń aplikacji systemu Windows do wykrywania zdarzeń wygenerowanych przez SQL Server. Zdarzenia są przekazywane do dziennika aplikacji, gdy zostają zapisane w dzienniku błędów SQL Servera. Oznacza to, że jeśli zdarzenie nie zostanie zarejestrowane w dzienniku, nie spowoduje uruchomienia alarmu. Dlatego stosując tę metodę, musimy się upewnić, że nasze komunikaty o błędach są zgłaszane za pomocą RAISERROR() z opcją WITH LOG. Dzięki temu zostaną one zarejestrowane, a alarm zostanie uruchomiony nawet wtedy, gdy poziom krytyczności błędu jest niższy niż 19.

Wykorzystajmy skrypt z listingu 7.11 do utworzenia operatora o imieniu Pete, który będzie powiadamiany w przypadku wyzwolenia alarmu.

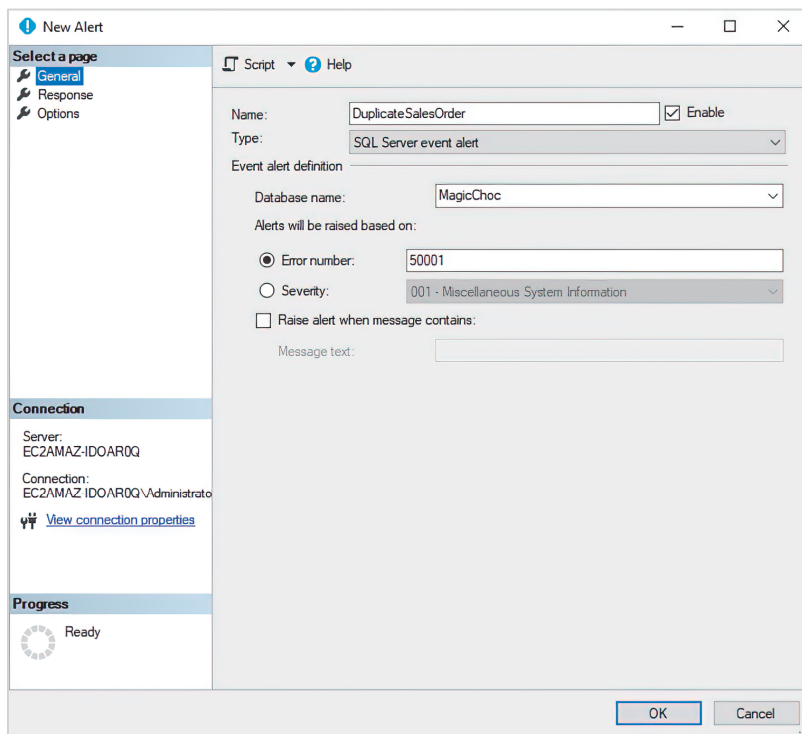
WSKAZÓWKA W rzeczywistym scenariuszu najlepiej byłoby użyć adresu e-mail grupy lub listy dystrybucyjnej.

Listing 7.11. Tworzenie operatora SQL Server Agent

```
EXEC msdb.dbo.sp_add_operator  
    @name= 'Pete'  
    , @enabled=1  
    , @email_address= 'pete@onehundredsq1mistakes.com' ;
```

WSKAZÓWKA SQL Server obecnie obsługuje również powiadomienia na pager, a także adres e-mail, pod warunkiem że mamy oprogramowanie do konwersji wiadomości pagera na e-maile. Ta funkcja jest jednak przestarzała i nie powinna być używana.

Aby utworzyć alarm, można wybrać opcję *New Alert* z węzła *Alerts* w gałęzi *SQL Server Agent* w programie SSMS. Spowoduje to wyświetlenie okna dialogowego *New Alert*. Stronę *General* tego okna dialogowego pokazano na rysunku 7.2. W tym miejscu nadajemy alarmowi nazwę i określamy, że ma on reagować na zdarzenie SQL Servera. Inne opcje to reakcja na zdarzenie WMI lub warunek wydajnościowy SQL Server. Na rozwijanej liście *Database name* możemy pozostawić domyślną opcję *all databases* lub wybrać konkretną bazę danych, w której alarm ma być wyzwalany. Na koniec możemy zdecydować, czy chcemy, żeby alarm był wyzwalany w odpowiedzi na określony numer błędu, czy na konkretny poziom krytyczności błędu.



Rysunek 7.2. Okno dialogowe *New Alert* programu *Database Mail* – strona *General*

Na stronie *Response* w oknie dialogowym możemy wybrać, czy w reakcji na alarm chcemy uruchomić zadanie SQL Server Agent, które spróbuje rozwiązać problem, czy wysłać powiadomienie do operatora, czy jedno i drugie. W naszym przypadku zaznaczymy opcję *Notify Operators*. Spowoduje to aktywowanie listy operatorów, gdzie możemy wybrać *Email* obok operatora *Pete*.

Na stronie *Options* w oknie dialogowym możemy utworzyć dodatkowy tekst powiadomienia, który zostanie wysłany wraz z wiadomością e-mail. Możemy również określić opóźnienie między wysyłanymi alertami. Funkcję tę można wykorzystać do ograniczenia „szumu alarmowego”, kiedy alert jest wyzwalany wielokrotnie w krótkich odstępach czasu.

Zamiast korzystać z interfejsu graficznego, moglibyśmy utworzyć alert przy użyciu procedur składowanych `sp_add_alert` i `sp_add_notification` w bazie danych `msdb`. Pokazano to na listingu 7.12.

Listing 7.12. Tworzenie alarmu z użyciem T-SQL

```
EXEC msdb.dbo.sp_add_alert
    @name= 'DuplicateSalesOrder'
    , @message_id=50001
    , @enabled=1
    , @database_name= 'MagicChoc' ;
```

```
EXEC msdb.dbo.sp_add_notification  
    @alert_name= 'DuplicateSalesOrder'  
    , @operator_name= 'Pete'  
    , @notification_method = 1 ;
```

Zawsze powinniśmy rozważyć tworzenie alarmów dla kodu w aplikacjach bazodanowych, które są uruchamiane przez zautomatyzowane procesy. Najlepiej, jeśli te alarmy będą tworzone w zaawansowanym narzędziu do monitorowania. Jeśli jednak nie mamy takiego narzędzia, możemy wykorzystać alarmy SQL Server Agent i funkcję Database Mail do wysyłania powiadomień przez istniejący serwer SMTP.

7.3. Numer 28 — nieużywanie funkcji debugowania

Czytelnicy mający doświadczenie z platformą .NET prawdopodobnie znają wiele narzędzi do debugowania dostępnych w Visual Studio. Wiele z tych funkcji jest również dostępnych podczas pracy z SQL Serverem, jednak rzadko są one wykorzystywane przez niedoświadczonych programistów T-SQL. Powodem tego jest często brak znajomości dostępnych możliwości, ale rezygnację z ich używania trzeba uznać za pomyłkę.

Widziałem programistów, którzy rwali sobie włosy z głowy, próbując znaleźć błąd w napisanej przez siebie procedurze składowanej. Wielokrotnie wykonywali tę procedurę, oznaczając kwerendy komentarzami i wstawiając instrukcje PRINT. Tworzyli zmienne symulujące parametry procedury i uruchamiali pojedyncze kwerendy w kodzie, desperacko próbując ustalić, gdzie tkwi problem. Nierzadko zdarza się, że faza debugowania zajmuje tyle samo czasu, co tworzenie procedury.

Programiści T-SQL mogliby zaoszczędzić sporo czasu, gdyby korzystali z funkcji debugowania dostępnych w Visual Studio w połączeniu z szablonem SQL Server Database Project. Aby wypróbować przykłady z tego podrozdziału, powinieneś utworzyć projekt SQL Server Database o nazwie *Marketing*, a następnie zaimportować bazę danych *Marketing*, którą utworzyliśmy w rozdziale 6. Możesz zaimportować bazę danych, wybierając opcję *Project/Import/Database* z menu kontekstowego projektu *Marketing*. Będziesz musiał utworzyć połączenie z bazą danych.

Po skonfigurowaniu projektu dodajmy do niego nowy element i utwórzmy procedurę składowaną zgodnie z definicją przedstawioną na listingu 7.13. Procedura ta przyjmuje parametry *CampaignID* i *Budget*, które wykorzystuje do utworzenia podsumowania kampanii. Generowane są dwa zestawy wyników. Pierwszy to finansowe podsumowanie kampanii reklamowej, a drugi to lista unikatowych wartości *ReferralURL* i *RenderingID*.

Listing 7.13. Tworzenie procedury składowanej CalculateCampaignSummary

```

CREATE PROCEDURE dbo.CalculateCampaignSummary
    @CampaignID INT,
    @Budget MONEY
AS
BEGIN
    DECLARE @AvgCPM DECIMAL ;
    DECLARE @CampaignCost MONEY ;
    DECLARE @NoOfImp INT ;
    DECLARE @AvgBidPrice MONEY ;
    DECLARE @CostOfBidPerc DECIMAL ;
    DECLARE @BudgetDifference MONEY ;

    SELECT
        @NoOfImp = COUNT(*)
        , @CampaignCost = (SUM(CostPerMille) / 1000) * COUNT(*)
    FROM marketing.Impressions
    WHERE CampaignID = @CampaignID
    GROUP BY CampaignID ;

    SELECT
        @avgcpm = AvgCostPerMille
        , @avgbidprice = AvgBidPrice
    FROM reporting.ImpressionAggregates
    WHERE CampaignID = @CampaignID ;

    SET @CostOfBidPerc = (@AvgBidPrice / @AvgCPM) * 100 ;
    SET @BudgetDifference = @budget - @CampaignCost ;

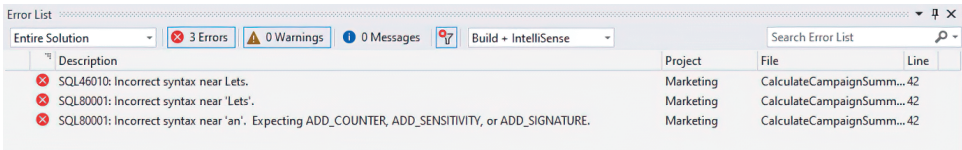
    SELECT
        @CampaignID CampaignID
        , @AvgCPM AvgCPM
        , @CampaignCost CampaignCost
        , @NoOfImp ImpQty
        , @CostOfBidPerc CostToBidPercentage
        , @BudgetDifference ;

    SELECT DISTINCT
        ReferralURL
        , RenderingID
    FROM marketing.Impressions
    WHERE CampaignID = @CampaignID ;
END

```

Pierwszym narzędziem do debugowania, które powinniśmy omówić, jest okno *Error List* dostępne w menu *View*. To niezwykle przydatne okno, w którym następuje analiza naszego kodu w czasie rzeczywistym i przekazywane są informacje o błędach kompilacji oraz błędach IntelliSense. Na przykład, gdybym dodał tekst *Lets add an error to our code* pomiędzy ostatnią kwerendą a instrukcją *END*, okno *Error List* zaktualizowałoby się w ciągu kilku chwil, jak pokazano na rysunku 7.3.

Teraz opublikujmy naszą bazę danych. Możemy to zrobić, wybierając opcję *Publish* z menu kontekstowego projektu. W oknie dialogowym *Publish Database* wybierz (wcześniej utworzone) połączenie z bazą danych *Marketing* na serwerze. W rezultacie nasza procedura składowana zostanie wdrożona na instancji *SQL Servera*.



Rysunek 7.3. Błędy w oknie Error List

OSTRZEŻENIE Jeśli wstawiłeś błędny tekst, tak jak ja w powyższym przykładzie, musisz go usunąć przed publikacją bazy danych. W przeciwnym razie kompilacja zakończy się niepowodzeniem.

Wykonajmy teraz naszą nową procedurę składowaną za pomocą polecenia pokazanego na listingu 7.14. Otwórz nowe okno kwerendy z menu kontekstowego instancji SQL Servera w SQL Server Object Explorer, a następnie wykonaj procedurę przez kliknięcie jasnozielonego przycisku *Start* na pasku zadań okna *Query*.

Listing 7.14. Wykonywanie procedury składowanej CalculateCampaignSummary

```
USE Marketing ;
GO

EXEC dbo.CalculateCampaignSummary
    22961,
    52 ;
```

W tym momencie zobaczymy, że procedura zgłasza błąd dzielenia przez zero w wierszu 26.

Wykonajmy więc ponownie naszą procedurę składowaną, tym razem korzystając z debugera. Jeśli klikniemy strzałkę obok jasnozielonego przycisku uruchamiania, zobaczymy dodatkową opcję. Jest to ciemnozielony przycisk oznaczony jako *Execute With Debugger*.

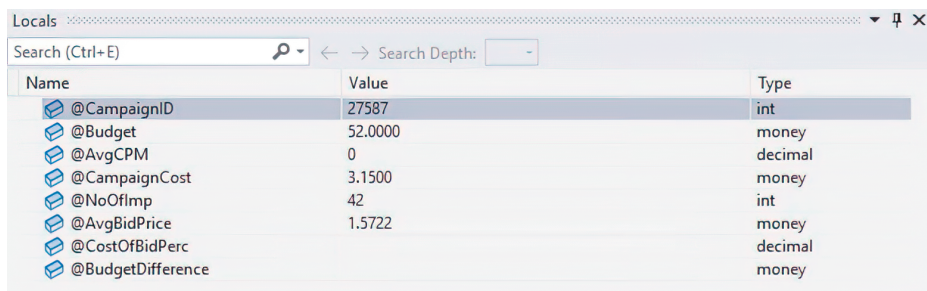
Po wybraniu tej opcji rozpocznie się wykonywanie kodu, ale pierwszy wiersz (w naszym przypadku instrukcja `USE Marketing`) zostanie podświetlony na żółto. Oznacza to, że w tym miejscu wykonywanie zostało wstrzymane. Mamy teraz kilka możliwości, które zostały opisane w tabeli 7.1.

Tabela 7.1. Opcje nawigacji podczas debugowania kodu

| Opcja | Skrót | Opis |
|--------------------------|-------|--|
| Continue (kontynuuj) | F5 | Kontynuuje wykonywanie programu aż do następnego punktu przerwania*. |
| Step Over (przekrocz) | F10 | Wykonuje krok po kroku wierzchnią warstwę kodu, ale kontynuuje przez niższe poziomy, takie jak procedury czy funkcje, bez ich debugowania (chyba że natrafi na punkt przerwania)*. |
| Step Into (wkrocz) | F11 | Pozwala na przechodzenie przez wszystkie warstwy kodu. Gdy napotkana zostanie procedura lub funkcja, debugger wejdzie do tego bloku kodu i zatrzyma wykonywanie na pierwszej instrukcji. |

* Omówione dalej.

Ponieważ chcemy zdiagnozować naszą procedurę składowaną, najlepszym wyborem będzie opcja *Step Into*. Użyjmy więc klawisza *F11*, aby wkroczyć do wnętrza procedury. Przechodząc przez kolejne kwerendy, zauważymy, że wartości poszczególnych zmiennych (i parametrów) są aktualizowane w oknie *Locals*. Po wykonaniu pierwszych dwóch kwerend okno *Locals* powinno wyglądać tak, jak na rysunku 7.4.



The screenshot shows the 'Locals' window in Visual Studio. It has a search bar at the top with 'Search (Ctrl+E)' and a 'Search Depth' dropdown. Below is a table of local variables:

| Name | Value | Type |
|-------------------|---------|---------|
| @CampaignID | 27587 | int |
| @Budget | 52.0000 | money |
| @AvgCPM | 0 | decimal |
| @CampaignCost | 3.1500 | money |
| @NoOfImp | 42 | int |
| @AvgBidPrice | 1.5722 | money |
| @CostOfBidPerc | | decimal |
| @BudgetDifference | | money |

Rysunek 7.4. Okno *Locals*

Widzimy teraz przyczynę błędu dzielenia przez zero. Wartość zmiennej `@AvgCPM` wynosi 0, a następna instrukcja będzie dzielić `@AvgBidPrice` przez tę wartość. Spróbujmy dotrzeć do sedna problemu. W oknie *Immediate* możemy uruchamiać kwerendy bez przerywania działania programu. Użyjmy tego okna do uruchomienia kwerendy z listingu 7.15, aby sprawdzić, czy druga kwerenda w naszej procedurze zwraca tylko jeden wiersz.

Listing 7.15. Wykonywanie kwerendy w oknie *Immediate*

```
SELECT COUNT(*) FROM reporting.ImpressionAggregates WHERE CampaignID = @CampaignID
```

W wyniku otrzymamy 42. Cóż, to wiele wyjaśnia! Drugie zapytanie w naszej procedurze zwraca 42 wiersze; mówiąc inaczej, próbujemy przypisać 42 różne wartości do każdej z naszych zmiennych.

Bonusowa pomyłka

Jeśli zastanawiasz się, dlaczego `@AvgCPM` i `@AvgBidPrice` zachowują się inaczej, zwróć uwagę na typy danych. `@AvgCPM` jest typu `DECIMAL`, podczas gdy `@AvgBidPrice` jest typu `MONEY`. W związku z tym `@AvgBidPrice` wykorzystuje końcową wartość do zwrócenia.

To był niezamierzony błąd, który popełniłem podczas tworzenia tych przykładów, dlatego nie pojawia się on w przykładzie debugowania. Postanowiłem go jednak zostawić, ponieważ jest dość interesujący i daje nieoczekiwane objawy. Niemniej jednak powinniśmy zmienić `@AvgCPM` również na typ `MONEY`.

Powinniśmy zaktualizować drugą kwerendę w naszej procedurze składowanej tak, aby uzyskać średnie wartości, których potrzebujemy. Możemy to zrobić przez opublikowanie definicji procedury z listingu 7.16.

Listing 7.16. Poprawianie procedury składowanej

```

CREATE PROCEDURE dbo.CalculateCampaignSummary
    @CampaignID INT,
    @Budget MONEY
AS
BEGIN
    DECLARE @AvgCPM MONEY ;
    DECLARE @CampaignCost MONEY ;
    DECLARE @NoOfImp INT ;
    DECLARE @AvgBidPrice MONEY ;
    DECLARE @CostOfBidPerc DECIMAL ;
    DECLARE @BudgetDifference MONEY ;

    SELECT
        @NoOfImp = COUNT(*)
        , @CampaignCost = (SUM(CostPerMille) / 1000) * COUNT(*)
    FROM marketing.Impressions
    WHERE CampaignID = @CampaignID
    GROUP BY CampaignID ;

    SELECT
        @avgcpm = AVG(AvgCostPerMille)
        , @avgbidprice = AVG(AvgBidPrice)
    FROM reporting.ImpressionAggregates
    WHERE CampaignID = @CampaignID ;

    SET @CostOfBidPerc = (@AvgBidPrice / @AvgCPM) * 100 ;
    SET @BudgetDifference = @budget - @CampaignCost ;

    SELECT
        @CampaignID CampaignID
        , @AvgCPM AvgCPM
        , @CampaignCost CampaignCost
        , @NoOfImp ImpQty
        , @CostOfBidPerc CostToBidPercentage
        , @BudgetDifference ;
    SELECT DISTINCT
        ReferralURL
        , RenderingID
    FROM marketing.Impressions
    WHERE CampaignID = @CampaignID ;
END

```

UWAGA W przypadku projektów SQL Server Database okno *Immediate* ma pewne ograniczenia. W szczególności nie możemy uruchamiać w nim niczego, co wymagałoby wczytania środowiska wykonawczego. Ogólnie rzecz biorąc, oznacza to, że możemy zwracać wartości skalarne, takie jak liczniki, ale nie zbiory wyników. Oznacza to również, że niektóre inne mechanizmy, takie jak funkcje systemowe `ERROR_NUMBER()` czy `ERROR_MESSAGE()`, nie są dostępne. Zmienne systemowe są jednak dostępne. Dlatego możemy na przykład wykonać zapytanie `SELECT @@trancount`.

Teraz, gdy poprawiliśmy naszą procedurę składowaną, wykonajmy ją ponownie w trybie debugowania. Zanim to jednak zrobimy, porozmawiajmy krótko o punktach przerywania. *Punkt przerywania* to miejsce w kodzie, w którym

programista chce wstrzymać wykonywanie programu. Jest to niezwykle przydatna funkcja, ponieważ pozwala programiście na wykonywanie takich czynności, jak sprawdzanie wartości zmiennych i analizowanie ścieżek wykonania w logice programu.

Teraz, gdy wiemy już, czym jest punkt przerwania, ustawmy taki punkt w wierszu 26 (`SET @CostOfBidPerc = (@AvgBidPrice / @AvgCPM) * 100 ;`). Aby utworzyć prosty punkt przerwania, wystarczy kliknąć lewy margines obok miejsca, w którym chcemy zatrzymać wykonywanie kodu. Jeśli teraz rozpoczniemy debugowanie naszej procedury składowanej i użyjemy klawisza *F5* zamiast *F11*, wykonanie będzie kontynuowane aż do napotkania punktu przerwania. W tym momencie możemy sprawdzić w oknie *Locals*, czy nasze zmienne `@AvgCPM` i `@Avg ↵ BidPrice` mają oczekiwane wartości, zanim ponownie naciśniemy *F5*, aby pozwolić na dokończenie wykonania procedury.

OSTRZEŻENIE Visual Studio oferuje zaawansowane punkty przerwania, które pozwalają na określanie warunków. Warunki nie są jednak obsługiwane w T-SQL.

Wykorzystaj nowoczesne techniki debugowania w szablonie SQL Server Database Project w Visual Studio, aby skutecznie analizować kod. Oszczędzi to czas i sprawi, że proces debugowania stanie się znacznie mniej frustrujący.

7.4. Numer 29 — niestosowanie narzędzia Schema Compare

Przy wprowadzaniu dużych zmian w złożonym obiekcie programistycznym trudno jest śledzić wszystkie dokonane modyfikacje. Częstym błędem, który obserwuję u programistów T-SQL, jest wdrażanie obszernych zmian w kodzie bez pełnego zrozumienia ich konsekwencji. Taka pomyłka może prowadzić do wielu problemów w środowisku produkcyjnym.

Ponadto słabe procesy zarządzania zmianami i wdrażania, w połączeniu z różnymi poprawkami, mogą prowadzić do rozbieżności kodu między różnymi środowiskami. Na przykład jeśli mamy środowisko deweloperskie, testowe, przygotowawcze i produkcyjne, możliwe jest, że wdrożenia będą wykonywane w niewłaściwej sekwencji.

Aby uniknąć tych problemów, programiści mogą skorzystać z narzędzia *SSDT Schema Compare*. Pozwala ono porównać bazy danych w dwóch różnych środowiskach i wskazać różnice między nimi. Narzędzia Schema Compare można też użyć do zsynchronizowania baz danych.

Aby to zbadać, zaktualizujmy procedurę składowaną `CalculateCampaignSummary` w projekcie Marketing. W naszym podsumowaniu brakuje aliasu dla `@Budget ↵ Difference`, a ponadto chcemy dodać parametr, który będzie określał, czy

zwracany ma być drugi zbiór wyniku. Możemy wprowadzić te zmiany w definicji procedury, korzystając ze skryptu przedstawionego na listingu 7.17. Zapisz wprowadzone zmiany, ale na razie ich nie publikuj.

Listing 7.17. Poprawianie definicji procedury CalculateCampaignSummary

```
CREATE PROCEDURE dbo.CalculateCampaignSummary
    @CampaignID INT,
    @Budget MONEY,
    @Detailed BIT
AS
BEGIN
    DECLARE @AvgCPM MONEY ;
    DECLARE @CampaignCost MONEY ;
    DECLARE @NoOfImp INT ;
    DECLARE @AvgBidPrice MONEY ;
    DECLARE @CostOfBidPerc DECIMAL ;
    DECLARE @BudgetDifference MONEY ;

    SELECT
        @NoOfImp = COUNT(*)
        , @CampaignCost = (SUM(CostPerMille) / 1000) * COUNT(*)
    FROM marketing.Impressions
    WHERE CampaignID = @CampaignID
    GROUP BY CampaignID ;

    SELECT
        @avgcpm = AVG(AvgCostPerMille)
        , @avgbidprice = AVG(AvgBidPrice)
    FROM reporting.ImpressionAggregates
    WHERE CampaignID = @CampaignID ;

    SET @CostOfBidPerc = (@AvgBidPrice / @AvgCPM) * 100 ;
    SET @BudgetDifference = @budget - @CampaignCost ;

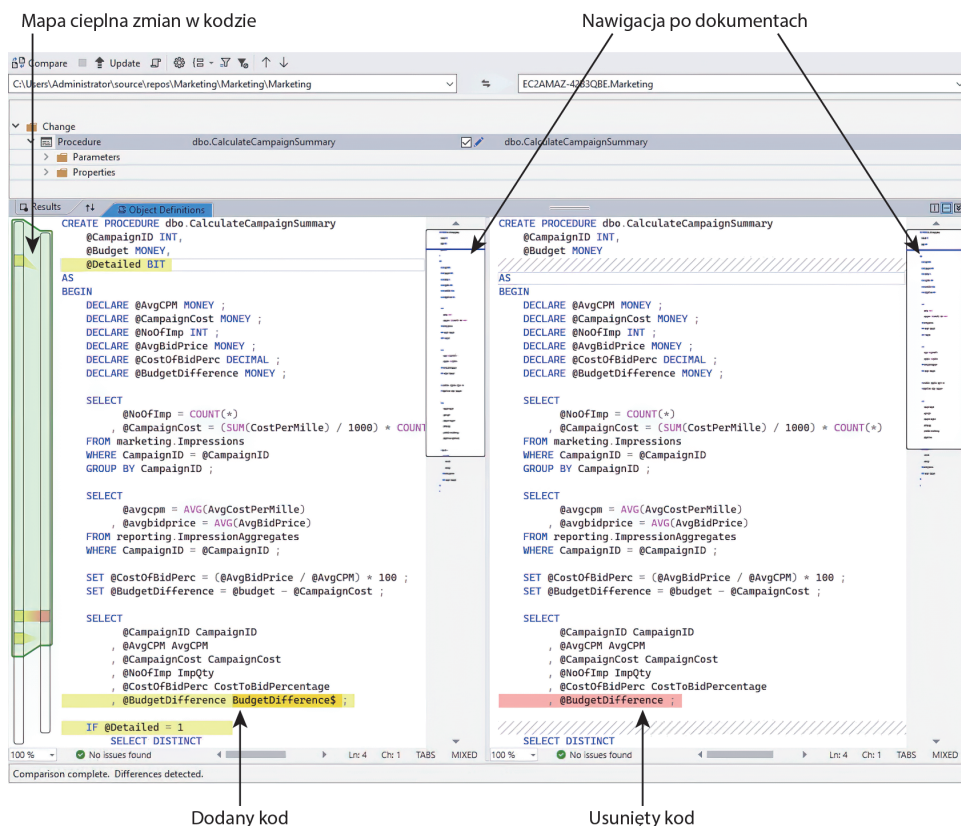
    SELECT
        @CampaignID CampaignID
        , @AvgCPM AvgCPM
        , @CampaignCost CampaignCost
        , @NoOfImp ImpQty
        , @CostOfBidPerc CostToBidPercentage
        , @BudgetDifference BudgetDifference$ ;

    IF @Detailed = 1
    BEGIN
        SELECT DISTINCT
            ReferralURL
            , RenderingID
        FROM marketing.Impressions
        WHERE CampaignID = @CampaignID ;
    END
END
```

Przed wdrożeniem zaktualizowanej procedury możemy przeanalizować wprowadzane zmiany. W tym celu wybierz opcję *Schema Compare* z menu kontekstowego projektu Marketing w SQL Server Object Explorer w Visual Studio.

Spowoduje to otwarcie okna *Compare*, w którym możesz użyć przycisku *Select Target* w prawym górnym rogu, aby wskazać bazę danych Marketing w naszej instancji. Kliknięcie przycisku *Compare* na górnym pasku narzędzi okna porównania uruchomi proces porównywania.

Wyniki tego porównania przedstawiono na rysunku 7.5.



Rysunek 7.5. Analiza porównawcza schematów

Górny panel okna pokazuje wszystkie zmienione obiekty. Po wybraniu konkretnego obiektu w dolnym panelu wyświetlane są definicje źródła i celu.

Zauważysz, że zmiany są podświetlone na żółto dla dodanych elementów i na czerwono dla usuniętych. Pasek po lewej stronie dolnego panelu pokazuje, w którym miejscu kodu nastąpiły zmiany. Możemy kliknąć ten pasek, aby przejść do interesującego nas obszaru. Pole po prawej stronie paneli źródłowego i docelowego wskazuje, która część kodu jest aktualnie widoczna na ekranie. Możesz je kliknąć, aby przenieść się do innych miejsc w kodzie.

Pasek narzędzi u góry okna zawiera przyciski do aktualizacji celu i generowania zaktualizowanego skryptu. Przycisk opcji pozwala ustawić reguły wykluczania obiektów i zdefiniować różnice, które powinny być ignorowane. Przycisk

grupowania obiektów przełącza między grupowaniem według typu obiektu a schematem. Dostępne są również przyciski do włączania i wyłączania wyświetlania nieobsługiwanych akcji oraz obiektów, które nie uległy zmianie. Możemy także użyć przycisków strzałek w górę i w dół, aby poruszać się między zmienionymi obiektami.

Przy wprowadzaniu zmian w dużych, złożonych obiektach powinniśmy przed wdrożeniem skorzystać z narzędzia Schema Compare. Pozwoli to upewnić się, że w pełni rozumiemy wprowadzane zmiany i nie nadpiszemy żadnych doraźnych poprawek, które nie zostały jeszcze uwzględnione w cyklu wdrożeniowym.

7.5. Numer 30 — niepisanie testów jednostkowych

Programiści psują rzeczy. To fakt. Bez względu na to, jak dobrzy jesteśmy, zawsze istnieje ryzyko, że podczas modyfikacji kodu w celu dodania nowej funkcji lub nawet naprawy błędu uszkodzimy coś innego. Może się to zdarzyć z wielu powodów — od zwykłej pomyłki po brak pełnej wiedzy o funkcjach i przeznaczeniu danego modułu kodu.

W tradycyjnym środowisku programistycznym SQL Servera, zwłaszcza takim, które rozwijało się organicznie przez lata, jest mało prawdopodobne, aby którykolwiek z programistów nadal pracujących nad aplikacją bazodanową był w pełni świadomy wszystkich wymagań spełnianych przez każdy moduł kodu w dużym projekcie. Jest to jeszcze bardziej widoczne w przypadku rozbudowanych i złożonych systemów.

Rozwiązaniem tego problemu jest napisanie testów jednostkowych. *Test jednostkowy* służy do sprawdzenia konkretnej funkcjonalności. Daje to dwie główne korzyści. Po pierwsze, zapobiega przypadkowemu uszkodzeniu istniejącej funkcjonalności przez programistę podczas aktualizacji modułu kodu. Po drugie, w duchu samodokumentacji, tworzy powiązanie między wymaganiem biznesowym a modułem kodu.

W nowoczesnej erze rozwoju oprogramowania niepisanie testów jednostkowych jest pomyłką, którą niestety obserwuję aż zbyt często w społeczności SQL Servera. Pomyłka ta jest jeszcze poważniejsza, jeśli aplikacja bazodanowa jest dostarczana w ramach *zwinnej metodyki projektowej*. W metodykach zwinnych, takich jak Scrum czy Kanban, aplikacja jest dostarczana w małych, iteracyjnych cyklach. Przynosi to wiele korzyści, takich jak możliwość szybszego dostarczenia minimalnego działającego produktu oraz łatwiejsze dostosowywanie się do zmieniających się potrzeb biznesowych.

Przy takiej metodyce projektowej istnieje jednak większe prawdopodobieństwo, że później trzeba będzie wracać do większej liczby modułów kodu, co zwiększa ryzyko uszkodzenia wcześniej napisanej funkcjonalności. To sprawia, że testy jednostkowe stają się jeszcze ważniejsze.

Przykład z życia wzięty

W okresie od 2000 do 2010 r. miałem przywilej kierować rozwojem niektórych z największych aplikacji bazodanowych tworzonych w tym czasie w Londynie. Różnica w metodyce na przestrzeni tego okresu była ogromna. Zarówno pierwszy, jak i ostatni z tych projektów trwały kilka lat, angażowały wiele firm i w szczytowym momencie były realizowane przez ponad 20 programistów. Oba projekty dotyczyły przetwarzania od 40 do 50 TB danych, co wymagało złożonych operacji.

Jeden z projektów obejmował tworzenie i analizę ścieżek kliknięć na podstawie rozproszonych źródeł danych, obejmujących dwie wyszukiwarki i cztery serwery plików cookie. Trzeba było obliczyć pełną ścieżkę użytkownika, od wyświetlenia baneru reklamowego lub wyszukiwania w wyszukiwarce aż do faktycznego zakupu produktu na stronie reklamodawcy.

Drugi projekt zawierał złożone dane, w tym 30-godzinny zegar i hierarchię 36 nakładających się regionów reklamowych, przy czym odsłony reklamy trzeba było odwzorowywać na odpowiedni region i właściwy poziom (lub czasem poziomy) hierarchii. Obie aplikacje zostały napisane w środowisku SQL Servera i ostatecznie każda z nich liczyła ponad milion wierszy kodu.

Pierwszy z tych projektów wykorzystywał tradycyjną metodykę rozwoju. Kod pisaaliśmy na serwerze deweloperskim. Moduły kodu były przechowywane w Team Foundation Server (TFS). TFS miał wiele funkcji; jedną z nich była forma pesymistycznej kontroli wersji. Przenosiliśmy kod do środowiska testowego (ang. *User Acceptance Testing*, UAT) za pomocą ręcznie przygotowanych skryptów. Następnie zespół biznesowy testował nowe funkcje, zanim kod został ręcznie przeniesiony do środowiska przedprodukcyjnego.

Wszystko szło dobrze aż do ostatniej fazy testów projektu. W tym momencie odkryto wiele błędów, które powstały, gdy programiści dodawali nowe funkcje, psując przy tym te już istniejące. Nikt tego nie zauważył, ponieważ przy wdrażaniu nowego kodu testowano tylko nowe funkcje. Co gorsza, ponieważ projekt trwał ponad dwa lata, niektórzy programiści, którzy napisali część początkowego kodu, już odeszli z firmy. Musieliśmy więc na nowo rozgryźć, do czego miały służyć poszczególne moduły. W rezultacie uruchomienie systemu opóźniło się o dziewięć miesięcy. Nie chciałbym przeżyć tego ponownie!

W ostatnim z tych projektów, korzystając z nabytych doświadczeń, zastosowaliśmy bardziej nowoczesne podejście. Przechowywaliśmy nasz kod na GitHubie, co pozwoliło nam na efektywniejszą pracę zespołową (więcej na ten temat w następnym podrozdziale). Wykorzystaliśmy również Jenkins i Octopus Deploy do stworzenia potoku CI/CD (patrz punkt 7.6.2). Utworzyliśmy testy jednostkowe dla każdego modułu kodu, który napisaliśmy. Potok CI/CD wykonywał te testy, a proces budowania kończył się niepowodzeniem, jeśli testy nie kończyły się pomyślnie. Wymagało to nieco więcej wysiłku na początku, ale oznaczało, że pod koniec projektu nie mieliśmy ogromnej liczby błędów do naprawienia, a projekt został uruchomiony na czas.

Zastosujmy dobre praktyki i utwórzmy test jednostkowy dla naszej procedury składowanej `CalculateCampaignSummary`. Aby rozpocząć ten proces, wybierzemy opcję *Create Unit Tests* z menu kontekstowego naszej procedury w oknie *SQL Server Object Explorer* w Visual Studio. Spowoduje to wyświetlenie okna dialogowego *Create Unit Tests*. W górnej części tego okna możemy wybrać obiekty do testowania. W dolnej części możemy zdecydować, czy projekt testowy, który

zostanie utworzony jako dodatkowy projekt w naszym rozwiązaniu, ma być oparty na C# czy Visual Basicu. Szczerze mówiąc, w przypadku pisania testów jednostkowych dla SQL Servera nie ma to większego znaczenia. Ja zazwyczaj wybieram C#, ale tylko dlatego, że lepiej znam ten język. Jeśli masz już utworzony projekt testowy, będziesz mógł wybrać go z tej listy rozwijanej. Nadaj projektowi nazwę i określ nazwę klasy wyjściowej.

Plik testu jednostkowego zostanie wyświetlony w wygodnym edytorze. Jeśli mamy wiele testów, możemy przełączać się między nimi za pomocą rozwijanej listy w lewym górnym rogu okna. Na liście rozwijanej po prawej stronie możemy wybrać skrypt wstępny, skrypt testowy lub skrypt końcowy.

Skrypt wstępny służy do przygotowania środowiska testowego. Może to obejmować takie zadania jak wypełnienie tabeli danymi przed wykonaniem testu. Ponieważ będziemy uruchamiać nasz skrypt na instancji SQL Servera, nie ma potrzeby tworzenia osobnego skryptu wstępnego.

Skrypt końcowy służy do robienia porządków. Gdybyśmy na przykład testowali procedurę składowaną, która dodaje zamówienie sprzedaży, użylibyśmy skryptu końcowego do usunięcia tego zamówienia.

W skrypcie testowym będziemy wykonywać nasz moduł kodu. Szablon kodu zostanie przygotowany przez Visual Studio, więc wystarczy, że dodamy wartości wymaganych parametrów. W naszym teście użyjemy CampaignID o wartości 27587 oraz Budget o wartości 10, a parametr Detailed ustawimy na 1.

Następnie chcemy stworzyć cztery warunki testowe. Pierwsze trzy będą warunkami dla wartości skalarnych, które skonfigurujemy tak, aby sprawdzić, czy następujące stwierdzenia są prawdziwe:

- Column 2 = 1,7876,
- Column 5 = 195,
- Column 6 = 6,85.

Ostatnim warunkiem będzie *Row Count*. Sprawdzi on, czy drugi zbiór wyników zwraca dokładnie 42 wiersze. Możemy usunąć automatycznie wygenerowany warunek nierozstrzygający.

Aby usunąć warunek nierozstrzygający, zaznacz go w dolnym panelu i użyj przycisku z czerwonym krzyżykiem. Następnie możesz dodać pożądane warunki, wybierając odpowiedni typ z listy rozwijanej i klikając zielony przycisk dodawania. Każdy warunek można skonfigurować w oknie właściwości. Rysunek 7.6 przedstawia okno kodu testowego oraz warunki testowe.

Po zapisaniu testu możemy go uruchomić, przechodząc do okna *Test Explorer* i klikając ikonę odtwarzania na pasku narzędzi po lewej stronie. Wyłączenie filtrów poprzez kliknięcie ikony lejka na pasku narzędzi spowoduje wyświetlenie wszystkich testów wraz z ich statusem. Możemy przejrzeć hierarchię testów i kliknąć wybrany test, aby zobaczyć szczegółowe informacje.

The screenshot shows the SQL Server Enterprise Manager interface for a unit test named 'dbo.CalculateCampaignSummaryTest1'. The test code is as follows:

```
-- database unit test for dbo.CalculateCampaignSummary
DECLARE @RC AS INT, @CampaignID AS INT, @Budget AS MONEY, @Detailed AS BIT;

SELECT @RC = 0,
       @CampaignID = 27587,
       @Budget = 10,
       @Detailed = 1;

EXECUTE @RC = [dbo].[CalculateCampaignSummary] @CampaignID, @Budget, @Detailed;

SELECT @RC AS RC;
```

Below the code, the test conditions are listed in a table:

| Name | Type | Value | Enabled |
|-----------------------|--------------|---|---------|
| scalarValueCondition1 | Scalar Value | The condition fails if the value in ResultSet 1 Row 1 Column 2 is not '1.7876'. | True |
| scalarValueCondition2 | Scalar Value | The condition fails if the value in ResultSet 1 Row 1 Column 5 is not '195'. | True |
| scalarValueCondition3 | Scalar Value | The condition fails if the value in ResultSet 1 Row 1 Column 6 is not '6.85'. | True |
| rowCountCondition1 | Row Count | 42 rows must be returned in ResultSet 2. | True |

Rysunek 7.6. Tworzenie testu jednostkowego

Aby przekonać się, jak wygląda test, który zakończył się niepowodzeniem, wróć do utworzonego wcześniej testu, zmień wartość jednego z oczekiwanych wyników i uruchom test ponownie.

WSKAZÓWKA Warto pamiętać, że jeśli test kończy się niepowodzeniem, nie zawsze oznacza to błąd w kodzie. Możliwe, że błędny jest sam test!

Zawsze powinniśmy pisać testy jednostkowe dla naszych modułów kodu T-SQL. Jest to szczególnie ważne w przypadku modułów, które zawierają rozgałęzienia lub modyfikują dane. Taka praktyka zmniejsza ryzyko błędów spowodowanych dodawaniem nowych funkcji oraz pomaga w tworzeniu samodokumentującego się kodu poprzez powiązanie wymagań biznesowych z konkretnymi modułami.

7.6. Nowoczesne techniki programowania

W kolejnych punktach omówimy nowoczesne techniki programistyczne, które deweloperzy SQL Server powinni stosować do zarządzania swoim kodem, choć często tego nie robią. Pierwszą z tych technik jest wykorzystanie systemu kontroli wersji. Drugą jest użycie potoku CI/CD do zarządzania wdrożeniami kodu.

Dlaczego właściwie powinniśmy zajmować się tymi technikami? Co jest nie tak ze starymi, dobrze znanymi praktykami stosowanymi przez specjalistów od baz danych? W przeszłości często zdarzało się, że jedyna kopia schematu bazy danych poza środowiskiem produkcyjnym znajdowała się na serwerze deweloperskim. Programiści wprowadzali zmiany bezpośrednio na tym serwerze aż do wdrożenia na produkcji. W niektórych środowiskach, po utworzeniu obiektów, tworzono ich skrypty i przechowywano je jako kod na innej platformie. Czasami był to komputer programisty, czasami SharePoint, a w bardziej zaawansowanych przypadkach skrypty mogły być nawet przechowywane w Team Foundation Server.

Jeśli chodzi o wdrożenia, w przeszłości powszechną praktyką było tworzenie bardzo skomplikowanych skryptów wdrożeniowych, które programiści następnie przekazywali administratorom baz danych (DBA) do wykonania na systemach produkcyjnych, często bez odpowiedniego kontekstu. Wszyscy zainteresowani mocno trzymali kciuki, licząc na to, że w trakcie wdrożenia nie pojawi się żaden problem specyficzny dla danego środowiska.

Dlaczego te techniki były tak problematyczne? Rozważmy następujący przykład. Programiści MagicChoc postanowili zastosować tradycyjne podejście do przechowywania i wdrażania kodu. Konkretnie, przechowują swój kod w instancji deweloperskiej. Uważają to za akceptowalne rozwiązanie, ponieważ wykonują cotygodniową kopię zapasową środowiska programistycznego, dzięki czemu w razie potrzeby mogą odtworzyć definicje obiektów z backupu.

Stosują również ręczny proces wdrażania. Polega on na tym, że programiści piszą złożone skrypty, które eksportują dane do tymczasowych tabel, usuwają i odtwarzają tabele oraz obiekty programowalne, a następnie importują dane z powrotem.

Zespół jest gotowy do wdrożenia nowej wersji bazy danych w środowisku produkcyjnym. W tym celu przesyła skrypty do administratora bazy danych z prośbą o ich uruchomienie podczas najbliższego okna konserwacji. Gdy nadchodzi czas prac konserwacyjnych, administrator uruchamia skrypt. Niestety, w trakcie jego wykonywania system Windows rozpoczyna instalację aktualizacji i restartuje serwer. W rezultacie baza danych pozostaje w niespójnym stanie. Część obiektów została zaktualizowana, inne nie. Co gorsza, część danych została utracona.

Okno konserwacji jest krótkie, więc w powietrzu czuć napięcie. Administrator bazy danych postanawia przywrócić bazę z kopii zapasowej wykonanej tuż przed incydem. W pośpiechu odtwarza jednak bazę w instancji deweloperskiej, a nie produkcyjnej. Zorientowawszy się w pomyłce, przywraca bazę również w środowisku produkcyjnym.

Programiści ze zrozumieniem podchodzą do pomyłki administratora bazy danych i proszą go o przywrócenie bazy rozwojowej na serwerze deweloperskim. Niestety, gdy administrator próbuje to zrobić, okazuje się, że utworzenie ostatniej kopii zapasowej serwera deweloperskiego nie powiodło się, więc najnowsza kopia

pochodzi sprzed 13 dni. W rezultacie programiści stracili prawie dwa tygodnie pracy, a cały wysiłek włożony w przygotowanie nowej wersji poszedł na marne, i będą musieli zacząć wszystko od nowa.

UWAGA Choć ten scenariusz może wydawać się mało prawdopodobny, jest oparty na doświadczeniu mojego znajomego, który pracuje jako programista SQL.

Jak więc firma MagicChoc mogła uniknąć tych problemów? Mówiąc prosto, wystarczyło zastosować nowoczesne praktyki programistyczne. Gdyby firma przechowywała swoje dane w systemie kontroli wersji, mogłaby szybko i łatwo odzyskać swój kod. Gdyby korzystała z potoku CI/CD, mogłaby zminimalizować ryzyko wystąpienia takich problemów. Oczywiście, lepsze zarządzanie wydaniem mogłoby całkowicie zapobiec tej sytuacji, ale to już zupełnie inna historia!

W kolejnych punktach przyjrzymy się pracy z GitHubem oraz koncepcjom ciągłej integracji i ciągłego wdrażania (CI/CD) w kontekście wdrożeń SQL Servera. Pozwoli Ci to zrozumieć, jak firma MagicChoc mogła lepiej zorganizować swoje działania. Zachęcam Cię jednak do dalszego zgłębiania obu tych tematów.

7.6.1. Numer 31 — niekorzystanie z kontroli wersji kodu źródłowego

W naszym scenariuszu firma MagicChoc nie była w stanie odzyskać kodu, ponieważ polegała wyłącznie na serwerze deweloperskim z kopiami zapasowymi. Aby uniknąć tego błędu, programiści powinni byli przechowywać kod w systemie kontroli wersji.

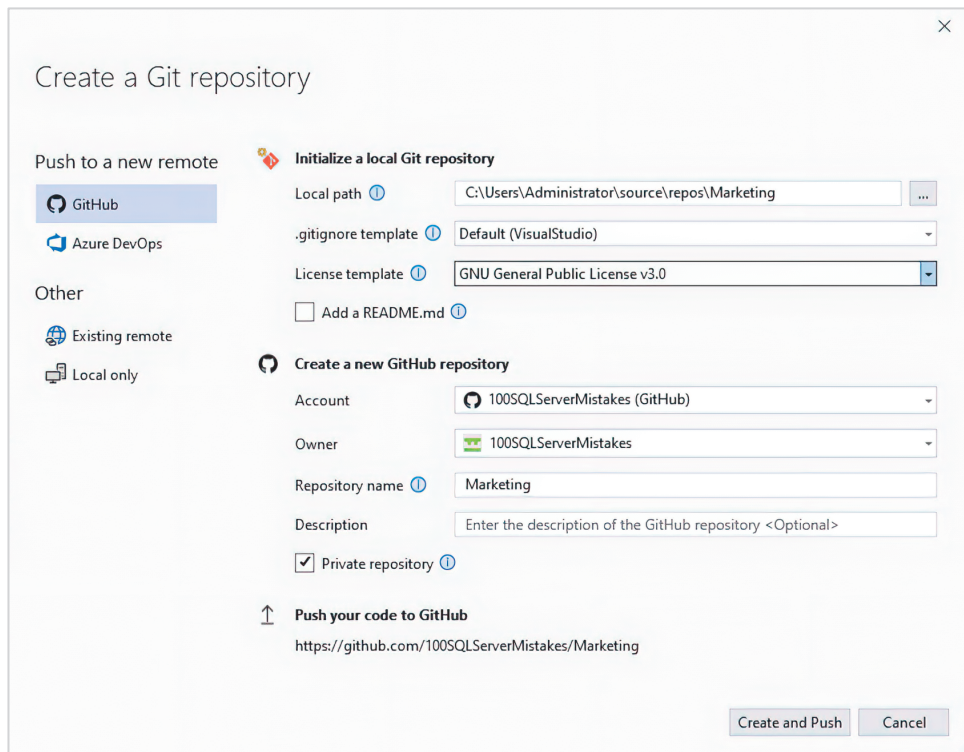
Wraz z rozwojem **Gita**, który jest systemem kontroli wersji o otwartym kodzie źródłowym, oraz od czasu wprowadzenia usług takich jak **GitHub**, czyli hostowany system SaaS do zarządzania repozytoriami Git, kontrola wersji stała się znacznie lepszym sposobem zarządzania kodem.

Integracja z Gitem umożliwia wielu programistom równoczesną pracę nad tymi samymi elementami projektu. Kontrola wersji działa w sposób optymistyczny, co oznacza, że pliki nie są blokowane, więc mogą pojawić się konflikty, którymi jednak można zarządzać. Mamy dostęp do historii zmian, dzięki czemu w przypadku wprowadzenia błędu można łatwo przywrócić poprzednią wersję. Oznacza to również, że kopia naszego kodu jest przechowywana w znanym, bezpiecznym i niezawodnym miejscu. Wszystkie te korzyści płynące z kontroli wersji sprawiają, że rezygnacja z jej stosowania w dzisiejszych czasach jest oczywistą pomyłką, którą niestety wciąż popełnia wielu programistów SQL.

Aby jej uniknąć, w projekcie SQL Server Database w Visual Studio przejdź do okna *Git Changes* i wybierz opcję *Create Git Repository*. Spowoduje to wyświetlenie okna dialogowego *Create Git Repository*.

Jak pokazano na rysunku 7.7, górna część ekranu pozwala określić ustawienia początkowe repozytorium. Konkretnie chodzi o lokalną ścieżkę do repozytorium

na komputerze, szablon licencji (jeśli jest używany) oraz szablon pliku `.gitignore`. Ten ostatni służy do wskazania w repozytorium plików, które nie powinny być śledzone przez Gita. W dolnej części okna określamy konto GitHub, którego chcemy użyć. Kliknięcie w tym miejscu otworzy stronę logowania do GitHuba. Podajemy też nazwę repozytorium, opcjonalny opis oraz właściciela.



Rysunek 7.7. Tworzenie i konfigurowanie repozytorium na GitHubie

Teraz, po skonfigurowaniu GitHuba, naszym pierwszym zadaniem będzie utworzenie gałęzi. Będzie to nasza gałąź robocza, którą możemy dopracować przed połączeniem jej z główną gałęzią projektu, zwykle nazywaną `master` lub `main`. Aby ją utworzyć, kliknij rozwijaną listę gałęzi u góry okna *Git Changes* i wybierz opcję *New branch*. W oknie dialogowym *Create a New Branch* nadaj nazwę gałęzi (jeśli chcesz robić wszystko tak, jak w przykładzie, możesz nazwać ją *CalculateCampaignSummary*). Następnie wybierz gałąź bazową, którą w naszym przypadku będzie gałąź główna. Pamiętaj, aby zaznaczyć opcje wyrejestrowania nowej gałęzi i śledzenia zmian w zdalnym repozytorium.

UWAGA W momencie pisania tego tekstu domyślną nazwą gałęzi głównej w Gicie wciąż jest `master`, ale na szczęście planowane jest wkrótce zastąpienie jej bardziej inkluzywną nazwą.

Aby zobaczyć, jak zmiany w kodzie działają w praktyce, dokonajmy niewielkiej modyfikacji w naszej procedurze składowanej `CreateCampaignSummary`. Skrypt z listingu 7.18 dodaje blok `BEGIN...END` wokół gałęzi `IF`.

WSKAZÓWKA W bloku `IF` użycie `BEGIN...END` jest konieczne, gdy gałąź zawiera wiele instrukcji, ale opcjonalne, gdy zawiera tylko jedną instrukcję.

Listing 7.18. Dokonywanie zmiany w procedurze `CalculateCampaignSummary`

```
CREATE PROCEDURE dbo.CalculateCampaignSummary
    @CampaignID INT,
    @Budget MONEY,
    @Detailed BIT
AS
BEGIN
    DECLARE @AvgCPM MONEY ;
    DECLARE @CampaignCost MONEY ;
    DECLARE @NoOfImp INT ;
    DECLARE @AvgBidPrice MONEY ;
    DECLARE @CostOfBidPerc DECIMAL ;
    DECLARE @BudgetDifference MONEY ;

    SELECT
        @NoOfImp = COUNT(*)
        , @CampaignCost = (SUM(CostPerMille) / 1000) * COUNT(*)
    FROM marketing.Impressions
    WHERE CampaignID = @CampaignID
    GROUP BY CampaignID ;

    SELECT
        @avgcpm = AVG(AvgCostPerMille)
        , @avgbidprice = AVG(AvgBidPrice)
    FROM reporting.ImpressionAggregates
    WHERE CampaignID = @CampaignID ;

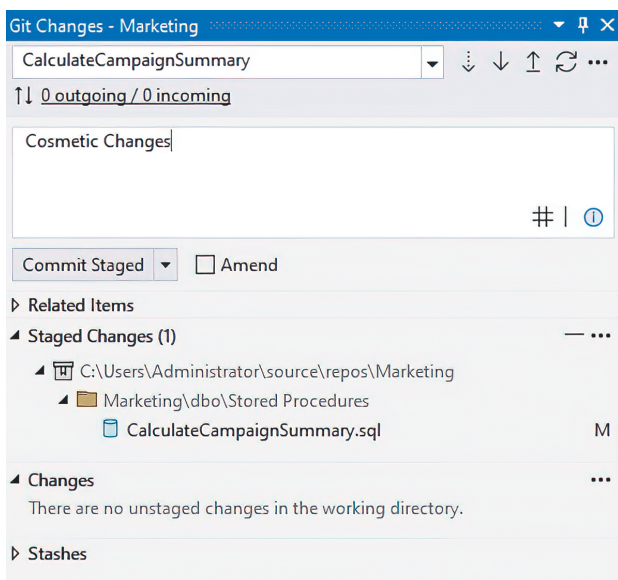
    SET @CostOfBidPerc = (@AvgBidPrice / @AvgCPM) * 100 ;
    SET @BudgetDifference = @budget - @CampaignCost ;

    SELECT
        @CampaignID CampaignID
        , @AvgCPM AvgCPM
        , @CampaignCost CampaignCost
        , @NoOfImp ImpQty
        , @CostOfBidPerc CostToBidPercentage
        , @BudgetDifference BudgetDifference$ ;

    IF @Detailed = 1
    BEGIN
        SELECT DISTINCT
            ReferralURL
            , RenderingID
        FROM marketing.Impressions
        WHERE CampaignID = @CampaignID ;
    END
END
```

Po zapisaniu pliku spójrzmy ponownie na okno zmian *Git Changes*. Zostało ono odświeżone, aby pokazać zmodyfikowany element. Kliknięcie plusa obok zmiany spowoduje jej dodanie do poczekalni. Odwrócona strzałka spowoduje cofnięcie zmiany. Kliknijmy ikonę plusa, aby dodać zmianę do poczekalni.

Zmiany zostaną przeniesione do folderu *Staged Changes*. Teraz możemy dodać komentarz do zatwierdzenia i zatwierdzić zmiany, klikając przycisk *Commit Staged*, jak pokazano na rysunku 7.8.



Rysunek 7.8. Zatwierdzanie zmian

Kliknięcie przycisku strzałki w górę, znajdującego się obok rozwijanej listy gałęzi, spowoduje przekazanie zmian do zdalnej gałęzi. W rezultacie na górze okna *Git Changes* pojawi się żółty baner z wiadomością zawierającą link do utworzenia prośby o scalenie zmian (ang. *pull request*).

Jeśli przekazaliśmy kod do gałęzi głównej, możemy po prostu użyć rozwijanej listy gałęzi, aby przełączyć się na gałąź główną, a następnie użyć przycisku ze strzałką w dół, aby pobrać zmiany z gałęzi głównej do naszego lokalnego repozytorium. Jeśli jednak przekazaliśmy zmiany do gałęzi zdalnej, możemy skorzystać z opcji *Create Pull Request*, która przeniesie nas na stronę GitHuba. Tam możemy utworzyć prośbę o przejrzenie naszych zmian przez współtwórcę repozytorium i scalenie ich z gałęzią główną.

WSKAZÓWKĄ Git i GitHub to tematy, którym można by poświęcić osobną książkę.

Jeśli nie korzystasz z tych narzędzi, gorąco zachęcam, żeby się z nimi bliżej zapoznać, ponieważ w tym podrozdziale opisałem tylko absolutne podstawy. Dobrym punktem wyjścia jest strona z materiałami edukacyjnymi dotyczącymi Gita i GitHuba dostępna pod adresem <https://mng.bz/JN90>.

7.6.2. Numer 32 — nieużywanie potoku CI/CD do wdrażania kodu

W naszym scenariuszu problem z wdrożeniem wynikał częściowo z metodyki wprowadzania nowej wersji bazy danych do środowiska produkcyjnego. Firma MagicChoc mogła uniknąć tego błędu i zmniejszyć ryzyko wdrożeniowe przez „zapakowanie” swojej aplikacji i wdrożenie jej za pomocą potoku CI/CD.

Procesy te przynoszą wiele korzyści — skracają czas wprowadzania kolejnych wersji oprogramowania, zmniejszają ryzyka wprowadzania błędów do środowiska produkcyjnego oraz upraszczają i przyspieszają proces wdrożeniowy. Podejście to doskonale wpisuje się w dzisiejsze zwinne metodyki projektowe.

Tworzenie potoku CI/CD to złożony proces wymagający użycia wielu narzędzi, które zazwyczaj są strategicznie dobierane przez organizację. Pełne wyjaśnienie konfiguracji tych narzędzi musiałoby być dostosowane do ich konkretnego zestawu i zasługiwałoby na osobną książkę. Wybrane elementy potoku zależałyby również od wymagań danego przedsiębiorstwa. Dlatego zamiast próbować wyjaśniać szczegółową konfigurację, omówię tylko ogólne koncepcje; zachęcam czytelników do dalszego zgłębiania tematu we własnym zakresie.

Zacznijmy jednak od omówienia *aplikacji warstwy danych* (ang. *data-tier application*, DAC). Aplikacje DAC odgrywają istotną rolę w środowisku ciągłego dostarczania, ponieważ tworzą pakiet zawierający wszystkie elementy składające się na DAC i nadają mu unikatowy numer wersji, który może być wykorzystywany w procesie CD. Pakiet ten obejmuje wszystkie obiekty bazy danych, takie jak tabele, procedury i użytkownicy, a także obiekty na poziomie instancji, jak identyfikatory logowania, od których zależy baza danych

Bazę danych można zarejestrować jako aplikację DAC na kilka sposobów. Możemy na przykład zarejestrować DAC z menu kontekstowego bazy danych w SSMS lub, zgodnie z koncepcjami omawianymi w tym rozdziale, zarejestrować projekt jako DAC w Visual Studio.

Aby zarejestrować DAC w Visual Studio, wystarczy opublikować projekt. Po jego zbudowaniu pojawi się okno dialogowe *Deployment*. W oknie tym znajduje się pole wyboru *Register As a Data-tier Application*. Jeśli zaznaczymy tę opcję, pojawi się dodatkowe pole wyboru umożliwiające zablokowanie wdrożenia, jeśli docelowa baza danych uległa zmianie.

Plik *dacpac*, który jest wynikiem procesu kompilacji, to w rzeczywistości archiwum ZIP zawierające pliki XML z definicjami obiektów bazy danych. Ponieważ jest to zwykły plik ZIP, możemy łatwo zbadać jego zawartość, zmieniając rozszerzenie pliku z *.dacpac* na *.zip*.

Główne elementy w procesie CI/CD dla baz danych SQL to:

- stanowiska deweloperskie,
- usługi kompilacyjne,
- usługi wdrożeniowe,

- kontrola wersji,
- środowiska.

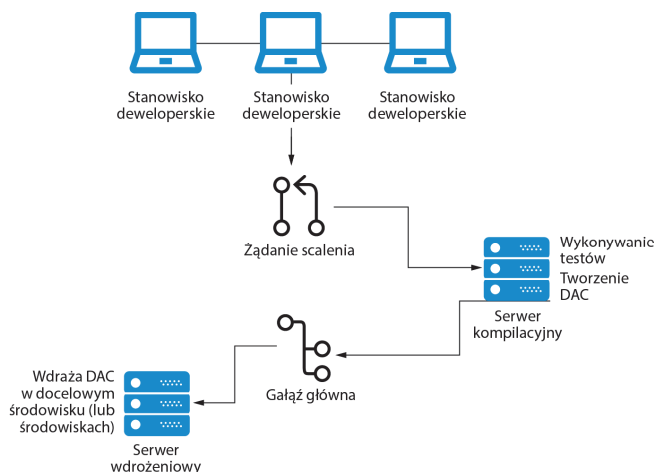
Stanowiska programistyczne są wyposażone w narzędzia deweloperskie, takie jak Visual Studio, SSDT, Git, Visual Studio Code. Programiści tworzą elementy lokalnie, a następnie zatwierdzają je w gałęziach w swoim repozytorium kontroli wersji i zgłaszają prośbę o scalenie zmian. Prośba ta często wyzwala usługi kompilacji. Usługi te mogą być koordynowane za pomocą narzędzi takich jak GitHub Actions, potoki Azure DevOps czy Jenkins.

Usługi kompilacji uruchamiają testy jednostkowe, o których mówiliśmy w poprzednim podrozdziale, co daje pewność, że nowy kod nie wprowadził żadnych błędów. Następnie zostanie utworzony pakiet DAC. Wynikiem tego procesu będzie plik dacpac.

Jeśli kompilacja zakończy się powodzeniem i kod zostanie scalony z główną gałęzią repozytorium, można przystąpić do wdrożenia DAC. Usługi wdrożeniowe, które mogą obejmować takie narzędzia jak Octopus Deploy, Azure DevOps czy Redgate Deploy, odpowiadają za wdrożenie DAC w najniższym środowisku na naszym stosie wydań. W niektórych projektach może to być kolejne środowisko deweloperskie, w innych — środowisko do testów integracyjnych, a w nielicznych przypadkach — środowisko UAT lub przygotowawcze. Po zatwierdzeniu kodu w danym środowisku usługi wdrożeniowe można wykorzystać do przeniesienia aplikacji DAC do kolejnego środowiska, aż ostatecznie trafi ona na produkcję.

UWAGA Chociaż poziom automatyzacji powinien być z założenia wysoki, różne środowiska będą miały różne punkty kontrolne wymagające interwencji człowieka. Na przykład możesz skonfigurować swoje środowisko tak, aby człowiek zatwierdzał żądanie scalenia, a usługi kompilacji budowania były uruchamiane po scaleniu zmian z gałęzią główną.

Typowy proces CI/CD dla bazy danych SQL Servera pokazano na rysunku 7.9.



Rysunek 7.9. Proces CI/CD

WSKAZÓWKA Dodatkowe materiały, które warto przestudiować, zależą prawdopodobnie od narzędzi, których używa Twoja organizacja. Jednak na blogu Microsoftu znajdziesz serię interesujących przewodników, które pokazują, jak korzystać z GitHuba, GitHub Actions i Visual Studio Code. Pierwszy wpis z tej serii znajdziesz pod adresem <https://mng.bz/M18W>. Możesz również sięgnąć po książkę wydawnictwa Manning *Grokking Continuous Delivery* autorstwa Christie Wilson, dostępną na stronie <https://www.manning.com/books/grokking-continuous-delivery>.

Powinniśmy rozważyć wdrożenie potoków CI/CD dla naszych projektów bazodanowych. Takie podejście pomoże działać elastyczniej, skrócić czas wprowadzania produktów na rynek oraz uniknąć ryzyka związanego ze złożonością systemów. Wykorzystanie DAC może uprościć proces wdrażania poprzez pakowanie wszystkich niezbędnych elementów i nadawanie im numeru wersji.

Podsumowanie

- ACID, zbiór podstawowych reguł dla transakcji, mówi, że powinny one być atomowe, spójne, izolowane i trwałe.
- Opcja `XACT_ABORT` służy do określenia, czy cała transakcja zostanie przerwana, gdy błędy o niskiej krytyczności spowodują niepowodzenie wykonania instrukcji.
- Zawsze należy implementować obsługę błędów w kodzie. Do przechwytywania błędów używaj konstrukcji `TRY...CATCH`.
- Używaj `THROW` i `RAISERROR()` do zgłaszania zrozumiałych komunikatów o błędach.
- Poziomy krytyczności błędu, w zakresie od 0 do 24, określają, jak poważny jest błąd — od komunikatów informacyjnych aż po krytyczne awarie sprzętu lub oprogramowania.
- Jeśli aplikacja ma procesy działające bez nadzoru, warto rozważyć skonfigurowanie alarmów, które powiadamią o błędach. Dzięki temu zespół wsparcia będzie mógł szybko zareagować na problemy.
- Jeśli to możliwe, użyj narzędzia do monitorowania środowiska przedsiębiorstwa w celu generowania alarmów. W przeciwnym razie skorzystaj z funkcji Database Mail oraz podsystemu alarmów SQL Server Agent.
- Wykorzystaj Visual Studio do debugowania kodu. Może to zaoszczędzić czas i ułatwić rozwiązywanie problemów.
- Projekty bazodanowe w Visual Studio mają wbudowaną funkcję porównywania schematów SQL. Warto z niej korzystać, aby upewnić się, że kod nie uległ niepożądanym zmianom przed wdrożeniem.

- Wykorzystaj funkcję Schema Compare w SSDT, aby dokładnie zrozumieć planowane zmiany i ich wpływ na funkcjonalność systemu.
- Zawsze przechowuj kod T-SQL w repozytorium kontroli wersji. Zapewni to historię zmian i mechanizm wycofywania modyfikacji, a także ułatwi pracę nad projektem, w którym uczestniczy wielu programistów.
- Zawsze pisz testy jednostkowe dla obiektów programowalnych. Choć początkowo potrwa to trochę dłużej, to w dalszej perspektywie zaoszczędzisz znacznie więcej czasu przy poprawianiu błędów wpływających na istniejące funkcje.
- Stosuj nowoczesne techniki wdrożeniowe. Potok CI/CD może zapewnić wiele korzyści, w tym skrócenie czasu wprowadzania produktu na rynek i zmniejszenie złożoności procesu wdrożeniowego.

Instalacja SQL Servera



W tym rozdziale:

- Znaczące nazwy instancji
- Wybór odpowiedniej wersji systemu operacyjnego i edycji SQL Servera
- Instalacja zautomatyzowana
- Rozmiar SQL Servera
- Aspekty instalacji w środowisku chmurowym

W tym rozdziale omówimy szeroką gamę opcji dostępnych przy instalacji SQL Servera. Jeszcze dekadę temu planowanie wdrożenia SQL Servera było stosunkowo proste. Należało wybrać najbardziej odpowiednią edycję SQL Servera i zastanowić się nad sposobem instalacji (graficzny interfejs użytkownika czy skrypt). Trzeba było rozważyć, jakie funkcje będą potrzebne dla danej instancji i nadać jej odpowiednią nazwę, ale to w zasadzie było wszystko.

Przenieśmy się do czasów obecnych, w których możliwości instalacji SQL Servera są praktycznie nieograniczone. Może to jednak prowadzić do licznych pomyłek, z których wiele wynika z przeprowadzania tradycyjnych instalacji bez rozważenia lepszych alternatyw.

W tym rozdziale zainstalujemy kilka instancji SQL Servera dla firmy MagicChoc i przyjrzymy się pułapkom, które mogą czyhać na przypadkowego administratora bazy danych. Omówimy różne aspekty planowania i przeprowadzania wdrożeń, od nadawania nazw instancjom po wybór odpowiedniego środowiska systemu operacyjnego i edycji SQL Servera. Przyjrzymy się również wyzwaniom związanym z ręcznymi instalacjami, instalowaniem niepotrzebnych funkcji oraz nadmierną konsolidacją instancji.

8.1. Numer 33 — stosowanie niejasnych nazw instancji

W rozdziale 2. omówiliśmy standardy nazewnictwa i konsekwencje wynikające z używania nieprzemyślanych nazw obiektów. Podobne problemy mogą wystąpić w przypadku instancji SQL Servera, jeśli nie zastanowimy się nad ich nazwami. Aby to zilustrować, wyobraźmy sobie, że firma MagicChoc poprosiła nas o utworzenie czterech instancji SQL Servera rozdzielonych między dwa serwery. Instancje te będą obsługiwać bazy danych używane przez następujące aplikacje:

- Choc Maker, stosowaną do produkcji,
- Magic Sales, aplikację dla zespołów sprzedażowych,
- Temperature Smart, aplikację wykorzystywaną w procesie produkcyjnym,
- HR Manager, narzędzie wykorzystywane przez zespół kadr.

Mamy cztery aplikacje i cztery instancje, więc postanawiamy utworzyć cztery instancje na dwóch serwerach i nadać im nazwy odpowiadające nazwom aplikacji:

- MagicServer1\ChocMaker,
- MagicServer1\MagicSales,
- MagicServer2\TempSmart,
- MagicServer2\HRManager.

Początkowo wydawało się to całkiem sensowne. Kilka miesięcy później okazało się jednak, że popełniliśmy błąd. Główny architekt bazy danych poprosił nas o skonsolidowanie aplikacji Choc Maker i Temperature Smart w jednej instancji.

Mamy teraz trzy możliwości. Możemy uruchomić aplikację Choc Maker w instancji o nazwie TempSmart, co prawdopodobnie spowoduje pewne zamieszanie w przyszłości. Możemy też uruchomić Temperature Smart w instancji o nazwie ChocMaker, co wywoła podobny poziom dezorientacji. Trzecią opcją jest zmiana nazwy instancji SQL Servera na Manufacturing, co może wiązać się z dużą ilością dodatkowej pracy, w zależności od tego, w jaki sposób aplikacje łączą się z instancją i jakie funkcje są wykorzystywane.

Czy zatem zalecam nazywanie instancji SQL Servera według działów, a nie według aplikacji? Nie, zdecydowanie nie. Takie podejście mogłoby przynieść inne problemy, na przykład istnienie wielu instancji o tej samej nazwie na różnych serwerach — albo, co gorsza, potrzebę utworzenia dwóch instancji o tej samej nazwie na jednym serwerze (co jest niemożliwe). Istnieje też ryzyko zmiany nazwy działu w ramach restrukturyzacji. Na przykład dział sprzedaży mógłby zostać podzielony na dwa odrębne działy: sprzedaży i marketingu.

Sposób nazywania instancji zależy od specyfiki naszej organizacji i struktury środowiska SQL Servera. Na przykład oprócz nadawania instancjom nazw związanych z aplikacjami lub działami można je również nazywać zgodnie ze świadczonymi usługami biznesowymi.

Może być także wymagane stosowanie konwencji nazewnictwa, która uwzględnia różne aspekty w nazwie instancji. Jako przykład rozważmy nazwę instancji MAPRDENTLON01. Ta nazwa odzwierciedla dział za pomocą dwuliterowego kodu, gdzie MA oznacza produkcję (ang. *manufacturing*). Następnie określa środowisko trzyliterowym kodem — w tym przypadku PRD oznacza środowisko produkcyjne. Kolejne trzy litery określają licencję przypisaną do instancji. W tym przykładzie ENT oznacza wersję Enterprise. Kolejne trzy znaki identyfikują lokalizację instancji: LON to Londyn. Na końcu znajduje się dwucyfrowy, rosnący identyfikator.

Podsumowując — radzę po prostu przykładać odpowiednią wagę do nazewnictwa i nadawać instancjom nazwy w przemyślany sposób, który nie będzie powodował zamieszania w przyszłości. Konwencja nazewnictwa powinna być spójna w całym środowisku.

8.2. Numer 34 — bezkrytyczne używanie systemu Windows

Do 2017 r. SQL Server mógł działać tylko w systemie Windows. Ma to sens, biorąc pod uwagę, że oba produkty były rozwijane przez Microsoft. Jednak w SQL Server 2017 nastąpiła istotna zmiana — Microsoft umożliwił uruchamianie SQL Servera w dystrybucjach Linuksa, takich jak Ubuntu, SUSE i Red Hat. Można go nawet uruchomić w kontenerach Dockera. Ale czy to nie wydaje się trochę dziwne? Czemu mielibyśmy używać SQL Servera w systemie innym niż Windows? Cóż, przyjrzyjmy się pewnemu scenariuszowi na przykładzie firmy MagicChoc.

W rozdziale 4. wyjaśniliśmy, że firma MagicChoc uznała, iż jej procesy są zbyt rozdrobnione, dlatego zleciła napisanie nowej aplikacji, która łączyłaby funkcje sprzedaży i zaopatrzenia w jeden interfejs z jednym zapleczem. Nie wspomnieliśmy jednak, że aplikacja ta ma zostać napisana w języku Go i będzie działać na maszynach wirtualnych z systemem Ubuntu. Kluczowymi priorytetami dla tej aplikacji są wydajność i stabilność.

Zespół programistów początkowo planował wykorzystać MySQL, jednak administratorzy DBA w firmie MagicChoc nie mieli doświadczenia z tą bazą danych, a ponadto chcieli używać zaawansowanych funkcji SQL Servera, takich jak AlwaysOn. Programiści doszli do wniosku, że mogliby również skorzystać z innych zaawansowanych możliwości T-SQL, jak typ danych HIERARCHYID czy funkcja okienkowania.

W związku z tym podjęto decyzję, że zespół administratorów DBA zbuduje instancje SQL Servera na maszynach wirtualnych z systemem Windows Server 2022. Jednak w tym przypadku była to pomyłka. Wskazówki wyjaśniające, dlaczego to był błąd, można znaleźć w studium przypadku.

Po pierwsze, zespół DBA nie chciał obsługiwać MySQL ze względu na brak odpowiednich umiejętności, a jednocześnie oczekiwał, że programiści pracujący w Linuksie przejdą na środowisko Windows dla silnika bazy danych. Gdyby SQL Server działał w Linuksie, próg wejścia byłby niższy.

Drugim powodem jest to, że aplikacja była budowana w systemie Ubuntu. Jest to darmowa dystrybucja Linuksa, co sugeruje, że koszty odgrywają istotną rolę. Wiele aplikacji korporacyjnych powstaje w dystrybucjach Red Hat lub SUSE, ponieważ mimo kosztów licencyjnych oferują one wsparcie techniczne. Gdyby SQL Server został zainstalowany w Ubuntu zamiast Windows, można byłoby uniknąć kosztów licencji systemu Windows Server.

Ostatnim powodem jest wymóg szybkości i stabilności aplikacji. Choć SQL Server spełnia oba te wymagania w każdym systemie operacyjnym, to w przypadku Linuksa są one nieco wzmocnione ze względu na charakter tego systemu operacyjnego.

Linux to bardzo lekki system operacyjny. Oznacza to, że mniej rzeczy może pójść nie tak. Choć stabilność systemu Windows znacznie wzrosła w ciągu ostatniej dekady, dystrybucje Linuksa wciąż wiodą prym pod tym względem. Lekka natura systemu może również zapewnić przewagę wydajnościową. W 2021 r. oficjalne testy wydajności wykazały, że Red Hat Linux jest najbardziej wydajnym systemem operacyjnym dla SQL Servera.

Oczywiście wszystkie te korzyści w tym scenariuszu nie oznaczają, że Linux powinien być domyślnym wyborem do instalacji SQL Servera. Oznacza to jednak, że przy projektowaniu środowiska platformowego dla warstwy danych aplikacji korzystającej z SQL Servera warto rozważyć system Linux. Istnieją sytuacje, w których Linux jest optymalnym wyborem, ale w wielu przypadkach Windows nadal będzie lepszą opcją.

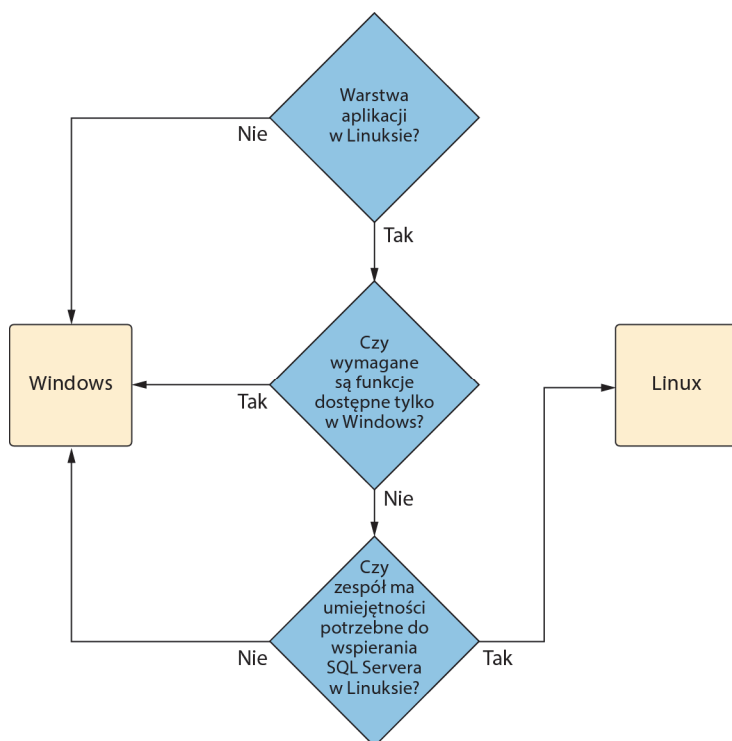
Jeśli na przykład aplikacja jest oparta na systemie Windows, wprowadzenie serwera linuksowego tylko niepotrzebnie skomplikuje sprawę. Mogłoby to również obniżyć poziom bezpieczeństwa, ponieważ musielibyśmy włączyć uwierzytelnianie SQL Servera, zamiast polegać wyłącznie na uwierzytelnianiu systemu Windows.

UWAGA Możliwe jest dołączenie serwerów linuksowych do domeny Active Directory (AD), jednak jest to nietypowa konfiguracja.

W zależności od naszej organizacji możemy nie posiadać wystarczających umiejętności w zakresie obsługi Linuksa. Jeśli tak jest, prawdopodobnie nie będziemy chcieli instalować w systemie operacyjnym złożonego produktu, takiego jak SQL Server, którego obsługa może sprawiać nam problemy.

System Linux ma również pewne ograniczenia funkcjonalne w kontekście SQL Servera. Na przykład nie obsługuje usług raportowania (SQL Server Reporting Services). Inne narzędzia graficzne, takie jak SQL Server Management Studio (SSMS), również nie są dostępne na serwerach linuksowych. Możemy jednak zainstalować SSMS na urządzeniu z systemem Windows i połączyć się z instancją SQL Servera działającą na Linuksie, tak samo jak w przypadku każdej innej instancji.

Przykładowe drzewo decyzyjne, które pomoże w wybraniu najlepszej opcji, przedstawiono na rysunku 8.1.



Rysunek 8.1. Drzewo decyzyjne: Windows kontra Linux

Podsumowując: trzeba uważnie przemyśleć środowisko hostingowe serwera SQL podczas planowania instalacji. Należy przed podjęciem decyzji wziąć pod uwagę charakter i wymagania warstwy aplikacji, wymagania techniczne serwera SQL, koszty licencji systemu operacyjnego, model wsparcia oraz umiejętności zespołu. Zawsze jednak warto pamiętać o opcji instalacji w systemie Linux.

WSKAZÓWKA Opis instalacji SQL Servera w Linuksie wykracza poza ramy niniejszej książki. Microsoft udostępnia przewodnik krok po kroku pod adresem <https://mng.bz/aVO7>. Szczegółowe instrukcje oraz przewodnik można też znaleźć w mojej książce *Pro SQL Server 2022 Administration* dostępnej pod adresem <https://mng.bz/gA9V>.

8.3. Numer 35 — zapominanie, jak użyteczne mogą być kontenery

Gdy zaczynałem pracę z SQL Serverem, zawsze instalowało się go na fizycznym serwerze, czyli „gołym metalu”. Z czasem maszyny wirtualne stawały się coraz popularniejsze i choć SQL Server dołączył do tego trendu nieco później, ostatecznie normą stało się instalowanie go w maszynach wirtualnych. W ostatnich latach coraz większą popularność zyskują kontenery, a SQL Server obsługuje środowiska kontenerowe od wersji 2017.

W przeciwieństwie do maszyny wirtualnej, która wirtualizuje sprzęt, kontener wirtualizuje jądro systemu operacyjnego. Dzięki temu kontenery są przenośne, odizolowane i lekkie. Kontenery mają wiele zalet, m.in. możliwość wykorzystania w aplikacjach opartych na mikrousługach, a także znacznie upraszczają wdrażanie aplikacji, szczególnie w scenariuszach DevOps.

Choć Microsoft dość szybko dołączył do świata kontenerów, często spotykam się z tym, że administratorzy baz danych niechętnie z nich korzystają. Wynika to zazwyczaj z braku zrozumienia i niedostrzegania korzyści, jakie kontenery mogą przynieść w przypadku SQL Servera. Mając to na uwadze, przyjrzyjmy się pewnemu scenariuszowi na przykładzie firmy MagicChoc.

Firma MagicChoc właśnie podpisała umowę z firmą konsultingową na stworzenie nowej, zintegrowanej aplikacji do zarządzania procesem produkcji. Jest to duży projekt, którego realizacja ma potrwać ponad rok. Zespół projektowy będzie składał się z 15 programistów i 5 testerów. Każdy członek zespołu projektowego potrzebuje dedykowanej instancji SQL Servera.

Niektórzy członkowie zespołu projektowego korzystają z laptopów z systemem Windows, ale większość używa komputerów Mac. W związku z tym nie mogą po prostu zainstalować SQL Servera na swoich laptopach. Jeden z programistów wspomina coś o użyciu kontenerów, ale administratorzy baz danych szybko ucinają tę dyskusję, argumentując, że kontenery są bezstanowe, więc programiści nie mogliby zapisywać swojej pracy. Zamiast tego administrator baz danych poprosił zespół Windows o uruchomienie 15 maszyn wirtualnych. Niestety w klastrze VMware zabrakło zasobów. Uruchomiono zatem jedną dużą maszynę wirtualną, na której administrator zainstalował 20 instancji SQL Servera.

Ten scenariusz jest dość typowy dla działów IT w dużych organizacjach, ale niestety pełen błędnych przekonań i pomyłek.

Optymalnym rozwiązaniem w tym scenariuszu byłoby użycie kontenerów. Przyjrzyjmy się bliżej związanym z tym nieporozumieniom.

Administratorzy baz danych podchodzą niechętnie do używania kontenerów, ponieważ są one bezstanowe. Z tego powodu uważają, że programiści nie będą

w stanie zapisać swojej pracy. Choć rzeczywiście kontenery są bezstanowe, to założenie o niemożności zapisywania pracy programistów jest błędne. Podczas korzystania z SQL Servera w kontenerze kluczowe jest zapewnienie trwałości danych. W przeciwnym razie, gdy kontener zostanie usunięty, wszystko, co się w nim znajduje, włącznie z bazami danych i kodem, zostanie utracone.

Aby przechowywać dane poza kontenerem, użyjemy *woluminu Dockera*, który jest zasobem zarządzanym przez Dockera i wskazuje na lokalizację w systemie plików poza kontenerem. Wbudowany sterownik jest lokalny. W praktyce oznacza to, że instancja SQL Servera będzie uruchomiona wewnątrz kontenera, ale wolumin Dockera będzie istniał na hoście. Możemy utworzyć wolumin Dockera o nazwie `sqldata` za pomocą polecenia przedstawionego na listingu 8.1.

Listing 8.1. Tworzenie woluminu Dockera

```
sudo docker volume create sqldata
```

WSKAZÓWKA Dostępne są również inne wtyczki sterowników Dockera. Na przykład Flocker umożliwia tworzenie woluminów Dockera w zewnętrznych magazynach danych, takich jak Elastic Block Storage w AWS.

Po utworzeniu woluminu Dockera możemy użyć przełącznika `-v` podczas tworzenia kontenera, aby odwzorować wewnętrzny system plików na zewnętrzny wolumin, jak pokazano na listingu 8.2. Spowoduje to utworzenie kontenera o nazwie `production` i przypisanie folderu, w którym przechowywane są bazy danych systemu, do woluminu Dockera o nazwie `sqldata`.

UWAGA W tym przykładzie zakładam, że Docker jest zainstalowany i uruchomiony oraz że pobrałeś domyślny obraz kontenera SQL Server 2022.

Listing 8.2. Tworzenie kontenera SQL Server z woluminem Dockera

```
sudo docker run -e "ACCEPT_EULA=Y" \  
-e "MSSQL_SA_PASSWORD=Ha$10" \  
-p 1433:1433 \  
--name production \  
--hostname production \  
-v sqldata:/var/opt/mssql \  
-d mcr.microsoft.com/mssql/server:2022-latest
```

Alternatywnym rozwiązaniem jest bezpośrednie podłączenie katalogu hosta jako woluminu danych wewnątrz kontenera. Korzystanie z woluminów Dockera ma jednak kilka zalet. Gdy używamy woluminu Dockera, możemy zarządzać nim z interfejsu wiersza poleceń Dockera. Ułatwia to przenoszenie woluminów lub współdzielenie ich między kontenerami. W przeciwieństwie do podłączania katalogu hosta woluminy Dockera mają dodatkową zaletę — nie są zależne od struktury katalogów w hoście.

Kontenery z pewnością mają swoje miejsce w pracy współczesnego administratora baz danych. Zapewniają izolację procesów oraz użytkowników i mogą

znacznie zmniejszyć nakład pracy związany z wdrażaniem i zarządzaniem środowiskami deweloperskimi w dużych zespołach. Dane mogą być przechowywane trwale poza kontenerem, mimo że silnik SQL Servera działa wewnątrz niego. Oznacza to, że dane mogą przetrwać usunięcie kontenera.

8.4. Numer 36 — niepotrzebne używanie środowiska Desktop Experience

Wcześniej w tym rozdziale wyjaśniłem, dlaczego warto rozważyć Linuksa jako potencjalną platformę hostingową przy planowaniu instalacji SQL Servera. Jeśli jednak zdecydujemy, że Windows jest niezbędny, powinniśmy się zastanowić, czy rzeczywiście potrzebujemy graficznego interfejsu użytkownika. Często odpowiedź będzie negatywna, a w takim przypadku SQL Server można zainstalować w Windows Server Core.

Aby lepiej to zrozumieć, przyjrzyjmy się innemu wymaganiu firmy MagicChoc. W rozdziale 6. utworzyliśmy bazę danych Marketing. Zbudowaliśmy również pakiet SSIS, który miał wypełniać tę bazę informacjami o odsłonach. Deweloperzy analityki biznesowej nie mają doświadczenia z systemem Linux i niechętnie podejmują się wsparcia aplikacji na tej platformie. W związku z tym zespół administratorów baz danych zdecydował, że instancja SQL Servera powinna być zainstalowana w systemie Windows Server 2022.

Utworzono maszynę wirtualną i użyto kreatora instalacji do zainstalowania domyślnej instancji SQL Servera. Podjęto taką decyzję, ponieważ pakiety SSIS tworzy się w środowisku graficznym Visual Studio. Było to jednak błędne podejście. Dlaczego?

Windows Server Core to Windows Server, ale bez graficznego interfejsu użytkownika. Zarządza się nim za pomocą PowerShella. Usunięcie interfejsu graficznego eliminuje wiele zbędnych elementów systemu Windows, czyniąc go lepszym i bezpieczniejszym systemem operacyjnym. Zajmuje mniej miejsca na dysku i zużywa mniej zasobów procesora i pamięci niż wersja Desktop Experience, co oznacza, że jest bardziej wydajny. Ponieważ w systemie działa mniej komponentów i usług, powierzchnia ataku jest zredukowana. Innymi słowy, jest mniej elementów, które mogłyby wykorzystać potencjalny napastnik. Te cechy sprawiają, że jest to doskonały system operacyjny do uruchamiania SQL Servera.

Programiści będą jednak używać Visual Studio do tworzenia pakietów, a Visual Studio nie działa w Server Core, więc czy w tym przypadku nie potrzebujemy interfejsu graficznego? Otóż nie, to kolejna „minipomyłka”. Wielu administratorów uważa, że programiści potrzebują Visual Studio zainstalowanego na serwerach

produkcyjnych, w których działają SSIS lub inne elementy warstwy analityki biznesowej. W rzeczywistości należy tego unikać. Nie ma ku temu żadnego technicznego uzasadnienia. Zintegrowane środowiska programistyczne (IDE) powinny działać na stacjach roboczych programistów lub w wyjątkowych przypadkach na serwerze deweloperskim.

Środowiska programistyczne na serwerach deweloperskich

Idealnym scenariuszem jest sytuacja, w której programiści korzystają wyłącznie ze środowisk IDE na swoich stacjach roboczych i łączą się z bazami danych hostowanymi na serwerze deweloperskim. Spotkałem się jednak z przypadkami, w których takie rozwiązanie nie było praktyczne.

Jeden taki scenariusz dotyczył tworzenia pakietów SSIS dla bardzo dużej hurtowni danych. Musiałem pracować na pełnym zbiorze danych, ponieważ niektóre transformacje wymagały danych z kilku miesięcy w celu uzyskania sensownych wyników, a analitycy biznesowi nie byli w stanie stworzyć przykładowego zbioru danych, który odzwierciedlałby wszystkie reguły biznesowe.

Ilość danych była tak ogromna, że moja stacja robocza po prostu nie miała wystarczająco dużo pamięci RAM, aby efektywnie uruchamiać potrzebne narzędzia. Co więcej, często pracowałem zdalnie, a przesyłanie takiej ilości danych przez VPN przy wolnym łączu internetowym nie sprawdzało się najlepiej.

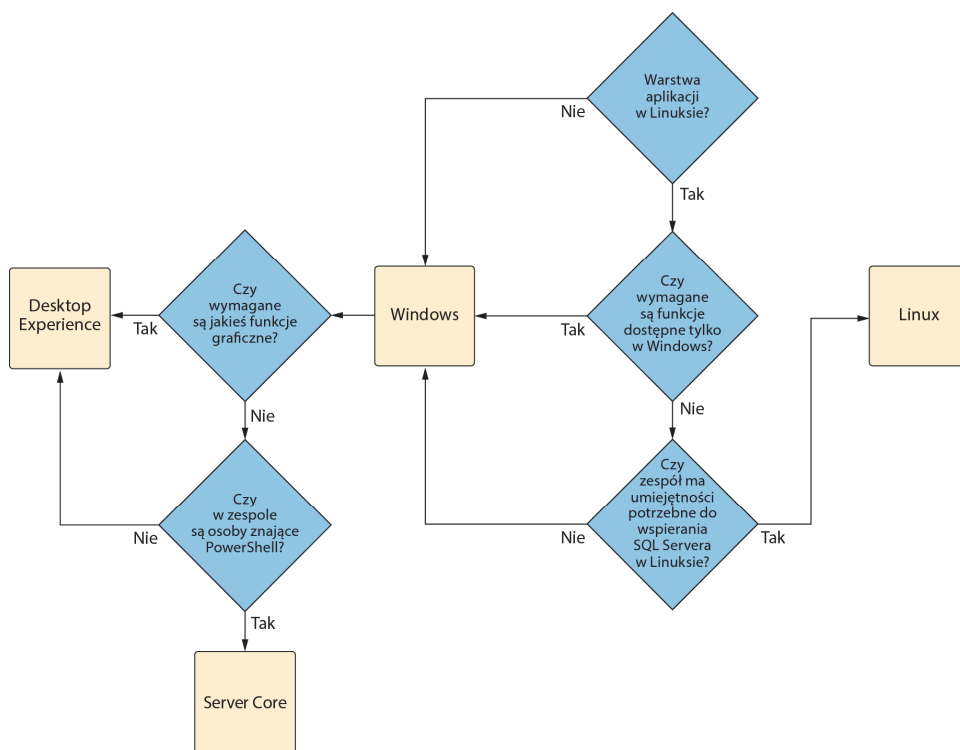
W związku z tym, aby napisać pakiety na czas, musiałem użyć protokołu zdalnego pulpitu (RDP) do łączenia się z serwerem deweloperskim i lokalnego uruchamiania pakietów. Jest to jednak sytuacja wyjątkowa, a nie standardowa procedura.

Pozwolenie programistom na zainstalowanie środowiska programistycznego na serwerze produkcyjnym zwiększa powierzchnię ataku w środowisku produkcyjnym, a jednocześnie daje im tylne wejście do tego środowiska. Oznacza to, że programiści mogą wykonywać jakieś działania lub nawet wdrażać kod bez wiedzy administratorów bazy danych, co zagraża bezpieczeństwu usługi i utrudnia wsparcie serwera. Naraża to również organizację na naruszenie wymogów regulacyjnych dotyczących właściwego podziału obowiązków.

Z uwagi na tę „minipomyłkę” w naszym scenariuszu nie ma właściwie potrzeby stosowania pełnego systemu operacyjnego, a administratorzy powinni byli zainstalować instancję SQL Servera w serwerze z systemem Windows Server Core.

Rysunek 8.2 rozszerza rysunek 8.1, dodając do drzewa decyzyjnego system Windows Server Core.

Czy instalowanie SQL Servera w systemie Windows Server ze środowiskiem Desktop Experience zawsze jest pomyłką? Nie, w niektórych scenariuszach jest to właściwy wybór. Błędem jest instalowanie SQL Servera w systemie Windows Server ze środowiskiem Desktop Experience tylko dlatego, że „zawsze tak się robiło”. Za każdym razem należy rozważyć zainstalowanie SQL Servera w Windows Server Core i odrzucić ten pomysł tylko wtedy, gdy istnieją ku temu uzasadnione powody. Takim powodem może być konieczność korzystania z funkcji graficznych, na przykład gdy wymagane jest użycie SQL Server Data Quality Services (DQS) lub Master Data Services.



Rysunek 8.2. Drzewo decyzyjne uwzględniające Server Core

8.5. Numer 37 — bezkrytyczne stosowanie wersji Enterprise Edition

SQL Server sporo kosztuje. Aby uczynić SQL Server bardziej dostępnym i zmniejszyć koszty ponoszone przez klientów, Microsoft oferuje wiele edycji tego produktu, każdą z różnymi funkcjami i ograniczeniami licencyjnymi. Na przykład edycja Express jest darmowa, ale ma istotne ograniczenia techniczne, takie jak maksymalny rozmiar bazy danych wynoszący 10 GB i maksymalna moc obliczeniowa jednego gniazda procesora lub czterech rdzeni. Posiada również wiele ograniczeń funkcjonalnych. Z drugiej strony edycja Developer jest również bezpłatna i oferuje pełen zestaw funkcji dostępnych w edycji Enterprise. Jedynym ograniczeniem jest to, że licencja ta nie zezwala na użytkowanie w środowisku produkcyjnym.

Firma MagicChoc próbuje wybrać odpowiednią edycję SQL Servera do obsługi warstwy danych swojej aplikacji marketingowej. Postanowiono wykorzystać

wersję Enterprise, ponieważ baza danych ma rozmiar 2 TB, wymaga hostingu w grupie dostępności z synchroniczną repliką oraz wykorzystuje SSIS. Była to pomyłka. Aby zrozumieć dlaczego, przeanalizujmy wymagania.

Jeśli chodzi o rozmiar bazy danych, zarówno edycja Standard, jak i Enterprise obsługują bazy danych o pojemności do 524 PB. Można by sądzić, że tak duża baza danych wymaga znacznej mocy obliczeniowej procesora i dużej ilości pamięci RAM. Zasoby te są jednak ograniczone w edycji Standard.

To prawda, ale wymagania nie wspominają o planowaniu pojemności. To tylko założenie. Biorąc pod uwagę, że edycja Standard obsługuje 128 GB pamięci RAM i 24 rdzenie (lub 4 gniazda) procesora, to założenie powinno zostać zweryfikowane poprzez planowanie pojemności. W przeciwnym razie istnieje duże ryzyko, że nasza organizacja będzie niepotrzebnie wydawać znaczne kwoty. Kiedy piszę tę książkę, cena katalogowa dwurdzeniowego pakietu licencji Enterprise wynosi 15 123 dolary. Dla porównania, dwurdzeniowy pakiet licencji Standard kosztuje 3945 dolarów. Jeśli planowanie pojemności wykaze, że aplikacja bazodanowa wymaga 16 rdzeni, użycie edycji Standard przyniesie oszczędności rzędu 89 424 dolarów.

Koszty kapitałowe i operacyjne

Jeśli tworzymy maszyny wirtualne w środowisku chmury publicznej, takiej jak Azure czy AWS, możemy wliczyć koszt licencji SQL Servera w godzinową stawkę instancji EC2. Ma to pewne zalety, takie jak możliwość zaprzestania płacenia za licencję, gdy nie jest już potrzebna, oraz brak opłat licencyjnych w godzinach, w których instancja jest wyłączona. Ta zamiana kosztów kapitałowych (CAPEX) na operacyjne (OPEX) jest również zgodna z modelem finansowym chmury, w przeciwieństwie do modelu opartego na hostingu w centrum danych.

Z drugiej strony, jeśli wiemy, że nasza aplikacja będzie używana przez dłuższy czas, to w dłuższej perspektywie zapłacimy więcej niż za licencję bezterminową. Powinniśmy wziąć to pod uwagę w modelu finansowym przy tworzeniu nowych aplikacji lub przenoszeniu istniejących do chmury.

Zastanówmy się również nad wymaganiami dotyczącymi funkcjonalności. Firma MagicChoc określiła, że aplikacja powinna być hostowana w grupach dostępności, co pozornie wymaga użycia edycji Enterprise. Jednak firma ta potrzebuje tylko jednej, synchronicznej repliki, a edycja Standard obsługuje funkcję zwaną *podstawowymi grupami dostępności*. Jest to uproszczona wersja grup dostępności, która oferuje pojedynczą replikę (synchroniczną lub asynchroniczną). Replika musi pozostać nieaktywna do momentu przełączenia awaryjnego. Oznacza to, że nie obsługuje zaawansowanych funkcji, takich jak repliki do odczytu, tworzenie kopii zapasowych na replice czy kontrola integralności repliki. Spełnia jednak wymagania firmy MagicChoc, więc nie ma potrzeby korzystania z edycji Enterprise. Więcej o grupach dostępności powiemy w rozdziale 13.

Ostatnim powodem wyboru edycji Enterprise był wymóg użycia SSIS. To częsta pomyłka, którą popełniają nawet doświadczeni administratorzy baz danych. W rzeczywistości SSIS jest obsługiwany również w edycji Standard, z pewnymi ograniczeniami. Dotyczą one głównie możliwości działania jako głównego węzła w konfiguracji skalowania horyzontalnego oraz niektórych zaawansowanych zadań przepływu sterowania i transformacji źródeł danych. Istnieją na przykład ograniczenia w zadaniach przechwytywania zmian danych (ang. *change data capture*, CDC) oraz transformacjach obejmujących dopasowywanie i ekstrakcję tekstu. Występują też ograniczenia dotyczące zaawansowanych źródeł i miejsc docelowych, takich jak Oracle, Teradata, SAP czy SSAS.

Najczęstszym uzasadnieniem tego, że aplikacje typu brownfield wymagają wersji Enterprise, jest stwierdzenie: „Może być potrzebna jakaś funkcja bazodanowa dostępna tylko w Enterprise, a ja nie jestem pewien i nie chcę ryzykować”. Ten argument można jednak w prosty sposób obalić, ponieważ łatwo to sprawdzić. Skrypt z listingu 8.3 tworzy bazę danych z tabelą wykorzystującą partycjonowanie, które jest funkcją dostępną wyłącznie w wersji Enterprise.

Listing 8.3. Tworzenie bazy danych wymagającej edycji Enterprise

```
CREATE DATABASE EnterpriseDatabase ;
GO

USE EnterpriseDatabase ;
GO

CREATE PARTITION FUNCTION PartFunc (INT)
    AS RANGE LEFT FOR VALUES (100, 200, 300) ;

CREATE PARTITION SCHEME PartScheme
    AS PARTITION PartFunc
    ALL TO ('PRIMARY') ;

CREATE TABLE VeryLargeTable (
    KeyColumn INT PRIMARY KEY IDENTITY,
    OtherColumn VARCHAR(50)
) ON PartScheme(KeyColumn) ;
```

Teraz możemy po prostu odpytać widok dynamicznego zarządzania (ang. *dynamic management view*, DMV) `sys.dm_db_persisted_sku_features`, aby uzyskać listę funkcji klasy Enterprise. Jak pokazano na listingu 8.4, jeśli odpytamy ten widok DMV z wcześniej utworzonej bazy danych, zwróci on pojedynczy wiersz dla partycjonowania. Gdyby baza danych nie korzystała z żadnych funkcji dostępnych wyłącznie w wersji Enterprise, kwerenda zwróciłaby pusty zbiór wyników.

Listing 8.4. Odkrywanie funkcji dostępnych tylko w wersji Enterprise

```
USE EnterpriseDatabase ;
GO

SELECT feature_name
FROM sys.dm_db_persisted_sku_features ;
```

Podsumowując — nie powinniśmy traktować wersji Enterprise jako opcji domyślnej. Jest ona znacznie droższa od wersji Standard i należy ją stosować tylko wtedy, gdy wymagane są funkcje dostępne wyłącznie w Enterprise. W środowiskach deweloperskich powinno się używać wersji Developer, która oferuje pełną funkcjonalność, z jedynym ograniczeniem polegającym na tym, że nie można jej wykorzystywać „na produkcji”.

8.6. Numer 38 — instalowanie instancji, kiedy wystarczyłoby rozwiązanie DBaaS lub PaaS

Trzej główni dostawcy chmury publicznej — GCP, AWS i Azure — oferują SQL Server jako usługę bazodanową (DBaaS), a Azure dodatkowo udostępnia instancje SQL Servera w modelu platformy jako usługi (PaaS). Rozwiązania te mogą przynieść wiele korzyści biznesowych dla niektórych (ale nie wszystkich) obciążeń.

Pomyślmy o firmie MagicChoc i nowej aplikacji, którą tworzy w chmurze. Aplikacja ta ma na celu automatyzację procesów w celu zmniejszenia obciążeń administracyjnych. Ma być zbudowana w Azure i będzie wykorzystywać różne komponenty chmurowe, w tym *Azure Event Grid*, czyli chmurową usługę dystrybucji komunikatów opartą na modelu publikowania/subskrybowania; *Azure Functions*, czyli bezserwerową opcję do wykonywania kodu sterowanego zdarzeniami; oraz *Azure Kubernetes Service*, czyli zarządzaną usługę kontenerową. Ponieważ aplikacja jest tworzona od podstaw, nikt tak naprawdę nie wie, ile mocy obliczeniowej będzie potrzebne do obsługi bazy danych.

Zespół administratorów bazy danych w firmie MagicChoc jest przeciążony. To mała grupa specjalistów, która oprócz realizowania wielu nowych projektów musi rozwiązywać liczne problemy operacyjne, związane przede wszystkim z optymalizacją wydajności. Główny administrator baz danych dostrzega korzyści płynące z natywnej aplikacji chmurowej, jednak zamiast zalecić korzystanie z Azure SQL Database, proponuje utworzenie maszyny wirtualnej SQL Server w Azure. To pomyłka, która wciąż jest dość powszechna. Problem polega na tym, że wielu administratorów baz danych odnosi się z rezerwą do rozwiązań chmurowych. Zauważyłem dwa główne powody takiego podejścia. Pierwszym jest brak lub postrzegany brak umiejętności w zakresie chmury, połączony z niechęcią do nauki całkowicie nowego stosu technologicznego. Drugim powodem jest obawa, że chmurowe rozwiązania SQL sprawią, iż administratorzy staną się zbędni.

Pierwszy z tych lęków zależy w pewnym stopniu od konkretnej osoby. Nauka nowego stosu technologicznego zawsze jest trudnym zadaniem, ale chmura nie zniknie, a my jako specjaliści od baz danych możemy albo się dostosować, albo

pójść w ślady dinozaurów. Drugi z tych lęków jest jednak zdecydowanie bezpodstawny. Chmura zmieni zestaw umiejętności wymaganych od administratora baz danych, ale z pewnością nie sprawi, że stanie się on zbędny.

WSKAZÓWKA Administratorzy baz danych powinni wiedzieć, że choć SQL Azure stanowi spory przeskok w porównaniu z instancją działającą w centrum danych, to podstawowy silnik bazy danych jest bardzo podobny do tego, który znają z codziennej pracy.

W omawianym tutaj przypadku zespół administratorów baz danych znajduje się pod dużą presją. Administratorzy mają do wykonania specjalistyczne zadania, takie jak diagnozowanie i rozwiązywanie problemów z wydajnością oraz wspieranie firmy w projektach wymagających nowych baz danych. Unikając rozwiązań typu DBaaS, po prostu dorzucają sobie kolejną instancję, która wiąże się ze wszystkimi przyziemnymi zadaniami — koniecznością zarządzania, konfigurowania, naprawiania, aktualizowania i planowania przestojów. Gdyby zdecydowali się na DBaaS, mieliby więcej czasu na skupienie się na zadaniach o wyższej wartości dla firmy.

Dodatkową korzyścią z wyboru Azure SQL Database w tym scenariuszu jest automatyczne skalowanie. Wymagana moc obliczeniowa w naszym przypadku nie jest znana. Jeśli zbudujemy bazę danych Azure SQL z wykorzystaniem bezserwerowej warstwy obliczeniowej, możemy zaimplementować autoskalowanie. Zaoszczędzi to jeszcze więcej czasu, ponieważ będziemy mieli znacznie mniej zmartwień związanych z doбором odpowiedniej wielkości zasobów. Możemy uniknąć niekontrolowanego wzrostu kosztów, określając maksymalną liczbę rdzeni wirtualnych do wykorzystania. Możemy również zapobiec nadmiernemu zmniejszeniu skali bazy danych, ustalając minimalną liczbę rdzeni wirtualnych. Skrypt przedstawiony na listingu 8.5 wykorzystuje PowerShell do utworzenia nowej bazy danych Azure SQL, która będzie automatycznie skalować się w zakresie od 2 do 16 rdzeni wirtualnych. Kod wykorzystuje technikę zwaną „splatting”, która poprawia czytelność kodu poprzez przekazywanie parametrów polecenia za pomocą tablicy asocjacyjnej.

Listing 8.5. Tworzenie automatycznie skalującej się bazy danych Azure SQL

```
$sqlAdminName = 'MagicChocAdmin'  
$sqlAdminPassword = 'Passw0rd!'  
  
$credentials = $(New-Object -TypeName System.Management.Automation.  
    PSCredential -ArgumentList $sqlAdminName,  
    $(ConvertTo-SecureString -String $sqlAdminPassword -AsPlainText -Force))  
  
$serverParameters = @{  
    ServerName = 'processautomationsql'  
    Location = 'eastus2'  
    SqlAdministratorCredentials = $credentials  
    ResourceGroupName = 'MagicChocApps'  
}
```

```
New-AzSqlServer @serverParameters

$databaseParameters = @{
    ResourceGroupName = 'MagicChocApps'
    ServerName = 'processautomationsql'
    DatabaseName = 'AutoPro'
    Edition = 'GeneralPurpose'
    ComputeModel = 'Serverless'
    ComputeGeneration = 'Gen5'
    MinimumCapacity = 2
    Vcore = 16
}

New-AzSqlDatabase @databaseParameters
```

Chociaż DBaaS nie jest odpowiednim rozwiązaniem dla każdej bazy danych, stanowi użyteczne narzędzie w arsenale administratora baz danych. Nie należy się go bać, a specjaliści od baz danych, którzy planują pozostać w zawodzie w nadchodzących latach i dekadach, powinni poświęcić czas na naukę rozwiązań chmurowych.

8.7. Numer 39 – instalowanie wszystkich funkcji

Podczas instalacji SQL Servera za pomocą interfejsu graficznego bardzo kuszące jest coś, co nazywam instalacją „dalej, dalej, zainstaluj”. Polega to na klikaniu przycisku *Dalej* na każdej stronie kreatora instalacji bez zmiany ustawień domyślnych (o czym więcej powiemy w następnym podrozdziale), a następnie zaznaczeniu wszystkich elementów na liście funkcji do zainstalowania.

Podczas instalacji SQL Servera z poziomu wiersza poleceń kuszące jest użycie parametru `/ROLE` z wartością `AllFeatures_WithDefaults`. Ma to taki sam efekt jak zaznaczenie wszystkich funkcji z listy w instalatorze graficznym.

Oczywiście nie spotkałem wielu osób, które w profesjonalnym środowisku instalują oprogramowanie metodą „dalej, dalej, zainstaluj”, ale zetknąłem się z wieloma zespołami, które standardowo instalują wszystkie dostępne funkcje. Zastanówmy się, dlaczego tak się dzieje.

Zespół programistów z firmy MagicChoc zwraca się do zespołu administratorów baz danych z prośbą o nową maszynę wirtualną z zainstalowanym SQL Serverem. Administratorzy pytają, jakie funkcje są im potrzebne, a programiści odpowiadają, że nie są pewni. „Czy możemy na początek dostać wszystko, bo jeszcze nie zaprojektowaliśmy całego systemu?”

Kilka miesięcy później, gdy projekt dobiega końca, zespół prosi o przygotowanie serwera produkcyjnego. „Chcielibyśmy, żeby był dokładnie taki sam jak serwer deweloperski, bo wiemy, że on działa”.

Zespół administratorów baz danych zwykle idzie po linii najmniejszego oporu i po prostu akceptuje tego typu prośby. W końcu, co złego może się stać, prawda? W niektórych przypadkach zespoły DBA po prostu budują instancje SQL Servera ze wszystkimi funkcjami jako firmowy standard. Robią tak, ponieważ zmniejsza to nakład pracy, jeśli nowe funkcje będą potrzebne w przyszłości. Takie podejście jest jednak pomyłką — szczególnie w środowisku produkcyjnym.

SQL Server to rozbudowany zestaw aplikacji, z których każda działa jako osobna usługa. Usługa to program działający w tle systemu operacyjnego. Jeśli zainstalujemy wszystkie komponenty SQL Servera, otrzymamy następujący zestaw usług:

- SQL Server Database Engine,
- SQL Server Agent,
- PolyBase Engine,
- PolyBase Data Movement,
- Analysis Services,
- SQL Browser,
- SQL Server Full Text,
- Integration Services.

Zainstalowane zostaną też usługi Master Data Services oraz aplikacje Data Quality Client. Jeśli instalujemy starsze wersje SQL Servera, zainstalujemy też cały pakiet narzędzi (wersja 2019 i starsze), a może nawet usługi Reporting Services (wersja 2016 i starsze).

To mnóstwo usług i aplikacji. Stwarza to dwa problemy. Po pierwsze, usługi zużywają zasoby, nawet gdy nie są wykorzystywane. Oznacza to, że uruchamiając niepotrzebne usługi, marnujemy zasoby — mogłyby być one wykorzystane przez funkcje, z których faktycznie korzystamy.

WSKAZÓWKA Oczywiście możemy obejść ten problem, utrzymując niepotrzebne usługi w stanie zatrzymanym, ale po co dokładać sobie pracy? Nie rozwiązuje to również problemu zajmowanej niepotrzebnie przestrzeni dyskowej. Choć zazwyczaj nie jest to ogromna ilość miejsca i nie stanowi dużego problemu, w niektórych przypadkach może zadecydować o tym, czy będziemy musieli powiększać dysk, czy nie.

Drugi, znacznie poważniejszy problem związany z uruchamianiem tak dużej liczby usług dotyczy bezpieczeństwa. Im więcej usług zainstalowanych na serwerze, tym większa powierzchnia ataku. Jeśli napastnik zdoła włamać się do jednej z uruchomionych usług, może uzyskać dostęp do innych serwerów. Dlatego zawsze dobrą praktyką jest instalowanie tylko tych funkcji, które są rzeczywiście potrzebne.

WSKAZÓWKA Dodatkowym aspektem związanym z bezpieczeństwem jest instalowanie poprawek. Jeśli zainstalujesz dodatkowe funkcje SQL Servera, które działają jako osobne usługi, ale nie są potrzebne, to i tak będą one wymagały aktualizacji. Oznacza to dodatkową pracę przy obsłudze niepotrzebnych usług.

Pełną listę funkcji, które można zainstalować za pomocą kreatora instalacji SQL Server 2022, znajdziesz w tabeli 8.1.

Tabela 8.1. Lista funkcji SQL Servera

| Funkcja | Funkcja nadrzędna | Typ funkcji | Opis |
|---|--------------------------|---|--|
| Database Engine Services | Nie dotyczy | Poziom instancji | Zapewnia podstawową funkcjonalność relacyjnej bazy danych. |
| SQL Server Replication | Database Engine Services | Poziom instancji | Umożliwia dystrybucję danych między różnymi instancjami przy użyciu modelu wydawca-subskrybent. |
| Machine Learning Services and Language Extensions | Database Engine Services | Poziom instancji | Wspiera rozproszone rozwiązania uczenia maszynowego z obsługą R i Pythona. |
| Full-Text and Semantic Extractions for Search | Database Engine Services | Poziom instancji | Zapewnia zaawansowane funkcje wyszukiwania i dopasowywania w danych binarnych oraz kolumnach tekstowych, w tym dopasowywanie przybliżone i obsługę tezauryasa. |
| Data Quality Services | Database Engine Services | Poziom instancji | Podstawowa funkcjonalność DQS, w tym funkcje kontroli jakości i przechowywania danych. |
| PolyBase Query Service for External Data | Database Engine Services | Poziom instancji | Umożliwia odpytywanie wielu heterogenicznych źródeł danych. |
| Analysis Services | Nie dotyczy | Poziom instancji | Zapewnia funkcje modelowania i odpytywania danych wielowymiarowych i tabelarycznych. |
| Data Quality Client | Nie dotyczy | Współdzielony między wszystkimi instancjami na serwerze | Samodzielna aplikacja do interakcji z usługami DQS oraz przygotowywania i czyszczenia danych. |
| Integration Services | Nie dotyczy | Współdzielony między wszystkimi instancjami na serwerze | Narzędzie do ekstrakcji, transformacji i wczytywania danych, używane do koordynowania importu i transformacji danych oraz eksportu do innych systemów. Więcej szczegółów znajdziesz w rozdziale 6. |
| Scale Out Master | Integration Services | Współdzielony między wszystkimi instancjami na serwerze | Skalowanie SSIS może zmniejszyć wąskie gardła poprzez horyzontalne skalowanie wykonywania pakietów. Główny węzeł skalowania odpowiada za zarządzanie tym procesem. |

Tabela 8.1. Lista funkcji SQL Servera – ciąg dalszy

| Funkcja | Funkcja nadrzędna | Typ funkcji | Opis |
|----------------------|----------------------|---|---|
| Scale Out Worker | Integration Services | Współdzielony między wszystkimi instancjami na serwerze | Węzły robocze skalowania wykonują zadania pobrane z głównego węzła skalowania. |
| Master Data Services | Nie dotyczy | Współdzielony między wszystkimi instancjami na serwerze | Pozwala tworzyć modele umożliwiające zarządzanie danymi w całym przedsiębiorstwie. Jest to szczególnie przydatne w rozbudowanych organizacjach, gdzie różne działy używają różnej terminologii dla tych samych elementów danych, ponieważ pozwala odwzorowywać te terminy na jedno źródło prawdy. |

Czytelnicy, którzy zetknęli się z filozofią programowania ekstremalnego, z pewnością słyszeli o zasadzie YAGNI. *Zasada YAGNI* (od ang. *You Aren't Going to Need It* – nie będziesz tego potrzebować) ma na celu eliminację nieefektywności w procesie tworzenia oprogramowania. Tę samą zasadę warto stosować również przy instalacji SQL Servera.

8.8. Numer 40 – niestosowanie skryptów do instalacji SQL Servera

Ręczne instalowanie SQL Servera, a następnie konfigurowanie instancji zgodnie z najlepszymi praktykami to żmudny proces. Jest on również podatny na ludzkie błędy. Nawet jeśli zespół administratorów baz danych posiada szczegółową instrukcję instalacji i konfiguracji instancji SQL Servera, to przy wielu osobach ręcznie instalujących wiele instancji z czasem pojawią się błędy. W rezultacie środowisko stanie się niespójne, co utrudni jego utrzymywanie.

Weźmy jako przykład firmę MagicChoc. Jej zespół DBA zarządza 120 instancjami SQL Servera rozlokowanymi na 100 maszynach wirtualnych. Część z nich znajduje się w infrastrukturze lokalnej, a pozostałe w chmurze. Firma posiada standard konfiguracji oraz szczegółową instrukcję wdrożenia, jednak każda instancja jest budowana i konfigurowana ręcznie. Ogólne standardy konfiguracji są następujące:

- Zainstaluj instancję domyślną z użyciem odpowiedniej edycji; jeśli jednak serwer ma mieć wiele instancji, możesz zmienić nazwę instancji domyślnej.
- Zabezpiecz instancję.
- Zainstaluj tylko silnik bazy danych, chyba że inne funkcje są wyraźnie wymagane.

WSKAZÓWKA W swoim środowisku zdecydowanie powinieneś rozważyć również optymalizację wydajności. Dotyczy to zarówno poziomu systemu operacyjnego, jak i SQL Servera. Wiele z tych opcji omówimy w rozdziale 10.

Niestety, niedawny audyt instancji wykazał, że żadna z tych zasad nie była konsekwentnie stosowana na wszystkich serwerach. Wynika to z faktu, że na przestrzeni lat wielu administratorów baz danych, działając w pośpiechu, nie przestrzegało ściśle procedur przy tworzeniu nowych instancji. Założenie, że ręczne instalowanie i konfigurowanie instancji pozwoli dotrzymać standardów, było pomyłką.

Zamiast ręcznego instalowania i konfigurowania instancji zespół DBA mógłby zastosować podejście oparte na skryptach. W tej metodyce skrypt, często napisany w PowerShellu, instaluje i konfiguruje instancje w sposób automatyczny. Oznacza to, że gdy potrzebna jest nowa instancja SQL Servera, administrator może po prostu uruchomić gotowy skrypt. Takie rozwiązanie nie tylko poprawia spójność, ale także znacznie skraca czas i zmniejsza nakład pracy potrzebny do tworzenia instancji, co pozwala administratorom skupić się na bardziej wartościowych zadaniach.

Złote obrazy

Niektórzy nadal instalują SQL Server z wykorzystaniem „złotego obrazu”. Polega to na utworzeniu obrazu systemu Windows Server ze wstępnie zainstalowanym SQL Serverem. Podejście oparte na skryptach jest jednak znacznie bardziej elastyczne.

Złote obrazy z pewnością poprawiają spójność, ale czasem aż nadto. Często prowadzą do instalowania wszystkich instancji SQL Servera w wersji Enterprise i konfigurowania ich pod kątem przetwarzania transakcyjnego (OLTP), nawet gdy firma posiada wiele hurtowni danych.

Obejściem tego problemu jest posiadanie wielu złotych obrazów, ale generuje to dużo pracy — na przykład gdy wymagana jest aktualizacja obrazu w celu zastosowania najnowszego pakietu serwisowego.

Dlatego zalecam stosowanie podejścia opartego na skryptach zamiast korzystania z gotowych obrazów. Znacznie łatwiej jest osiągnąć odpowiednią równowagę między spójnością a elastycznością, używając parametrów, które można przekazać do skryptu.

Zastanówmy się, jak moglibyśmy ulepszyć ten proces, tworząc skrypt PowerShella, który zainstaluje instancję SQL Servera, przeprowadzi podstawowe testy i skonfiguruje tę instancję. Najpierw jednak pomyślmy o standardach, jakie powinniśmy przyjąć, aby odpowiednio zabezpieczyć nasze instancje:

- Wyłączenie dostępu zdalnego.
- Zmiana nazwy konta sa.

OSTRZEŻENIE W tym przykładzie ustawiamy tylko dwie opcje związane z zabezpieczaniem systemu. Ma to na celu zapewnienie jasnego i zwięzłego przykładu. W rzeczywistym środowisku należy rozważyć znacznie więcej ustawień bezpieczeństwa. Zalecam zapoznanie się ze wzorcowymi konfiguracjami dla SQL Server 2022 opracowanymi przez Center for Internet Security (CIS), które są dostępne na stronie <https://mng.bz/eVOQ>.

Oprócz zabezpieczenia instancji musimy również zastanowić się nad wyborem odpowiedniej edycji. Na maszynach programistycznych będziemy chcieli zainstalować wersję Developer, natomiast w przypadku instancji produkcyjnych przyda się możliwość wyboru między wersjami Enterprise a Standard.

Zastanówmy się najpierw nad parametrami, które będziemy przekazywać do naszego skryptu. Biorąc pod uwagę nasze opcje konfiguracyjne, będziemy potrzebować możliwości przekazania nazw użytkowników i haseł dla kont usługowych, które będą używane do wykonywania silnika bazy danych i narzędzia SQL Server Agent. Będziemy też chcieli przekazać edycję oraz wybraną nazwę i hasło dla konta sa. Dlatego na początku naszego skryptu umieścimy blok param, aby zdefiniować oczekiwane parametry. Musimy również uwzględnić parametr dla nazwy instancji, ale ponieważ zgodnie z przyjętymi wymaganiami standardem powinno być instalowanie instancji domyślnej, a nazwana instancja powinna być wyjątkiem, nadamy temu parametrowi domyślną wartość MSSQLSERVER. Oznacza to, że zostanie zainstalowana instancja domyślna, chyba że celowo zmienimy tę wartość, ręcznie określając parametr podczas uruchamiania skryptu.

Pierwszym zadaniem naszego skryptu będzie zainstalowanie modułu sqlserver PowerShella. Umożliwi to późniejsze użycie polecenia `Invoke-SqlCmd` w skrypcie. Instalację modułu wykonamy za pomocą polecenia `Install-Module`.

Nasze drugie zadanie polega na określeniu wartości zmiennej `$pid` na podstawie parametru `edition`. Instalator SQL Servera ustala, którą wersję zainstalować, na podstawie klucza produktu. Dlatego klucz produktu musi zostać przekazany do parametru `/PID`. Jeśli nie zostanie podany żaden klucz, instalator domyślnie wybierze edycję Evaluation, która przestanie działać po okresie próbnym.

OSTRZEŻENIE W rzeczywistym środowisku klucze produktów, znane również jako identyfikatory produktów (ang. *product ID*, PID), powinny być przechowywane w menedżerze haseł i odczytywane z niego przez skrypt. Pozwala to uniknąć przechowywania informacji firmowych w systemie kontroli wersji.

Wiele organizacji korzysta ze zbiorczych kluczy licencyjnych. Oznacza to, że używają tego samego klucza dla wszystkich instalacji SQL Servera. Możemy więc uprościć proces instalacji przez odwzorowanie klucza produktu na edycję. Dzięki temu administratorzy baz danych mogą po prostu wybrać edycję, którą chcą zainstalować, zamiast każdorazowo szukać klucza licencji zbiorczej.

OSTRZEŻENIE Klucz produktu 00000-00000-00000-00000-00000 użyty w przykładowym skrypcie dla wersji Enterprise i Standard to w rzeczywistości klucz dla wersji Evaluation. Założenie jest takie, że przed uruchomieniem skryptu zastąpisz ten klucz odpowiednim kluczem produktu dla Twojej organizacji.

Następnie zainstalujemy SQL Server przez uruchomienie programu *setup.exe*. Skrypt zakłada, że pobraliśmy pliki instalacyjne i są one dostępne w folderze, z którego uruchamiamy skrypt. Polecenie to przyjmuje większość parametrów, które przekazujemy do naszych skryptów: nazwę instancji, nazwy kont usługowych, hasła, a także edycję. Dodajemy również parametry `/IACCEPTSQLSERVERLICENSETERMS` oraz `/Q`, które są wymagane do zaakceptowania warunków licencji i przeprowadzenia cichej instalacji. Oba są niezbędne podczas instalacji nienadzorowanej. Polecenie wykorzystuje także parametr `$pid`, który obliczamy na podstawie parametru `$edition`. Przekazujemy również wartość SQL do parametru `/SECURITYMODE`, aby włączyć uwierzytelnianie mieszane. Oznacza to, że musimy także podać hasło dla konta `sa`.

UWAGA Bezpośrednio po instalacji konto `sa` będzie miało swoją domyślną nazwę. Zmienimy ją później w skrypcie.

Po zainstalowaniu instancji należy przeprowadzić proste testy sprawdzające poprawność instalacji. Następnie użyjemy narzędzia `Invoke-SqlCmd` do skonfigurowania zabezpieczeń naszej instancji. Pierwsze polecenie `sqlcmd` wyłącza dostęp zdalny. Dezaktywuje to przestarzałą funkcję umożliwiającą zdalne wykonywanie lokalnych procedur składowanych poprzez połączenia między serwerami. Drugie polecenie zmienia nazwę konta `sa` na nową, którą prześlemy do skryptu jako parametr. Uzasadnieniem tej zmiany jest fakt, że `sa` to powszechnie znana nazwa konta administratora. Ponieważ konto `sa` jest używane podczas uwierzytelniania drugiego poziomu, zmiana tej nazwy zmniejsza ryzyko przeprowadzenia przez atakującego ataku typu brute force w celu uzyskania uprawnień administracyjnych.

Skrypt z listingu 8.6 łączy wszystkie te elementy w jednym pliku, który zespół DBA może uruchamiać za każdym razem, gdy potrzebuje zainstalować i skonfigurować nową instancję SQL Servera.

WSKAZÓWKa Pamiętaj, aby zmienić identyfikatory PID na klucze produktów zakupione przez Twoją organizację.

Listing 8.6. Tworzenie skryptu do instalowania SQL Servera

```
[CmdletBinding()]
param(
    [string] $SQLServiceAccount,
    [string] $SQLServiceAccountPassword,
    [string] $AgentServiceAccount,
    [string] $AgentServiceAccountPassword,
    [string] $SaName,
    [string] $SaPassword,
    [string] $edition,
    [string] $InstanceName = 'MSSQLSERVER'
)

if ($Edycja -eq 'Developer') {
    $sqlPid = '22222-00000-00000-00000-00000'
} elseif ($Edycja -eq 'Standard') {
    $sqlPid = '00000-00000-00000-00000-00000'
} elseif ($Edycja -eq 'Enterprise') {
    $sqlPid = '00000-00000-00000-00000-00000'
}

If(-not(Get-Module NuGet -ErrorAction silentlycontinue)){
    Install-Module NuGet -Confirm:$False -Force
} <----- Cicha instalacja NuGet, wymagana do instalacji modułu SQLServer.
If(-not(Get-Module SQLServer -ErrorAction silentlycontinue)){
    Install-Module SQLServer -Confirm:$False -Force -AllowClobber
} <----- Cicha instalacja modułu SQLServer PowerShella.

./setup.exe /ACTION=Install /FEATURES=SQLENGINE /
    INSTANCENAME=$InstanceName /SQLSVCACCOUNT=$SQLServiceAccount
    /SQLSVCACCOUNT=$SQLServiceAccountPassword /
    AGTSSVCACCOUNT=$AgentServiceAccount /
    AGTSSVCACCOUNT=$AgentServiceAccountPassword /PID=$sqlPid /
    IACCEPTSQLSERVERLICENSETERMS /Q /SECURITYMODE=SQL /SAPWD=$SaPassword
/SQLSYSADMINACCOUNTS=Administrator <----- Instalacja instancji SQL Servera.

if ($InstanceName -ne 'MSSQLSERVER') { <----- Początek wstępnych testów.
    $ServiceName = 'MSSQL$' + $InstanceName
    $ServerInstance = 'localhost\' + $InstanceName
} else {
    $ServiceName = $InstanceName
    $ServerInstance = 'localhost'
}

$ServiceStatus = Get-Service -servicename $ServiceName

if ($ServiceStatus.Status -eq 'Running') {
    $output = "The service for SQL Server instance: {0} is Running" -f
$InstanceName
    Write-Output $output
} else {
    $output = "The service for SQL Server instance: {0} is NOT Running.
Script terminating" -f $InstanceName
    Write-Output $output
    exit
}
```

```

$ServerName = Invoke-Sqlcmd -ServerInstance $ServerInstance -query 'SELECT
@@SERVERNAME ;' -Username 'sa' -Password $SaPassword

if ($ServerName -ne $null) {
    $output = "The SQL Server instance: {0} is accessible" -f $InstanceName
    Write-Output $output
} else {
    $output = "The SQL Server instance: {0} is NOT accessible. Script
terminating" -f $InstanceName
    Write-Output $output
    exit
} ← Koniec wstępnych testów.

Invoke-Sqlcmd -ServerInstance $ServerInstance -Query "EXEC sp_configure 'show
advanced options', 1 ; RECONFIGURE ; EXEC sp_configure 'remote
access', 0 ; RECONFIGURE ; EXEC sp_configure 'show advanced options', 0 ;
RECONFIGURE ;" -Username 'sa' -Password $SaPassword ← Wyłączenie dostępu zdalnego.

$Query = "ALTER LOGIN sa WITH NAME = {0}" -f $SaName

Invoke-Sqlcmd -ServerInstance $ServerInstance -Query $Query -Username 'sa'
-Password $SaPassword ← Zmiana nazwy konta sa.

```

Jeśli skrypt zostanie zapisany jako *InstallSQLServer.ps1* w katalogu głównym zawierającym nośnik instalacyjny, będzie można uruchomić go za pomocą polecenia przedstawionego na listingu 8.7.

WSKAZÓWKA Skrypt należy uruchomić z uprawnieniami administratora.

Listing 8.7. Wykonywanie skryptu instalującego SQL Server

```

./InstallSQLServer.ps1 -SqlServiceAccount 'sqlServiceAccount' -SqlServiceAccountPassword
'Pa$$w0rd' -AgentServiceAccount 'sqlServiceAccount' -AgentServiceAccountPassword
'Pa$$w0rd' -SaName 'SQLAdmin' -SaPassword 'Pa$$w0rd' -Edition 'Developer'

```

Automatyzacja instalacji SQL Servera powinna być jednym z priorytetów każdego administratora baz danych. Pozwala to ograniczyć ręczną pracę i sprzyja standaryzacji. Przynosi również dodatkowe korzyści, takie jak możliwość przechowywania konfiguracji w repozytorium na GitHubie, co zapewnia kontrolę wersji i prosty mechanizm wprowadzania zmian.

8.9. Numer 41 — zakładanie, że zarządzanie konfiguracją nie dotyczy SQL Servera

Jednym z głównych powodów automatyzowania instalacji SQL Servera jest zapewnienie spójności. Jest to istotne z perspektywy wspierania rozwiązań stosowanych w przedsiębiorstwie oraz zagwarantowania podstaw bezpieczeństwa. Problem polega na tym, że po zakończeniu procesu instalacji każdy użytkownik z odpowiednim poziomem dostępu może po prostu zmienić te ustawienia.

Wyobraźmy sobie na przykład, że właśnie zainstalowaliśmy nową instancję SQL Servera. Jest ona poprawnie skonfigurowana i spełnia wymogi CIS poziomu 1. Niestety kilka dni po jej uruchomieniu niedoświadczony administrator baz danych włącza funkcję `xp_cmdshell`. Więcej informacji o tej rozszerzonej procedurze składowanej podam w rozdziale 14., ale na razie wystarczy, że zapamiętasz dwie rzeczy. Po pierwsze, umożliwia ona wykonywanie poleceń systemu operacyjnego bezpośrednio z poziomu instancji SQL Servera. Po drugie, stanowi poważną lukę w zabezpieczeniach i nie powinna być włączona.

Zatem zaledwie kilka dni po utworzeniu instancji jej stan nie jest już zgodny z oczekiwaniami i pojawia się luka w zabezpieczeniach. Moglibyśmy częściowo rozwiązać ten problem poprzez odpowiednie ograniczenie uprawnień i audytowanie działań. Jednak żadna z tych metod nie rozwiązuje problemu w pełni.

Pełnym rozwiązaniem jest zastosowanie techniki zwanej *zarządzaniem konfiguracją*. Polega to na tym, że konfiguracja jest w pełni stosowana podczas tworzenia serwera, a następnie okresowo weryfikowana. Jeśli podczas weryfikacji zostanie stwierdzone, że nastąpiło odchylenie od założonej konfiguracji, jest ono automatycznie korygowane. Gdyby więc zarządzanie konfiguracją sprawdzało stan naszego serwera co 30 minut, to każda nieprawidłowość w konfiguracji, taka jak luka w zabezpieczeniach, zostałaby automatycznie naprawiona w ciągu maksymalnie 30 minut od jej wystąpienia.

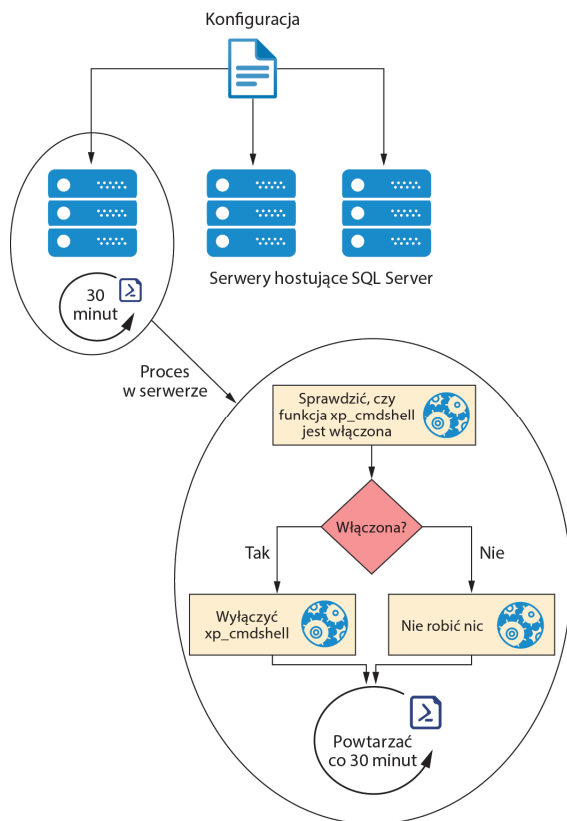
Zarządzanie konfiguracją cieszy się popularnością w środowiskach Linuksa od ponad dekady i jest kluczowym elementem wielu środowisk DevOps. Jego wykorzystanie w środowiskach Windows rośnie od kilku lat, ale wciąż nie jest tak powszechne jak w przypadku Linuksa. Dostępnych jest wiele narzędzi do zarządzania konfiguracją, a do najpopularniejszych należą Puppet, Ansible oraz PowerShell Desired State Configuration (DSC).

Na najbardziej podstawowym poziomie narzędzia te opierają się na następujących koncepcjach:

- Zasób to jednostka kodu, która ocenia i zmienia bieżący stan określonej konfiguracji serwera. Ta konfiguracja może dotyczyć różnych elementów, od klucza rejestru po folder, a nawet instalację oprogramowania. Każdy zasób będzie posiadał następujące metody:

- Metoda testowa, która sprawdza poprawność bieżącej konfiguracji. Metoda testowa jest zazwyczaj uruchamiana przy każdej ocenie zasobu.
- Metoda ustawiająca, która aktualizuje konfigurację, jeśli ta nie jest w pożądanym stanie. Metoda ta jest zazwyczaj uruchamiana tylko wtedy, gdy metoda testowa wykryje dryf konfiguracji.
- Manifest, który określa zasoby niezbędne do skonfigurowania serwera oraz wartości, które należy przekazać tym zasobom.

Ilustruje to diagram z rysunku 8.3. Przedstawia on pojedynczą konfigurację DSC zastosowaną do wielu serwerów, w których działają instancje SQL Servera. Diagram pokazuje proces zachodzący na każdym z tych serwerów w celu wymuszenia najlepszych praktyk dotyczących wyłączania funkcji `xp_cmdshell`. Co 30 minut uruchamiana jest funkcja testowa DSC, która sprawdza, czy funkcja `xp_cmdshell` jest włączona. Jeśli wykryty zostanie pożądaný stan (wyłączona), proces kończy się i przechodzi do następnej konfiguracji w manifeście. Jeśli jednak konfiguracja nie jest w pożądanym stanie, uruchamiana jest funkcja ustawiająca, która wyłącza tę funkcję.



Rysunek 8.3. Proces zarządzania konfiguracją

Zarządzanie konfiguracją to świetne podejście do instalowania i konfigurowania instancji SQL Servera. Problem polega jednak na tym, że wielu administratorów baz danych nie wie o nim albo nie korzysta z niego, ponieważ nie rozumie jego założeń. To błąd. Wykorzystanie zarządzania konfiguracją może nie tylko zmniejszyć nakład pracy i uczynić środowisko łatwiejszym i bardziej spójnym w zarządzaniu, ale także pomóc w takich aspektach, jak zgodność z przepisami i bezpieczeństwo.

Pełne omówienie DSC w PowerShellu wykracza poza ramy tej książki i zasługuje na osobną publikację, przyjrzyjmy się więc, jak utworzyć prosty dokument DSC w formacie MOF (ang. *Managed Object Format*), będący implementacją manifestu w DSC. Utworzony przez nas plik MOF sprawdzi, czy utworzona została instancja SQL Servera, a następnie wyłączy dostęp zdalny i utworzy identyfikator logowania SQL dla lokalnego użytkownika Windows o nazwie Pete. Na koniec zobaczymy, jak skompilować i zastosować tę konfigurację.

Zanim zaczniemy, musimy się odpowiednio przygotować. Będziemy potrzebować modułu PowerShell o nazwie `SqlServerDsc`. Możemy go zainstalować za pomocą skryptu z listingu 8.8.

Listing 8.8. Instalowanie modułów DSC PowerShella

```
Install-Module SqlServerDsc
```

WSKAZÓWKA Gdybyśmy chcieli skonfigurować system operacyjny, powinniśmy również zainstalować moduł PowerShella o nazwie `PSDscResources`.

Konfiguracja, którą utworzymy, nazywa się `InstallSql` i przyjmuje parametry w postaci nazwy instancji (domyślnie ustawionej na instancję domyślną) oraz edycji, którą chcemy zainstalować. Pierwsza instrukcja w konfiguracji importuje moduł `SqlServerDsc`. W tworzeniu konfiguracji dobrą praktyką jest importowanie modułów, które będą używane. Następnie na podstawie wybranej edycji określamy identyfikator PID, który powinien być zastosowany. Wykorzystujemy tutaj tę samą logikę, którą zastosowaliśmy w poprzednim podrozdziale.

Następnie zdefiniujemy węzeł. Jest to nazwa serwera, na którym chcemy uruchomić konfigurację. W naszym przypadku zdefiniowaliśmy tylko węzeł dla lokalnego hosta, ale w bardziej złożonych scenariuszach moglibyśmy stworzyć centralną usługę DSC, która będzie przysyłać konfigurację do wielu serwerów. W takiej sytuacji możemy zdefiniować wiele węzłów.

W definicji węzła określimy trzy zasoby DSC. Pierwszy z nich zapewnia, że instancja SQL Servera została utworzona. Jeśli instancja już istnieje, zasób nie podejmie żadnych działań. Natomiast w przypadku braku instancji zasób utworzy ją automatycznie.

Kolejne dwa zasoby, odpowiednio, wyłączają dostęp zdalny i tworzą konto dla użytkownika Pete. Gdyby te zasoby zostały wykonane przed utworzeniem instancji, oczywiście wystąpiłby błąd, ponieważ zasoby nie mogłyby połączyć się

z instancją. Dlatego używamy właściwości `DependsOn`, aby upewnić się, że nie zostaną one wykonane przed utworzeniem instancji. Całą konfigurację pokazano na listingu 8.9.

Listing 8.9. Tworzenie konfiguracji DSC

```
Configuration InstallSql {
    param (
        $SqlInstanceName = 'MSSQLSERVER',
        $Edition
    )

    Import-DscResource -ModuleName SqlServerDsc

    if ($Edition -eq 'Developer') {
        $ProductKey = '22222-00000-00000-00000-00000'
    } elseif ($Edition -eq 'Standard') {
        $ProductKey = '00000-00000-00000-00000-00000'
    } elseif ($Edition -eq 'Enterprise') {
        $ProductKey = '00000-00000-00000-00000-00000'
    }

    node localhost {
        SqlSetup 'InstallInstance' {
            InstanceName      = $SqlInstanceName
            Features           = 'SQLENGINE'
            SourcePath         = 'C:\SQL Media'
            SQLSysAdminAccounts = @('Administrator')
            ProductKey         = $ProductKey
        }

        SqlConfiguration 'RemoteAccess' {
            InstanceName = $SqlInstanceName
            OptionName   = 'remote access'
            OptionValue  = 1

            DependsOn    = '[SqlSetup]InstallInstance'
        }

        SqlLogin 'AddSqlAdmin' {
            Ensure        = 'Present'
            Name          = 'Pete'
            InstanceName  = $SqlInstanceName
            LoginType     = 'WindowsUser'

            DependsOn    = '[SqlSetup]InstallInstance'
        }
    }
}
```

Teraz, gdy mamy już zdefiniowaną konfigurację, musimy ją skompilować, co spowoduje utworzenie pliku MOF. Aby skompilować konfigurację, najpierw musimy wczytać plik źródłowy do bieżącego kontekstu. Następnie uruchomimy konfigurację `InstallSql`, przekazując wartości parametrów tak, jak w przypadku

poleceń PowerShella. Przy założeniu, że zapisaliśmy naszą konfigurację jako *SqlInstallDsc.ps1* w folderze *c:\DSC*, skrypt przedstawiony na listingu 8.10 skompiluje plik MOF.

WSKAZÓWKA Jeśli wykonywałeś wcześniejsze przykłady z tego rozdziału i masz już zainstalowaną domyślną instancję SQL Servera, możesz zmodyfikować przykłady kodu w tej sekcji, aby utworzyć instancję nazwaną. Dzięki temu unikniesz konieczności odinstalowywania istniejącej instancji lub używania innej maszyny wirtualnej. Pamiętaj jednak, że będziesz musiał podać własne ścieżki do plików.

Listing 8.10. Kompilowanie konfiguracji

```
. c:\DSC\SqlInstallDsc.ps1
```

```
InstallSql -Edition Developer
```

W katalogu roboczym zostanie utworzony folder o nazwie odpowiadającej konfiguracji. W folderze tym znajdziemy skompilowany plik MOF. Plik MOF utworzony w wyniku naszej kompilacji wygląda tak:

```
/*
@TargetNode='localhost'
@GeneratedBy=Administrator
@GenerationDate=11/05/2023 11:49:28
@GenerationHost=EC2AMAZ-43B3PBE
*/

instance of DSC_SqlSetup as $DSC_SqlSetup1ref
{
    SourcePath = "C:\\SQL Media";
    InstanceName = "MSSQLSERVER";
    ProductKey = "22222-00000-00000-00000-00000";
    SourceInfo = "C:\\dsc\\SqlInstallDsc.ps1::18::11::SqlSetup";
    ResourceID = "[SqlSetup]InstallInstance";
    ModuleName = "SqlServerDsc";
    SQLSysAdminAccounts = {
        "Administrator"
    };
    ModuleVersion = "16.5.0";
    Features = "SQLENGINE";

    ConfigurationName = "InstallSql";
};

instance of DSC_SqlConfiguration as $DSC_SqlConfiguration1ref
{
    ResourceID = "[SqlConfiguration]RemoteAccess";
    InstanceName = "MSSQLSERVER";
    SourceInfo = "C:\\dsc\\SqlInstallDsc2.ps1::26::11::SqlConfiguration";
    OptionValue = 1;
    ModuleName = "SqlServerDsc";
    OptionName = "remote access";
    ModuleVersion = "16.5.0";

    DependsOn = {
```

```

"[SqlSetup]InstallInstance");

ConfigurationName = "InstallSql";

};
instance of DSC_SqlLogin as $DSC_SqlLogin1ref
{
ResourceID = "[SqlLogin]AddSqlAdmin";
InstanceName = "MSSQLSERVER";
Ensure = "Present";
SourceInfo = "C:\\dsc\\SqlInstallDsc2.ps1::34::11::SqlLogin";
Name = "Pete";
ModuleName = "SqlServerDsc";
LoginType = "WindowsUser";
ModuleVersion = "16.5.0";

DependsOn = {

    "[SqlSetup]InstallInstance";
    ConfigurationName = "InstallSql";

};
instance of OMI_ConfigurationDocument
{
    Version="2.0.0";
    MinimumCompatibleVersion = "1.0.0";

CompatibleVersionAdditionalProperties={"Omi_BaseResource:ConfigurationName"};

    Author="Administrator";

    GenerationDate="11/05/2023 11:49:28";

    GenerationHost="EC2AMAZ-42B3QBE";

    Name="InstallSql";

};

```

Katalog można zastosować na serwerze za pomocą polecenia przedstawionego na listingu 8.11. Polecenie to można następnie dodać do zaplanowanego zadania, które będzie uruchamiane według harmonogramu i zapewni, że serwer będzie zawsze skonfigurowany zgodnie z wymaganiami.

Listing 8.11. Stosowanie konfiguracji

```
Start-DscConfiguration -Path 'C:\dsc\InstallSql' -Verbose
```

WSKAZÓWKA Nie jesteśmy ograniczeni tylko do zasobów DSC dostępnych w module `SqlServerDsc`. DSC jest w pełni rozszerzalny i możemy tworzyć własne, niestandardowe zasoby DSC. Chociaż wykracza to poza ramy niniejszej książki, zachęcam do zgłębienia tego tematu. Jako punkt wyjścia polecam rozdział poświęcony DSC w książce *Windows PowerShell in Action, 3rd ed.*, którą można znaleźć pod adresem <https://mng.bz/px5z>. Można też skorzystać z podstawowego przewodnika przygotowanego przez Microsoft, dostępnego pod adresem <https://mng.bz/OmNE>.

Zarządzanie konfiguracją to potężne narzędzie, które administratorzy baz danych powinni rozważyć jako sposób zapewnienia spójności środowiska SQL, a przez to zmniejszenia obciążeń administracyjnych. Jest to nowoczesne podejście do automatyzacji, w pełni rozszerzalne, niezależnie od przyjętej platformy.

WSKAZÓWKA Jeśli Twoja organizacja korzysta z platformy Azure, możesz zastosować DSC na serwerach z włączoną funkcją Azure Arc, używając usługi Azure Automange Machine Configuration (wcześniej znanej jako Azure Policy Guest Configuration).

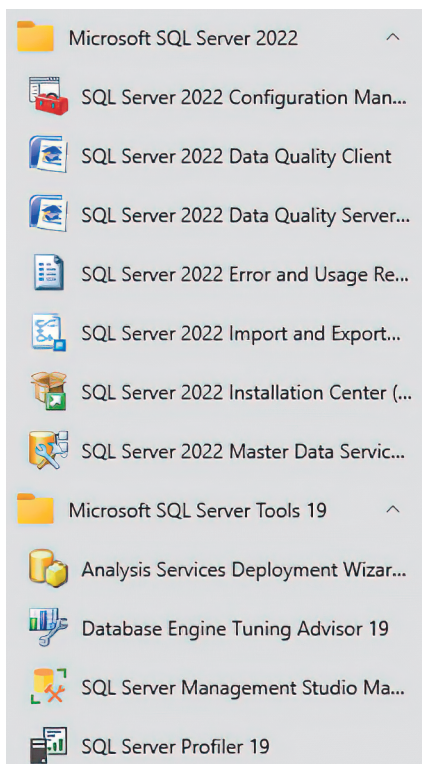
8.10. Numer 42 — używanie chmurowych obrazów SQL Servera bez ich modyfikowania

Wszyscy główni dostawcy usług chmurowych oferują obrazy ze wstępnie zainstalowanym SQL Serverem. Dzięki temu uruchamianie maszyn wirtualnych z SQL Serverem w chmurze jest bardzo proste. Korzystanie z obrazów chmurowych jest również istotne, jeśli chcemy skorzystać z modelu płatności za SQL Server, w którym licencja jest wliczona w cenę usługi. W tym modelu nie korzystamy z własnej licencji SQL Servera, lecz koszt licencji jest wliczony w stawkę godzinową maszyny wirtualnej. Aby skorzystać z tego modelu licencjonowania, musimy użyć obrazu SQL Servera dostarczanego przez dostawcę usług chmurowych, ponieważ to właśnie uruchomienie takiego obrazu aktywuje wliczoną licencję.

Bardzo często organizacje używają tego obrazu w takiej postaci, w jakiej został dostarczony, aby po prostu uruchomić maszynę wirtualną w wybranej chmurze, gdy potrzebna jest nowa maszyna z SQL Serverem. Jest to jednak pomyłka.

Aby wyjaśnić dlaczego, zastanówmy się nad niektórymi kwestiami omówionymi już w tym rozdziale. Wyjaśniliśmy na przykład, dlaczego nie jest dobrym pomysłem instalowanie wszystkich funkcji SQL Servera i czemu lepiej ograniczyć instalację tylko do tych, które są rzeczywiście potrzebne. Omówiliśmy również wykorzystanie zarządzania konfiguracją do utrzymywania pożądanych ustawień serwera i instancji SQL Servera. W rozdziale 10. wyjaśnię też, dlaczego należy unikać używania flag śledzenia 1117 i 1118.

Pomimo stosowania tych dobrych praktyk, gdybyśmy chcieli zbudować nową instancję SQL Server EC2 w chmurze AWS, zauważylibyśmy, że zostały zainstalowane wszystkie funkcje, w tym oprogramowanie SSMS, które powinno być używane jako narzędzie klienckie, a nie instalowane w serwerze. Na rysunku 8.4 pokazano nowo utworzoną instancję EC2, która wykorzystuje dostarczony przez AWS obraz SQL Server AMI.

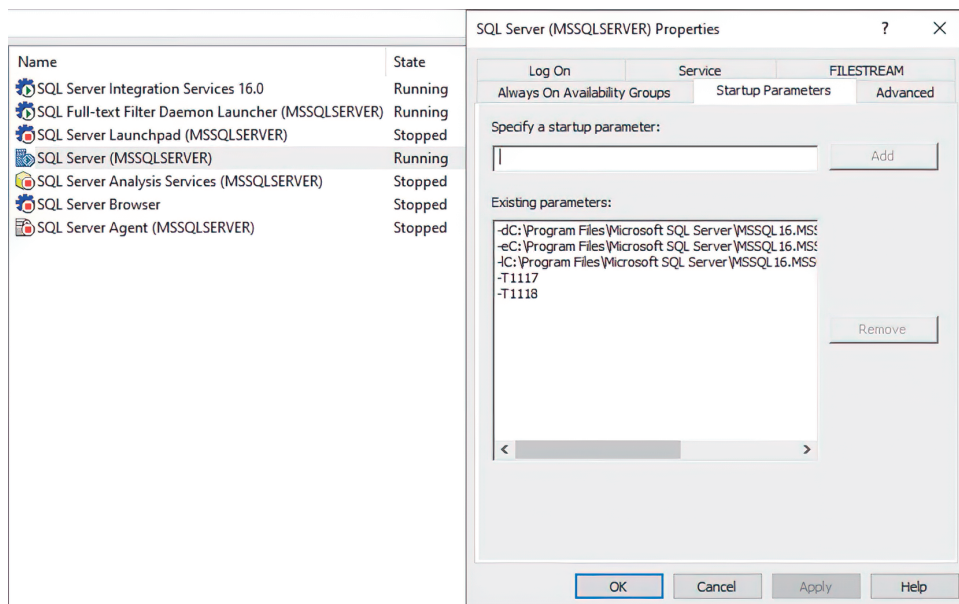


Rysunek 8.4. Nowa instancja EC2 z obrazem SQL Server AMI

W przypadku Azure sytuacja jest jeszcze gorsza. Nie dość, że zainstalowane zostaną wszystkie funkcje i narzędzia, to dodatkowo w instancji SQL Servera skonfigurowane będą flagi śledzenia T1117 i T1118. Rysunek 8.5 przedstawia zrzut ekranu z nowo utworzonej maszyny wirtualnej Azure, korzystającej z obrazu dostarczonego przez Azure. Widać na nim menedżera konfiguracji SQL Servera ze wszystkimi uruchomionymi usługami SQL oraz rozwinięte parametry uruchomieniowe usługi Database Engine, pokazujące skonfigurowane flagi śledzenia.

Dlaczego dostawcy chmury to robią? Odpowiedź jest prosta — próbują ułatwić życie wszystkim klientom, niezależnie od ich wymagań. Wiąże się to jednak z problemami, o których wspomnieliśmy wcześniej w tym rozdziale. Co więcej, ponieważ instancja SQL Servera jest wbudowana w obraz, podlega ona w pełni zjawisku dryfu konfiguracji.

Oznacza to, że korzystanie z gotowego obrazu SQL Servera od dostawcy usług chmurowych można uznać za błąd — i to dość powszechny. Aby uniknąć tego problemu, zachowując jednocześnie zgodność z licencją, należy utworzyć własny niestandardowy obraz, bazując na obrazie SQL Servera udostępnianym przez dostawcę usług.



Rysunek 8.5. Nowo utworzona maszyna wirtualna Azure z obrazem SQL Servera

W tym celu możemy utworzyć nową maszynę wirtualną na podstawie obrazu SQL Servera udostępnionego przez dostawcę usług chmurowych, a następnie dostosować ją do naszych potrzeb. Możemy na przykład odinstalować komponenty takie jak Analysis Services i Integration Services, jeśli nie są nam potrzebne. Możemy również skonfigurować instancję tak, aby była zgodna ze standardem CIS. Możemy nawet stworzyć plik DSC MOF, aby upewnić się, że instancja po zostanie skonfigurowana zgodnie z naszymi wymaganiami.

Po zakończeniu modyfikacji maszyny wirtualnej możemy utworzyć nowy obraz na jej podstawie. Proces ten różni się w zależności od dostawcy usług chmurowych, ale zawsze jest dość prosty. Na przykład w Azure możemy przejść do maszyny wirtualnej w portalu Azure i wybrać opcję *Capture* z górnego menu. Spowoduje to wyświetlenie strony *Create an Image*, na której możemy skonfigurować właściwości obrazu, takie jak grupa zasobów, oraz zdecydować, czy chcemy udostępnić obraz w galerii zasobów obliczeniowych Azure (co uczyni go publicznym), czy zachować go jako (prywatny) obraz zarządzany (to prawie na pewno jest opcja, którą wybierzesz).

Aby utworzyć obraz z instancji EC2 w AWS, przejdź do listy instancji EC2 w konsoli AWS. Następnie kliknij prawym przyciskiem myszy wybraną instancję EC2 i z menu kontekstowego wybierz opcję *Image and Templates/Create Image*. Spowoduje to wyświetlenie okna *Create Image*. W tym miejscu możesz nadać obrazowi nazwę i dodać tagi. Możesz też określić dodatkowe woluminy i skonfigurować ich rozmiary.

OSTRZEŻENIE Jeśli eksperymentujesz z tworzeniem obrazów maszyn wirtualnych w chmurze, pamiętaj, że SQL Server jest kosztowny, a gdy opłaty za jego użycie są wliczone w godzinową stawkę maszyny wirtualnej, koszty mogą szybko rosnąć. Upewnij się, że wyłączasz nieużywane instancje oraz usuwasz je, gdy już nie są potrzebne.

Przy tworzeniu maszyn wirtualnych SQL Server w chmurze nie powinniśmy korzystać bezpośrednio z obrazów udostępnianych przez dostawcę usług chmurowych. Lepiej jest dostosować serwer do naszych wymagań, a następnie utworzyć własny obraz.

Podsumowanie

- Podczas tworzenia instancji SQL Servera ważne jest stosowanie znaczących nazw, które ułatwiają identyfikację środowisk i pomagają uniknąć nieporozumień.
- Nadawanie odpowiednich nazw instancjom to bardziej sztuka niż nauka.
- Planując wdrożenie SQL Servera, warto rozważyć instalację w Linuksie.
- SQL Server 2022 jest obsługiwany w systemach Red Hat, SUSE i Ubuntu.
- Zamiast instalować SQL Server w maszynie wirtualnej lub na „gołym metalu”, warto rozważyć użycie kontenerów — szczególnie w przypadku dużych środowisk wdrożeńowych.
- Podczas korzystania z SQL Servera w kontenerze warto użyć woluminu Dockera do przechowywania danych poza kontenerem. Dzięki temu dane zostaną zachowane nawet po usunięciu kontenera.
- SQL Server jest w pełni obsługiwany w kontenerach linuksowych. Obecnie nie jest oficjalnie obsługiwany w kontenerach Windows, ale możemy używać kontenerów Windows do uruchamiania SQL Servera w środowiskach nieprodukcyjnych, jeśli utworzymy własny obraz kontenera.
- Przy planowaniu instalacji SQL Servera warto rozważyć użycie wersji Server Core. Instalacja SQL Servera w systemie Windows Server z Desktop Experience (interfejsem graficznym) nie powinna być domyślnym wyborem. Server Core jest dobrym rozwiązaniem, chyba że z jakiegoś konkretnego powodu wymagany jest interfejs graficzny.
- Pozbawiona interfejsu graficznego wersja Server Core jest wydajniejsza i bezpieczniejsza niż Windows Server z pełnym środowiskiem graficznym.

- Edycja Enterprise SQL Server jest kosztowna i powinna być stosowana tylko w przypadkach, gdy w środowisku produkcyjnym wymagane są funkcje klasy korporacyjnej.
- W środowisku produkcyjnym należy stosować edycję Standard, gdy jest to wskazane.
- W środowisku nieprodukcyjnym należy stosować edycję Developer.
- Planując wdrożenia SQL Servera, warto rozważyć oferty DBaaS (baza danych jako usługa) od dostawców usług chmurowych. Zmniejszają one nakład pracy, ponieważ nie musimy zarządzać systemem operacyjnym ani samą instancją SQL Servera.
- Chociaż chmurowe rozwiązania SQL Servera mogą wpłynąć na zmianę zestawu umiejętności wymaganych od administratora baz danych, nie eliminują potrzeby zatrudniania takich specjalistów. Dlatego administratorzy baz danych nie powinni obawiać się chmury.
- Instaluj tylko niezbędne funkcje, a nie wszystkie, które są dostępne. Pozwoli to zmniejszyć powierzchnię ataku i uniknąć niepotrzebnego zużycia zasobów.
- Ręczna instalacja SQL Servera jest czasochłonna i podatna na ludzkie błędy.
- Automatyzacja instalacji SQL Servera zmniejsza nakład pracy administracyjnej i ułatwia zarządzanie całym środowiskiem.
- Wdrażanie z użyciem skryptów zapewnia spójność i dobre praktyki tylko w momencie budowania serwera. Po wdrożeniu może dojść do dryfu konfiguracji.
- Stosowanie technik zarządzania konfiguracją pozwala utrzymać prawidłową konfigurację serwera i instancji SQL Servera przez cały cykl życia aplikacji.
- SQL Server można konfigurować za pomocą narzędzi do zarządzania konfiguracją, takich jak PowerShell DSC i Puppet.
- Zarządzanie konfiguracją pomaga zachować zgodność z wymogami bezpieczeństwa i zmniejsza nakłady administracyjne związane z utrzymaniem środowiska SQL Server.
- Narzędzia do zarządzania konfiguracją obejmują Ansible i Puppet. Microsoft oferuje również zarządzanie konfiguracją z wykorzystaniem PowerShella. Funkcja ta nosi nazwę PowerShell DSC.
- Rozwiązanie PowerShell DSC jest w pełni rozszerzalne, a w razie potrzeby można tworzyć własne zasoby DSC.

- Obrazy chmurowe ze wstępnie zainstalowanym SQL Serverem mają zainstalowane wszystkie funkcje, co zwiększa zużycie zasobów oraz powierzchnię ataku.
- Zaleca się tworzyć własne obrazy chmurowe na podstawie obrazów dostarczanych przez dostawcę i wykorzystywać je zamiast oryginalnych.
- Obrazy niestandardowe muszą bazować na obrazach dostarczanych przez producenta, gdy wymagany jest model licencjonowania „z wliczoną licencją”. Koszt licencji SQL Servera jest wówczas wliczony w godzinową opłatę za maszynę wirtualną.

Zarządzanie instancją i bazą danych

W tym rozdziale:

- Typowe pomyłki i nieporozumienia związane z konserwacją
- Planowanie wydajności
- Uszkodzenie bazy danych
- Skrypty administracyjne
- Instalowanie poprawek

W tym rozdziale omówimy typowe pomyłki związane z konserwacją i konfiguracją, jakie popełniają przypadkowi administratorzy baz danych (DBA). Przyjrzymy się skutkom automatycznego zmniejszania baz danych, a następnie zajmiemy się niektórymi błędnymi przekonaniami dotyczącymi plików dziennika transakcji, które to błędy prowadzą do problemów takich jak obniżenie wydajności.

Następnie omówimy planowanie wydajności. Jest to zadanie lekceważone przez wielu administratorów baz danych, a my przyjrzymy się potencjalnym konsekwencjom tego zaniedbania. Potem zbadamy niektóre typowe pomyłki związane ze skryptami i automatyzacją, w tym używanie kursorów oraz całkowity brak automatyzacji czynności konserwacyjnych.

Na koniec przyjrzymy się problemowi nieaktualizowania serwerów. Zastanowimy się nad przyczynami unikania instalacji poprawek oraz konsekwencjami takiego postępowania. Wyjaśnimy sobie również, jak uniknąć tego błędu w przyszłości.

Wiele tematów w tym rozdziale skupia się na codziennej pracy administratora baz danych. Dlatego nie wszystkie omówione zagadnienia bezpośrednio wpływają na biznesową działalność firmy. Trzeba jednak pamiętać, że z pewnością mają pośredni wpływ na firmę. Jeśli na przykład zaniedbamy instalację poprawek, zwiększa się ryzyko ataku na serwery, co może prowadzić do poważnych zakłóceń w działalności i uszczerbku na reputacji firmy.

Zmienia to charakter naszych interakcji z zespołami biznesowymi. Do tej pory większość omawianych działań była inicjowana przez biznes. Natomiast z wieloma inicjatywami omawianymi w tym rozdziale będzie wychodzić zespół DBA w celu uniknięcia lub rozwiązania problemów operacyjnych wpływających na biznes. Na przykład fragmentacja dzienników nie jest problemem, o którym wie albo o którego rozwiązanie mógłby poprosić zespół biznesowy. To my dbamy o tego typu zagadnienia, ponieważ jeśli tego nie zrobimy, biznes pośrednio odczuje skutki w postaci problemów z wydajnością.

W przykładach, które wymagają bazy danych, będziemy korzystać z bazy danych MarketingArchive. Aby ją utworzyć, potrzebna będzie baza Marketing, którą przygotowaliśmy w rozdziale 4. W przykładach wykorzystujących instancję SQL Servera zalecam użycie tej samej instancji, w której znajduje się baza Marketing, choć nie jest to konieczne.

9.1. Numer 43 — automatyczne zmniejszanie baz danych

Większość specjalistów IT znalazła się kiedyś w sytuacji, w której zaczyna brakować miejsca na dysku i trzeba ręcznie usuwać stare dane, aby uniknąć jego całkowitego zapelnienia. Wielu administratorom baz danych zdarzyło się poprosić administratora sieci SAN o rozszerzenie woluminu i usłyszeć ciężkie westchnienie, ponieważ w sieci SAN kończyła się wolna przestrzeń.

Ponadto większość specjalistów od baz danych zdaje sobie sprawę, że gdy maszyna wirtualna z SQL Serverem jest tworzona w środowisku chmurowym, takim jak AWS czy Azure, infrastruktura bazowa nie należy do organizacji. Oznacza to, że rozszerzenie woluminu wiąże się z bezpośrednimi kosztami.

Kiedy więc ktoś, kto dopiero zaczyna pracę jako administrator SQL Servera, dowiaduje się o funkcji automatycznego zmniejszania baz danych i odzyskiwania niewykorzystanej przestrzeni, można mu wybaczyć, że uznaje to za świetny pomysł. Wielokrotnie spotkałem się z sytuacjami, w których przypadkowy administrator włączał tę funkcję dla wszystkich swoich baz danych, nie rozumiejąc konsekwencji takiego działania.

Wyobraźmy sobie, że firma MagicChoc ma instancję SQL Servera hostowaną w chmurze Azure. Instancja ta obsługuje cztery bazy danych:

- Marketing,
- SalesManagerPlus,
- TargetManager,
- MarketingArchive.

Wykorzystanie przestrzeni dyskowej w woluminie danych oscyluje wokół 75%, co stanowi próg ostrzegawczy w narzędziu monitorującym. Niedoświadczony administrator baz danych reaguje na to ostrzeżenie i odkrywa opcję automatycznego zmniejszania baz danych, którą następnie włącza. W ciągu godziny jednak wiele zespołów aplikacyjnych zaczyna zgłaszać problemy, narzekając na drastyczny spadek wydajności ich aplikacji. Włączenie automatycznego zmniejszania baz danych było bowiem pomyłką. Aby zrozumieć dlaczego, trzeba poznać sposób działania tej funkcji.

W SQL Serverze w tle działa proces, który co jakiś czas sprawdza, czy któraś z baz danych ma włączoną opcję automatycznego zmniejszania. Jeśli tak, to znajduje pierwszą bazę z takim ustawieniem i sprawdza, czy można odzyskać wolne miejsce. Jeśli jest taka możliwość, wykonuje operację zmniejszenia tej bazy. Następnie proces usypia na kilka minut. Po przebudzeniu sprawdza i ewentualnie zmniejsza kolejną bazę z włączoną opcją automatycznego zmniejszania. Proces działa w ten sposób nieprzerwanie, przechodząc cyklicznie przez wszystkie bazy danych.

Podczas operacji automatycznego zmniejszania bazy danych system nakłada blokady, które mogą powodować zatrzymanie innych aplikacji wymagających założenia blokady. Może to prowadzić do problemów z wydajnością, a nawet do przekroczenia limitu czasu, jeśli skonfigurowano go dla blokad.

Operacje zmniejszania bazy danych są też bardzo zasobochłonne. W trakcie ich wykonywania znacznie wzrasta wykorzystanie procesora, co może wpłynąć na inne procesy, zwłaszcza gdy używane są inne wymagające funkcje, takie jak szyfrowanie czy kompresja. Podsystem dyskowy również staje się bardzo zajęty, co objawia się dużym skokiem aktywności dysku i może powodować dodatkowe problemy z wydajnością aplikacji. Sytuację pogarsza fakt, że operacja zmniejszania bazy jest w pełni rejestrowana w dzienniku, co oznacza intensywny transfer danych zarówno do pliku dziennika transakcji, jak i do plików danych.

Po zakończeniu operacji zmniejszania ogólna wydajność bazy danych prawdopodobnie pogorszy się w wyniku zjawiska zwanego *fragmentacją indeksów*. Fragmentacja jest spowodowana sposobem, w jaki operacja zmniejszania reorganizuje strony w plikach danych. Nie będziemy się tym jednak zajmować w tym miejscu, ponieważ omówimy to dokładniej w następnym podrozdziale.

Choć automatyczne zmniejszanie baz danych wydaje się kuszącym pomysłem kontrolowania zajętości dysku, znacznie lepszym rozwiązaniem jest zaplanowanie wydajności w celu zapewnienia odpowiednich rozmiarów woluminów danych. Planowanie wydajności omówimy dalej w tym rozdziale.

Za pomocą kwerendy z listingu 9.1 możesz sprawdzić, czy masz jakiekolwiek bazy danych z włączoną funkcją automatycznego zmniejszania.

Listing 9.1. Sprawdzanie, czy włączone jest automatyczne zmniejszanie

```
SELECT
    name
    , is_auto_shrink_on
FROM sys.databases ;
```

Funkcję automatycznego zmniejszania można wyłączyć za pomocą instrukcji ALTER DATABASE. Poniższa kwerenda (listing 9.2) wyłącza automatyczne zmniejszanie bazy danych Marketing:

Listing 9.2. Wyłączanie automatycznego zmniejszania

```
ALTER DATABASE Marketing
SET auto_shrink OFF WITH NO_WAIT ;
```

WSKAZÓWKA W rozdziale 8. omówiliśmy egzekwowanie najlepszych praktyk konfiguracyjnych z użyciem narzędzi Desired State Configuration (DSC). Jeśli stosujesz podejście oparte na zarządzaniu konfiguracją, wymuszenie wyłączenia funkcji automatycznego zmniejszania baz danych jest doskonałym kandydatem do automatyzacji.

9.2. Numer 44 — zaniedbywanie odbudowy indeksów po zmniejszeniu pliku danych

W poprzednim podrozdziale wyjaśniłem, dlaczego włączanie automatycznego zmniejszania bazy danych nie jest dobrym pomysłem. Administratorzy SQL Servera mogą jednak również ręcznie zmniejszać bazy danych lub pliki. Ogólnie rzecz biorąc, należy unikać zmniejszania plików danych, nawet ręcznie. Istnieją jednak sytuacje, w których ręczne zmniejszenie bazy danych może być pomocne.

WSKAZÓWKA Zmniejszanie plików dziennika wiąże się z innymi kwestiami, które omówimy dalej w tym rozdziale.

Rozważmy instancję SQL Servera omówioną w poprzednim podrozdziale. Znajduje się ona w maszynie wirtualnej Azure i obsługuje cztery bazy danych. Jedna z nich, o nazwie MarketingArchive, służy do przechowywania historycznych danych marketingowych. Baza ta działa od dłuższego czasu i obecnie zawiera dane z ostatnich dziewięciu lat. Wymagania biznesowe zakładają jednak przechowywanie danych tylko z ostatnich trzech lat. Wolumin danych osiągnął już 75% pojemności, więc istnieją dwie opcje. Można zwiększyć pojemność

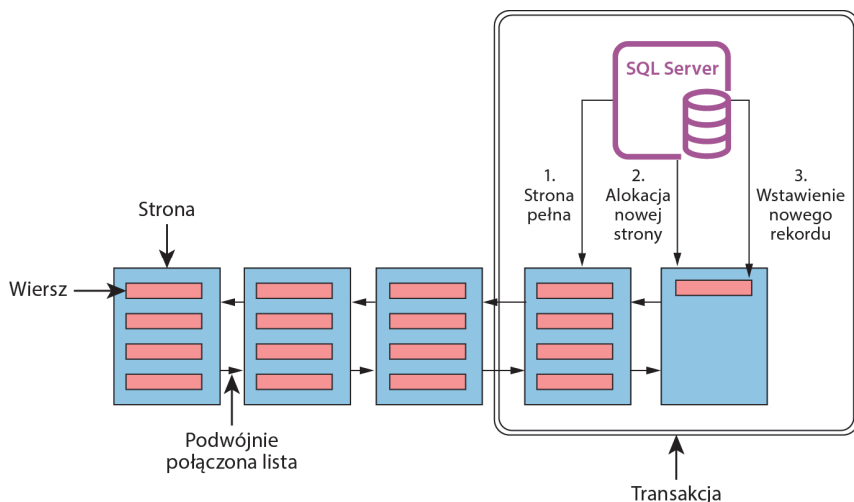
woluminu albo usunąć sześć lat danych z bazy MarketingArchive. Rozszerzenie woluminu wiązałoby się z dodatkowymi opłatami za przechowywanie w Azure, a przechowywane dane nie są już potrzebne. W tej sytuacji sensowne jest więc usunięcie niepotrzebnych danych i przeprowadzenie jednorazowej operacji zmniejszenia bazy danych w celu odzyskania miejsca.

W tym scenariuszu administrator bazy danych usuwa dane i zmniejsza pliki danych. Wie, że operacja zmniejszania plików wpłynie na wydajność systemu, dlatego wykonuje to zadanie w czasie przeznaczonym na konserwację.

Niestety, po zakończeniu operacji użytkownicy zaczynają narzekać na spadek wydajności i dłuższy czas wykonywania wielu kwerend. Powodem tego jest fragmentacja indeksów w bazie danych. Aby to zrozumieć, prześledźmy kroki podjęte przez administratora bazy danych w celu zmniejszenia plików danych i przyjrzyjmy się, co się stało podczas tej operacji. Najpierw jednak przypomnijmy sobie podstawowe pojęcia związane z podziałami stron i fragmentacją indeksów.

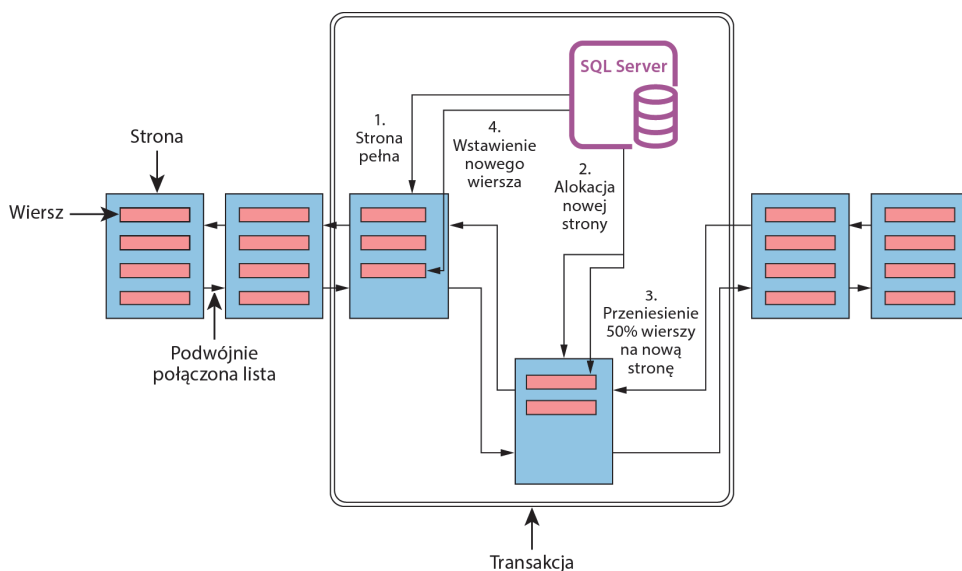
Dane są przechowywane w serii stron o rozmiarze 8 KB, a ciąg ośmiu kolejnych stron 8 KB nazywany jest *ekstentem*. Strony danych tworzące indeks są połączone w dwukierunkową listę. Oznacza to, że każda strona zawiera wskaźnik zarówno do następnej, jak i poprzedniej strony. Gdy strona danych zostanie zapełniona, a SQL Server musi dodać do niej kolejne dane, konieczne jest utworzenie nowej strony. Proces ten nazywany jest *podziałem strony*. Wyróżniamy dwa rodzaje podziału strony: „dobry” i „zły”.

Aby zrozumieć, na czym polega dobry podział strony, przyjrzyj się diagramowi na rysunku 9.1. Diagram ten przedstawia indeks, który został całkowicie zapełniony. SQL Server musi wstawić nowy wiersz na ostatnią stronę, ale nie ma tam już wolnego miejsca. W takiej sytuacji przydzielana jest kolejna strona z ekstentu i nowy wiersz zostaje wstawiony na tę nową stronę.



Rysunek 9.1. Dobre podziały stron

Zły podział strony ma miejsce wtedy, gdy operacja wstawiania lub duża aktualizacja wymaga podziału strony znajdującej się w środku indeksu. W takiej sytuacji nowa strona, którą musi utworzyć SQL Server, będzie w niewłaściwej kolejności. Strony ułożone w niewłaściwej kolejności powodują *fragmentację zewnętrzną*. Dodatkowo połowa wierszy z pierwotnej strony zostaje przeniesiona na nową stronę, aby zwolnić miejsce na stronie źródłowej. To zjawisko nazywane jest *fragmentacją wewnętrzną* i może prowadzić do sytuacji, w której SQL Server musi odczytać więcej stron, aby zwrócić tę samą ilość danych. Zły podział strony zilustrowano na rysunku 9.2.



Rysunek 9.2. Zły podział strony

Zanim bliżej zbadamy ten scenariusz — jeśli chcesz samodzielnie wykonywać przykłady z tego rozdziału, użyj skryptu z listingu 9.3, aby utworzyć i wypełnić danymi bazę MarketingArchive.

WSKAZÓWKA W skrypcie z listingu 9.3 używana jest baza Marketing, którą utworzyliśmy w rozdziale 6.

Listing 9.3. Tworzenie bazy danych MarketingArchive

```
CREATE DATABASE MarketingArchive ;
GO

USE MarketingArchive ;
GO

CREATE TABLE dbo.ImpressionsArchive (
```

| | | |
|-------------------|------------------|--------------------------------|
| ImpressionID | BIGINT | NOT NULL IDENTITY PRIMARY KEY, |
| ImpressionUID | UNIQUEIDENTIFIER | NOT NULL, |
| ReferralURL | VARCHAR(512) | NOT NULL, |
| CookieID | UNIQUEIDENTIFIER | NOT NULL, |
| CampaignID | BIGINT | NOT NULL, |
| RenderingID | BIGINT | NOT NULL, |
| CountryCode | TINYINT | NULL, |
| StateID | TINYINT | NULL, |
| BrowserVersion | BIGINT | NOT NULL, |
| OperatingSystemID | BIGINT | NOT NULL, |
| BidPrice | MONEY | NOT NULL, |
| CostPerMille | MONEY | NOT NULL, |
| EventTime | DATETIME | NOT NULL, |

```
) ;
```

```
DECLARE @Numbers TABLE (
    Number INT
);
```

```
INSERT INTO @Numbers
VALUES (-1),(-2),(-3),(-4),(-5),(-6),(-7),(-8),(-9) ;
```

```
INSERT INTO MarketingArchive.dbo.ImpressionsArchive (
    ImpressionUID,
    ReferralURL,
    CookieID,
    CampaignID,
    RenderingID,
    CountryCode,
    StateID,
    BrowserVersion,
    OperatingSystemID,
    BidPrice,
    CostPerMille,
    EventTime
)
```

```
SELECT
    ImpressionUID,
    ReferralURL,
    CookieID,
    CampaignID,
    RenderingID,
    CountryCode,
    StateID,
    BrowserVersion,
    OperatingSystemID,
    BidPrice,
    CostPerMille,
    DATEADD(YEAR, n.Number, i.EventTime)
FROM Marketing.Marketing.Impressions i
CROSS JOIN @Numbers n ;
```

```
CREATE NONCLUSTERED INDEX EventTimeNCI
    ON dbo.ImpressionsArchive(EventTime) ;
```

```
CREATE NONCLUSTERED INDEX ImpressionUIDNCI
    ON dbo.ImpressionsArchive(ImpressionUID) ;
```


Najpierw użyjemy funkcji zarządzania dynamicznego (ang. *dynamic management function*, DMF) `sys.dm_db_index_physical_stats`. Funkcja ta zwraca szczegółowe informacje o fragmentacji indeksów i przyjmuje następujące parametry:

- identyfikator bazy danych,
- identyfikator obiektu tabeli,
- identyfikator indeksu w tabeli,
- numer partycji,
- tryb.

Przekazanie wartości NULL do parametrów `object_id`, `index_id` i `partition_number` powoduje zwrócenie wyników dla wszystkich obiektów, indeksów i partycji. Parametr `mode` określa poziom dokładności wyników, który wpływa na czas wykonania. `LIMITED` jest najszybszym, ale najmniej dokładnym trybem. Tryb ten generuje statystyki tylko przez analizowanie poziomów innych niż liście w strukturze B-drzewa. Tryb `SAMPLED` generuje statystyki na podstawie jednego procenta stron w indeksie, a tryb `DETAILED` skanuje wszystkie strony B-drzewa, aby wygenerować statystyki.

DMF zwraca jeden wiersz dla każdego poziomu B-drzewa każdego indeksu w zakresie. Dotyczy to zarówno indeksów klastrowych, jak i nieklastrowych.

Informacje o wyjściowym poziomie fragmentacji zewnętrznej możemy uzyskać przez wykonanie kwerendy przedstawionej na listingu 9.4. Kwerenda ta wykorzystuje funkcję `sys.dm_db_index_physical_stats` do zwrócenia statystyk fragmentacji dla każdego indeksu. Dodatkowo łączy się z widokiem systemowym `sys.indexes`, aby pobrać nazwy indeksów. Kolumna `index_level_size_MB` jest generowana przez podzielenie liczby stron w poziomie indeksu przez 128, ponieważ 1 MB może pomieścić 128 stron o rozmiarze 8 KB. Kolumna `avg_fragmentation_in_percent` zwraca poziom fragmentacji zewnętrznej. Filtrujemy również wyniki, aby pokazać tylko poziom 0, co oznacza, że otrzymamy informacje tylko o poziomie liści każdego indeksu.

WSKAZÓWKA W DMF istnieje kolumna o nazwie `avg_page_space_used_in_percent`, która informuje o stopniu wewnętrznej fragmentacji. Nie jest to istotne w tym rozdziale, fragmentacją wewnętrzną zajmiemy się w rozdziale 11.

Listing 9.4. Pomiar wyjściowego poziomu fragmentacji

```
SELECT
    OBJECT_NAME(ips.object_id) AS table_name
    , i.name AS index_name
    , ips.index_type_desc
    , ips.index_level
    , ips.page_count
    , ips.page_count /128 AS index_level_size_MB
    , ips.avg_fragmentation_in_percent
FROM sys.dm_db_index_physical_stats(DB_ID('MarketingArchive'), NULL, NULL,
```

```

NULL, 'Detailed') ips
INNER JOIN sys.indexes i
    ON i.object_id = ips.object_id
    AND i.index_id = ips.index_id
WHERE index_level = 0 ;

```

Zauważ, że wartość `avg_fragmentation_in_percent` jest bardzo niska dla każdego indeksu. W moich wynikach indeks klastrowy i indeks `EventTimeNCI` miały po 0,01%, a indeks `ImpressionUIDNCI` — 0,12%. Twoje wyniki mogą się jednak różnić w zależności od liczby rdzeni dostępnych dla instancji SQL Servera, a także innych czynników, takich jak obecność innych transakcji wykonywanych w bazie danych.

Teraz, gdy jesteśmy już przygotowani, wykonajmy te same kroki, które podjąłby administrator bazy danych MagicChoc. Na początek usuniemy niepotrzebne dane z ostatnich dziewięciu lat; następnie wykonamy operację zmniejszenia plików danych. Kwerendę usuwającą stare dane pokazano na listingu 9.5.

Listing 9.5. Usuwanie starych danych

```

DELETE
FROM dbo.ImpressionsArchive
WHERE EventTime < '20200101' ;

```

Teraz użyjemy zapytania z listingu 9.6, aby sprawdzić, ile wolnego miejsca możemy odzyskać.

Listing 9.6. Sprawdzanie wolnego miejsca

```

USE MarketingArchive ;
GO

SELECT
    name
    , AvailableSpaceMB
    , CurrentSizeMB
    , (CurrentSizeMB AvailableSpaceMB) * 1.2 AS TargetSizeMB
FROM (
    SELECT
        name
        , size / 128 CAST(FILEPROPERTY(name, 'SpaceUsed') AS INT) / 128
    AS AvailableSpaceMB
        , size / 128 AS CurrentSizeMB
    FROM sys.database_files
    WHERE type = 0
) df ;

```

Twoje wyniki mogą się nieco różnić, ale w moim przypadku docelowa przestrzeń, z uwzględnieniem 20% na naturalny wzrost, wynosi 838,8 MB. Mając tę informację, możemy teraz zmniejszyć plik danych bazy za pomocą polecenia `DBCC SHRINKFILE`. W przykładzie z listingu 9.7 przekazujemy temu poleceniu logiczną nazwę pliku danych oraz docelowy rozmiar. Używamy również opcji

WAIT_AT_LOW_PRIORITY. Spowoduje to, że operacja zmniejszania zostanie przerwana po 1 minucie, jeśli długo wykonująca się kwerenda będzie uniemożliwiała założenie blokady. Ta opcja jest nowością w SQL Server 2022 i pomaga zminimalizować ryzyko negatywnego wpływu na działanie aplikacji.

Listing 9.7. Zmniejszanie pliku danych

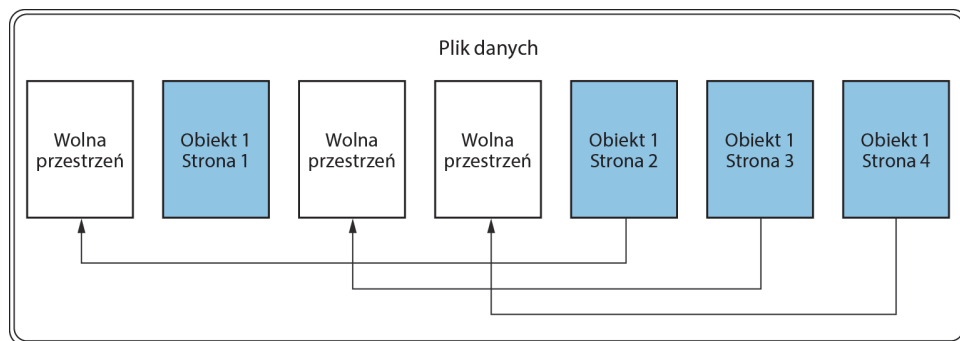
```
DBCC SHRINKFILE ('MarketingArchive' , 838) WITH WAIT_AT_LOW_PRIORITY ;
GO
```

Gdybyśmy teraz ponownie uruchomili kwerendę z listingu 9.6, zobaczylibyśmy, że nasza baza danych ma mniej więcej docelowy rozmiar. Wszystko wydaje się działać zgodnie z oczekiwaniami, ale pomyłką jest niezrozumienie wpływu, jaki będzie to miało na fragmentację naszych indeksów.

Gdybyśmy ponownie uruchomili kwerendę z listingu 9.4, zobaczylibyśmy, że niektóre lub wszystkie nasze indeksy mają fragmentację bliską 100%. Oznacza to, że strony indeksu są całkowicie nieuporządkowane, co może powodować problemy z wydajnością kwerend wykonujących odczyty sekwencyjne, takich jak skanowanie indeksu.

WSKAZÓWKA Powszechnym błędnym przekonaniem jest to, że fragmentacja indeksów powoduje problemy z wydajnością wszystkich kwerend. Nie jest to prawda. Nie ma ona wpływu na operacje wyszukiwania, które zwracają pojedynczy wiersz. Spadek wydajności będzie widoczny w kwerendach wykonujących operacje skanowania indeksu lub przeszukiwania obejmującego wiele stron.

Aby zrozumieć, dlaczego tak się dzieje, przyjrzyjmy się diagramowi z rysunku 9.3. Ilustruje on właśnie przeprowadzoną operację zmniejszania. Proces rozpoczął się od końca pliku i za każdym razem, gdy napotkał stronę, przeniósł ją na pierwsze dostępne miejsce w pliku. W przypadku wielu obiektów oznaczało to całkowite odwrócenie kolejności stron, co doprowadziło do dużej fragmentacji.



Rysunek 9.3. Operacja zmniejszania pliku

Kiedy więc musisz zmniejszyć rozmiar pliku danych, zawsze powinieneś odbudować wszystkie indeksy po zakończeniu operacji zmniejszania. Najprostszy sposób, aby to osiągnąć, przedstawiono na listingu 9.8, choć w rozdziale 11. omówimy alternatywne podejście. Polecenie to uruchamia wbudowaną procedurę składowaną, która przechodzi przez wszystkie tabele w bazie danych. Przekazuje ona instrukcję, która odbudowuje wszystkie indeksy dla danej tabeli, używając znaku ? jako symbolu zastępczego dla nazwy tabeli.

Listing 9.8. Odtwarzanie wszystkich indeksów w bazie danych

```
EXEC sp_msforeachtable 'ALTER INDEX ALL ON ? REBUILD' ;
```

Plików danych nie należy zmniejszać regularnie. Należy robić to jednorazowo, tylko w przypadku rzeczywistej potrzeby. Jeśli jednak zajdzie konieczność zmniejszenia pliku danych, ważne jest, aby po tej operacji odbudować wszystkie indeksy. Pozwoli to uniknąć problemów z wydajnością spowodowanych dużą fragmentacją indeksów.

9.3. Numer 45 — poleganie na automatycznym powiększaniu

Domyślnie, gdy SQL Server wyczerpie miejsce w pliku, automatycznie powiększy go o 64 MB. Standardowo pliki danych mogą rosnać aż do wypełnienia całego woluminu, natomiast pliki dziennika mogą osiągnąć rozmiar 2 TB lub wypełnić cały wolumin. Korzystanie z tych domyślnych ustawień może prowadzić do problemów zarówno w przypadku plików danych, jak i plików dziennika.

Baza danych MarketingArchive firmy MagicChoc to duży zbiór danych, który powiększa się o około 5 GB dziennie w wyniku procesu ekstrakcji, transformacji i wczytywania (ang. *extract, transform, load*, ETL), w ramach którego pobierane są historyczne dane z bazy Marketing. Niektóre transakcje są bardzo duże, a ponadto wykonywane równolegle. Prowadzi to do automatycznego rozszerzania zarówno plików danych, jak i dziennika. Zespół administratorów bazy danych korzysta z domyślnych ustawień automatycznego powiększania.

Procesy ETL zajmują dużo czasu, a analiza wykazuje, że operacje losowego dostępu, niektóre bardzo małe, również trwają długo i blokują wykonanie innych transakcji. Dzieje się tak, ponieważ poleganie na automatycznym zwiększaniu rozmiaru plików do zarządzania ich wielkością jest pomyłką.

Automatyczne zwiększanie rozmiaru plików może powodować problemy w różnych okolicznościach. Powiększanie plików zużywa zasoby dysku i procesora.

Co więcej, jeśli transakcja musi powiększyć plik dziennika, blokuje ona inne transakcje próbujące zapisać dane w tym pliku, dopóki nie dobiegnie końca zarówno operacja powiększania, jak i modyfikacja danych.

Używanie domyślnego ustawienia przyrostu plików danych o 64 MB w bazie danych, która rośnie o 5 GB dziennie, tylko pogłębia problemy związane z kosztami dodatkowymi. Co więcej, taki wzorec małych przyrostów plików może prowadzić do fragmentacji dysku, co z kolei może powodować problemy z wydajnością operacji odczytu sekwencyjnego z tradycyjnych dysków twardych. Zezwalanie, aby pliki dziennika rosły w małych krokach, może prowadzić do fragmentacji dziennika transakcji, co szczegółowo omówimy dalej w tym rozdziale.

Gdy plik rośnie, system musi „wyzerować” nową przestrzeń w tym pliku. Polega to dosłownie na zapisywaniu zer w pliku, aż zostanie on wypełniony. Właśnie to stanowi znaczną część kosztów dodatkowych związanych ze wzrostem pliku. Można częściowo złagodzić ten problem, stosując natychmiastową inicjalizację plików, o której będzie mowa w rozdziale 10.

Czy zalecam wyłączenie funkcji automatycznego powiększania? Nie, absolutnie nie. Jestem zdecydowanym zwolennikiem włączenia tej funkcji, ale uważam, że powinna ona służyć jako zabezpieczenie w przypadku nagłego, nieoczekiwanego wzrostu bazy danych, a nie jako narzędzie do codziennego zarządzania rozmiarem plików.

WSKAZÓWKA W ramach rutynowego zarządzania bazą danych powinniśmy z góry ustalić rozmiar plików danych i dziennika, biorąc pod uwagę ich przewidywany wzrost. Następnie możemy włączyć funkcję automatycznego powiększania, aby zabezpieczyć się przed nieoczekiwanymi sytuacjami. Należy też dostosować przyrost plików do rozmiaru bazy danych. Jeśli na przykład spodziewamy się, że baza MarketingArchive będzie rosła o 5 GB dziennie, możemy ustawić przyrost na 5 GB, a proces ETL uruchamiany w nocy będzie mógł powiększać plik tylko raz na dobę. Jeśli monitorujemy wzrost plików, następnego ranka zauważymy wszelkie anomalie i będziemy mogli je zbadać. Takie podejście zapobiega niepowodzeniom procesu ETL z powodu zapełnienia pliku, dając nam jednocześnie możliwość zaimplementowania strategicznej poprawki.

Instrukcja z listingu 9.9 konfiguruje plik danych w bazie danych Marketing Archive tak, aby za każdym razem rósł o 5 GB.

Listing 9.9. Zmiana przyrostu automatycznego powiększania

```
ALTER DATABASE MarketingArchive  
  MODIFY FILE ( NAME = 'MarketingArchive', FILEGROWTH = 5GB ) ;
```

Gdybyśmy zmienili domyślny przyrost rozmiaru plików w systemowej bazie danych model, stałby się on domyślnym przyrostem dla nowych baz danych tworzonych w danej instancji. Oczywiście nadal moglibyśmy to zmienić podczas tworzenia bazy danych, używając opcji WITH FILEGROWTH w instrukcji CREATE DATABASE. W rzeczywistości zdecydowanie zalecam, aby zawsze określać rozmiar

plików i ustawienia ich przyrostu przy tworzeniu bazy danych, szczególnie w środowisku produkcyjnym.

Nie powinniśmy pozwalać, aby funkcja automatycznego powiększania zarządzała na co dzień rozmiarem naszych plików danych i dziennika. Opcja ta powinna być włączona, ale pliki powinny być wstępnie zwymiarowane poprzez planowanie wydajności. Automatyczne powiększanie powinno być jedynie rozwiązaniem awaryjnym.

9.4. Numer 46 — używanie wielu plików dziennika

Bazy danych SQL Servera mogą korzystać z wielu plików danych i plików dziennika. W niektórych przypadkach rozdzielenie danych na wiele plików może być korzystne. W szczególności może to poprawić równoległość przetwarzania, zmniejszyć rywalizację o zasoby systemowe i ułatwić migrację bazy danych. Może to również pomóc w zaawansowanych strategiach przywracania danych.

A co z plikami dziennika? Rozważmy przykład z firmy MagicChoc. Baza danych MarketingArchive otrzymuje dane z wielu równoległych transakcji podczas nocnego procesu ETL. Proces ten jest dość powolny, a analizując statystyki oczekiwania, administrator bazy danych zauważył znaczące opóźnienia związane z typem oczekiwania WRITELOG, co wskazuje na problem z wydajnością podczas zapisów do dziennika transakcji. W związku z tym administrator postanowił utworzyć dodatkowy plik dziennika, aby zmniejszyć obciążenie i poprawić wydajność. Jest to jednak pomyłka. Wyjaśnijmy dlaczego.

W przeciwieństwie do plików danych dzielenie dziennika transakcji na wiele plików nie przynosi żadnych korzyści. Pozycje dziennika są zawsze zapisywane sekwencyjnie. Oznacza to, że nawet jeśli dodamy kolejny plik dziennika transakcji, SQL Server nadal będzie zapisywał wszystkie transakcje w pierwszym pliku dziennika. Drugi plik zostanie użyty dopiero wtedy, gdy pierwszy się zapełni. Dlatego też nie ma żadnych korzyści wydajnościowych wynikających z używania wielu plików dziennika, ale nie wpływa to również negatywnie na wydajność. Czy zatem ta praktyka jest rzeczywiście tak zła?

Problem pojawia się, gdy zachodzi potrzeba przywrócenia bazy danych. Jeśli dziennik transakcji nie istnieje, SQL Server musi go utworzyć. Jak wspomniałem w poprzednim podrozdziale, zazwyczaj wiąże się to z koniecznością wyzerowania pliku. W przypadku dwóch plików dziennika transakcji konieczne jest wyzerowanie obu plików zamiast jednego. Wydłuża to czas przywracania bazy.

Jedyną sytuacją, w której powinniśmy rozważyć użycie wielu plików dziennika, jest tymczasowe rozwiązanie problemu polegającego na tym, że dziennik transakcji zajął całą dostępną przestrzeń na dysku. W takim przypadku dopuszczalne

jest utworzenie drugiego pliku dziennika na innym woluminie, aby umożliwić kontynuację operacji bazodanowych.

Jeśli zdecydujemy się na takie podejście, powinniśmy korzystać z niego tylko do czasu opracowania bardziej strategicznego rozwiązania. Może ono obejmować zwiększenie pojemności woluminu, na którym znajduje się pierwotny plik dziennika, przeniesienie dziennika na nowy, większy wolumin lub w niektórych przypadkach zbadanie przyczyn tak szybkiego rozrostu pliku. Może to wskazywać na ukryty problem, na przykład niepoprawnie działające kopie zapasowe.

Co administrator bazy danych MagicChoc powinien był zrobić inaczej, aby poprawić wydajność procesu ETL? Nie ma tu jednoznacznej reguły, ale w naszym scenariuszu, gdzie mamy do czynienia z hurtownią danych wypełnianą przez proces ETL, istnieje duże prawdopodobieństwo, że będą występować hurtowe wstawienia danych. Wynika to stąd, że hurtownie danych są często zasilane hurtowo z baz danych typu OLTP lub innych źródeł według ustalonego harmonogramu, najczęściej w nocy.

Jeśli tak jest, możemy rozważyć zmianę modelu przywracania bazy danych na BULK_LOGGED na początku procesu ETL. Spowoduje to, że operacje hurtowe będą rejestrowane w dziennikach w minimalnym stopniu. Zmniejszy to liczbę pozycji zapisywanych w dzienniku transakcji. W rezultacie zostanie wykorzystane mniej miejsca na dziennik, a także zmniejszy się obciążenie podsystemu dyskowego związane z operacjami wejścia-wyjścia na dzienniku transakcji. Po zakończeniu procesu ETL można ustawić model przywracania z powrotem na FULL i wykonać kopię zapasową dziennika transakcji.

Przełączenie z powrotem na model pełnego przywracania i wykonanie kopii zapasowej dziennika transakcji jest ważne, ponieważ model z hurtowym rejestrowaniem nie pozwala na przywracanie do określonego punktu w czasie. Oznacza to, że w scenariuszu przywracania bazy danych moglibyśmy odzyskać dane tylko do momentu wykonania ostatniej kopii zapasowej dziennika transakcji. Nie byłoby możliwe odtworzenie bazy do stanu z dowolnego punktu pomiędzy kopiami.

Model przywracania bazy danych MarketingArchive można zmienić na BULK_LOGGED za pomocą polecenia przedstawionego na listingu 9.10.

Listing 9.10. Ustawianie modelu przywracania bazy danych na tryb hurtowego rejestrowania

```
ALTER DATABASE MarketingArchive SET RECOVERY BULK_LOGGED WITH NO_WAIT ;
```

Innym sposobem na poprawę wydajności operacji wejścia-wyjścia może być przeniesienie dziennika zdarzeń na szybszy dysk. Wielu administratorów systemów SAN stosuje jednolite zasady używania macierzy RAID-5 lub RAID-6 dla wszystkich woluminów. Jednak ze względu na sekwencyjny charakter zapisów do dziennika transakcji optymalnym poziomem RAID jest RAID-1.

UWAGA Administratorzy SAN nie zawsze mają możliwość zapewnienia obsługi wielu poziomów RAID.

Podsumowując — bazy danych powinny mieć tylko jeden plik dziennika transakcji. Używanie wielu dzienników transakcji nie poprawia wydajności, a może wydłużyć czas przywracania danych. Jeśli wolumin z dziennikiem się zapełni, można tymczasowo utworzyć drugi plik dziennika na innym woluminie, ale powinno to być jedynie doraźne rozwiązanie.

9.5. Numer 47 — dopuszczanie do fragmentacji dzienników

Chociaż większość administratorów baz danych zdaje sobie sprawę z fragmentacji indeksów, to tylko nieliczni wiedzą o fragmentacji dziennika transakcji i problemach, jakie może ona powodować. Zastanówmy się nad bazą danych MarketingArchive. Utworzyliśmy ją, korzystając z domyślnych ustawień rozmiaru i przyrostu. W rezultacie plik dziennika został początkowo utworzony z rozmiarem 8 MB, a w przypadku wyczerpania miejsca będzie rósł o 64 GB aż do maksymalnej wielkości 2 TB. Administratorzy bazy danych MagicChoc pozostawili te domyślne ustawienia bez zmian. Wcześniej w tym rozdziale wyjaśniłem, jak w niektórych okolicznościach rozrost pliku dziennika może prowadzić do zwiększenia intensywności operacji wejścia-wyjścia i blokowania innych transakcji.

W tym konkretnym przypadku, przy założeniu że używamy wersji SQL Server 2022 lub nowszej oraz że włączone jest natychmiastowe inicjowanie plików (o czym będzie mowa w rozdziale 11.), plik dziennika nie będzie musiał być zerowany, ponieważ rośnie on w małych, 64-megabajtowych kawałkach. Jednak zezwolenie na rozrost dziennika przy użyciu ustawień domyślnych jest pomyłką, ponieważ prowadzi do fragmentacji pliku dziennika. To z kolei może powodować problemy z wydajnością. Aby zrozumieć dlaczego, musisz wiedzieć co nieco o architekturze dziennika transakcji.

W pliku dziennika transakcji SQL Server tworzy kilka *wirtualnych plików dziennika* (ang. *virtual log file*, VLF). Wewnątrz VLF znajduje się szereg *bloków dziennika*, które są pojemnikami na rekordy dziennika transakcji i jednostkami, w których rekordy są zapisywane na dysk. Te *rekordy dziennika* zawierają informacje o wszystkich operacjach DML (ang. *data manipulation language*) i DDL (ang. *data definition language*) wykonanych w bazie danych. W modelu przywracania Full rejestrowane są również wszystkie przydzielone i zwolnione strony oraz ekstenty. Każdy rekord dziennika otrzymuje *numer sekwencyjny dziennika* (ang. *log sequence number*, LSN). LSN składa się z trzech części:

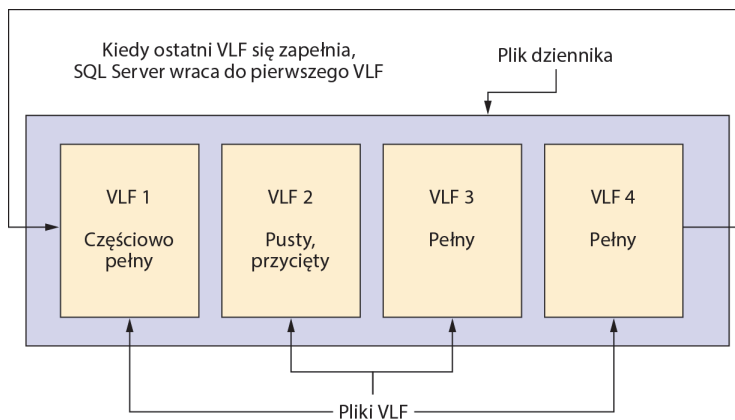
numeru sekwencyjnego VLF, identyfikatora bloku dziennika oraz identyfikatora rekordu dziennika.

Podczas przycinania dziennika transakcji przycinany jest nie sam plik dziennika, lecz zawarte w nim pliki VLF. Przycięte zostaną wszystkie pliki VLF, które nie zawierają żadnych aktywnych transakcji. Transakcja może być oznaczona jako aktywna z wielu powodów, między innymi:

- VLF zawiera transakcje, które nie zostały jeszcze dokończone.
- VLF zawiera transakcje, które nie zostały jeszcze przesłane do replik w topologii grupy dostępności.
- VLF zawiera transakcje, które nie zostały jeszcze przechwycone przez mechanizm śledzenia zmian w danych.

Plik dziennika transakcji nigdy nie jest opróżniany całkowicie; zawsze istnieje co najmniej jeden aktywny VLF, ale w danym momencie może być ich wiele.

Plik dziennika transakcji ma charakter cykliczny. Gdy ostatni VLF w pliku dziennika zostanie zapełniony, SQL Server wraca na początek dziennika i zaczyna używać pierwszego VLF, który został przycięty. Pokazano to na rysunku 9.4.



Rysunek 9.4. Cykl dziennika transakcji

Jeśli w pliku dziennika transakcji nie ma żadnych przyciętych wirtualnych plików dziennika (VLF), to gdy ostatni VLF zostanie zapełniony, serwer SQL Server spróbuje powiększyć plik dziennika zgodnie z ustawieniami automatycznego wzrostu dla tego pliku. Jeśli nie uda się powiększyć pliku, ponieważ ustawienia wzrostu na to nie pozwalają lub dysk jest pełny, zostanie zgłoszony błąd 9002 informujący o zapełnieniu dziennika transakcji.

Gdy plik dziennika rośnie, liczba tworzonych plików VLF zależy od aktualnego rozmiaru dziennika transakcji oraz przyrostu wielkości. Zasady określające liczbę tworzonych VLF mówią, że jeśli przyrost jest mniejszy niż $\frac{1}{8}$ aktualnego rozmiaru dziennika, zostanie utworzony jeden VLF. W przeciwnym razie:

- Jeśli przyrost jest mniejszy niż 64 MB, utworzone zostaną 4 pliki VLF.
- Jeśli przyrost wynosi od 64 MB do 1 GB, zostanie utworzonych 8 plików VLF.
- Jeśli przyrost przekracza 1 GB, zostanie utworzonych 16 plików VLF.

Ten wzór może prowadzić do tworzenia dziennika zawierającego bardzo dużą liczbę plików VLF. *Fragmentacja dziennika* występuje wtedy, gdy dziennik transakcji zawiera nieproporcjonalnie dużą liczbę plików VLF. Może to powodować problemy z wydajnością podczas przywracania bazy danych, dołączania bazy danych oraz odzyskiwania bazy danych przy uruchamianiu instancji SQL Servera. Może to również wpływać negatywnie na wydajność technologii takich jak grupy dostępności AlwaysOn.

Nie ma ustalonej liczby plików VLF, które powinien zawierać plik dziennika, ale często zaczynają one powodować problemy, gdy jest ich więcej niż 1000. Każdą bazę danych należy rozpatrywać indywidualnie, ale ogólnie zaleca się utrzymywanie od dwóch do czterech plików VLF na każdy GB całkowitego rozmiaru pliku dziennika. Należy również unikać zwiększania dużego pliku dziennika o więcej niż 8 GB naraz, ponieważ zbyt mała liczba plików VLF również może prowadzić do problemów.

Kwerenda z listingu 9.11 pozwala sprawdzić, czy plik dziennika bazy danych MarketingArchive jest pofragmentowany. Podkwerenda zlicza liczbę plików VLF dla wskazanej bazy przez wywołanie funkcji `sys.dm_db_log_info`. Kwerenda zewnętrzna łączy te wyniki z widokiem systemowym `sys.master_files` i wykonuje proste obliczenia, aby przekonwertować rozmiar pliku z KB na GB oraz obliczyć optymalną minimalną i maksymalną liczbę plików VLF.

Listing 9.11. Sprawdzanie fragmentacji dziennika

```
SELECT
    li.database_id
    , li.ActualVLFs
    , mf.size/1024/1024 AS SizeInGB
    , mf.size/1024/1024*2. AS MinIdealVLFs
    , mf.size/1024/1024*4. AS MaxIdealVLFs
FROM (
    SELECT database_id, COUNT(*) AS ActualVLFs
    FROM sys.dm_db_log_info(DB_ID('MarketingArchive'))
    GROUP BY database_id
) li
INNER JOIN sys.master_files mf
    ON li.database_id = mf.database_id
WHERE mf.type = 1 ;
```

Jeśli uznasz, że konieczne jest rozwiązanie problemu fragmentacji pliku dziennika, będziesz musiał go opróżnić. W przypadku bazy danych z pełnym modelem odzyskiwania (FULL) możesz to zrobić poprzez utworzenie kopii zapasowej, natomiast w przypadku prostego modelu odzyskiwania (SIMPLE) — poprzez

przycięcie dziennika. Następnie musisz zmniejszyć plik do najmniejszego możliwego rozmiaru za pomocą polecenia DBCC SHRINKFILE. Na koniec ustaw odpowiednią wartość przyrostu, w zależności od docelowego rozmiaru dziennika transakcji.

Powinniśmy zawsze starać się zapobiegać fragmentacji dzienników. Jeśli do niej dojdzie, możemy napotkać problemy z wydajnością podczas operacji przywracania lub korzystania z funkcji takich jak grupy dostępności. Aby usunąć istniejącą fragmentację, możemy zmniejszyć plik dziennika, a następnie pozwolić mu rosnąć z odpowiednim przyrostem.

9.6. Numer 48 — zaniedbywanie planowania wydajności

Planowanie wydajności to proces szacowania ilości zasobów, których aplikacja będzie potrzebować po upływie określonego czasu. Okres ten zazwyczaj pokrywa się z cyklem wymiany sprzętu w organizacji. Wyobraźmy sobie na przykład, że firma MagicChoc ma trzyletni cykl wymiany sprzętu. Dwa lata po rozpoczęciu cyklu powstaje nowa aplikacja, która ma działać na maszynach wirtualnych w firmowym centrum danych. W fazie planowania wydajności dla tego projektu zadaniem będzie oszacowanie, ile mocy obliczeniowej, pamięci i przestrzeni dyskowej aplikacja będzie wymagać po roku użytkowania. Pozwoli to organizacji ocenić, czy posiada wystarczające zasoby w macierzy SAN i klastrze wirtualizacyjnym, aby obsługiwać aplikację do czasu następnej wymiany sprzętu, czy też konieczne będzie nabycie nowego sprzętu.

Rok później, pod koniec cyklu odświeżania sprzętu, konieczne będzie przeprowadzenie kolejnej sesji planowania wydajności. Tym razem zadanie będzie polegało na określeniu ilości zasobów, których aplikacja będzie wymagała za trzy lata. Na tej podstawie organizacja ustali, ile dodatkowej mocy obliczeniowej i pojemności pamięci masowej należy dokupić podczas modernizacji.

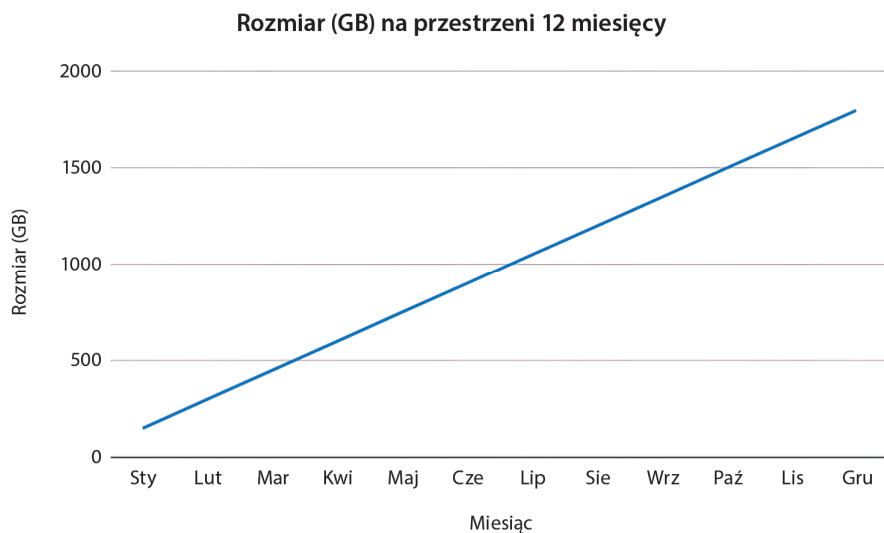
Przeanalizujemy scenariusz, o którym właśnie wspomnieliśmy, aby zrozumieć, dlaczego zaniedbanie planowania wydajności jest błędem. Minęły dwa lata od początku cyklu i właśnie uruchomiliśmy aplikację do analizy historycznych danych marketingowych. Baza danych MarketingArchive została przeniesiona ze środowiska rozwojowego do produkcyjnego, z plikami danych o rozmiarze 200 GB. Zauważamy, że w plikach danych jest 50 GB wolnego miejsca, więc uznajemy, że jest wystarczająco dużo przestrzeni na naturalny wzrost. Ponieważ jesteśmy zajęci, a planowanie wydajności jest trudne, rezygnujemy z tego procesu.

Zapomnieliśmy, że baza danych codziennie rośnie o 5 GB, a zasady wymagają przechowywania danych przez trzy lata. Oznacza to, że po roku, pod koniec cyklu życia sprzętu, pliki danych w bazie będą zajmować łącznie 1,78 TB.

Dodatkowo nie wzięliśmy pod uwagę, jak duże rozmiary osiągną systemowa baza danych TempDB i dziennik transakcji. Przy uwzględnieniu miejsca na dziennik transakcji i TempDB, a także zapasu na nieoczekiwany wzrost, w przypadku tak dużego magazynu danych możliwe jest, że będziemy potrzebować około 3,2 TB przestrzeni. Większym problemem jest to, że w firmowej macierzy SAN pozostały tylko 2 TB dostępnej pojemności zarezerwowanej już dla innych aplikacji, które zgłosiły zapotrzebowanie na przestrzeń dyskową.

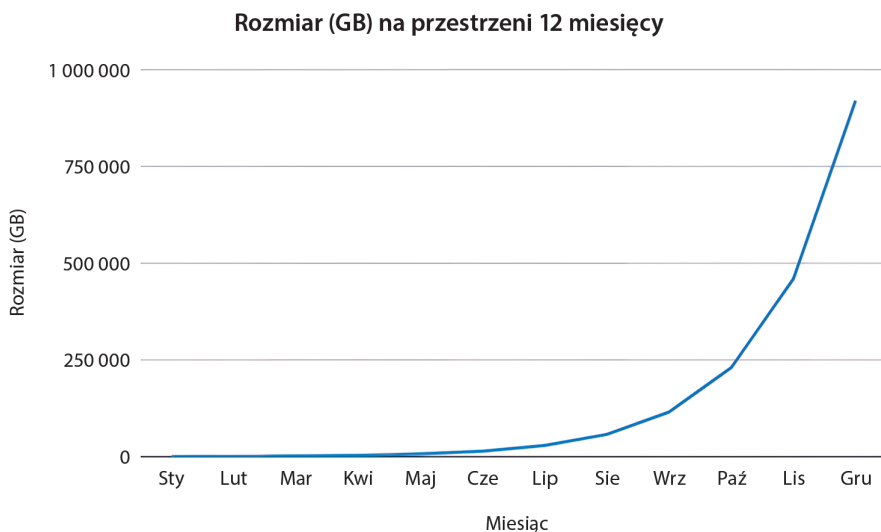
Ten scenariusz prawdopodobnie doprowadzi do bardzo trudnej rozmowy z działem biznesowym. Może również skutkować długim okresem przestoju, podczas którego zespół odpowiedzialny za pamięć masową będzie musiał zamówić i zainstalować dodatkową pojemność SAN. Nie ulega wątpliwości, że brak planowania wydajności jest pomyłką, ale niestety jest to sytuacja, którą obserwuję niepokojąco często.

Aby uniknąć tego błędu, zespół administratorów baz danych powinien był oszacować przyszłe wymagania we współpracy z zespołem programistów. Obliczenie wymaganej pojemności pamięci masowej dla plików danych tej aplikacji jest dość proste. Aplikacja zużywa około 5 GB danych dziennie w stałym tempie. Jest to tak zwany *liniowy wzorzec wzrostu*, co zilustrowano na rysunku 9.5.



Rysunek 9.5. Wzorzec wzrostu liniowego

Wyobraźmy sobie jednak sytuację, w której ilość przetwarzanych danych nie rośnie w stałym tempie, lecz podwaja się co miesiąc. Zamiast wzrostu liniowego obserwowalibyśmy wtedy wzrost wykładniczy, który pokazano na rysunku 9.6.



Rysunek 9.6. Wzorzec wzrostu wykładniczego

Metoda planowania wydajności, którą zastosowaliśmy dla plików danych, jest zazwyczaj związana z istniejącymi systemami działającymi od pewnego czasu. Mamy wtedy do dyspozycji serię historycznych punktów danych, które możemy przedstawić na wykresie i prognozować na przyszłość. Niestety nie zawsze tak jest w przypadku nowych aplikacji (typu greenfield). W wielu scenariuszach nie ma też równomiernego napływu danych każdego dnia.

W przypadku projektów typu greenfield zwykle musimy porozmawiać z osobą odpowiedzialną za aplikację po stronie biznesowej i ocenić spodziewany rozwój działalności. Jeśli na przykład rozwiązanie dotyczy aplikacji sprzedażowej, moglibyśmy zapytać o prognozy sprzedaży na pierwszy rok funkcjonowania aplikacji.

Gdy zbierzemy jak najwięcej informacji od działów biznesowych, możemy zdefiniować reguły i rozszerzyć dane w bazie. Na przykład w nowej bazie danych sprzedażowych moglibyśmy ręcznie dodać dane 10 klientów, z przeciętną liczbą adresów dostawy ustaloną na podstawie rozmów z przedstawicielami biznesu. Później moglibyśmy ręcznie wprowadzić po jednym zamówieniu dla każdego klienta.

Następnie możemy rozszerzyć dane zgodnie z oczekiwaniami biznesowymi. Na przykład, jeśli prognoza sprzedaży zakłada, że w pierwszym roku zostanie pozyskanych 5000 klientów, a każdy z nich złoży średnio 10 zamówień, możemy wykorzystać złączenia krzyżowe (CROSS JOIN) i tabele liczb do powielenia danych klientów 500 razy, a danych zamówień 5000 razy. Pozwoli to nam uzyskać przybliżone oszacowanie spodziewanej wielkości bazy danych.

Wykonanie typowych operacji na bazie po rozszerzeniu danych pozwoli nam również ocenić wymagania dotyczące rozmiaru bazy danych TempDB i pliku dziennika transakcji. Jest to ważna kwestia, szczególnie w scenariuszach hurtowni

danych, ponieważ ilość miejsca zajmowanego przez te pliki może być większa, niż się spodziewasz. Z mojego doświadczenia wynika, że nie jest niczym niezwykłym, gdy każdy z nich wymaga około 25% rozmiaru plików danych bazy.

Należy również pamiętać o konieczności zapewnienia odpowiedniej rezerwy. Jeśli na przykład przewidujemy, że łączna wielkość naszych plików danych, pliku dziennika i bazy TempDB wyniesie 1 TB, powinniśmy dodać dodatkowe 20% na nieplanowany wzrost. W rezultacie nasze oszacowanie wyniosłoby 1,2 TB.

WSKAZÓWKA Planowanie wydajności to zawsze pisanie palcem po wodzie, co szczególnie dotyczy nowych systemów. Nigdy jednak nie jest to strata czasu. Znacznie lepiej jest mieć choćby przybliżone prognozy wzrostu niż nie mieć żadnego pojęcia o tym, czego się spodziewać.

W przypadku scenariuszy chmurowych kwestie związane z planowaniem wydajności wyglądają nieco inaczej, ale nadal są równie istotne. Musimy pamiętać, że choć łatwo możemy zwiększyć rozmiar instancji, dodając więcej pamięci i mocy obliczeniowej, wiąże się to z koniecznością ponownego uruchomienia maszyny wirtualnej. Oznacza to, że w przypadku obciążeń działających 24/7 zwiększenie pojemności będzie wymagało przerwy w działaniu usługi.

Skończona chmura

Elastyczna natura przechowywania danych w chmurze może prowadzić do przekonania, że chmura ma „nieskończone zasoby”, co nie jest prawdą. W rzeczywistości, podczas pandemii COVID-19, co najmniej jeden z dużych dostawców usług chmurowych doświadczył problemów z wydajnością w niektórych regionach. Było to spowodowane gwałtownym wzrostem wykorzystania chmury obliczeniowej, wynikającym z masowego przejścia na pracę zdalną.

Jeszcze ważniejszą kwestią jest zarządzanie finansami w chmurze. W przeciwieństwie do centrów danych organizacji, które zazwyczaj podlegają modelowi wydatków kapitałowych, w chmurze przechodzimy na model wydatków operacyjnych. Oznacza to, że istnieje rzeczywisty koszt związany z przewymiarowanymi instancjami i prawdopodobnie będziesz regularnie rozmawiać z zespołem ds. zakupów lub zarządzania finansami w chmurze w sprawie optymalizacji (zmniejszenia) obciążeń. Na szczęście możemy aktywnie przeciwdziałać problemom przez analizowanie zaleceń dotyczących rozmiaru instancji dostarczanych przez natywne narzędzia chmurowe, takie jak Azure Advisor i AWS Trusted Advisor.

Są to całkowicie uzasadnione rozważania, które stoją w sprzeczności z tym, co właśnie omówiliśmy. Odpowiedź musi być ustalona przez organizację indywidualnie dla każdego przypadku z uwzględnieniem rezerwacji zasobów w chmurze, planów oszczędnościowych i zniżek, zobowiązań wydatkowych, a ostatecznie kompromisu między kosztami a potencjalnymi krótkimi przestojami.

Może na przykład zapisać decyzja, że system o krytycznym znaczeniu, który wymaga dostępności na poziomie pięciu dziewiątek (o czym będzie mowa w rozdziale 13.), będzie działać w maszynie wirtualnej, która początkowo będzie prze-wymiarowana w oparciu o wyniki planowania wydajności. Jednocześnie można zdecydować, że mniej istotne aplikacje będą hostowane na maszynach wirtualnych o optymalnie dobranych parametrach, i zaakceptować ewentualne przestoje w przypadku konieczności ich rozbudowy.

Podsumowując — zawsze powinniśmy planować wydajność naszych baz danych. W przypadku rozwiązań lokalnych należy patrzeć na to przez pryzmat wydatków inwestycyjnych, natomiast w chmurze powinniśmy rozpatrywać to w kontekście wydatków operacyjnych i uwzględniać zarządzanie kosztami w planowaniu i podejmowaniu decyzji.

9.7. Numer 49 — umieszczanie TempDB i plików dziennika zawsze na osobnych dyskach

Od dawna zaleca się, aby pliki danych, dzienniki zdarzeń i baza TempDB były zawsze rozdzielone na różne woluminy, ponieważ ma to poprawiać wydajność. Czy jednak ta rada jest ciągle aktualna? Wróćmy do przykładu MagicChoc, aby przyjrzeć się temu zagadnieniu nieco dokładniej.

Wszystkie serwery lokalne firmy MagicChoc to maszyny wirtualne, których pamięć masowa jest hostowana w sieci SAN. Zbudowaliśmy nowy serwer do obsługi bazy danych MarketingArchive i skonfigurowaliśmy w nim instancję SQL Servera. Zgodnie z oficjalnymi zaleceniami poprosiliśmy zespół ds. pamięci masowej o udostępnienie trzech woluminów danych i woluminu systemowego. Pliki danych MarketingArchive umieściliśmy na jednym z woluminów danych, plik dziennika MarketingArchive na drugim, a bazę TempDB na trzecim.

Zastanówmy się najpierw nad wydajnością tego rozwiązania. Wszystkie trzy woluminy znajdują się w macierzy SAN. Aby uzyskać dostęp do danych ze wszystkich trzech woluminów, będziemy korzystać z tej samej karty sieciowej, przechodzić przez tę samą ścieżkę sieciową i używać tej samej infrastruktury SAN. Okazuje się też, że wszystkie trzy woluminy zostały utworzone w tej samej macierzy RAID. W takim scenariuszu nie zobaczymy żadnej poprawy wydajności.

Czy istnieją jednak jakieś negatywne konsekwencje? Zazwyczaj odpowiedź jest przecząca, ale spotkałem się z oprogramowaniem SAN, które wymaga, aby pliki danych i dzienników znajdowały się na tym samym woluminie, aby możliwe było wykonanie migawki spójnej z aplikacją. Taka migawka może być bardzo przydatna podczas aktualizacji systemu, ponieważ zapewnia szybki i łatwy punkt przywracania.

Migawki spójne z aplikacją

Migawka to specjalny rodzaj kopii zapasowej, wykonywany na poziomie pamięci masowej, który wykorzystuje technologię kopiowania przy zapisie, aby bardzo szybko utworzyć obraz woluminu w danym punkcie czasu. Takie standardowe migawki nazywane są migawkami spójnymi po awarii. Nie nadają się one jednak do użytku z SQL Serverem, ponieważ jeśli w momencie rozpoczęcia tworzenia migawki trwają jakieś transakcje, to po przywróceniu takiej kopii bazy danych mogą być uszkodzone.

W przeciwieństwie do zwykłych migawek migawka spójna z aplikacją współpracuje z usługą kopiowania woluminów w tle (Volume Shadow Copy Service), aby zapisać zawartość pamięci na dysku przed wykonaniem migawki. Oznacza to, że w odróżnieniu od standardowych migawek może być wykorzystana do odtworzenia baz danych SQL Servera nawet wtedy, gdy w momencie rozpoczęcia tworzenia migawki trwały niezakończone transakcje.

Nie oznacza to oczywiście, że bazy TempDB i pliki dziennika nigdy nie powinny być rozdzielone. Na przykład standardem organizacji może być przechowywanie wszystkich baz danych w macierzy SAN. Jednak w przypadku konkretnej aplikacji baza TempDB może mieć bardzo wysokie wymagania dotyczące przepływności. W takiej sytuacji korzystne może być umieszczenie bazy TempDB na lokalnym dysku M.2.

Podsumowując — jeśli Twoje bazy danych są hostowane w sieci SAN, nie ma korzyści wydajnościowych z rozdzielania plików danych, plików dziennika i bazy TempDB na różne woluminy. Zwykle nie powoduje to problemów, ale w nielicznych przypadkach może uniemożliwić wykonywanie migawek spójnych z aplikacją. Mogą jednak istnieć inne uzasadnione powody, żeby przenieść dzienniki i bazę TempDB na inne woluminy.

9.8. Numer 50 — zaniedbywanie regularnego sprawdzania bazy pod kątem uszkodzeń

Bazy danych mogą ulec uszkodzeniu. To fakt. Uszkodzenie może być spowodowane różnymi czynnikami, takimi jak nagle wyłączenie systemu, wadliwy nośnik danych, a nawet celowe działanie. Uszkodzenie bazy danych to prawdziwy koszmar. Jedyną gorszą rzeczą od tego jest nieświadomość, że do takiego uszkodzenia doszło.

Jest piątkowy wieczór, a dyżurny administrator baz danych firmy MagicChoc właśnie zasypia, gdy nagle dzwoni telefon. To zgłoszenie o najwyższym priorytecie. Podczas korzystania z krytycznej dla firmy bazy danych użytkownik napotkał błąd i nie może uzyskać dostępu do potrzebnych informacji.

To ostatnia rzecz, jakiej potrzebuje administrator bazy danych, gdy szykuje się do weekendowego odpoczynku w piątkowy wieczór. Można było tego łatwo uniknąć, regularnie sprawdzając integralność bazy danych. Zespół administratorów mógłby też znacznie lepiej przysłużyć się firmie, gdyby zauważył problem, zanim odczuli go użytkownicy.

Możemy zasymulować problem, z którym boryka się użytkownik, uruchamiając skrypt z listingu 9.12, aby spowodować uszkodzenie bazy danych Marketing ↪ Archive. Pierwszą czynnością, jaką wykonuje skrypt, jest utworzenie kopii zapasowej bazy MarketingArchive. Ten krok jest kluczowy, ponieważ umożliwi późniejsze przywrócenie bazy z backupu. Następnie skrypt dynamicznie tworzy instrukcję SQL, która uruchamia polecenie DBCC WRITEPAGE zapisujące losową wartość na stronie danych w tabeli ImpressionsArchive. Zanim przejdziemy dalej, należy podkreślić, że DBCC WRITEPAGE jest nieudokumentowaną i *niezwykle niebezpieczną funkcją*, która bezpośrednio zapisuje stronę danych z pominięciem podręcznej pamięci buforów. Jest świetna do testowania scenariuszy awarii, ale powinniśmy jej używać tylko pod warunkiem, że wiemy, co robimy, i że mamy kopię zapasową bazy danych. Absolutnie nie wolno jej stosować w środowisku produkcyjnym.

OSTRZEŻENIE Polecenie DBCC WRITEPAGE jest *niebezpieczne*. Nie używaj go, chyba że masz absolutną pewność co do jego działania i jesteś przygotowany na możliwość utraty danych. Nigdy nie stosuj go w środowisku produkcyjnym.

Listing 9.12. Symulowanie uszkodzenia danych

```
USE master ;
GO

BACKUP DATABASE MarketingArchive
TO DISK = 'D:\Backups\MarketingArchive.bak' ;
GO

ALTER DATABASE MarketingArchive SET SINGLE_USER WITH NO_WAIT ;
GO

DECLARE @SQL NVARCHAR(MAX) ;

SELECT @SQL = 'DBCC WRITEPAGE(' +
(
    SELECT CAST(DB_ID('MarketingArchive') AS NVARCHAR)
) +
', ' +
(
    SELECT TOP 1 CAST(file_id AS NVARCHAR)
    FROM MarketingArchive.dbo.ImpressionsArchive
    CROSS APPLY sys.fn_PhysLocCracker(%%physloc%%)
) +
', ' +
(
    SELECT TOP 1 CAST(page_id AS NVARCHAR)
```

```

FROM MarketingArchive.dbo.ImpressionsArchive
CROSS APPLY sys.fn_PhysLocCracker(%%physloc%%)
) +
', 2000, 1, 0x61, 1)' ;

EXEC(@SQL) ;

ALTER DATABASE MarketingArchive SET MULTI_USER ;
GO

```

Jeśli teraz wykonamy kwerendę `SELECT` na tabeli `ImpressionsArchive`, zobaczymy błąd podobny do poniższego:

```

SQL Server detected a logical consistency-based I/O error: incorrect checksum (expected:
0x968b1542; actual: 0x96845542). It occurred during a read of page (1:71224) in database
ID 11 at offset 0x00000022c70000 in file 'C:\Program Files\Microsoft SQL
Server\MSSQL16.MSSQLSERVER\MSSQL\DATA\MarketingArchive.mdf'.

```

Oczywiście istnieje możliwość, że błąd wystąpił tuż przed dostępem do danych. W takim przypadku nie mielibyśmy szansy go wykryć. Jednak nierzadko uszkodzenie danych pozostaje niezauważone przez dłuższy czas. To znacznie utrudnia dokładne ustalenie, kiedy do niego doszło, a co za tym idzie — utrudnia naprawę błędu.

Aby sprawdzić bazę danych pod kątem uszkodzeń, wystarczy uruchomić polecenie z listingu 9.13. Jeśli zautomatyzujemy ten proces i zaplanujemy jego regularne wykonywanie, możemy skonfigurować nasze narzędzia monitorujące tak, aby alarmowały nas w przypadku wykrycia uszkodzeń. Gdy system znajdzie błędy, polecenie zwróci odpowiedni komunikat.

Listing 9.13. Sprawdzanie uszkodzeń bazy danych

```
DBCC CHECKDB('MarketingArchive') ;
```

Regularne sprawdzanie integralności bazy danych jest niezwykle istotne. Im szybciej wykryjemy uszkodzenie, tym łatwiej będzie je naprawić. Co więcej, lepiej gdy to administrator bazy danych zauważy problem, zanim zrobi to ktoś z działu biznesowego. W przypadku wykrycia uszkodzenia bazy danych najlepszym rozwiązaniem jest przywrócenie konkretnych stron. Takie podejście minimalizuje, a często całkowicie eliminuje utratę danych. Jeśli nie jest to możliwe, można użyć narzędzia `DBCC CHECKDB` do naprawy uszkodzeń. Należy jednak pamiętać, że jeśli problem dotyczy stron z danymi, zastosowanie tego narzędzia spowoduje utratę informacji przechowywanych na tych stronach.

9.9. Numer 51 — zaniedbywanie automatyzacji

Często słyszy się powiedzenie, że „dobrzy programiści są leniwi”. Nie jest to bynajmniej obraźliwe stwierdzenie. Chodzi raczej o to, że dobry programista poświęci dwa razy więcej czasu na napisanie kodu, który będzie można wielokrotnie wykorzystywać, aby uniknąć ciągłego pisania podobnych fragmentów. To samo dotyczy administratorów baz danych. Dobry administrator poświęci czas na automatyzację jak największej liczby zadań, aby uniknąć ręcznego ich wykonywania. W rozdziale 8. omówiliśmy już automatyzację instalacji SQL Servera, a tutaj chciałbym skupić się na automatyzacji konserwacji.

Wyobraźmy sobie, że administrator bazy danych MagicChoc wykonuje większość czynności konserwacyjnych SQL Servera ręcznie. Tworzenie kopii zapasowych jest zautomatyzowane, ponieważ zajmuje się tym firmowe narzędzie do backupu (takie jak Commvault czy NetBackup), ale wszystkie zadania konserwacyjne na poziomie SQL Servera, takie jak przebudowa indeksów, aktualizacja statystyk czy sprawdzanie spójności, są wykonywane przez administratorów w ramach ich codziennych obowiązków. Widziałem niejednego przypadkowego administratora bazy danych, który wpadł w tę pułapkę, a z wielu powodów jest to poważny błąd.

Po pierwsze, wykonywanie nudnych, powtarzalnych zadań jest naprawdę mało satysfakcjonujące. Może to prowadzić do problemów, takich jak zwiększona rotacja pracowników. Po drugie, istnieje koszt alternatywny. Jeśli czynności konserwacyjne zostaną zautomatyzowane, administratorzy baz danych będą mogli poświęcić swój czas na bardziej wartościowe działania, takie jak diagnozowanie złożonych problemów wydajnościowych. Po trzecie, często po prostu nie dochodzi do wykonania tych zadań. Pojawia się pilne zadanie operacyjne i administrator odkłada rutynową konserwację, aby się nim zająć. Wreszcie, może to prowadzić do błędów ludzkich spowodowanych wykonywaniem powtarzalnych czynności „na autopilocie”.

Jeszcze częstszym błędem — tak powszechnym, że występuje prawdopodobnie w większości środowisk, z jakimi się zetknąłem — jest zdecentralizowana automatyzacja. Aby zrozumieć to podejście, wyobraźmy sobie, że administrator bazy danych MagicChoc ma dość codziennego ręcznego wykonywania czynności konserwacyjnych. Postanawia więc wprowadzić pewną automatyzację. Przechodzi przez wszystkie instancje SQL Servera i starannie tworzy zadania w SQL Server Agent, które wykonują rutynowe czynności konserwacyjne na każdym serwerze. Dlaczego jest to pomyłka?

Otóż dlatego, że obecnie każdego ranka nasz administrator bazy danych musi przeglądać wszystkie instancje SQL Servera i sprawdzać historię zadań agenta

SQL Server dla każdego utworzonego zadania konserwacyjnego. Choć jest to nieco szybsze niż faktyczne wykonywanie konserwacji, wiąże się z tymi samymi problemami. Konkretnie, polega to na monotonnej, czasochłonnej, ręcznej pracy, która niekiedy jest pomijana z powodu pilnych spraw operacyjnych.

Jak to usprawnić? Rozwiązaniem jest centralizacja zarządzania. Jeśli infrastruktura SQL Servera jest lokalna, może to oznaczać konieczność utworzenia centralnego serwera zarządzającego, który będzie cyklicznie łączył się z każdym egzemplarzem SQL Servera w firmie i wykonywał na nim zadania konserwacyjne.

Konfiguracja ta działa najlepiej, gdy w skrypcie lub manifeście DSC używanym do instalacji nowych instancji SQL Servera uwzględnimy krok rejestracji. Wiąże się to jednak z pewnymi komplikacjami. Kiedy wdrażałem to rozwiązanie w przeszłości, musiałem zbudować prosty mechanizm harmonogramowania, wykorzystując tabele, widoki i procedury składowane, aby upewnić się, że odpowiednie zadania zostały wykonane na właściwych instancjach zgodnie z ustalonym planem.

Jeśli infrastruktura SQL Servera znajduje się w chmurze, istnieją znacznie bardziej eleganckie sposoby centralizacji zadań konserwacyjnych. Jeśli na przykład serwery są hostowane w Azure, możemy wykorzystać Azure Runbooks, Azure Functions lub Logic Apps do konserwowania maszyn wirtualnych z SQL Serverem. W przypadku korzystania z usługi Azure SQL Database dobrym rozwiązaniem do konfiguracji zadań konserwacyjnych byłoby użycie zadań elastycznych (ang. *elastic jobs*).

Niezależnie od wybranego narzędzia kluczowe jest posiadanie scentralizowanego, zautomatyzowanego systemu konserwacji SQL Servera. Takie podejście zmniejsza nakład pracy ręcznej i pozwala uniknąć wielu problemów. Wśród tych problemów można wymienić dużą rotację personelu, pomijanie zadań konserwacyjnych oraz straty wynikające z niemożności poświęcenia czasu na bardziej wartościowe zadania.

9.10. Numer 52 — używanie kursorów do celów administracyjnych

W rozdziale 5. omówiłem kursory z perspektywy programistycznej i wyjaśniłem, dlaczego należy ich unikać. Zaobserwowałem, że wielu administratorów baz danych słusznie egzekwuje zasadę nieużywania kursorów, ale sami wykorzystują je w skryptach administracyjnych. Zazwyczaj usprawiedliwiają to tym, że skrypt nie iteruje po wielu obiektach, więc nie ma to znaczenia.

Wyobraź sobie, że zespół DBA w firmie MagicChoc wprowadza standard zabraniający używania kursorów w ich środowisku. Mają do tego pełne prawo, ponieważ to oni muszą rozwiązywać problemy z wydajnością w środowisku produkcyjnym — czasem o nietypowych porach. Decydują się jednak na wykorzystanie kursorów we własnych skryptach administracyjnych.

Jeden z programistów zauważa, że administratorzy baz danych używają kursorów, i wyraża swoje zaniepokojenie. Jest nieco poirytowany i chce wiedzieć, dlaczego administratorzy mogą korzystać z kursorów, podczas gdy jego zespołowi nie wolno. Administratorzy wyjaśniają programiście, że ich skrypty są inne, ponieważ przetwarzają tylko kilka obiektów, więc wpływ na wydajność jest niezauważalny.

Programista kwestionuje to podejście, słusznie argumentując, że niektóre z jego kwerend muszą przetworzyć zaledwie kilka wierszy, więc co za różnica? W rezultacie administratorzy niechętnie decydują się na przyznanie wyjątku, zezwalając programiście na użycie kursorów pod warunkiem, że nie będą one iterować po zbyt dużej liczbie wierszy.

Rozpoczyna to efekt kuli śnieżnej i wyjątki pojawiają się jeden za drugim. Egzekwowanie zasady braku kursorów staje się niemożliwe. MagicChoc ostatecznie używa wielu kursorów przetwarzających zmienne ilości wierszy. Co więcej, aplikacja z czasem rozrasta się organicznie, a niewielka liczba wierszy przeradza się w znaczną ilość danych.

Wszystkich tych problemów można było uniknąć, gdyby administratorzy baz danych po prostu przestrzegali własnych standardów kodowania. Jeśli zna się alternatywne techniki, nie ma powodu, by używać kursorów. Na przykład kwerenda z listingu 9.14 pokazuje, jak administrator bazy danych mógłby użyć kursora do przeglądania każdego indeksu w bazie danych i jego przebudowy.

WSKAZÓWKA Administratorzy baz danych często stosują logikę warunkową przy przebudowie indeksów. Przykłady w tym rozdziale po prostu przebudowują wszystkie indeksy. O warunkowej przebudowie indeksów porozmawiamy szerzej w rozdziale 11.

Listing 9.14. Używanie kursora do przebudowy wszystkich indeksów

```
DECLARE @Command NVARCHAR(MAX) ;
DECLARE @Table NVARCHAR(256) ;

DECLARE Tables CURSOR READ_ONLY FOR
SELECT name
FROM sys.tables ;

OPEN Tables ;

FETCH NEXT FROM Tables INTO @Table
WHILE @@FETCH_STATUS = 0
BEGIN
    SET @Command = 'ALTER INDEX ALL ON ' + QUOTENAME(@Table) + ' REBUILD ; '
```

```

;
EXEC(@Command) ;

FETCH NEXT FROM Tables INTO @Table ;
END

CLOSE Tables ;
DEALLOCATE Tables ;

```

UWAGA Warto zaznaczyć, że wcześniej w tym rozdziale używaliśmy procedury składowanej `sp_msforeachdb` do iterowania po tabelach w celu przebudowy indeksów. Procedura ta również wykorzystuje kursor.

Jak zatem zmienić ten skrypt, aby wykorzystywał bardziej wydajną metodę unikającą kursorów? Rozwiązaniem jest zastosowanie sztuczki z typem danych XML, jak pokazano na listingu 9.15. Podkwerenda generuje listę poleceń, które chcemy wykonać, a klauzula `FOR XML` przekształca tę listę w format XML. Ponieważ używamy trybu `PATH` bez wyrażeń `XPath`, powstaje pojedynczy, ciągły węzeł bez znaczników. Jako że kwerenda zewnętrzna oczekuje typu `NVARCHAR`, XML jest niejawnie konwertowany z powrotem na ten typ danych, dzięki czemu może zostać wykonany jako dynamiczny kod SQL.

Listing 9.15. Zastępowanie kursora bardziej efektywną metodą

```

DECLARE @SQL NVARCHAR(MAX) ;

SET @SQL = (
    SELECT 'ALTER INDEX ALL ON ' + QUOTENAME(name) + ' REBUILD ; '
    FROM sys.tables
    FOR XML PATH('')
) ;

EXEC(@SQL) ;

```

Ta kwerenda nie tylko jest bardziej wydajna, ale również skraca kod o ponad połowę. Jednak w przypadku małych zbiorów danych jej główną korzyścią jest to, że daje dobry przykład zespołowi programistów.

Nie ma żadnego technicznie uzasadnionego powodu, żeby używać kursorów. Zawsze istnieje lepsze rozwiązanie. Chociaż administratorzy baz danych czasami stosują kursory w skryptach administracyjnych, stanowi to zły przykład dla programistów i może prowadzić do narastających problemów organizacyjnych. Zamiast tego administratorzy powinni korzystać z bardziej efektywnych technik, takich jak metoda XML przedstawiona w tym rozdziale.

9.11. Numer 53 — nieinstalowanie poprawek

Instalowanie poprawek to prawdziwa udręka. Każdy administrator baz danych to potwierdzi i nie zamierzam z tym polemizować. Zawsze znajdzie się jakiś system działający całodobowo, którego firma nie chce wyłączać, albo aplikacja hurtowni danych z trwającym 12 godzin procesem ETL, który musi się zakończyć przed rozpoczęciem dnia roboczego.

Prowadzi to do błędu popełnianego przez wielu administratorów baz danych, zespoły programistyczne oraz osoby odpowiedzialne za aplikacje, polegającego na unikaniu aktualizacji „złotych” systemów. Jakie są konsekwencje tego błędu? Przyjrzyjmy się temu z perspektywy administratora bazy danych w firmie MagicChoc.

MagicChoc ma dwa systemy bazodanowe, których nie aktualizuje. Firma nie instaluje żadnych poprawek — ani systemu operacyjnego, ani oprogramowania pośredniego, ani SQL Servera. Jeden z serwerów obsługuje bazę danych sprzedaży online, która musi być dostępna przez całą dobę. Drugi to hurtownia danych marketingowych. System ten przetwarza ogromne ilości danych, a procesy ETL działają przez całą noc i muszą być gotowe przed rozpoczęciem kolejnego dnia roboczego.

Jest trzecia w nocy, gdy kierownik ds. dostarczania usług budzi dyżurnego administratora bazy danych. Pojawił się poważny problem — nikt w całej firmie nie ma dostępu do żadnej bazy danych. Napastnik wykorzystał niezaktualizowane serwery i uzyskał możliwość przemieszczania się po sieci, co pozwoliło mu przeprowadzić atak typu ransomware w całej infrastrukturze firmy.

Piszę ten tekst po regularnej wtorkowej aktualizacji z listopada 2023 roku. W tym miesiącu sama zbiorcza aktualizacja systemu Windows Server naprawiła ponad 50 luk w zabezpieczeniach. Dlatego rezygnacja z comiesięcznego instalowania poprawek stanowi ogromne zagrożenie dla bezpieczeństwa i naraża organizację na wszelkiego rodzaju cyberprzestępstwa.

Zaniedbanie instalowania poprawek naraża nasze organizacje nie tylko na problemy związane z bezpieczeństwem, ale także na inne rodzaje ryzyka. Producenci oprogramowania, tacy jak Microsoft, wykorzystują aktualizacje zbiorcze do naprawiania błędów i poprawy stabilności systemów. Dlatego jeśli nie aktualizujemy naszych systemów, zwiększamy ryzyko wystąpienia awarii, których można by uniknąć. Czasami poprzez aktualizacje zbiorcze udostępniane są również drobne nowe funkcje, z których nie będziemy mogli skorzystać, jeśli nie zainstalujemy poprawek.

Jak zatem możemy zagwarantować, że wszystkie nasze serwery będą aktualizowane, jednocześnie spełniając wymagania biznesowe? Przyjrzyjmy się każdemu

ze scenariuszy po kolei. Na początek omówimy wymagania dla systemu działającego przez całą dobę, którego nie można wyłączyć na czas aktualizacji.

Jeśli baza danych wymaga dostępności przez całą dobę, musi być oparta na infrastrukturze o wysokiej niezawodności. Oznacza to wykorzystanie technologii takich jak *grupy dostępności AlwaysOn*, które synchronizują repliki baz danych na wielu serwerach za pomocą strumieni dzienników. Można je stosować do przywracania po awarii lub zapewnienia wysokiej dostępności. W przypadku wysokiej dostępności technologia ta umożliwia automatyczne przełączenie awaryjne baz danych w razie awarii serwera głównego. W przypadku przywracania po awarii przełączenie wymaga ręcznej interwencji. Technologia ta może również pomóc w rozwiązywaniu problemu z aktualizacjami systemu.

W tym scenariuszu możemy zaktualizować węzły wtórne w klastrze. Po ich aktualizacji możemy przeprowadzić kontrolowane przełączenie awaryjne z repliki głównej na replikę pomocniczą, zanim zaktualizujemy serwer, który pierwotnie hostował bazy danych. Technicznie rzecz biorąc, powoduje to przerwę w działaniu, ale trwa ona zaledwie kilka sekund, a nie potencjalnie kilka godzin w przypadku standardowego okna aktualizacji. Z perspektywy użytkownika końcowego przerwa jest tak krótka, że zwykle niezauważalna.

Drugi przypadek dotyczy hurtowni danych, w której proces ETL trwa całą noc i musi zostać ukończony do rana. Hurtownia danych może nie być odpowiednia dla grup dostępności, o czym będziemy mówić szerzej w rozdziale 13., ale może być hostowana w klastrze. *Instancja klastrowa z przełączaniem awaryjnym* (ang. *failover clustered instance*, FCI) przechowuje bazy danych na współdzielonym zasobie dyskowym, dostępnym dla wielu węzłów klastra. Oznacza to, że w przypadku awarii jednego węzła inny może automatycznie przejąć kontrolę nad bazą danych, zapewniając automatyczne przełączenie awaryjne. Jednak nawet przy zastosowaniu FCI krótka przerwa w działaniu, trwająca około 30 sekund (tyle wynosi średni czas potrzebny na przełączenie FCI), może spowodować poważne problemy z procesem ETL.

Rozwiązanie problemu instalowania poprawek sprowadza się zatem do kwestii harmonogramu. W przypadku hurtowni danych może być dopuszczalna 30-sekundowa przerwa w działaniu systemu w ciągu dnia roboczego. Jeśli tak jest, można zastosować podobną metodę aktualizacji etapowych, jaką opisaliśmy dla grup dostępności. Różnica polega na tym, że przerwa w działaniu byłaby zaplanowana na czas pracy, a nie na okres najmniejszej aktywności.

Jeśli ta opcja nie wchodzi w grę, to warto mieć na uwadze, że często podczas weekendów są okresy, kiedy systemy takie jak hurtownie danych są nieużywane albo wykorzystywane przez znacznie mniejszą liczbę użytkowników. W takim przypadku najprościej jest zaplanować przerwę w działaniu systemu na weekend.

Podsumowując — aktualizowanie serwerów jest bardzo istotne, co dotyczy nawet „złotych” systemów o krytycznym znaczeniu dla firmy. Ryzyko związane z brakiem aktualizacji znacznie przewyższa niedogodności wynikające z regularnych, comiesięcznych cykli instalowania poprawek.

Podsumowanie

- Zmniejszenie rozmiaru bazy danych spowoduje niemal 100-procentową fragmentację indeksów.
- Nigdy nie używaj funkcji automatycznego zmniejszania baz danych, ponieważ prowadzi to do niekończącego się cyklu spadku wydajności i zwiększonego zużycia zasobów.
- Jeśli musisz ręcznie zmniejszyć bazę danych, pamiętaj o konieczności przebudowania wszystkich indeksów po zakończeniu tej operacji.
- Nigdy nie stosuj funkcji automatycznego powiększania jako sposobu na zarządzanie rozmiarem bazy danych, ponieważ może to prowadzić do problemów z wydajnością.
- Nie używaj wielu plików dziennika transakcji, ponieważ nie przynosi to żadnych korzyści. Jeśli istnieje kilka plików dziennika transakcji, pierwszy plik musi zostać zapełniony, zanim serwer zacznie używać drugiego.
- Każdej transakcji zapisanej w dzienniku transakcji przypisywany jest numer LSN. LSN identyfikuje transakcję i zawiera identyfikatory pliku VLF, bloku dziennika oraz samej transakcji.
- Pliki dziennika transakcji zawierają wiele plików VLF. Dodatkowe pliki VLF są tworzone automatycznie, gdy plik dziennika rośnie.
- Zbyt duża liczba plików VLF, znana jako fragmentacja dziennika, może prowadzić do problemów z wydajnością.
- Fragmentacja wewnętrzna indeksu odnosi się do stopnia zapełnienia poszczególnych stron indeksu.
- Fragmentacja zewnętrzna indeksu odnosi się do tego, ile stron indeksu jest ułożonych w niewłaściwej kolejności.
- Aby usunąć fragmentację dziennika, musimy zmniejszyć dziennik transakcji, a następnie pozwolić mu rosnąć w większych przyrostach.
- Metadane indeksu można znaleźć w widokach zarządzania dynamicznego (DMV) i funkcjach zarządzania dynamicznego (DMF).
- Przy analizie fragmentacji indeksów szczególnie przydatny jest widok DMF o nazwie `sys.dm_db_index_physical_stats`. Widok ten może zwrócić jeden wiersz dla każdego poziomu każdego indeksu w bazie danych.
- Brak odpowiedniego planowania wydajności może prowadzić do problemów biznesowych, takich jak nieplanowane wydatki. Może również skutkować przestojami, jeśli czas oczekiwania na nowy sprzęt jest długi.

- Podczas planowania wydajności staraj się identyfikować wzorce wzrostu, takie jak wzrost liniowy lub wykładniczy.
- Nie zawsze konieczne jest umieszczanie plików bazy danych, plików dziennika transakcji i bazy TempDB na osobnych woluminach.
- Jeśli wszystkie dane są przechowywane w macierzy SAN, to często zdarza się, że wiele woluminów znajduje się w tej samej fizycznej macierzy RAID.
- Należy regularnie sprawdzać, czy baza danych nie uległa uszkodzeniu. Może to być spowodowane różnymi czynnikami, takimi jak błędy dysku.
- Wczesne wykrycie i naprawa uszkodzeń bazy danych są łatwiejsze i wiążą się z mniejszym ryzykiem utraty danych niż w przypadku, gdy problem pozostaje niezauważony przez dłuższy czas.
- Staraj się automatyzować wszystko, co tylko możliwe. Automatyzacja zmniejsza nakład pracy ręcznej i ogranicza ryzyko błędów ludzkich. Pozwala też administratorom baz danych skupić się na bardziej wartościowych zadaniach, takich jak analizowanie problemów z wydajnością.
- Administratorzy baz danych nie powinni używać kursorów. Zamiast tego powinni dawać dobry przykład i pokazywać zespołom programistycznym, że zawsze istnieje bardziej efektywny sposób wykonania zadania.
- Nawet „złote” systemy wymagają aktualizacji. Nie pozwalaj, aby jakiegokolwiek serwery były wyłączone z regularnego procesu instalowania poprawek.
- Możesz wykorzystać technologie takie jak grupy dostępności AlwaysOn lub instancje klastrowe z przełączaniem awaryjnym (FCI), aby skrócić przestoje związane z aktualizacjami.
- Wiele wyzwań związanych z wprowadzaniem poprawek można rozwiązać poprzez kreatywne harmonogramowanie.

10

Optymalizacja

W tym rozdziale:

- Optymalizacja na poziomie instancji
- Optymalizacja kwerend
- Optymalizacja tabel
- Poziomy izolacji transakcji
- Rozwiązywanie problemów z wydajnością przez dodawanie sprzętu

W tym rozdziale przeanalizujemy pomyłki i nieporozumienia związane z optymalizacją wydajności SQL Servera. Zaczniemy od optymalizacji na poziomie instancji, poruszając takie zagadnienia, jak przestarzałe flagi śledzenia, błędne przekonania dotyczące natychmiastowej inicjalizacji plików oraz błędy w konfiguracji pamięci. Pomyłki mogą przybierać dwie główne formy: brak optymalizacji lub nieprawidłowa optymalizacja.

Brak optymalizacji

Niedawno byłem świadkiem braku optymalizacji. Poproszono mnie o przeanalizowanie procesu ETL, który zajmował dużo czasu w związku z wprowadzaniem informacji do hurtowni danych. Po dokładnym zbadaniu procesu stało się jasne, że zespół nie przeprowadził planowania wydajności (patrz rozdział 9.). Choć na dysku było jeszcze wolne miejsce, nie dostosowano rozmiaru plików do dużej ilości informacji wprowadzanych do bazy każdej nocy. W związku z tym plik danych musiał być wielokrotnie powiększany w ciągu nocy, za każdym razem o niewielki przyrost. Co gorsza, nie włączono funkcji natychmiastowej inicjalizacji plików, co oznaczało, że przy każdym powiększeniu nowa przestrzeń musiała być zerowana — problem ten zostanie omówiony jako druga pomyłka w tym rozdziale. Włączenie natychmiastowej inicjalizacji plików od razu rozwiązało problem, ale poleciłem też zespołowi, by odpowiednio zaplanował wydajność i zmienił rozmiar dysku.

Następnie przejdziemy do błędów popełnianych przy optymalizacji kwerend. Pierwszym z nich jest utrudnianie zamiast ułatwiania pracy optymalizatorowi. Omówimy też informacje zwrotne dotyczące przetwarzania kwerend, które to informacje są często pomijane przez administratorów baz danych.

Później przejdziemy do omówienia optymalizacji tabel. Skupimy się na częstej pomyłce, jaką jest pomijanie użytecznych technik poprawy wydajności, takich jak partycjonowanie i kompresja tabel.

W następnej kolejności przyjrzymy się poziomom izolacji transakcji i omówimy typowe błędy i nieporozumienia. Na koniec opowiem, co się dzieje, gdy próbuje się rozwiązać problemy z wydajnością przez rozbudowę sprzętu.

Aby omówić błędy w tym rozdziale, wykorzystamy bazę danych Marketing, którą utworzyliśmy w rozdziale 6., oraz bazę danych MarketingArchive, którą utworzyliśmy w rozdziale 9. Jeśli wykonywałeś przykłady z rozdziału 8. i nie naprawiłeś jeszcze bazy MarketingArchive, na początek uruchom skrypt z listingu 10.1.

OSTRZEŻENIE W normalnych okolicznościach nie powinieneś używać opcji REPAIR_↪ALLOW_DATA_LOSS, chyba że nie masz innych możliwości odzyskania danych, takich jak przywrócenie z kopii zapasowej. Użycie tej opcji może prowadzić do utraty danych.

Listing 10.1. Naprawianie bazy danych MarketingArchive

```
USE master ;
GO

ALTER DATABASE MarketingArchive SET SINGLE_USER ;
GO

DBCC CHECKDB (MarketingArchive, REPAIR_ALLOW_DATA_LOSS) ;
GO

ALTER DATABASE MarketingArchive SET MULTI_USER ;
GO
```

W przykładach, które wprowadzają zmiany w instancji, możesz skorzystać z dowolnej instancji, ale dobrym wyborem będzie ta, w której znajduje się baza danych MarketingArchive.

10.1. Numer 54 – włączanie flag TF1117 i TF1118

Flagi śledzenia to opcje, które pozwalają administratorom na zmianę określonych konfiguracji. Każda flaga ma trzy- lub czterocyfrowy numer. Choć wiele flag śledzenia jest nieudokumentowanych albo dotyczy starszych wersji SQL Servera, niektóre są bardzo przydatne. Na przykład flaga śledzenia 3226 pozwala wyciszyć komunikaty o udanych kopiach zapasowych. Zmniejsza to ilość „szumu” w dziennikach SQL Servera.

W zależności od charakteru opcji śledzenia możliwe jest zastosowanie konfiguracji lokalnie dla danej sesji lub globalnie dla wszystkich sesji. Na przykład wspomnianą wcześniej opcję 3226 można włączyć tylko globalnie.

Flagi śledzenia można włączać i wyłączać poleceniami DBCC TRACEON i DBCC TRACEOFF. W celu skonfigurowania sesji w poleceniu należy podać numer flagi śledzenia. W przypadku konfiguracji globalnej przekazuje się drugi parametr, o wartości -1. Poniższy skrypt pokazuje, jak włączyć flagę śledzenia 1224 w ramach sesji, a następnie globalnie. Flaga ta wyłącza eskalację blokad opartą na liczbie blokad:

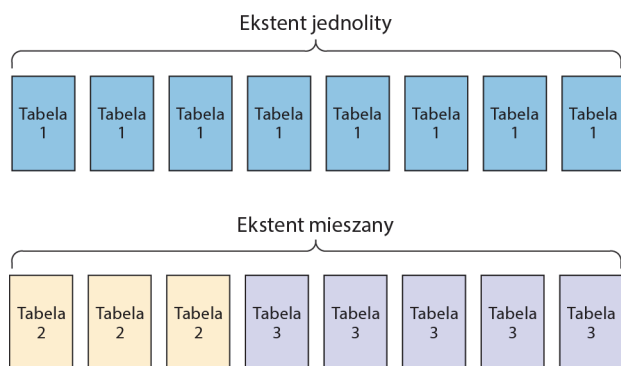
```
DBCC TRACEON (1224) ;  
DBCC TRACEON (1224,-1) ;
```

Problem z globalnymi opcjami śledzenia polega na tym, że nie są one zachowywane po ponownym uruchomieniu usługi Database Engine. Dlatego jeśli chcemy, aby dana konfiguracja pozostała aktywna po restarcie, musimy dodać ją jako parametr uruchomieniowy usługi SQL Server.

Jakiś czas temu powszechne było stosowanie dwóch flag śledzenia w celu poprawy wydajności aplikacji w stylu hurtowni danych, a w niektórych przypadkach także bazy TempDB w środowiskach OLTP. W szczególności flaga T1117 (skrót od ang. *trace flag 1117*) była używana do powiększania wszystkich plików w grupie plików, kiedy którykolwiek z nich osiągnął próg automatycznego wzrostu. SQL Server wykorzystuje algorytm proporcjonalnego wypełniania do rozdzielania danych między pliki w grupie plików. Dlatego powiększenie wszystkich plików jednocześnie zapobiega „faworyzowaniu” większego pliku przez SQL Server, co mogłoby uniemożliwić pełne wykorzystanie zalet równoległego odczytu danych z wielu plików.

Flagi T1118 używano do wymuszania jednolitych ekstentów. Oznacza to, że wiele obiektów nie może przydzielać stron w tym samym ekstencie. Domyślnie dla małych obiektów, mniejszych niż 64 KB, możliwe były ekstenty mieszane. Wymuszenie jednolitych ekstentów pomaga zapobiec konfliktom na stronach systemowych, takich jak strony globalnej mapy alokacji (ang. *Global Allocation Map*, GAM), współdzielonej globalnej mapy alokacji (ang. *Shared Global Allocation Map*, SGAM) i wolnej przestrzeni stron (ang. *Page Free Space*, PFS), które znajdują się w każdym pliku danych. Różnice między obszarami jednolitymi a mieszanymi zostały zilustrowane na rysunku 10.1.

Zarówno flaga T1117, jak i T1118 zostały uznane za przestarzałe w SQL Serverze 2016, a ich funkcje przeniesiono na niższe poziomy, co umożliwia bardziej precyzyjną konfigurację. Jest to znacznie lepsze rozwiązanie, ponieważ ustawienia można zastosować do konkretnych baz danych, które mogą na tym skorzystać, w przeciwieństwie do flag śledzenia, które stosowały konfigurację do wszystkich baz danych.



Rysunek 10.1. Ekstenty jednolite i mieszane

Wciąż spotykam się z sytuacjami, w których administratorzy ustawiają opcje śledzenia T1117 i T1118 dla usługi Database Engine; są one również ustawione w obrazie Azure SQL Server. Jest to jednak pomyłka, ponieważ od wersji SQL Server 2016 te opcje śledzenia nie mają już żadnego wpływu na działanie systemu. Administratorzy nieświadomi tej zmiany sądzą, że skonfigurowali te ustawienia, podczas gdy w rzeczywistości tego nie zrobili.

Na szczęście równomierne powiększanie plików jest teraz domyślnym zachowaniem dla bazy TempDB, choć nadal trzeba skonfigurować je dla hurtowni danych. Ekstenty jednolite są teraz domyślnym ustawieniem zarówno dla TempDB, jak i baz użytkownika. To jednak tylko odwraca problem. Administrator bazy danych, który nie jest świadomy tych zmian, a chciałby używać ekstentów mieszanych i (lub) równomiernego powiększania plików, nie podejmie żadnych działań, nie wiedząc, że niepożądane działanie jest teraz opcją domyślną.

Co zatem powinniśmy zrobić, zamiast skonfigurować te dwie flagi śledzenia, jeśli chcemy zmienić działanie konkretnych baz danych? Skrypt przedstawiony na listingu 10.2 włącza równomierne powiększanie plików i wyłącza ekstenty jednolite dla bazy danych MarketingArchive. Pierwsze polecenie w skrypcie używa instrukcji ALTER DATABASE...MODIFY FILEGROUP, aby włączyć opcję AUTOGROW_ALL_FILES, która nadpisuje domyślne ustawienie AUTOGROW_SINGLE_FILE. W drugim poleceniu używamy opcji ALTER DATABASE SET, aby włączyć ekstenty mieszane.

Listing 10.2. Włączanie równomiernego powiększania plików i ekstentów jednolitych

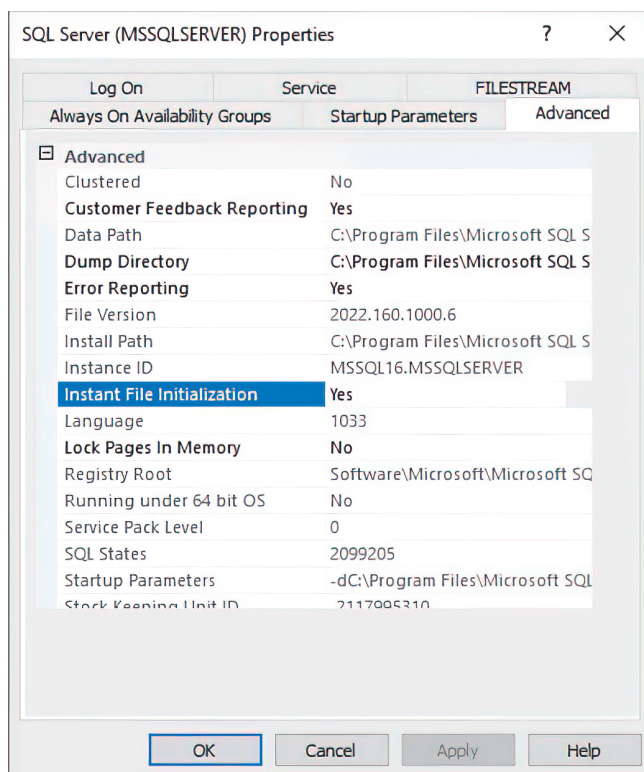
```
ALTER DATABASE MarketingArchive
    MODIFY FILEGROUP [PRIMARY] AUTOGROW_ALL_FILES ;

ALTER DATABASE MarketingArchive
    SET MIXED_PAGE_ALLOCATION ON ;
```

WSKAZÓWKA Polecenia te można również wykonać za pomocą DSC (patrz rozdział 8.).

10.2. Numer 55 – niestosowanie natychmiastowej inicjalizacji plików

Gdy SQL Server tworzy nowy plik lub powiększa istniejący, musi go wyzerować, co oznacza wypełnienie pustej przestrzeni zerami. Aby zoptymalizować wydajność, możliwe jest pominięcie tego procesu dla plików danych, a czasem także dla plików dziennika transakcji. Można to osiągnąć poprzez natychmiastowe inicjowanie plików. Aby to zrobić, wystarczy nadać kontu usługowemu, na którym działa usługa Database Engine, uprawnienie „wykonuj zadania konserwacji woluminów” (SeManageVolumePrivilege). SQL Server będzie wtedy automatycznie pomijać zerowanie tworzonych lub powiększanych plików. W nowszych wersjach SQL Servera Microsoft ułatwił to zadanie, dodając tę opcję do instalatora. Odpowiednia opcja została również dodana do karty *Advanced* usługi Database Engine w narzędziu SQL Server Configuration Manager, jak pokazano na rysunku 10.2.



Rysunek 10.2. Konfigurowanie natychmiastowej inicjalizacji plików

Wydaje się więc, że włączenie tej funkcji to oczywisty wybór, prawda? Niezupełnie. Pomyłką, którą popełniają niektórzy administratorzy baz danych, jest właśnie decyzja o zrezygnowaniu z tego rozwiązania. Aby zrozumieć, czemu tak się dzieje, musimy się zastanowić nad kwestiami bezpieczeństwa.

Wyobraź sobie, że mamy bazę danych zawierającą bardzo wrażliwe informacje. Baza ta zostaje przeniesiona na nowy serwer, a następnie usunięta z oryginalnego serwera. Oryginalny serwer jest później wykorzystywany do hostowania nowej bazy danych.

Gdy dane są usuwane z dysku, fizyczna zawartość nie jest kasowana — usuwane są jedynie wskaźniki do tych danych. Kiedy nowe pliki są tworzone w tym samym miejscu na dysku, obszar ten jest zerowany, co powoduje nadpisanie oryginalnych danych. W przypadku zastosowania natychmiastowego inicjowania plików dane nie są nadpisywane, dopóki SQL Server nie przydzieli ekstentów w tym samym fizycznym miejscu i nie zapisze w nich stron. Istnieje więc teoretyczne ryzyko, że osoba o złych zamiarach mogłaby odzyskać oryginalne dane.

Dlaczego sugeruję, że nieużywanie natychmiastowej inicjalizacji plików to błąd, skoro wiąże się to z potencjalnym zagrożeniem? Odpowiedź jest prosta — korzyści płynące z tej funkcji znacznie przewyższają związane z nią niewielkie ryzyko.

Aby zagrożenie się urzeczywistniło, muszą zostać spełnione wszystkie poniższe kryteria:

- Na dyskach znajdowały się wcześniej wrażliwe dane.
- Napastnik uzyskał podwyższone uprawnienia dostępu do serwera.
- Napastnik posiada specjalistyczne oprogramowanie i umiejętności niezbędne do odnalezienia i wydobywania danych.
- Napastnik ma czas na znalezienie i przechwycenie danych, zanim
 - administrator wykryje atak;
 - dane zostaną nadpisane przez nowe informacje.

Jeśli więc nie przechowujesz kodów do broni jądrowej lub prawdy o tym, co w istocie wydarzyło się w Roswell (i po wszystkim nie zdecydowałeś się zniszczyć dysków), w kontekście zysków wydajnościowych ryzyko jest prawdopodobnie zbyt małe, by się nim przejmować.

10.3. Numer 56 — niepozostawianie wystarczającej ilości pamięci dla innych aplikacji

Wyobrażam sobie, że wielu czytelników mogło się zdziwić, widząc tytuł tego podręcznika. Chyba powinienem napisać, że pomyłką jest umieszczanie innych aplikacji na tym samym serwerze co instancja SQL Servera, zamiast omawiać przydzielanie pamięci dla innych programów?

Owszem, instalowanie innych aplikacji na tym samym serwerze co SQL Server to bardzo zła praktyka z wielu powodów, takich jak względy bezpieczeństwa, mieszanie różnych profili obciążenia oraz utrudnione rozwiązywanie problemów. Dlatego nigdy bym tego nie zalecał. Warto jednak pamiętać, że inne funkcje wchodzące w skład pakietu SQL Server, takie jak SQL Server Integration Services (SSIS) czy SQL Server Analysis Services (SSAS), zasadniczo również są odrębnymi aplikacjami. Mają własne usługi, działają jako oddzielne procesy i mają własne przydziały pamięci. W związku z tym, jeśli korzystamy z którejkolwiek z tych funkcji, powinniśmy rozważyć ich wymagania pamięciowe niezależnie od silnika bazy danych.

Oprócz tego powinniśmy również wziąć pod uwagę programy, które mogą być używane przez inne zespoły, na przykład oprogramowanie antywirusowe, narzędzia do monitorowania czy inwentaryzacji zasobów, by wymienić tylko kilka przykładów. Choć nie są one bezpośrednio związane z działalnością biznesową, stanowią bardzo ważne narzędzia dla całej organizacji i często ich instalacja jest wymagana na wszystkich serwerach. Oczywiście te narzędzia i programy powinny być lekkie, ale niejednokrotnie zauważyłem, że niektóre z nich zużywają znacznie więcej pamięci, niż można by się spodziewać.

Należy również wziąć pod uwagę ilość pamięci, którą pozostawiamy do dyspozycji systemu operacyjnego. Microsoft zaleca, aby dla systemu Windows pozostawić 25% dostępnej pamięci. Te 25% jest również wykorzystywane przez niektóre komponenty SQL Servera, takie jak rozszerzone procedury składowane i pliki wykonywalne.

Domyślnie, po zainstalowaniu SQL Servera, instancja będzie alokować tyle pamięci, ile potrzebuje na *pulę buforów*, czyli obszar pamięci, w którym przechowywane są strony danych i indeksów. Częstą pomyłką jest pozostawienie tej domyślnej konfiguracji przez administratorów z założeniem, że SQL Server jest jedyną aplikacją na serwerze i powinien wykorzystywać tyle pamięci, ile zechce. Takie podejście nie uwzględnia jednak wcześniej omówionych wymagań. Błąd ten jest jeszcze poważniejszy, jeśli włączono opcję *Lock Pages In Memory*,

ponieważ uniemożliwia to zmniejszanie zestawu roboczego, co zostanie omówione w następnym podrozdziale.

W związku z tym maksymalną ilość pamięci, którą może wykorzystać pula buforów, należy skonfigurować według następującego wzoru:

$$\text{Maksymalna pamięć} = (\text{Pamięć serwera} - \text{Pamięć wymagana przez inne procesy}) : 100) \cdot 75$$

Maksymalną ilość pamięci możemy skonfigurować za pomocą procedury składowanej `sp_configure`. Skrypt z listingu 10.3 ustawia maksymalną pamięć serwera na 48 GB, co byłoby odpowiednie dla pojedynczej instancji SQL Servera będącej jedyną aplikacją uruchomioną na serwerze z 64 GB pamięci RAM. Oczywiście, jeśli na serwerze działałoby kilka instancji, należałoby odpowiednio podzielić te 48 GB między nie, w zależności od ich wymagań.

WSKAZÓWKA Wartość na listingu 10.3 jest podana w MB.

Listing 10.3. Konfiguracja maksymalnej pamięci

```
sp_configure 'show advanced options', 1 ;  
RECONFIGURE ;  
GO
```

```
sp_configure 'max server memory', 49152 ;  
RECONFIGURE ;  
GO
```

Maksymalną ilość pamięci dla SQL Servera zawsze należy konfigurować tak, aby zostawić wystarczająco dużo pamięci dla systemu operacyjnego. Choć generalnie należy unikać instalowania innych aplikacji na serwerze z instancją SQL Servera, przy ustawianiu maksymalnej ilości pamięci warto wziąć pod uwagę wymagania pamięciowe innych komponentów z pakietu SQL Server, takich jak SSIS i SSAS. Należy również uwzględnić potrzeby niezbędnych narzędzi i agentów, które muszą działać na serwerze.

10.4. Numer 57 — nieblokowanie stron w pamięci

Gdy systemowi Windows zaczyna brakować pamięci, może on stać się niestabilny i mogą pojawiać się błędy braku pamięci. W takiej sytuacji Windows próbuje chronić się, zmniejszając zestaw roboczy procesów użytkownika. Innymi słowy, system wymusza przenoszenie danych procesów na dysk, aby odzyskać pamięć dla siebie.

Wyobraź sobie, że mamy instancję SQL Servera z ograniczonymi zasobami, a użytkownicy zaczynają zgłaszać poważne problemy z wydajnością, w tym

przekroczenia limitu czasu kwerend. Administratorzy baz danych badają problem i odkrywają następujący komunikat w dzienniku SQL Servera:

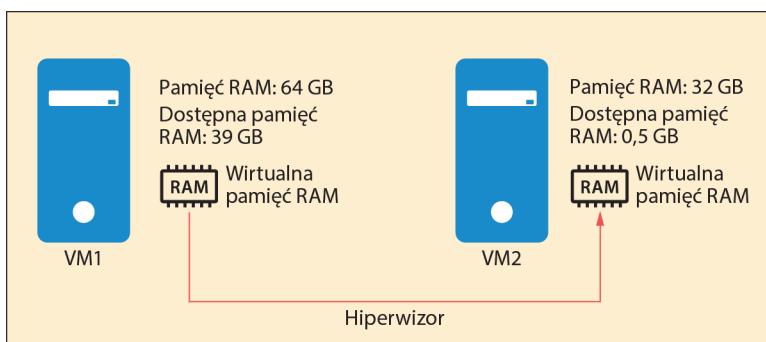
A significant part of SQL Server process memory has been paged out. This may result in a performance degradation. Duration: 0 seconds. Working set (KB): 6081740, committed (KB): 17175674, memory utilization: 35%.

Ten błąd wskazuje, że część pamięci podręcznej buforów została zapisana na dysk, a pamięć została odzyskana przez system operacyjny. Czy zatem blokowanie stron w pamięci to oczywiste rozwiązanie? Cóż, w środowisku specjalistów toczyła się na ten temat znacząca debata, która sprawiła, że wielu administratorów baz danych nie konfiguruje tego ustawienia. Dlaczego jest to przedmiotem dyskusji?

W starszych wersjach Windows Server system operacyjny bardzo agresywnie zarządzał zestawami roboczymi procesów. Z biegiem czasu to zachowanie stało się mniej agresywne, podczas gdy ogólne zarządzanie pamięcią uległo poprawie. Oznaczało to, że prawdopodobieństwo wystąpienia tego problemu znacznie się zmniejszyło.

Jednocześnie domyślnym środowiskiem hostingu SQL Servera stały się platformy wirtualizacyjne. Jedną z głównych zalet platform wirtualizacyjnych, takich jak VMware czy Hyper-V, jest możliwość nadmiernego przydzielania zasobów fizycznych. Oznacza to, że na przykład host wirtualizacyjny z 8 procesorami i 64 GB pamięci RAM mógłby obsługiwać maszyny wirtualne o łącznym zapotrzebowaniu na 16 wirtualnych procesorów i 96 GB pamięci RAM.

Zarządza tym hiperwizor, przydzielając zasoby maszynom wirtualnym wtedy, gdy są potrzebne. Kiedy maszyny wirtualne nie są używane, odzyskuje te zasoby i przydziela je innym, które ich bardziej potrzebują. Proces ten znany jest jako pompowanie (ang. *ballooning*) i został zilustrowany na rysunku 10.3.



Rysunek 10.3. Pompowanie pamięci

Jeśli maszyny wirtualne hostują instancje SQL Servera, które blokują strony w pamięci, to pamięć ta nie może zostać zwolniona. Może to prowadzić do problemów z wydajnością w całym środowisku wirtualizacyjnym.

Argument dotyczący maszyn wirtualnych jest w pełni uzasadniony. Podczas uruchamiania SQL Servera w środowisku wirtualnym warto ściśle współpracować z administratorami VMware, aby zapewnić użytkownikom niezawodną usługę bez negatywnego wpływu na całe środowisko. Jednakże argument dotyczący mniej agresywnego ograniczania zestawu roboczego nie jest wystarczająco mocny, by uzasadnić rezygnację z ustawiania opcji *Lock Pages In Memory* jako domyślnej przy instalacji. Fakt, że zdarza się to rzadziej, nie oznacza, że nigdy się nie zdarza — sam byłem świadkiem takich sytuacji. Powinniśmy starać się tego unikać, ponieważ są to awarie, którym można łatwo zapobiec.

Ponieważ standardem dla hostingu SQL Servera staje się chmura, warto wspomnieć, że główni dostawcy usług chmurowych, tacy jak AWS, zalecają konfigurację opcji *Lock Pages In Memory* jako najlepszą praktykę w swoim środowisku. Dzieje się tak, ponieważ mimo że jest to środowisko wirtualne, dostawcy chmury nie przydzielają zasobów nadmiarowo. W związku z tym kwestia „pompowania” jest tu nieistotna.

Funkcję *Lock Pages In Memory* konfiguruje się poprzez przyznanie uprawnień *SeLockMemoryPrivilege* kontu usługi, na którym działa usługa Database Engine. Można ją również skonfigurować podczas instalacji SQL Servera lub z zakładki *Advanced* dla usługi Database Engine w narzędziu SQL Server Configuration Manager.

10.5. Numer 58 — działanie wbrew optymalizatorowi

T-SQL to język opisowy. Oznacza to, że gdy programista pisze kwerendę, opisuje rezultaty, jakie chce otrzymać z SQL Servera, zamiast podawać mu szczegółowe instrukcje krok po kroku, jak wykonać dane zadanie. Następnie optymalizator kwerend oblicza najbardziej efektywny sposób zwrócenia danych, których programista poszukuje.

Optymalizator kwerend jest bardzo zaawansowanym narzędziem, które podejmuje wiele trafnych decyzji na podstawie indeksów, statystyk i licznych innych czynników. Nie jest jednak nieomylny i jeśli wybierze nieoptymalne rozwiązanie, może to negatywnie wpłynąć na wydajność kwerendy.

Programiści i administratorzy SQL Servera mogą wpływać na plan kwerendy generowany przez optymalizator, używając wskazówek do kwerend. Wskazówki te mogą zmieniać decyzje, które optymalizator podjąłby samodzielnie. Powinny być używane tylko w wyjątkowych okolicznościach, ale mogą okazać się nieocenione w rozwiązywaniu problemów z wydajnością.

OSTRZEŻENIE Wskazówek do kwerend należy używać tylko w wyjątkowych okolicznościach. W większości przypadków optymalizator sam wybiera najlepsze rozwiązanie.

Błędem, który często widzę przy stosowaniu wskazówek do kwerend, jest ich nadmierne wykorzystywanie lub, jak lubię mówić, działanie wbrew optymalizatorowi zamiast współpracy z nim. Najlepszym przykładem jest wskazówka określająca fizyczną operację łączenia tabel.

Do wykonywania logicznej operacji złączenia, takiej jak INNER JOIN lub OUTER JOIN między dwiema tabelami, optymalizator kwerend może wykorzystać cztery fizyczne operatory. Każdy z nich jest najbardziej efektywny w innym scenariuszu. Podsumowanie tych operatorów znajduje się w tabeli 10.1.

Tabela 10.1. Fizyczne operatory złączeń

| Operator | Opis | Najefektywniejsze zastosowanie |
|---|---|--|
| Nested Loops (złączenie w pętli zagnieżdżonej) | Wybiera jedną tabelę jako zewnętrzną, a drugą jako wewnętrzną. Dla każdego klucza w tabeli zewnętrznej przeszukuje wszystkie wiersze w tabeli wewnętrznej w poszukiwaniu pasujących wartości kluczy. | Małe tabele |
| Hash Join (złączenie haszujące) | Tworzy tabelę haszującą w pamięci, umieszczając każdy wiersz pierwszej tabeli w odpowiednim przedziale na podstawie wartości klucza. Następnie haszuje klucze drugiej tabeli i porównuje je z pierwszą tabelą wiersz po wierszu. Jeśli brakuje pamięci na wszystkie wiersze pierwszej tabeli, operacja jest wykonywana w kilku krokach. | Duże tabele nieposortowane według klucza złączenia lub znacznie różniącą się liczbą wierszy w tabelach |
| Merge Join (złączenie przez scalanie) | Sprawdza równość pierwszego klucza w obu tabelach. Następnie porównuje kolejny wiersz w drugiej tabeli z pierwszym wierszem w pierwszej tabeli. Przypomina to pętlę zagnieżdżoną, ale ponieważ dane wejściowe są posortowane według klucza złączenia, można przerwać operację po znalezieniu niepasującego wiersza w drugiej tabeli i przejść do następnego wiersza w pierwszej tabeli. | Duże tabele posortowane według klucza złączenia, o podobnej liczbie wierszy |
| Adaptive Join (złączenie adaptacyjne) | Kwerenda zaczyna od użycia złączenia haszującego, ale w trakcie wykonywania może przełączyć się na pętlę zagnieżdżoną, jeśli tabela używana do tworzenia tabeli haszującej jest na tyle mała, że pętla zagnieżdżona jest bardziej efektywna. | Kwerendy, w których liczba wierszy wejściowych znacznie się zmienia i używane jest przetwarzanie wsadowe |

Wyobraź sobie sytuację, w której kwerenda z poniższego przykładu (listing 10.4) działa bardzo wolno.

Listing 10.4. Kwerenda o niskiej wydajności

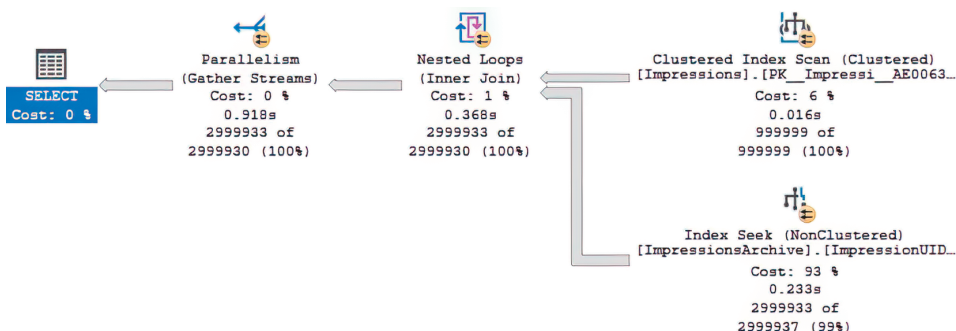
```

SELECT
    i.ImpressionUID
FROM Marketing.marketing.Impressions i
INNER JOIN MarketingArchive.dbo.ImpressionsArchive ia
    ON i.ImpressionUID = ia.ImpressionUID ;

```

Analizujemy problem i po zbadaniu planu kwerendy odkrywamy, że optymalizator wybiera złączenie typu Nested Loops jako operator łączenia, jak pokazano na rysunku 10.4. Na moim komputerze testowym wykonanie kwerendy z tym operatorem trwa 19 sekund.

WSKAZÓWKA Jeśli wykonujesz przykłady z tego rozdziału, pamiętaj, że plan wybrany przez SQL Server może różnić się od planu, który został wygenerowany w moim środowisku.



Rysunek 10.4. Plan z wykorzystaniem operatora Nested Loops

OSTRZEŻENIE Zanim przejdziemy dalej, należy podkreślić, że stosowanie wskazówek do kwerend powinno być ostatecznością. Tego typu problemy zazwyczaj można rozwiązać innymi metodami, na przykład poprzez aktualizację statystyk. Wskazówki do kwerend to ostatnia deska ratunku.

Błędem, który wielu administratorów baz danych popełnia w tej sytuacji, jest zmuszanie optymalizatora do użycia złączenia typu Hash Join, ponieważ w danym momencie wydaje się to najbardziej efektywną opcją. Istnieje kilka sposobów, aby to osiągnąć, w tym zamrożenie planu, wskazówka Query Store, wskazówka USE PLAN, wskazówki HASH MATCH lub wskazówka do złączenia. W tym przykładzie użyjemy wskazówki do złączenia, jak pokazano na listingu 10.5.

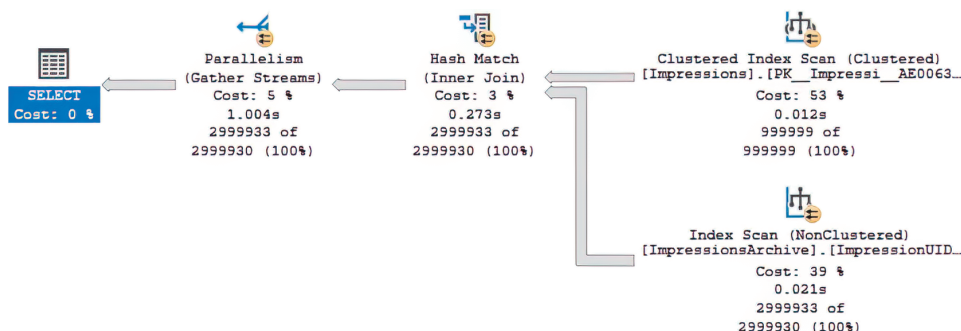
Listing 10.5. Wymuszenie złączenia typu Hash Join za pomocą wskazówki

```

SELECT
    i.ImpressionUID
FROM Marketing.marketing.Impressions i
INNER HASH JOIN MarketingArchive.dbo.ImpressionsArchive ia
    ON i.ImpressionUID = ia.ImpressionUID ;

```

W kwerendzie tej wykorzystano plan przedstawiony na rysunku 10.5, a na moim stanowisku testowym zajęła ona 5 sekund.



Rysunek 10.5. Plan wykorzystujący złączenie typu Hash Join na podstawie wskazówki

Świetny wynik! Dlaczego więc jest to pomyłka? Zastanówmy się nad tym. Przypuśćmy, że po wykonaniu tej kwerendy do kolumny `ImpressionUID` w tabeli `Impressions` w bazie danych `Marketing` dodano indeksy nieklastrowe. Można to osiągnąć za pomocą skryptu przedstawionego na listingu 10.6.

WSKAZÓWKA Nie trzeba tworzyć tego indeksu dla bazy danych `MarketingArchive`, ponieważ odpowiedni indeks został już utworzony w rozdziale 9.

Listing 10.6. Tworzenie indeksu na kolumnach używanych w złączeniu

```

USE Marketing ;
GO

CREATE NONCLUSTERED INDEX ImpressionUID
ON marketing.Impressions(ImpressionUID) ;
GO
  
```

Teraz byłoby najlepiej, gdyby optymalizator wybrał złączenie typu `Merge Join`, które wykorzystuje nowy indeks i na moim środowisku testowym zajmuje 3 sekundy. Niestety, nasza wskazówka zmusza optymalizator do użycia złączenia typu `Hash Join`, które na tym samym stanowisku trwa 4 sekundy, nawet z nowym indeksem. Dlatego lepszym podejściem byłoby zastosowanie wskazówki do kwerendy, która pozwala określić kilka opcji do wyboru dla optymalizatora. Rozważmy kwerendę przedstawioną na listingu 10.7.

Listing 10.7. Użycie wskazówki do kwerendy zamiast wskazówki do złączenia

```

SELECT
    i.ImpressionUID
FROM Marketing.marketing.Impressions i
INNER JOIN MarketingArchive.dbo.ImpressionsArchive ia
    ON i.ImpressionUID = ia.ImpressionUID OPTION (MERGE JOIN, HASH JOIN) ;
  
```

Ta wskazówka nadal wyklucza możliwość użycia zagnieżdżonych pętli, które — jak wiemy — nigdy nie będą dobrym rozwiązaniem ze względu na rozmiar tabel. Daje ona jednak optymalizatorowi wybór między złączeniem typu Merge Join a Hash Join, w zależności od tego, które uzna za bardziej efektywne w danej sytuacji.

Chociaż wskazówek do kwerend należy używać tylko w ostateczności, mogą być przydatnym narzędziem do rozwiązywania problemów z wydajnością. Jeżeli zdecydujesz się na ich zastosowanie, staraj się współpracować z optymalizatorem, a nie działać wbrew niemu. Jeśli to możliwe, daj mu do wyboru kilka opcji, zamiast wymuszać jedno konkretne rozwiązanie, które z czasem może stać się nieaktualne.

10.6. Numer 59 — niewykorzystywanie informacji zwrotnych DOP

W poprzednim podrozdziale rozmawialiśmy o tym, jak współpracować z optymalizatorem, zamiast działać wbrew niemu, gdy wydaje nam się, że nie mamy innego wyboru, jak tylko użyć wskazówek do kwerend. Czy życie nie byłoby o wiele prostsze, gdyby SQL Server uczył się, co działa najlepiej, i dostosowywał plany wykonania według potrzeb?

SQL Server 2022 rozszerza zbiór inteligentnych funkcji przetwarzania kwerend, które umożliwiają adaptację planów wykonania na podstawie wcześniej wykrytych nieefektywności. W wersji tej wprowadzono mechanizmy informacji zwrotnych o kwerendach, które wcześniej były dostępne w wersji testowej Azure SQL Database. Funkcje informacji zwrotnych o przydziałach pamięci stopniowo dodawano do podstawowej wersji produktu w ciągu ostatnich trzech wydań.

Funkcje informacji zwrotnych o kwerendach wykorzystują mechanizm Query Store do porównywania różnych planów wykonania i związanej z nimi wydajności. Aby skorzystać z przykładów przedstawionych w tej sekcji, musisz włączyć mechanizm Query Store dla bazy danych MarketingArchive. Można to zrobić za pomocą polecenia pokazanego na listingu 10.8.

WSKAZÓWKA Począwszy od wersji SQL Server 2022 mechanizm Query Store jest domyślnie włączony. Jest on również domyślnie aktywny w usługach Azure SQL Database i Azure Managed Instance.

Listing 10.8. Włączanie mechanizmu Query Store dla bazy danych MarketingArchive

```
ALTER DATABASE MarketingArchive  
SET QUERY_STORE = ON (OPERATION_MODE = READ_WRITE) ;
```


WSKAZÓWKA Pełne omówienie mechanizmu Query Store wykracza poza ramy niniejszej książki, ale Microsoft udostępnia szczegółowy przewodnik na stronie <https://mng.bz/YVmA>.

Wyobraź sobie sytuację, w której kwerendy w naszej instancji zaczynają działać wolno. Podczas analizy statystyk oczekiwania odkrywamy bardzo wysokie wartości dla typów oczekiwania `CX_PACKET` i `SOS_SCHEDULER_YIELD`. Zauważamy również, że ma to miejsce zawsze, gdy wykonywana jest duża kwerenda raportowa. Po sprawdzeniu planu kwerendy okazuje się, że jest ona wykonywana z maksymalnym stopniem zrównoleglenia (`MAXDOP`) równym 16.

WSKAZÓWKA Przed wydaniem SQL Server 2016 Service Pack 2 i SQL Server 2017 Cumulative Update 4 wyniki zwracane przez `CX_PACKET` były niekonsekwentne. Od czasu naprawienia błędu statystyka ta stała się bardzo użyteczna.

Wszystkie objawy wskazują na to, że problem jest spowodowany przez dużą kwerendę raportową, które działa ze stopniem zrównoleglenia faktycznie szkodliwym dla czasu zarówno jej własnego wykonania, jak i dla innych jednocześnie uruchamianych kwerend.

Wcześniej zmieniliśmy ustawienie `MAXDOP` na poziomie instancji na wartość 16, aby poprawić wydajność niektórych kwerend. Dlatego postanawiamy dodać wskazówkę `MAXDOP 8` do problematycznej kwerendy. Jest to jednak pomyłka. Dlaczego? Ponieważ korzystamy z wersji SQL Server 2022, która umożliwia włączenie mechanizmu informacji zwrotnych DOP. Pozwala on SQL Serverowi na ciągłą ocenę i zmianę stopnia zrównoleglenia (DOP) używanego w planie kwerendy na podstawie statystyk wykonania. Dlaczego administratorzy baz danych popełniają ten błąd? Są dwa powody: po pierwsze, informacje zwrotne DOP są jedną z trzech funkcji optymalizacji kwerend, która nie jest domyślnie włączona. Po drugie, w natłoku codziennych obowiązków administratorzy nie zawsze mają czas, by na bieżąco śledzić wszystkie nowe funkcje SQL Servera. To doskonały przykład tego, jak ważne jest monitorowanie i wdrażanie nowych funkcji. Może to zaoszczędzić administratorom wiele czasu i pozwolić im skupić się na bardziej wartościowych zadaniach.

SQL Server 2022 oferuje następujące opcje informacji zwrotnych o kwerendach, które są częścią pakietu inteligentnego przetwarzania kwerend:

- informacje zwrotne DOP,
- informacje zwrotne dotyczące szacowania licznosci,
- informacje zwrotne dotyczące przydziału pamięci.

Omówiliśmy już mechanizm informacji zwrotnych DOP. Pozwala on SQL Serverowi optymalizować stopień zrównoleglenia kwerendy na podstawie historycznych wykonania.

Szacowanie licznosci (ang. *cardinality estimate*, CE) pomaga SQL Serverowi w wyborze najbardziej odpowiedniego planu wykonania na podstawie przewidywanej liczby wierszy przetwarzanych na każdym etapie realizacji kwerendy.

Szacunki te, oparte na statystykach kolumn i indeksów oraz przetwarzane przez zaawansowany algorytm, mogą czasami być niedokładne. Wynika to z faktu, że nie da się stworzyć algorytmu uwzględniającego wszystkie możliwe wymagania i wzorce użytkowania. Niedokładne szacunki mogą prowadzić do nieefektywnych planów kwerend, a w konsekwencji do niskiej wydajności. Mechanizm informacji zwrotnych CE może pomóc SQL Serverowi w identyfikowaniu niedokładnych szacunków i podejmowaniu lepszych decyzji optymalizacyjnych.

Mechanizm *informacji zwrotnych dotyczących przydziału pamięci* dostosowuje ilość pamięci przydzielanej zapytaniu na podstawie danych historycznych. Pomaga to uniknąć sytuacji, w której przydziela się za mało pamięci, co skutkuje zapisywaniem danych na dysku. Zapobiega również przydzielaniu nadmiernej ilości pamięci, co może prowadzić do nieefektywnego zrównoleglenia operacji.

W przeciwieństwie do mechanizmów informacji zwrotnych DOP i CE, które są nowością w wersji SQL Server 2022, mechanizm informacji zwrotnych dotyczących przydziału pamięci był stopniowo wdrażany w trzech ostatnich głównych wersjach SQL Servera. W wersji SQL Server 2017 zaimplementowano mechanizm informacji zwrotnych dotyczących przydziału pamięci w trybie wsadowym, a w wersji SQL Server 2019 – w trybie wierszowym. W wersji SQL Server 2022 wdrożono tryby percentylowy i trwały. Dzięki temu statystyki wykorzystywane do podejmowania decyzji są przechowywane w magazynie Query Store i nie są tracone po ponownym uruchomieniu instancji.

Jeśli dla bazy danych włączono funkcję Query Store, domyślnie aktywne będą mechanizmy informacji zwrotnych dotyczących przydziału pamięci oraz szacowania liczności. Warunkiem jest ustawienie poziomu zgodności bazy danych na 140 lub wyższy (dla przydziału pamięci) oraz 160 lub wyższy (dla CE). Natomiast mechanizm informacji zwrotnych DOP jest domyślnie wyłączony. Można go aktywować za pomocą polecenia przedstawionego na listingu 10.9.

Listing 10.9. Włączanie informacji zwrotnych DOP

```
ALTER DATABASE SCOPED CONFIGURATION  
SET DOP_FEEDBACK = ON ;
```

Kwerenda z listingu 10.10 pozwala sprawdzić, które funkcje informacji zwrotnej o kwerendach są włączone dla bieżącej bazy danych.

Listing 10.10. Sprawdzanie, które opcje informacji zwrotnych o kwerendach są włączone

```
SELECT  
    name  
    , CASE  
        WHEN value = 1 THEN 'Enabled'  
        ELSE 'Disabled'  
    END as Enabled  
FROM sys.database_scoped_configurations  
WHERE name LIKE '%feedback%' ;
```

Dobłą praktyką jest włączenie informacji zwrotnych DOP w bazach danych o poziomie zgodności 160 lub wyższym. Eliminuje to konieczność ręcznej oceny i optymalizacji ustawień MAXDOP dla kwerend przez administratora bazy danych.

10.7. Numer 60 — niepartycjonowanie dużych tabel

SQL Server często jest ograniczony przez operacje wejścia-wyjścia. Oznacza to, że podsystem dyskowy stanowi wąskie gardło wpływające na wydajność kwerend. Jest to szczególnie widoczne w przypadku bardzo dużych baz danych, w których wykonywane są kwerendy raportowe na ogromnych tabelach.

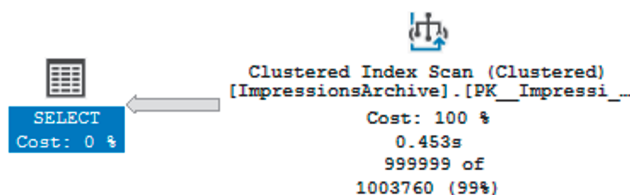
Rozważmy bazę danych MarketingArchive. Wyobraźmy sobie, że nasza instancja ma problemy z operacjami wejścia-wyjścia, a kwerenda z listingu 10.11 jest często wykonywana. Pierwsze polecenie w skrypcie włącza statystyki operacji wejścia-wyjścia dla bieżącej sesji.

Listing 10.11. Kwerenda w dużej tabeli

```
SET STATISTICS IO ON ;
GO

SELECT *
FROM dbo.ImpressionsArchive
WHERE EventTime >= '20210101' AND EventTime <= '20211231' ;
```

Jeśli przyjrzymy się planowi kwerendy przedstawionemu na rysunku 10.6, zauważymy, że wykonywane jest skanowanie indeksu klastrowego. Oznacza to, że odczytywany jest każdy wiersz w tabeli.



Rysunek 10.6. Plan wykonania kwerendy raportowej

Przyjrzyjmy się statystykom operacji wejścia-wyjścia dla tej kwerendy. Zostaną one wyświetlone na karcie *Messages*, ponieważ w tej sesji włączone jest raportowanie statystyk I/O:

```
Table 'ImpressionsArchive'. Scan count 1, logical reads 46979, physical reads 1, page
server reads 0, read-ahead reads 46981, page server read-ahead reads 0, lob logical reads
0, lob physical reads 0, lob page server reads 0, lob read-ahead reads 0, lob page server
read-ahead reads 0.
```

Pamięć podręczna buforów była pusta, więc SQL Server wykonał fizyczny odczyt wszystkich wymaganych stron (1 odczyt fizyczny i 46 981 odczytów wyprzedzających). W wyniku odczytów wyprzedzających pozostałe potrzebne strony zostały umieszczone w pamięci podręcznej buforów, co wyjaśnia, dlaczego statystyki pokazują 46 979 odczytów logicznych.

Opróżnianie pamięci podręcznej

W celach testowych czasami przydatne jest opróżnienie pamięci podręcznej buforów, by wymusić odczyt stron z dysku. Jeśli tego nie zrobisz, przy drugim uruchomieniu kwerendy strony będą już w pamięci podręcznej, co sprawi, że test nie będzie miarodajny.

Aby usunąć strony z pamięci podręcznej buforów, wykonaj poniższe polecenia:

```
CHECKPOINT  
DBCC DROPLEANBUFFERS
```

Pierwsze polecenie powoduje zapisanie na dysku stron „brudnych”, czyli tych, które zostały zmodyfikowane. Drugie polecenie usuwa z pamięci podręcznej strony „czyste”, czyli te, które nie zostały zmienione.

Jeśli planujesz przeprowadzać wielokrotne testy, dobrą praktyką jest umieszczenie tych poleceń na początku skryptu. Pamiętaj jednak, że powinieneś to robić wyłącznie na serwerze testowym lub deweloperskim. Uruchamianie takich poleceń na serwerze produkcyjnym może negatywnie wpłynąć na jego wydajność.

Często zdarza się, że administratorzy baz danych popełniają pomyłkę, zatrzymując się na etapie sprawdzania brakujących indeksów. Następnie albo rozkładają ręce, mówiąc: „Niestety, nic więcej nie da się zrobić”, albo kontaktują się z zespołem odpowiedzialnym za pamięć masową, licząc na wyciągnięcie dodatkowej wydajności z macierzy SAN. Tymczasem powinni skupić się na rozważeniu niedocenianej funkcji, jaką jest *partycjonowanie tabel*.

Partycjonowanie tabel to funkcja, która jest od dawna dostępna w SQL Serverze i dzieli tabelę na kilka mniejszych tabel na podstawie klucza partycjonowania. Podział ten jest całkowicie niewidoczny dla programisty, ponieważ tabela nadal wygląda tak, jakby była jedną całością, i nie wymaga żadnych zmian w kodzie.

Istnieją dwie główne korzyści wynikające z partycjonowania dużej tabeli, na której wykonywane są kwerendy raportowe. Pierwszą z nich jest możliwość umieszczenia każdej partycji w osobnej grupie plików. Jeśli pliki w tych grupach są przechowywane na różnych dyskach, partycje mogą być odczytywane równolegle.

Druga, moim zdaniem ważniejsza, zaleta to możliwość wyeliminowania partycji. Innymi słowy, SQL Server może odczytywać dane tylko z tych partycji, które są niezbędne, pomijając te, które nie zawierają istotnych informacji. Może to prowadzić do znacznego zmniejszenia liczby operacji odczytu, co odciąża podsystem wejścia-wyjścia. Bezpośrednio wpływa to również na poprawę wydajności kwerendy, ponieważ odczytywana jest mniejsza ilość danych.

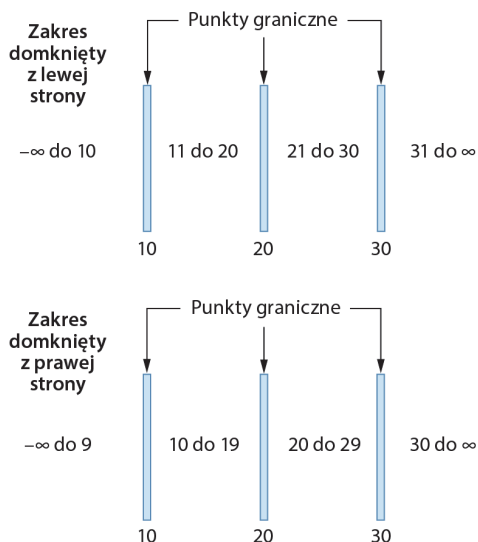
Dlaczego więc administratorzy baz danych nie korzystają z tej funkcji? Są trzy powody, z którymi się spotkałem. Pierwszy to po prostu nieznanomość tej funkcji. Zawsze mnie to zaskakuje, ponieważ została ona wprowadzona w SQL Serverze 2005 — prawie 20 lat temu! Drugi powód to kwestie polityczne. Zdarzały się sytuacje, w których zespół administratorów twierdził, że to zadanie programistów, bo oni nie znają specyfiki aplikacji. Jednocześnie programiści odmawiali wdrożenia tej funkcji, argumentując, że nie są administratorami i nie rozumieją partycjonowania. Problem ten można łatwo rozwiązać poprzez współpracę. Jeśli oba zespoły połączą siły, implementacja będzie prosta.

Ostatnim powodem jest niezrozumienie działania technologii lub sposobu jej implementacji. Zbadamy ten powód w pozostałej części tego podrozdziału.

Aby utworzyć tabelę partycjonowaną lub podzielić istniejącą tabelę na partycje, należy utworzyć funkcję partycjonującą oraz schemat partycjonowania. Aby lepiej zrozumieć te pojęcia, lubię posługiwać się analogią do gospodarstwa rolnego z wieloma polami. Pola są oddzielone od siebie płotami, co umożliwia uprawę różnych roślin na każdym z nich.

Używając tej analogii, można powiedzieć, że funkcja partycjonująca odpowiada płotom. Wyznacza ona punkty graniczne — miejsca, gdzie kończy się jedno pole, a zaczyna następne. Tworząc funkcję partycjonującą, określamy typ danych punktów granicznych, konkretne wartości tych punktów oraz to, czy przedział ma być domknięty z lewej czy z prawej strony.

Domknięcie z lewej lub prawej strony pozwala określić, czy wartości znajdujące się dokładnie na granicy przedziału mają być przypisane do lewej czy prawej strony tej granicy. Różnicę tę zilustrowano na rysunku 10.7. W tym przykładzie funkcja partycjonująca utworzyła punkty graniczne dla wartości całkowitych 10, 20 i 30.



Rysunek 10.7. Zakresy funkcji partycjonującej domknięte z lewej i prawej strony

Schemat partycjonowania określa, w której grupie plików umieszczona jest każda partycja. Mamy możliwość wskazania innej (wcześniej utworzonej) grupy plików dla każdej partycji lub użycia słowa kluczowego ALL, aby zaznaczyć, że chcemy przechowywać wszystkie partycje w tej samej grupie plików.

Na koniec, podczas tworzenia tabeli, zamiast klauzuli ON <GRUPA_PLIKÓW> używamy klauzuli ON <SCHEMAT_PARTYCJONOWANIA> (<KLUCZ_PARTYCJONOWANIA>). Taka architektura umożliwia utworzenie wielu partycji w ramach jednego schematu partycjonowania oraz wielu schematów partycjonowania z jedną funkcją partycjonującą.

Skrypt z listingu 10.12 tworzy funkcję partycjonującą o nazwie Impression ↪ DatesPF, która wykorzystuje zakres domknięty z prawej strony i typ danych DATETIME. Funkcja ta ustanawia punkty graniczne na pierwszy dzień stycznia w latach 2020, 2021, 2022 i 2023. Następnie tworzy schemat partycjonowania o nazwie ImpressionDatesPS, który odwzorowuje wszystkie partycje na grupę plików PRIMARY. W kolejnym kroku skrypt tworzy nową tabelę o nazwie ImpressionArchive ↪ Partitioned ze schematem partycjonowania ImpressionDatesPS, używając kolumny EventTime jako klucza podstawowego. Następnie tworzy klucz PRIMARY KEY w kolumnach ImpressionID i EventTime. Ważne jest, aby zrozumieć, że w przypadku tabel partycjonowanych klucz partycjonowania musi być podzbiorem dowolnego unikatowego klucza tabeli. W tym przypadku stworzymy nową tabelę, zamiast partycjonować istniejącą, z dwóch powodów. Po pierwsze, pozwala to na porównanie obu podejść, a po drugie, tabeli ImpressionArchive będziemy używać w późniejszych przykładach.

Listing 10.12. Tworzenie tabeli partycjonowanej

```
CREATE PARTITION FUNCTION ImpressionDatesPF (DATETIME) ← Tworzy funkcję partycjonującą.
AS RANGE RIGHT FOR VALUES ('20200101', '20210101', '20220101', '20230101');
GO

CREATE PARTITION SCHEME ImpressionDatesPS ← Tworzy schemat partycjonowania.
AS PARTITION ImpressionDatesPF
ALL TO ([PRIMARY]) ;
GO

CREATE TABLE dbo.ImpressionsArchivePartitioned(
    ImpressionID      BIGINT          NOT NULL IDENTITY(1,1),
    ImpressionUID      UNIQUEIDENTIFIER NOT NULL,
    ReferralURL        VARCHAR(512)    NOT NULL,
    CookieID           UNIQUEIDENTIFIER NOT NULL,
    CampaignID         BIGINT          NOT NULL,
    RenderingID        BIGINT          NOT NULL,
    CountryCode        TINYINT         NULL,
    StateID            TINYINT         NULL,
    BrowserVersion     BIGINT          NOT NULL,
    OperatingSystemID  BIGINT          NOT NULL,
    BidPrice           MONEY           NOT NULL,
    CostPerMille       MONEY           NOT NULL,
    EventTime          DATETIME        NOT NULL,
) ON ImpressionDatesPS(EventTime) ; ← Tworzy tabelę ze schematem partycjonowania.
```

GO

```
ALTER TABLE dbo.ImpressionsArchivePartitioned ADD CONSTRAINT PK_ImpressionsArchivePartitioned PRIMARY KEY (ImpressionID, EventTime) ;
```

GO

```
INSERT INTO dbo.ImpressionsArchivePartitioned ( ImpressionUID,
    ReferralURL,
    CookieID,
    CampaignID,
    RenderingID,
    CountryCode,
    StateID,
    BrowserVersion,
    OperatingSystemID,
    BidPrice,
    CostPerMille,
    EventTime
)
SELECT
    ImpressionUID
    , ReferralURL
    , CookieID
    , CampaignID
    , RenderingID
    , CountryCode
    , StateID
    , BrowserVersion
    , OperatingSystemID
    , BidPrice
    , CostPerMille
    , EventTime
FROM dbo.ImpressionsArchive ;
```

**Tworzy klucz
podstawowy.**

Wstawia dane do nowej tabeli.

Sprawdźmy teraz, jak wpłynęło to na wymagania dotyczące operacji wejścia-wyjścia, przez wykonanie kwerendy z listingu 10.13.

Listing 10.13. Kwerenda w partycjonowanej tabeli

```
SELECT *
FROM dbo.ImpressionsArchivePartitioned
WHERE EventTime >= '20210101' AND EventTime <= '20211231' ;
```

Statystyki operacji wejścia-wyjścia dla tej kwerendy są następujące:

```
Table 'ImpressionsArchivePartitioned'. Scan count 1, logical reads 15732, physical reads 2, page server reads 0, read-ahead reads 15736, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob page server reads 0, lob read-ahead reads 0, lob page server read-ahead reads 0.
```

Mimo że kwerenda zwraca dokładnie te same wiersze, to zamiast odczytywać 46 982 strony z dysku, odczytała tylko 15 738 stron. Jeśli przyjrzymy się planowi kwerendy przedstawionemu na rysunku 10.8, przyczyna stanie się oczywista. Chociaż nadal przeprowadzano skanowanie indeksu klastrowego, wyeliminowano wszystkie partycje oprócz jednej, co oznacza, że trzeba było odczytać tylko jedną

trzecią danych. Dzięki temu znacznie zmniejszyło się obciążenie dysku i poprawiła się wydajność kwerendy.

| Clustered Index Scan (Clustered) | |
|---|----------------------|
| Scanning a clustered index, entirely or only a range. | |
| Physical Operation | Clustered Index Scan |
| Logical Operation | Clustered Index Scan |
| Actual Execution Mode | Row |
| Estimated Execution Mode | Row |
| Storage | RowStore |
| Actual Number of Rows Read | 999999 |
| Actual Number of Rows for All Executions | 999999 |
| Actual Number of Batches | 0 |
| Estimated Operator Cost | 12.3975 (100%) |
| Estimated I/O Cost | 11.3217 |
| Estimated Subtree Cost | 12.3975 |
| Estimated CPU Cost | 1.07587 |
| Estimated Number of Executions | 1 |
| Number of Executions | 1 |
| Estimated Number of Rows for All Executions | 977923 |
| Estimated Number of Rows to be Read | 977923 |
| Estimated Number of Rows Per Execution | 977923 |
| Estimated Row Size | 366 B |
| Actual Rebinds | 0 |
| Actual Rewinds | 0 |
| Partitioned | True |
| Actual Partition Count | 1 |
| Ordered | True |
| Node ID | 0 |

Rysunek 10.8. Plan kwerendy dla tabeli partycjonowanej

Jeśli masz bardzo duże tabele ze wzorcem użycia typowym dla kwerend raportowych, warto rozważyć podzielenie ich na partycje. Takie podejście pozwala na wyeliminowanie partycji, które nie są potrzebne, co może znacząco odciążyć podsystem wejścia-wyjścia i poprawić wydajność kwerend.

10.8. Numer 61 — niezrozumienie ograniczeń eliminowania partycji

W poprzednim podrozdziale omówiliśmy, jak przydatne może być partycjonowanie w przypadku bardzo dużych tabel. Korzyści wynikają z eliminowania partycji — SQL Server może odczytać dane tylko z tych partycji, które są potrzebne. Ponieważ jednak każda partycja jest przechowywana jako osobne B-drzewo, to jeśli nasze kwerendy nie są w stanie skorzystać z eliminowania partycji, w rzeczywistości mogą działać wolniej niż w przypadku niepartycjonowanej tabeli. Dlatego

pomyłką jest wprowadzanie partycjonowania bez przemyślenia, jakie kwerendy będą z niego korzystać.

Warto podkreślić, że do skutecznego eliminowania partycji istotny jest klucz partycjonowania. Eliminowanie partycji najczęściej nie działa przez pisanie kwerend, które nie wykorzystują klucza partycjonowania. Jeśli na przykład filtrujemy dane według klucza partycjonowania ORAZ innej kolumny, eliminacja partycji zadziała. Jeśli jednak filtrujemy według klucza partycjonowania LUB innej kolumny, konieczne będzie odczytanie wszystkich partycji. Pokazuje to listing 10.14. Pierwsza kwerenda w skrypcie uzyska dostęp tylko do jednej partycji. Druga kwerenda, wykorzystujące logikę OR (LUB), będzie musiała odczytać wszystkie partycje w tabeli.

Listing 10.14. Logika AND I OR

```
SELECT *
FROM dbo.ImpressionsArchivePartitioned
WHERE CampaignID = 44538
AND EventTime >= '20210101' AND EventTime <= '20211231' ;

SELECT *
FROM dbo.ImpressionsArchivePartitioned
WHERE CampaignID = 44538
OR EventTime >= '20210101' AND EventTime <= '20211231' ;
```

Inną częstą przyczyną problemów są sytuacje, w których SQL Server musi dokonać konwersji danych. Na przykład w naszej tabeli ImpressionsArchivePartitioned kolumna EventTime używa typu danych DATETIME. W dotychczasowych przykładach SQL Server był w stanie porównywać wartości literalowe z tą kolumną, mimo że były one sformatowane jako DATE, ponieważ DATETIME ma wyższy priorytet niż DATE. Gdybyśmy jednak jawnie określili naszą wartość jako DATE, SQL Server musiałby dokonać konwersji, co uniemożliwiłoby wyeliminowanie partycji. Na przykład pierwsza kwerenda z listingu 10.15 odczytałaby wszystkie partycje tabeli, podczas gdy druga byłaby w stanie wyeliminować wszystkie partycje oprócz jednej.

Listing 10.15. Przekształcanie typów danych

```
DECLARE @StartDate DATE ;
SET @StartDate = '20210101' ;

DECLARE @EndDate DATE ;
SET @Enddate = '20211231' ;

SELECT *
FROM dbo.ImpressionsArchivePartitioned
WHERE EventTime >= @StartDate AND EventTime <= @EndDate ;
GO

DECLARE @StartDate DATETIME ;
```

```
SET @StartDate = '20210101' ;

DECLARE @EndDate DATETIME ;
SET @Enddate = '20211231' ;

SELECT *
FROM dbo.ImpressionsArchivePartitioned
WHERE EventTime >= @StartDate AND EventTime <= @EndDate ;
```

Ostatnią powszechną przyczyną problemów jest prosta parametryzacja. Jeśli SQL Server uzna, że może znowu wykorzystać plan, będzie promować jego ponowne użycie poprzez parametryzację wartości w filtrze. To uniemożliwi działanie mechanizmu eliminowania partycji.

Można obejść ten problem na dwa sposoby. Pierwszym z nich jest użycie opcji RECOMPILE. Zapobiegnie to prostej parametryzacji, ponieważ plan w ogóle nie zostanie zapisany w buforze. Ma to jednak oczywistą wadę — plan będzie musiał być kompilowany za każdym razem, gdy kwerenda zostanie uruchomiona.

Drugim rozwiązaniem jest dodanie do kwerendy statycznego operatora nierówności, na przykład $1 <> 2$. Zapobiegnie to prostej parametryzacji, ale plan kwerendy nadal będzie buforowany, co pozwoli ponownie go wykorzystać.

Listing 10.16 zawiera trzy kwerendy. Pierwsza z nich odczyta wszystkie partycje tabeli. Druga i trzecia kwerenda odczytują tylko jedną partycję.

Listing 10.16. Prosta parametryzacja

```
SELECT COUNT(*)
FROM dbo.ImpressionsArchivePartitioned
WHERE EventTime >= '20210101' AND EventTime <= '20211231' ;
GO
```

```
SELECT COUNT(*)
FROM dbo.ImpressionsArchivePartitioned
WHERE EventTime >= '20210101' AND EventTime <= '20211231'
OPTION(RECOMPILE) ;
GO
```

```
SELECT COUNT(*)
FROM dbo.ImpressionsArchivePartitioned
WHERE EventTime >= '20210101' AND EventTime <= '20211231'
AND 1<>2 ;
GO
```

Oba te rozwiązania spowodują dynamiczne eliminowanie partycji. Oznacza to, że SQL Server decyduje o wyeliminowaniu partycji w czasie wykonywania kwerendy, a nie w momencie jej kompilacji. Rysunek 10.9 przedstawia predykaty wyszukiwania w planie trzeciej kwerendy z listingu 10.16. Jak widać, do określenia partycji, która ma zostać użyta, wykorzystywany jest pojedynczy operator skalarny o wartości 3.

| Seek Predicates | Seek Keys[1]: Prefix: Ptnld1000 = Scalar Operator((3)) |
|--|--|
| [-] [1] | Seek Keys[1]: Prefix: Ptnld1000 = Scalar Operator((3)) |
| [-] [-] [1] | Prefix: Ptnld1000 = Scalar Operator((3)) |
| [-] [-] [-] Prefix | Ptnld1000 = Scalar Operator((3)) |
| [-] [-] [-] [-] Range Columns | Ptnld1000 |
| [-] [-] [-] [-] [-] Column | Ptnld1000 |
| [-] [-] [-] [-] [-] Range Expressions | Scalar Operator((3)) |
| [-] [-] [-] [-] [-] [-] Const | |
| [-] [-] [-] [-] [-] [-] [-] ConstValue | (3) |
| [-] [-] [-] [-] [-] [-] [-] ScalarString | (3) |
| [-] [-] [-] [-] [-] [-] [-] Scan Type | EQ |

Rysunek 10.9. Predykaty wyszukiwania w planie wykonania kwerendy

Partycjonowanie dużych tabel może przynieść znaczące korzyści wydajnościowe. Jednak pochopne partycjonowanie bez odpowiedniego przygotowania kodu zapewniającego eliminowanie partycji jest pomyłką. Należy pamiętać o ograniczeniach eliminowania partycji i mieć świadomość, że wydajność może się pogorszyć, jeśli nie dojdzie do wyeliminowania partycji.

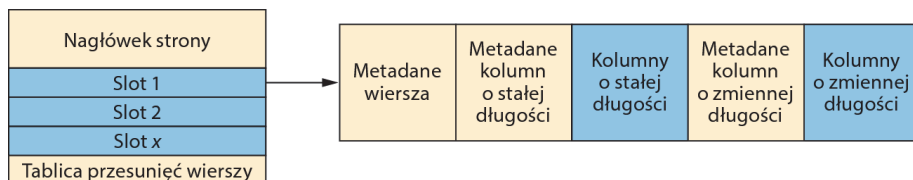
Wielu problemów związanych z eliminowaniem partycji można uniknąć przez rezygnację z prostej parametryzacji, zapobieganie konwersji wartości przez SQL Server oraz dbanie o to, by programiści pisali kod filtrujący lub łączący z wykorzystaniem klucza partycjonowania. Jeśli jednak kod nie jest kompatybilny z eliminowaniem partycji, lepiej zrezygnować z partycjonowania, ponieważ może ono negatywnie wpłynąć na wydajność.

10.9. Numer 62 — niekompresowanie dużych tabel

W poprzednim podrozdziale omówiliśmy partycjonowanie jako sposób na odciążenie podsystemu wejścia-wyjścia w celu poprawy wydajności kwerend. Inną metodą osiągnięcia tego celu jest zastosowanie kompresji. Większość administratorów baz danych, myśląc o kompresji, kojarzy ją ze zmniejszaniem rozmiaru danych kosztem wydajności. Jest to jednak błędne przekonanie, które prowadzi do niedostatecznego wykorzystania kompresji w SQL Serverze. W rzeczywistości kompresja może poprawić wydajność kwerend w przypadku niektórych obciążeń.

Jeśli SQL Server jest ograniczony przez operacje wejścia-wyjścia, ale ma wolne zasoby procesora, kompresja wierszy i stron może zmniejszyć liczbę stron, które SQL Server musi odczytać z dysku. Pozwala to odciążyć podsystem I/O kosztem dodatkowych cykli procesora.

Aby zrozumieć, jak działa kompresja, warto najpierw poznać strukturę danych w wierszu na zwykłej, nieskompresowanej stronie. Ilustruje to rysunek 10.10. Lewa strona diagramu przedstawia ogólną strukturę strony, gdzie slot oznacza fizyczny „pojemnik” na wiersz. Prawa strona diagramu pokazuje, jak zorganizowane są dane wewnątrz slotu.



Rysunek 10.10. Struktura strony danych

Zastanówmy się nad tabelą `ImpressionsArchive`. Z poprzedniego podrozdziału wiemy już, że aby odczytać wszystkie wiersze tej tabeli, trzeba odczytać 46 982 strony. A co by się stało, gdybyśmy zastosowali kompresję wierszy? Użyjmy polecenia z listingu 10.17, aby zaimplementować tę kompresję.

Listing 10.17. Implementacja kompresji wierszy

```
ALTER TABLE ImpressionsArchive
    REBUILD WITH (DATA_COMPRESSION = ROW) ;
```

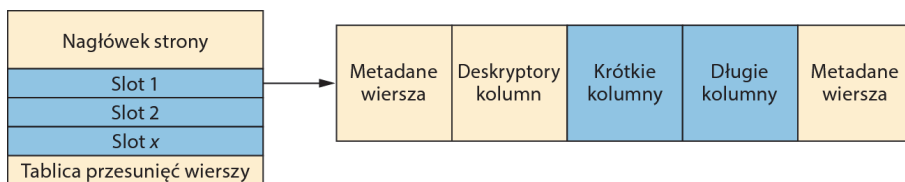
Skompresowana tabela zajmuje teraz 33 687 stron przechowujących dane, co oznacza redukcję o około 28%. Liczbę stron używanych przez tabelę możemy sprawdzić na dwa sposoby: przez wykonanie kwerendy bez klauzuli `WHERE` i analizę statystyk wejścia-wyjścia lub przez uruchomienie kwerendy z listingu 10.18, która pobiera liczbę stron z metadanych. Filtrowanie po `index_id = 1` zapewnia, że zliczamy tylko strony z indeksu klastrowego.

Listing 10.18. Określanie liczb stron w tabeli

```
SELECT
    in_row_used_page_count
FROM sys.dm_db_partition_stats
WHERE object_id = OBJECT_ID('ImpressionsArchive')
    AND index_id = 1 ;
```

Kompresja wierszy działa poprzez implementację systemu `VARDECIMAL`. Oznacza to, że oprócz typów `VARCHAR`, `NVARCHAR` i `VARBINARY`, które już są typami o zmiennej długości, również wartości dziesiętne stają się typem zmiennej długości. Na przykład w kolumnie typu `BIGINT` wartość 10 zajmie tylko 1 bajt, ponieważ zostanie zapisana jako `TINYINT`, a wartość 50 000 zajmie tylko 4 bajty, gdyż zostanie zapisana jako `INT`. Wartość `NULL` w ogóle nie zajmie miejsca. Kompresja wierszy usuwa również nadmiarowe spacje z kolumn znakowych o stałej długości i kompresuje wartości Unicode do 1 bajta na znak, jeśli to możliwe.

Mając to na uwadze, przyjrzyjmy się, jak dane są przechowywane na stronie danych z zastosowaną kompresją wierszy. Porównajmy tę strukturę, przedstawioną na rysunku 10.11, ze strukturą strony nieskompresowanej.



Rysunek 10.11. Struktura strony z kompresją wierszy

A gdybyśmy zastosowali kompresję stron? Polecenie z listingu 10.19 implementuje kompresję stron dla tabeli `ImpressionArchive`.

Listing 10.19. Implementacja kompresji stron

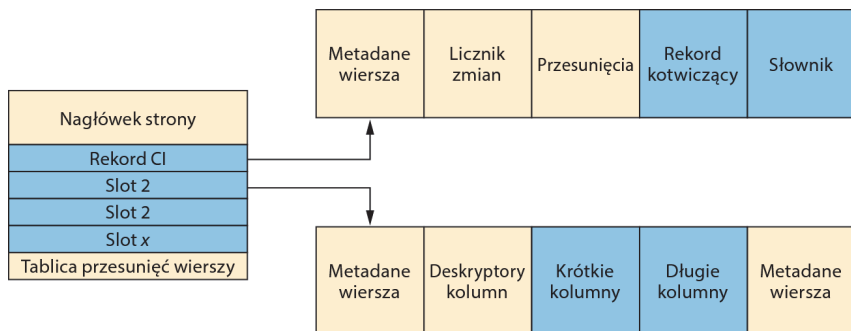
```
ALTER TABLE ImpressionArchive
REBUILD WITH (DATA_COMPRESSION = PAGE) ;
```

W przypadku kompresji stron stosowane są wcześniej opisane techniki kompresji wierszy. Następnie wykonywane są kompresja prefiksów oraz kompresja słownikowa.

W kompresji prefiksów następuje analiza wartości przechowywanych w wielu wierszach tej samej kolumny na stronie i próba zidentyfikowania wspólnego prefiksu dla tych wierszy. Wybierana jest najdłuższa wartość zawierająca pełny prefiks jako rekord kotwiczący. Wszystkie pozostałe wartości w tej kolumnie są przechowywane jako różnica względem rekordu kotwiczącego.

Po zastosowaniu kompresji prefiksów ostatnim etapem w procesie kompresji strony jest kompresja słownikowa. Na tym etapie analizowane są wszystkie wartości przechowywane na stronie w poszukiwaniu duplikatów. Co ciekawe, wartości te są oceniane na podstawie ich reprezentacji binarnej, co sprawia, że proces jest niezależny od typu danych i poprawia stopień kompresji. Pasujące wartości są przechowywane w słowniku na początku strony, a wiersze przechowują jedynie wskaźniki do lokalizacji wartości w słowniku.

Strukturę skompresowanej strony danych pokazano na rysunku 10.12. Jak widać, struktura slotów jest taka sama jak przy kompresji wierszy. Widoczny jest jednak dodatkowy rekord CI. Jest on wstawiany bezpośrednio po nagłówku strony i służy za punkt odniesienia dla kompresji prefiksów oraz jako słownik dla kompresji słownikowej. Licznik zmian jest wykorzystywany do śledzenia, ile razy strona została zaktualizowana, ponieważ może to wpłynąć na efektywność rekordu CI. SQL Server wykorzystuje tę informację do podjęcia decyzji, czy strona wymaga przebudowania.



Rysunek 10.12. Struktura strony z zaimplementowaną kompresją stron

Jak wpłynęła kompresja stron na rozmiar naszej tabeli `ImpressionArchive`? Wcale! Tabela nadal zajmuje 33 687 stron. Dlaczego tak się dzieje? Cóż, w naszym przypadku tabela została zbudowana z użyciem losowych wartości, takich jak identyfikatory GUID i losowo generowane ciągi znaków, co oznacza, że występuje tak mało wspólnych elementów, iż nie ma możliwości kompresji danych.

Zanim wprowadzimy zmiany, możemy wykorzystać procedurę składowaną `sp_estimate_data_compression_savings` do oszacowania rozmiaru tabeli przy różnych poziomach kompresji. W naszym przypadku pomogłoby to podjąć decyzję o zastosowaniu kompresji wierszy zamiast kompresji stron. Byłby to dobry wybór, ponieważ kompresja stron nie zmniejszyła rozmiaru tabeli bardziej niż kompresja wierszy, mimo że wymaga ona więcej cykli procesora do dekompresji strony.

Na listingu 10.20 pokazano, jak użyć procedury składowanej `sp_estimate_data_compression_savings` do oszacowania rozmiaru tabeli w przypadku usunięcia całej kompresji w porównaniu z jej obecnym rozmiarem z zastosowaną kompresją stron.

Listing 10.20. Szacowanie rozmiaru tabeli z kompresją

```
EXEC sp_estimate_data_compression_savings
    @schema_name = 'dbo',
    @object_name = 'ImpressionsArchive',
    @index_id = 1,
    @partition_number = NULL,
    @data_compression = 'none' ;
```

W przypadku dużych tabel w instancjach SQL Servera, w których wydajność jest ograniczona przez operacje wejścia-wyjścia, warto rozważyć kompresję danych jako technikę optymalizacji. Aby kompresja danych była opłacalna, instancja musi być ograniczona przez operacje wejścia-wyjścia, a jednocześnie musi dysponować zapasem mocy obliczeniowej procesora. Przed wdrożeniem kompresji należy ocenić wpływ zarówno kompresji wierszy, jak i stron. Wynika to z faktu, że każda z tych technik może osiągać różne stopnie kompresji w zależności od charakteru danych w tabeli. Kompresja stron zazwyczaj zapewnia wyższy stopień kompresji, ale kosztem dodatkowego obciążenia procesora podczas dekompresji.

10.10. Numer 63 — używanie poziomu izolacji Read Uncommitted

W rozdziale 5. wyjaśniłem, że stosowanie wskazówki NOLOCK jako sposobu na poprawę wydajności jest błędem, ponieważ brak blokad może prowadzić do zwracania niedeterministycznych wyników. Jednak wielu użytkownika SQL Servera popełnia podobną pomyłkę podczas pracy z mechanizmem blokowania na poziomie transakcji.

Poziom izolacji transakcji Read Uncommitted (odczytuj niezatwierdzone) działa podobnie do NOLOCK, ale na poziomie całej transakcji. Nie nakłada żadnych blokad podczas operacji odczytu. Pozwala to uniknąć konfliktów blokad, ale może prowadzić do „brudnych” odczytów, czyli odczytywania danych, które nigdy nie zostały zatwierdzone w bazie danych.

Oznacza to, że poziom izolacji Read Uncommitted jest odpowiedni wyłącznie w przypadku tabel przechowywanych w grupach plików tylko do odczytu, co uniemożliwia zapisywanie do nich danych. Niestety nierzadko można spotkać użytkowników SQL Servera, którzy stosują ten poziom izolacji jako sposób na optymalizację wydajności nawet w przypadku tabel, które dopuszczają aktualizacje.

Aby zrozumieć zjawisko brudnego odczytu, rozważmy następujący przykład. Administrator danych pracuje nad rozwiązaniem pewnych problemów z danymi w bazie MarketingArchive. W tym samym czasie użytkownik generuje raport.

WSKAZÓWKA Aby wykonać ten przykład, powinieneś mieć otwarte dwa okna kwerendy. W pierwszym oknie wykonaj polecenia z listingów 10.21 i 10.23. W drugim oknie wykonaj skrypt z listingu 10.22.

Kwerenda z listingu 10.21 symuluje sytuację, w której administrator danych aktualizuje wartość BidPrice w tabeli ImpressionsArchive dla wpisu o ImpressionID równym 100.

Listing 10.21. Symulacja aktualizacji dokonanej przez administratora danych

BEGIN TRANSACTION ← Wykonaj w pierwszym oknie kwerendy.

```
UPDATE dbo.ImpressionsArchive
SET BidPrice = 1.812
WHERE ImpressionID = 100 ;
```

Kwerenda z listingu 10.22 symuluje sytuację, w której użytkownik bazy danych generuje raport zwracający sumę wszystkich wartości BidPrice powiązanych z daną kampanią.

Listing 10.22. Symulacja kwerendy raportowej

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED ← Wykonaj w drugim oknie kwerendy.
```

```
BEGIN TRANSACTION
```

```
SELECT SUM(BidPrice)
FROM dbo.ImpressionsArchive
WHERE CampaignID = (
    SELECT
        CampaignID
    FROM dbo.ImpressionsArchive
    WHERE ImpressionID = 100
);
```

```
COMMIT
```

Polecenie z listingu 10.23 wycofuje transakcję administratora danych, który zorientował się, że popełnił błąd.

Listing 10.23. Wycofywanie transakcji administratora danych

```
ROLLBACK ;
```

Niestety taka sekwencja zdarzeń oznacza, że wynik kwerendy zwrócony użytkownikowi jest „niepoprawny” w tym sensie, że bazował na danych, które nigdy nie zostały zatwierdzone w bazie. Ponowne wykonanie tej samej kwerendy zwróciłoby inny rezultat.

W zwykłych okolicznościach nie należy stosować poziomu izolacji transakcji Read Uncommitted jako optymalizacji wydajności. Jest on odpowiedni jedynie przy odczycie danych z tabel przechowywanych w grupie plików tylko do odczytu lub w bardzo rzadkich przypadkach, gdy dokładność danych nie ma znaczenia.

10.11. Numer 64 — używanie nadmiernie restrykcyjnych poziomów izolacji

Domyślnym poziomem izolacji transakcji w SQL Serverze jest Read Committed, który chroni przed brudnymi odczytami, ale nadal naraża użytkowników na *odczyty niepowtarzalne*, czyli zjawisko, w którym transakcja odczytuje ten sam wiersz dwukrotnie, ale za każdym razem otrzymuje inną wartość. Ponadto naraża użytkowników na *odczyty fantomowe*, czyli sytuację, w której transakcja dwukrotnie odczytuje zbiór wierszy, ale za każdym razem otrzymuje inną ich liczbę.

Czasem specjaliści od baz danych uznają dane za tak istotne, że postanawiają stosować poziom izolacji Repeatable Read, który chroni zarówno przed odczytami

brudnymi, jak i niepowtarzalnymi, lub poziom izolacji Serializable, który zabezpiecza przed wszystkimi problemami ze spójnością, włącznie z odczytami fantomowymi, jako domyślne ustawienie dla wszystkich transakcji.

Stosowanie poziomów izolacji transakcji Repeatable Read i Serializable jest całkowicie uzasadnione i konieczne w niektórych sytuacjach, na przykład przy wykonywaniu obliczeń aktuarialnych. Jednak często ustawianie tych poziomów izolacji jest przesadą. W wielu typowych przypadkach konfigurowanie tych bardziej restrykcyjnych poziomów izolacji to pomyłka. W zdecydowanej większości sytuacji właściwym rozwiązaniem jest pozostawienie domyślnej konfiguracji Read Committed.

Aby nauczyć się świadomie wybierać poziom izolacji odpowiedni w konkretnych scenariuszach, przyjrzymy się konsekwencjom odczytu niepowtarzalnego i fantomowego. Do wykonania kolejnych dwóch przykładów będziesz potrzebować dwóch okien kwerendy. Każdy listing będzie wskazywał, w którym oknie kwerendy należy uruchomić dane polecenie.

Przyjrzymy się najpierw zjawisku odczytu niepowtarzalnego. Wyobraźmy sobie, że administrator danych znów aktualizuje wiersze w tabeli Impressions ↪ Archive, aby poprawić błędy. Tym razem użytkownik generujący raport będzie korzystał z domyślnego poziomu izolacji transakcji Read Committed. Transakcja użytkownika generującego raport rozpoczyna się od kodu przedstawionego na listingu 10.24.

Listing 10.24. Rozpoczynanie transakcji użytkownika generującego raport

BEGIN TRANSACTION ← Uruchom ten skrypt w pierwszym oknie kwerendy.

```
SELECT SUM(BidPrice)
FROM dbo.ImpressionsArchive
WHERE OperatingSystemID = (
    SELECT OperatingSystemID
    FROM dbo.ImpressionsArchive
    WHERE ImpressionID = 100
) ;
```

Skrypt przedstawiony na listingu 10.25 symuluje sytuację, w której administrator danych poprawia wartość BidPrice dla odsłony o identyfikatorze ImpressionID równym 100. W tym przypadku administrator zatwierdza zmianę natychmiast.

Listing 10.25. Administrator danych poprawia niektóre dane

BEGIN TRANSACTION ← Uruchom ten skrypt w drugim oknie kwerendy.

```
UPDATE dbo.ImpressionsArchive
SET BidPrice = 1.5
WHERE ImpressionID = 100 ;
```

COMMIT

Wreszcie skrypt z listingu 10.26 ilustruje zakończenie transakcji użytkownika generującego raport.

Listing 10.26. Koniec transakcji użytkownika generującego raport

```
SELECT SUM(BidPrice) ← Uruchom ten skrypt w pierwszym oknie kwerendy.
FROM dbo.ImpressionsArchive
WHERE CountryCode = (
    SELECT CountryCode
    FROM dbo.ImpressionsArchive
    WHERE ImpressionID = 100
) ;

COMMIT
```

W przedstawionej sekwencji zdarzeń użytkownik odczytał wartość BidPrice dla odsłony o ImpressionID równym 100 dwukrotnie w dwóch różnych kwerendach. Za drugim razem odczytana wartość BidPrice różniła się od tej z pierwszego odczytu. Żadna z tych wartości nie jest „błędna” — obie były poprawne w momencie ich odczytu z tabeli. Jeśli jednak użytkownik spróbuje uzgodnić wartości z obu kwerend, będzie to mylące i z pewnością niespójne.

W wielu przypadkach tego typu anomalia nie będzie miała znaczenia. Jeśli jednak wykonujemy na przykład regulowane przepisami obliczenia finansowe, może być konieczne ustawienie poziomu izolacji na Repeatable Read. Jeżeli takie anomalie są do przyjęcia, powinniśmy pozostawić domyślny poziom izolacji, czyli Read Committed.

Teraz przyjrzyjmy się zjawisku odczytu fantomowego. Nasz administrator danych wraca, ale tym razem usuwa niepoprawne dane. Skrypt z listingu 10.27 rozpoczyna transakcję użytkownika generującego raport.

Listing 10.27. Początek transakcji użytkownika generującego raport

```
BEGIN TRANSACTION ← Uruchom ten skrypt w pierwszym oknie kwerendy.

SELECT COUNT(*)
FROM dbo.ImpressionsArchive
WHERE OperatingSystemID = (
    SELECT OperatingSystemID
    FROM dbo.ImpressionsArchive
    WHERE ImpressionID = 100
) ;
```

Skrypt z listingu 10.28 symuluje usuwanie wierszy z tabeli przez administratora danych.

Listing 10.28. Transakcja administratora danych

```
BEGIN TRANSACTION ← Uruchom ten skrypt w drugim oknie kwerendy.

DELETE
```

```

FROM dbo.ImpressionsArchive
WHERE ImpressionID IN (
    SELECT TOP 3 ImpressionID
    FROM dbo.ImpressionsArchive
    WHERE OperatingSystemID = (
        SELECT OperatingSystemID
        FROM dbo.ImpressionsArchive
        WHERE ImpressionID = 100
    )
    AND ImpressionID <> 100
);

COMMIT

```

Wreszcie skrypt z listingu 10.29 pokazuje zakończenie transakcji użytkownika generującego raport.

Listing 10.29. Koniec transakcji użytkownika generującego raport

```

SELECT COUNT(DISTINCT ImpressionID) ← Uruchom ten skrypt w pierwszym oknie kwerendy.
FROM dbo.ImpressionsArchive
WHERE OperatingSystemID = (
    SELECT OperatingSystemID
    FROM dbo.ImpressionsArchive
    WHERE ImpressionID = 100
);

COMMIT

```

W tym scenariuszu, gdy użytkownik wykonuje drugą kwerendę, zwraca ona o trzy wiersze mniej niż pierwsza. Jest to znane jako odczyt fantomowy. Z odczytem fantomowym mielibyśmy do czynienia również w przypadku, w którym do tabeli zostałyby dodane nowe wiersze, co spowodowałoby zwrócenie dodatkowych wierszy przez drugą kwerendę.

W wielu sytuacjach odczytu fantomowego nie uważa się za poważny problem. Jeśli mamy do czynienia z takim scenariuszem, jak obliczenia aktuarialne, w którym odczyt fantomowy mógłby być problematyczny, warto rozważyć użycie poziomu izolacji transakcji Serializable. W przeciwnym razie powinniśmy stosować słabszy poziom izolacji, najlepiej Read Committed.

Wyższe poziomy izolacji zapobiegają większej liczbie anomalii, ale mogą prowadzić do problemów z wydajnością spowodowanych rywalizacją o blokady. Mogą nawet prowadzić do *zakleszczeń*, które występują, gdy jedna transakcja oczekuje na zakończenie drugiej, a druga na zakończenie pierwszej i żadna nie może kontynuować. W przypadku zakleszczenia SQL Server wycofuje transakcję, którą uznaje za najmniej kosztowną. Dlatego powinniśmy stosować wyższy poziom izolacji tylko wtedy, gdy jest to rzeczywiście uzasadnione.

10.12. Numer 65 — nieuwzględnianie optymistycznych poziomów izolacji

SQL Server obsługuje dwa typy poziomów izolacji transakcji: pesymistyczne i optymistyczne. Poziomy pesymistyczne wykorzystują blokady, aby zapobiec modyfikacji danych przez inne transakcje. Poziomy optymistyczne przechowują stare wersje wierszy w bazie TempDB. Pozwala to uniknąć anomalii w danych, jednocześnie zapobiegając problemom wydajnościowym związanym ze wzajemnym blokowaniem się operacji odczytu i zapisu.

W poprzednich dwóch podrozdziałach omówiliśmy pesymistyczne poziomy izolacji transakcji. W tym przyjrzymy się dwóm poziomom optymistycznym obsługiwanych przez SQL Server. Te poziomy izolacji to Read Committed Snapshot i Snapshot, odpowiadające poziomom Read Committed i Serializable. Zapewniają one ochronę przed anomaliami w danych.

Te poziomy izolacji przechowują starsze wersje wierszy w bazie TempDB, aż do najstarszej wersji wiersza w ramach najstarszej otwartej transakcji. Dzięki temu transakcje mogą odwoływać się do odpowiedniej wersji wiersza bez konieczności stosowania blokad i oczekiwania, które są charakterystyczne dla pesymistycznych poziomów izolacji.

Niestety, te przydatne poziomy izolacji są często pomijane przez większość administratorów baz danych. Myślę, że głównym powodem jest obawa przed popełnieniem błędu. Jest to zrozumiałe, ponieważ optymistyczna kontrola współbieżności nie jest w żadnym wypadku magicznym rozwiązaniem wszystkich problemów.

Nie ma magicznych rozwiązań

Choć w tym podrozdziale przedstawiam zalety optymistycznej kontroli współbieżności, należy pamiętać, że nie jest to magiczne rozwiązanie. Jak w przypadku większości aspektów SQL Servera, mamy tu do czynienia z pewnym kompromisem.

Wersje wierszy są przechowywane w obszarze TempDB znanym jako Version Store (magazyn wersji). W przypadku intensywnie wykorzystywanych systemów produkcyjnych może to oznaczać znaczne nasilenie operacji wejścia-wyjścia związanych z optymistycznymi poziomami izolacji. Zważywszy, że dwa podrozdziały tego rozdziału poświęcone są metodom redukcji liczby operacji wejścia-wyjścia, nietrudno zauważyć, że może to stanowić potencjalny problem. Trzeba również wziąć pod uwagę dodatkową przestrzeń dyskową, która będzie wykorzystywana do tego celu.

Rozważając zastosowanie mechanizmów takich jak optymistyczna kontrola współbieżności, musimy oceniać każdą sytuację indywidualnie. Na przykład, jeśli nasza baza TempDB jest przechowywana lokalnie na dysku NVMe z dużą ilością wolnego miejsca, to przejście na optymistyczną kontrolę współbieżności prawdopodobnie będzie dobrym pomysłem. Jeśli natomiast baza TempDB znajduje się w przeciążonej macierzy SAN i doświadczamy wąskich gardeł w operacjach wejścia-wyjścia, to wprowadzenie tego mechanizmu może przynieść więcej szkody niż pożytku.

Pomyłką opisywaną w tym podrozdziale jest nieuwzględnienie optymistycznej kontroli współbieżności w konkretnym scenariuszu. W niektórych sytuacjach poziomy izolacji Read Committed Snapshot oraz Snapshot mogą być bardzo przydatnymi narzędziami w naszym arsenale, kiedy próbujemy rozwiązać trudne problemy z wydajnością.

Jeśli mamy problemy z wydajnością spowodowane blokowaniem i wzajemnym wykluczaniem, a jednocześnie musimy zachować wysoki poziom izolacji, zdecydowanie powinniśmy rozważyć zastosowanie poziomów Read Committed Snapshot i (lub) Snapshot. Musimy jednak pamiętać o wymaganiach w zakresie wejścia-wyjścia i unikać wdrażania tego rozwiązania w środowisku, w którym operacje wejścia-wyjścia stanowią wąskie gardło.

Poziom izolacji Read Committed Snapshot można włączyć za pomocą polecenia przedstawionego na listingu 10.30.

Listing 10.30. Włączanie poziomu izolacji Read Committed Snapshot

```
ALTER DATABASE MarketingArchive  
SET READ_COMMITTED_SNAPSHOT ON WITH NO_WAIT ;
```

Po włączeniu poziomu izolacji Read Committed Snapshot nie można używać poziomu Read Committed, a Read Committed Snapshot staje się domyślnym poziomem izolacji dla wszystkich transakcji.

Skrypt z listingu 10.31 pokazuje, jak włączyć poziom izolacji Snapshot dla bazy danych.

Listing 10.31. Włączanie poziomu izolacji Snapshot

```
ALTER DATABASE MarketingArchive  
SET ALLOW_SNAPSHOT_ISOLATION ON ;
```

Ponieważ teraz domyślnym poziomem izolacji jest Read Committed Snapshot, możemy zbadać wiersze w Version Store przez rozpoczęcie dowolnej transakcji, a następnie przeanalizowanie metadanych systemowych. Skrypt z listingu 10.32 rozpoczyna transakcję i aktualizuje wiersz.

Listing 10.32. Rozpoczynanie transakcji z poziomem izolacji Read Committed Snapshot

```
BEGIN TRANSACTION  
  
UPDATE dbo.ImpressionsArchive  
SET BidPrice = BidPrice + 0.1
```

```
WHERE ImpressionID = 100 ;

UPDATE dbo.ImpressionsArchive
SET CostPerMille = CostPerMille + 0.1
WHERE ImpressionID = 100 ;
```

Teraz możemy przejrzeć zawartość Version Store za pomocą kwerendy z listingu 10.33.

Listing 10.33. Badanie Version Store

```
SELECT *
FROM sys.dm_tran_version_store ;
```

COMMIT

Zauważ, że pierwotna wersja wiersza jest przechowywana jako wartość binarna, a jej długość w bajtach jest zapisana w dodatkowej kolumnie.

Każda sytuacja jest inna, a jeśli masz do czynienia z dużym obciążeniem wejścia-wyjścia, to optymistyczne zarządzanie współbieżnością może nie być najlepszym rozwiązaniem. Niemniej jednak pomijanie tej metody w przypadku problemów z wydajnością spowodowanych pesymistycznym podejściem jest pomyłką. W odpowiednich okolicznościach poziomy izolacji Read Committed Snapshot i Snapshot mogą być przydatnymi narzędziami dla administratora bazy danych.

10.13. Numer 66 — rozwiązywanie problemów przez dokładanie sprzętu

Cały ten rozdział jest poświęcony optymalizacji wydajności SQL Servera, ale błędem, który widzę raz za razem, często popełnianym przez firmy zajmujące się wsparciem technicznym, jest brak dogłębnej analizy problemu wydajnościowego i próba rozwiązania go na poziomie sprzętowym.

Takie podejście nigdy nie jest dobrym pomysłem. Główny problem polega na tym, że w przypadku kłopotów z wydajnością dołożenie sprzętu prawdopodobnie nie przyniesie rozwiązania. A nawet jeśli pomoże, efekty będą najpewniej tymczasowe i wkrótce znów pojawi się potrzeba rozbudowy.

Przyjrzyjmy się wyjątkowo źle napisanej kwerendzie przedstawionej na listingu 10.34. Kwerenda ta po prostu aktualizuje kolumnę BidPrice we wszystkich wierszach tabeli MarketingArchive, ale robi to z użyciem kursora.

UWAGA Wykonanie tego skryptu może zająć dużo czasu.

Listing 10.34. Bardzo wolny kursor

```

DECLARE @ImpressionID BIGINT

DECLARE Impressions CURSOR FAST_FORWARD FOR
SELECT ImpressionID
FROM dbo.ImpressionsArchive ;

OPEN Impressions ;

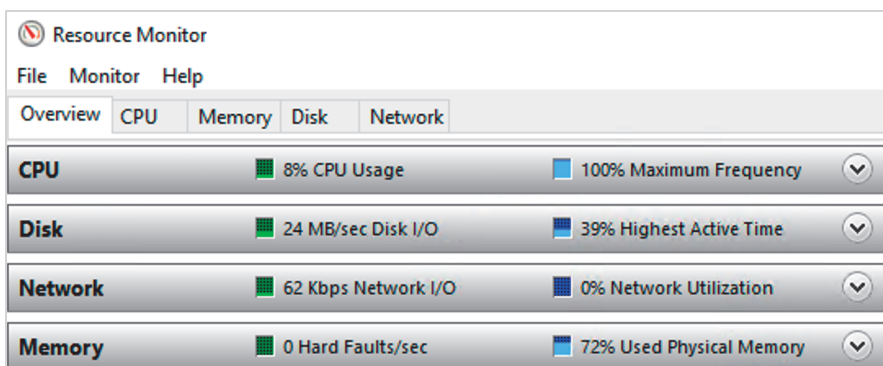
FETCH NEXT FROM Impressions INTO @ImpressionID
WHILE @@FETCH_STATUS = 0
BEGIN
    UPDATE dbo.ImpressionsArchive
    SET BidPrice = BidPrice + 0.1
    WHERE ImpressionID = @ImpressionID ;

    FETCH NEXT FROM Impressions INTO @ImpressionID ;
END

CLOSE Impressions ;
DEALLOCATE Impressions ;

```

Na moim stanowisku testowym kwerenda ta wykonywała się aż 10 minut i 15 sekund. Wykorzystanie zasobów komputera podczas wykonywania kwerendy pokazano na rysunku 10.13.



Rysunek 10.13. Wykorzystanie zasobów w czasie wykonywania kwerendy

Choć jest to nieco przesadzony scenariusz, często rozmawiam z młodszyimi administratorami baz danych, którzy w takiej sytuacji stwierdziliby, że problem z wydajnością jest spowodowany wykorzystaniem procesora, ponieważ wystąpił chwilowy skok do 100%, mimo że później ustabilizowało się na poziomie 8%. Zamiast zbadać przyczynę problemu, najchętniej dodaliby więcej procesorów.

Prawdopodobnie zdajesz sobie sprawę, że problemu tego nie rozwiąże nawet najmocniejszy sprzęt; jedynym sposobem na jego naprawę jest zmodyfikowanie kwerendy. Zoptymalizowaną wersję, która na moim stanowisku testowym wykonuje się w 3 sekundy, pokazano na listingu 10.35.

Listing 10.35. Aktualizowanie BidPrice techniką relacyjną

```
UPDATE dbo.MarketingArchive  
SET BidPrice = BidPrice + 0.1 ;
```

Takie podejście diagnostyczne nigdy nie było przydatne w erze maszyn wirtualnych, a w dobie chmury staje się również kwestią kosztów. Rozwiązywanie problemów z wydajnością poprzez dodawanie sprzętu oznacza niepotrzebne wydatki związane ze zwiększaniem rozmiaru maszyny wirtualnej w chmurze lub zakupem szybszej pamięci masowej.

Oczywiście zdarzają się sytuacje, gdy obciążenie przerasta dostępne możliwości serwera i konieczna jest jego rozbudowa. Pozostaje mieć nadzieję, że zauważymy to i zaplanujemy odpowiednie działania podczas planowania wydajności, o czym mówiliśmy w rozdziale 9.

Zawsze zanim przystąpimy do rozbudowy sprzętu, powinniśmy próbować rozwiązać problem z wydajnością. Mocniejszy sprzęt nie zawsze rozwiązuje problem lub może tylko tymczasowo złagodzić objawy. Zamiast tego powinniśmy dążyć do znalezienia źródła problemu. Możemy wykorzystać techniki optymalizacji omówione w tym rozdziale, upewnić się, że stosujemy najlepsze praktyki programistyczne opisane w rozdziale 5., lub skorzystać z bogatego zestawu funkcji i narzędzi diagnostycznych SQL Servera, aby rozwiązać problem.

Podsumowanie

- Flagi śledzenia służą do włączania i wyłączania określonych funkcji SQL Servera.
- Flagi śledzenia T1117 i T1118 zostały uznane za przestarzałe i nie mają już wpływu na działanie instancji SQL Servera. Zamiast nich należy stosować bardziej precyzyjne alternatywy, takie jak `AUTOGROW_ALL_FILES` i `UNIFORM_PAGE_ALLOCATION`.
- W zdecydowanej większości przypadków warto stosować natychmiastową inicjalizację plików. Należy jej unikać jedynie w środowiskach, które wymagają najwyższego poziomu bezpieczeństwa.
- Opcja blokowania stron w pamięci (Lock Pages In Memory) jest zalecana w większości sytuacji. Istnieją przypadki, gdy należy ją wyłączyć w chmurach prywatnych, jednak dostawcy chmur publicznych zalecają jej używanie.
- Zawsze pozostaw wystarczającą ilość pamięci RAM dla systemu operacyjnego i innych aplikacji działających w serwerze. Jest to szczególnie ważne, jeśli używana jest opcja Lock Pages In Memory.

- Użyj ustawienia Max Server Memory (maksymalna pamięć serwera), aby skonfigurować maksymalną ilość pamięci RAM, która może zostać przydzielona na pulę buforów SQL Servera.
- Staraj się zawsze współpracować z optymalizatorem, a nie działać wbrew niemu. W rzadkich sytuacjach, gdy konieczne jest użycie wskazówek do kwerend, postaraj się pozostawić optymalizatorowi jak najwięcej możliwości wyboru.
- Pamiętaj, aby korzystać z informacji zwrotnych DOP.
- Domyślnie mechanizm informacji zwrotnych DOP jest wyłączony, w przeciwieństwie do innych mechanizmów informacji zwrotnych o kwerendach.
- Rozważ podzielenie dużych tabel na partycje w celu poprawienia wydajności i zmniejszenia obciążenia podsystemu wejścia-wyjścia.
- Partycjonowanie tabel umożliwia eliminowanie partycji, co oznacza, że partycje niezawierające istotnych danych nie są odczytywane.
- Rozważ zastosowanie kompresji danych jako sposobu na poprawę wydajności dużych tabel.
- Kompresja danych najlepiej sprawdza się w przypadku obciążeń ograniczonych przez operacje wejścia-wyjścia, gdy procesor dysponuje wolnymi zasobami.
- Stopień kompresji zależy od danych przechowywanych w tabeli. Aby ocenić efekty kompresji wierszy i stron przed ich wdrożeniem, warto skorzystać z procedury składowanej `sp_estimate_data_compression_savings`.
- Unikaj poziomu izolacji transakcji Read Uncommitted, chyba że pracujesz z tabelami przechowywanymi w grupach plików tylko do odczytu.
- Poziom izolacji Read Uncommitted może prowadzić do brudnych odczytów, podobnie jak stosowanie wskazówki NOLOCK w kwerendach.
- Unikaj stosowania restrykcyjnych poziomów izolacji, takich jak Repeatable Read lub Serializable, chyba że są absolutnie niezbędne. Mogą one prowadzić do konfliktów blokad i zakleszczeń.
- Rozważ zastosowanie optymistycznych poziomów izolacji, jeśli napotykasz problemy z wydajnością spowodowane rywalizacją o blokady.
- Optymistyczne poziomy izolacji to Read Committed Snapshot i Snapshot.

- Optymistyczne poziomy izolacji najlepiej sprawdzają się w środowiskach, w których baza TempDB nie jest ograniczona przez operacje wejścia-wyjścia, a woluminie TempDB jest dużo wolnego miejsca.
- Unikaj rozwiązywania problemów z wydajnością przez dokładanie sprzętu. Zamiast tego postaraj się zdiagnozować źródło problemu i rozwiązać go u podstaw.

11 Indeksy

W tym rozdziale:

- Fragmentacja indeksów
- Konserwacja indeksów
- Jak indeksy współdziałają z procesami ekstrakcji, transformacji i wczytywania danych
- Database Engine Tuning Advisor
- Indeksy kolumnowe

W tym rozdziale najpierw przyjrzymy się typowym błędom i nieporozumieniom związanym z fragmentacją. Zrozumienie fragmentacji indeksów może mieć kluczowe znaczenie dla wydajności bazy danych. Na przykład fragmentacja indeksów jest często uważana za zjawisko jednoznacznie negatywne, jednak zobaczymy, że zbyt mała fragmentacja wewnętrzna może prowadzić do niekorzystnych podziałów stron.

Następnie skupimy się na konserwacji indeksów i przyjrzymy się różnym pomyłkom, które mogą prowadzić do zwiększonej fragmentacji indeksów i obniżenia wydajności. Pomyłki te obejmują zarówno całkowite zaniedbywanie przebudowy indeksów, jak i ich nadmierną, nieprzemyślaną przebudowę. Przeanalizujemy również, w jaki sposób konserwacja indeksów wpływa na aktualizację statystyk.

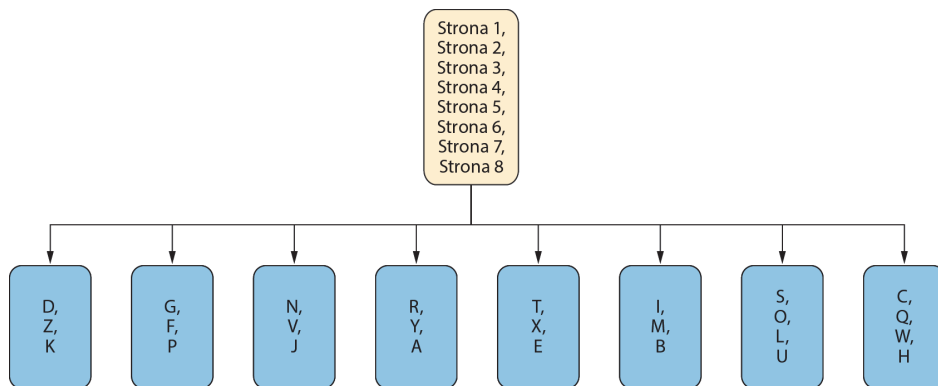
Przyjrzymy się, jak indeksy wpływają na operacje ekstrakcji, transformacji i wczytywania (ang. *extract, transform, load*, ETL). W szczególności skupimy się na operacjach hurtowego wczytywania danych i omówimy, dlaczego strategie

indeksowania mają kluczowe znaczenie dla wydajnego przebiegu procesów ETL. Następnie zajmiemy się narzędziem Database Tuning Advisor i zastanowimy się, dlaczego nadmierne poleganie na tej funkcji może nie być dobrym pomysłem.

Na koniec przyjrzymy się indeksom kolumnowym, które mogą przynieść ogromną poprawę wydajności w przypadku obciążeń analitycznych i hurtowni danych. Mimo że zostały wprowadzone do SQL Servera ponad dekadę temu, wciąż nie są powszechnie stosowane. Omówimy przyczyny tego stanu rzeczy i wyjaśnimy, dlaczego nieużywanie indeksów kolumnowych jest pomyłką.

W tym rozdziale w przykładach kodu będą używane bazy danych Marketing i MarketingArchive, które utworzyliśmy w poprzednich rozdziałach. Zalecam więc korzystanie z tych baz, jeśli chcesz samodzielnie wykonywać przykłady. Zanim jednak zaczniemy, przypomnę pokrótce koncepcje indeksów, które omówiliśmy w rozdziale 4.

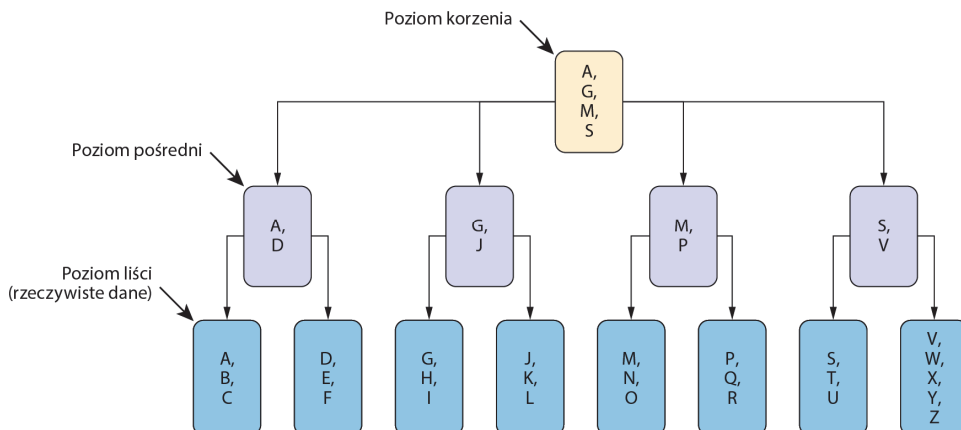
Tabelę utworzoną bez indeksu klastrowego nazywamy *stertą*. Dane na stercie są przechowywane bez określonego porządku. Zamiast korzenia indeksu istnieje prosta mapa alokacji indeksów (ang. *Index Allocation Map*, IAM), która przechowuje listę wszystkich stron przydzielonych stercie. W przypadku większych tabel oznacza to, że SQL Server musi włożyć dużo pracy w znalezienie konkretnej wartości. W praktyce prawie każda kwerenda SELECT wykonana na stercie wymaga odczytania wszystkich stron danych, które tworzą tabelę. Ilustracja sterty znajduje się na rysunku 11.1.



Rysunek 11.1. Sterta przechowuje strony w przypadkowej kolejności

WSKAZÓWKA Kwerendy zawierające słowa kluczowe takie jak TOP nie zawsze wymagają odczytania wszystkich stron tabeli.

Kiedy tworzymy w tabeli indeks klastrowy, SQL Server buduje strukturę B-drzewa, taką jak pokazana na rysunku 11.2. Operacja ta porządkuje strony tabeli według klucza klastrowego. Klucz klastrowy zazwyczaj tworzy się na podstawie klucza głównego tabeli. Jeśli jednak klucz główny jest szeroki, można zbudować indeks klastrowy na innej unikatowej kolumnie.



Rysunek 11.2. Indeks klastrowy tworzy strukturę B-drzewa i porządkuje strony danych

Indeks klastrowy umożliwia efektywniejsze wykonywanie operacji odczytu — różne operacje na indeksach omówimy dalej w tym rozdziale. Poziom liści struktury B-drzewa to faktyczne strony danych tabeli. Oznacza to, że strony danych tabeli są przechowywane w kolejności klucza klastrowego. Z tego powodu możemy mieć tylko jeden indeks klastrowy dla danej tabeli.

Indeks nieklastrowy to struktura B-drzewa zbudowana na innych kolumnach tabeli. SQL Server może wykorzystywać te indeksy do poprawy wydajności operacji takich jak złączenia, filtrowanie i agregacje. Poziom liści indeksu nieklastrowego zawiera wskaźniki do stron danych na sterce lub w indeksie klastrowym. Ponieważ nie wpływa on na kolejność rzeczywistych stron danych, możemy mieć wiele indeksów nieklastrowych w jednej tabeli. W rzeczywistości tabela może obsługiwać do 256 indeksów nieklastrowych, choć zbyt duża ich liczba może negatywnie wpłynąć na operacje zapisu, a także zajmuje miejsce na dysku i potencjalnie w pamięci.

11.1. Numer 67 — zakładanie, że fragmentacja wewnętrzna jest zawsze niekorzystna

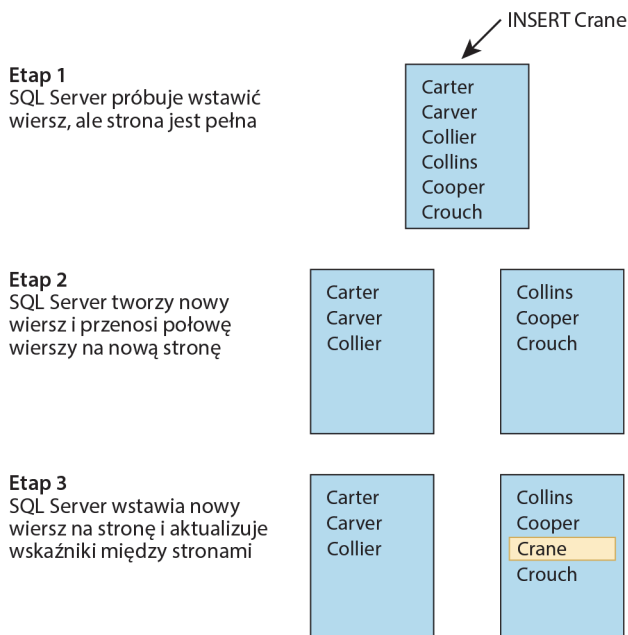
Jedną z pierwszych rzeczy, których uczymy się jako administratorzy baz danych, jest to, że fragmentacja to zło — nie powinniśmy dopuszczać do fragmentacji indeksów, bo ucierpi na tym wydajność. Nie jest to jednak pełny obraz sytuacji, a założenie to może prowadzić do problemów. Aby zrozumieć dlaczego, musimy pamiętać o dwóch rodzajach fragmentacji, które mogą wystąpić. *Fragmentacja zewnętrzna* odnosi się do sytuacji, gdy strony indeksu tracą fizyczny porządek.

Fragmentacja wewnętrzna opisuje ilość wolnego miejsca na stronach indeksu. W tej części skupimy się na tym, dlaczego fragmentacja wewnętrzna może być problematyczna.

Wyobraź sobie, że mamy bazę danych obsługującą obciążenia transakcyjne. Oznacza to, że do bazy danych często wysyłane są instrukcje INSERT, UPDATE i DELETE, w tym wstawienia danych do tabeli zawierającej informacje o klientach. Aby poprawić wydajność, utworzyliśmy indeks nieklastrowy na kolumnie Name. Indeks ten ma ustawienia domyślne, co oznacza, że podczas jego tworzenia serwer SQL Server dążył do całkowitego wypełnienia każdej strony indeksu, pozostawiając miejsce tylko na jeden dodatkowy wiersz.

Użytkownik wykonuje instrukcję INSERT, aby dodać nowego klienta. W rezultacie do tabeli zostaje wstawiony nowy wiersz zawierający m.in. nowe nazwisko. Ponieważ w kolumnie z nazwiskiem istnieje indeks, konieczne jest również dodanie nowego wpisu na stronie indeksu nieklastrowego.

Rysunek 11.3 przedstawia proces, który musi przejść, aby nowy wiersz został wstawiony do tabeli. Najpierw SQL Server próbuje umieścić wartość (w tym przypadku "Crane") na stronie danych, ale odkrywa, że nie ma na niej wystarczająco dużo miejsca. W związku z tym tworzy nową stronę, co powoduje fragmentację zewnętrzną, i przenosi połowę danych z pierwotnej strony na nową. Teraz na pierwotnej stronie jest już wystarczająco dużo miejsca, aby SQL Server mógł wstawić nowy wiersz. Na koniec muszą zostać zaktualizowane wskaźniki między stronami.



Rysunek 11.3. Dodawanie nowego wiersza do pełnej strony indeksu

Ten proces, nazywany *podziałem strony*, jest niekorzystnym rodzajem podziału. Generuje on dodatkowe operacje wejścia-wyjścia i powoduje fragmentację zewnętrzną. Powinniśmy starać się minimalizować występowanie tego typu podziałów stron, jeśli to możliwe.

WSKAZÓWKA Podziały stron mogą być spowodowane nie tylko przez wstawianie, ale także przez aktualizacje danych. Przykładowo, jeśli kolumna typu VARCHAR(60) zawierała wartość o długości 5 znaków, a następnie została zaktualizowana do wartości o długości 60 znaków, to zwiększenie rozmiaru danych może spowodować podział strony. Taką sytuację nazywamy *aktualizacją rozszerzającą*.

W poprzednim przykładzie niepożądany podział strony wystąpił w indeksie nieklastrowym, jednak podziały stron mogą również zachodzić w indeksach klastrowych. Dzieje się tak zwykle, gdy klucz klastrowy nie jest sekwencyjny, ale może to być również spowodowane aktualizacjami rozszerzającymi, nawet gdy klucz klastrowy jest sekwencyjny.

Minipomyłka — podziały stron nie zawsze są złe

Podobnie jak wiele osób uważa, że fragmentacja zawsze jest zła, wiele osób sądzi, że podziały stron są zawsze niekorzystne. W rzeczywistości podział strony to zupełnie normalna operacja w SQL Serverze. Gdyby podziały stron nigdy nie występowały, nie byłoby możliwe dodawanie nowych wierszy do tabeli czy indeksu.

Różnica między dobrym a złym podziałem strony zależy od tego, która strona w indeksie jest dzielona. Jeśli dzielona jest ostatnia strona indeksu z powodu dodania nowego wiersza do pełnego indeksu, wówczas na końcu indeksu alokowana jest nowa strona, na której można wstawiać nowe wiersze. W tym przypadku dane nie są przenoszone na nową stronę. Do tabeli po prostu przydzielana jest nowa strona. Taki podział strony uważa się za korzystny.

W poprzednim przykładzie podział nastąpił jednak w środku indeksu, co doprowadziło do fragmentacji zewnętrznej (strony nie są ułożone po kolei) i niepotrzebnych operacji wejścia-wyjścia. Jest to uznawane za niekorzystny podział strony i powinniśmy starać się unikać takiej sytuacji.

Skoro więc pełne strony powodują niekorzystne podziały, to czy powinniśmy po prostu pogodzić się z wysokim poziomem wewnętrznej fragmentacji we wszystkich indeksach? Niezupełnie. Wiele zależy od specyfiki naszego środowiska.

Wyobraźmy sobie na przykład bazę danych obsługującą obciążenia charakterystyczne dla hurtowni danych. Codziennie w nocy uruchamiany jest proces ETL, który usuwa indeksy, hurtowo wczytuje dane, a następnie odtwarza indeksy. W ciągu dnia na bazie danych nie są wykonywane żadne operacje INSERT, UPDATE ani DELETE. W takim scenariuszu nie ma możliwości wystąpienia podziałów stron, więc prawdopodobnie będziemy chcieli, aby nasze strony były wypełnione w jak największym stopniu.

W środowisku transakcyjnym sytuacja jest nieco bardziej skomplikowana i musimy iść na kompromisy. Jeśli strony mają bardzo niską fragmentację wewnętrzną

(są bardzo zapełnione), wzrasta ryzyko podziału stron. Jeśli jednak mają wysoką fragmentację wewnętrzną (dużo wolnego miejsca), potrzeba więcej stron do przechowywania tej samej ilości danych. Oznacza to, że SQL Server musi odczytywać więcej stron z dysku, co zwiększa liczbę operacji wejścia-wyjścia i skraca czas przebywania stron danych w pamięci podręcznej. W rezultacie spada współczynnik trafień w *pamięci podręcznej buforów*, co z kolei oznacza, że strony będą musiały być częściej odczytywane z dysku.

W związku z tym nie da się jednoznacznie określić zalecanego poziomu wewnętrznej fragmentacji. Właściwy poziom zależy od wielu czynników, począwszy od wydajności odczytu i zapisu podsystemu dyskowego aż po liczbę i rodzaj transakcji wykonywanych w bazie danych. Jako punkt wyjścia można jednak przyjąć, że dla obciążeń charakterystycznych dla hurtowni danych odpowiednie może być wypełnienie stron na poziomie 95 – 100%. W przypadku intensywnie wykorzystywanych systemów transakcyjnych optymalny poziom może wynosić zaledwie 60 – 70%.

Sporo mówiliśmy o fragmentacji wewnętrznej i optymalnej gęstości stron, czyli o tym, jak bardzo strony są *zapełnione*. Nie wyjaśniłem jednak jeszcze, jak skonfigurować docelową gęstość stron. Zajmiemy się tym w następnej kolejności.

SQL Server wykorzystuje ustawienie o nazwie `FILLFACTOR` do określenia docelowej gęstości stron na poziomie liści indeksu. Ma również ustawienie `PAD_INDEX`, które można wykorzystać do kontrolowania gęstości stron na pośrednich poziomach indeksu. Poziom korzenia zawsze składa się z dokładnie jednej strony, więc nie podlega ani wewnętrznej, ani zewnętrznej fragmentacji.

Domyślna wartość parametru `FILLFACTOR` wynosi 0, co oznacza stuprocentowe wypełnienie strony. Jeśli zostanie zastąpiona inną wartością, staje się ona docelowym procentem wypełnienia strony. Parametr `PAD_INDEX` może być ustawiony tylko na `ON` lub `OFF`, przy czym `OFF` jest opcją domyślną. Jeśli ustawimy `PAD_INDEX` na `ON`, to poziom `FILLFACTOR` będzie stosowany również do stron pośrednich. Nie jest możliwe skonfigurowanie innego stopnia wypełnienia dla stron pośrednich i dla stron liści.

Parametr `PAD_INDEX` jest prawdopodobnie mniej istotny niż `FILLFACTOR`, jednak w przypadku dużych indeksów, o wielu poziomach pośrednich, podział strony może rozprzestrzenić się w górę struktury B-drzewa, co znacznie zwiększa jego wpływ. Dlatego warto włączyć `PAD_INDEX` dla indeksów w bardzo dużych tabelach, które obsługują obciążenia transakcyjne i działają mało wydajnie z powodu częstych podziałów stron.

Aby skonfigurować parametr `FILLFACTOR` dla indeksu lub włączyć parametr `PAD_INDEX`, trzeba przebudować indeks. Polecenie z listingu 11.1 ustawia parametr `FILLFACTOR` dla indeksu `ImpressionUID` w tabeli `Marketing.Impressions` bazy danych `Marketing` na 90% oraz włącza opcję `PAD_INDEX`.

Listing 11.1. Ustawienie współczynnika wypełnienia i włączenie indeksowania z wypełnieniem

```
ALTER INDEX ImpressionUID ON marketing.Impressions REBUILD  
WITH (  
    FILLFACTOR = 90,  
    PAD_INDEX = ON  
) ;
```

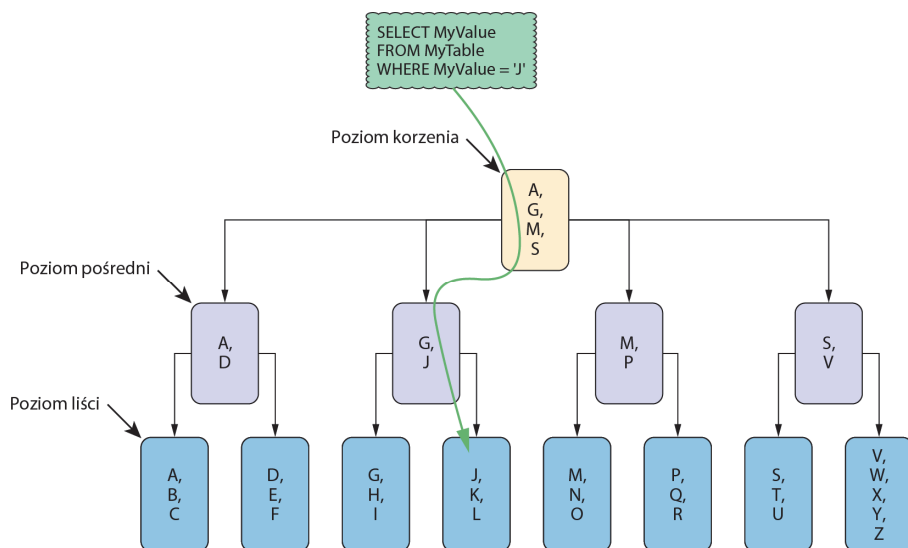
Administratorów baz danych często się uczy, że fragmentacja indeksów zawsze jest zjawiskiem negatywnym. Jednak fragmentacja wewnętrzna może być w rzeczywistości korzystniejsza niż stuprocentowe zapelnienie stron. Wynika to stąd, że nadmiernie wypełnione strony mogą prowadzić do dużej liczby niekorzystnych podziałów stron. Musimy znaleźć równowagę między liczbą podziałów stron a liczbą stron używanych do przechowywania indeksów. Optymalna wartość FILLFACTOR zależy od wymagań konkretnego obciążenia.

11.2. Numer 68 — pogląd, że fragmentacja zewnętrzna powoduje problemy we wszystkich kwerendach

Fragmentacja zewnętrzna powoduje spadek wydajności. To nie podlega dyskusji. Jednak pułapką, w którą często wpadają administratorzy baz danych, jest odruchowe przebudowywanie indeksów w reakcji na problemy z wydajnością zgłaszane przez użytkowników. W niektórych przypadkach może to zająć sporo czasu, a w sytuacjach, gdy fragmentacja indeksów nie jest główną przyczyną problemu, ryzykujemy zmarnowaniem cennego czasu.

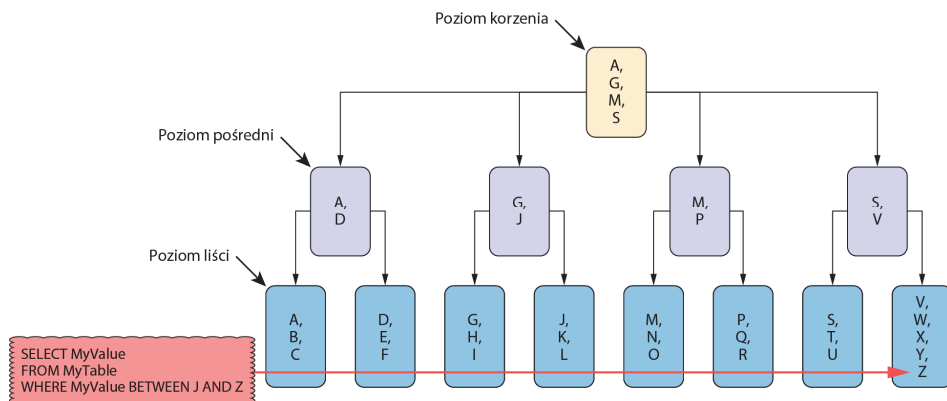
Wyobraź sobie sytuację, w której użytkownicy narzekają na długi czas wykonywania kwerend. Wiele z tych kwerend zwraca wartość skalarną albo pojedynczy wiersz. W takich przypadkach, jak zobaczymy za chwilę, mało prawdopodobne jest, aby przyczyną problemu była fragmentacja, więc przeprowadzenie awaryjnej przebudowy wszystkich indeksów w bazie danych raczej nie pomoże. Co więcej, w zależności od sposobu przebudowy indeksów, możemy jeszcze bardziej pogorszyć wydajność podczas tego procesu konserwacyjnego.

Aby zrozumieć to zagadnienie, musimy poznać różne operacje, które można wykonywać na indeksie. Są to przeszukiwanie, skanowanie i sprawdzanie. Operacja *przeszukiwania* zaczyna się od korzenia indeksu i przechodzi przez każdy poziom struktury B-drzewa, aby znaleźć poszukiwany wiersz. Ten proces, zilustrowany na rysunku 11.4, jest najszybszą metodą pobierania niewielkiej liczby wierszy.



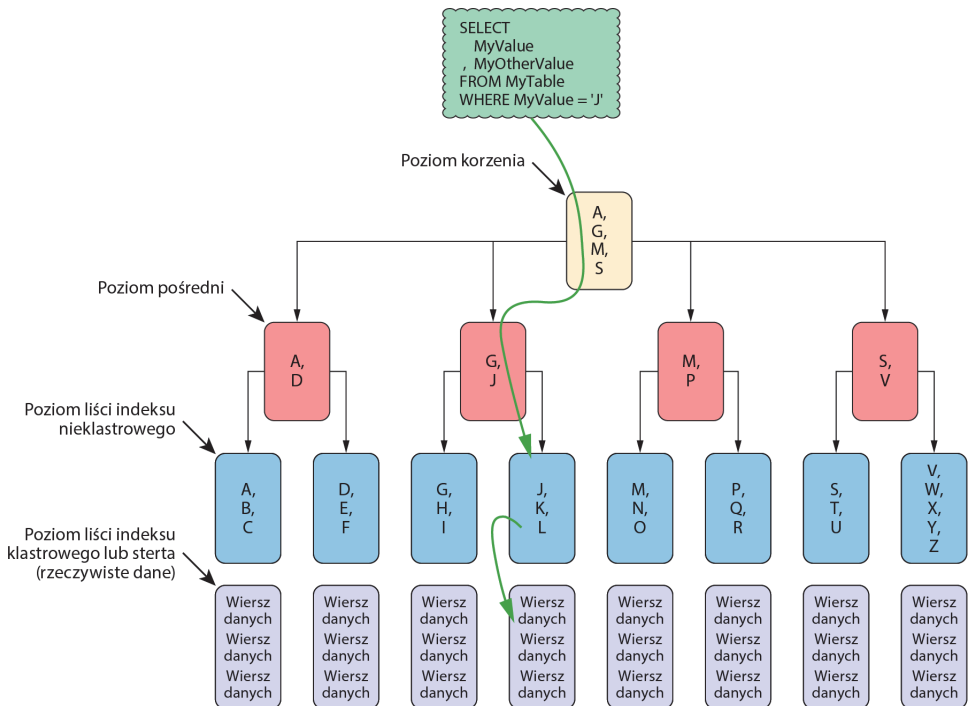
Rysunek 11.4. Przeszukiwanie indeksu

Skanowanie stosuje się, gdy SQL Server musi odczytać większy procent wierszy z indeksu. W takiej sytuacji przeszukiwanie byłoby nieefektywne, dlatego zamiast tego serwer skanuje wszystkie strony na poziomie liści w strukturze B-drzewa. Ilustruje to rysunek 11.5.



Rysunek 11.5. Skanowanie indeksu

Podczas gdy przeszukiwanie i skanowanie indeksu dotyczy zarówno indeksów klastrowych, jak i nieklastrowych, operacja *sprawdzania* indeksu ma zastosowanie tylko wtedy, gdy do realizacji kwerendy wykorzystywany jest indeks nieklastrowy. SQL Server używa indeksu nieklastrowego do wykonywania takich operacji, jak filtrowanie danych przez klauzulę *WHERE*, ale następnie musi wykonać operację przeszukiwania w stronach danych indeksu klastrowego lub sterty, aby pobrać dodatkowe kolumny. Zilustrowano to na rysunku 11.6.

**Rysunek 11.6.** Sprawdzenie indeksu

Oczywiście, jeśli strony indeksu są nieuporządkowane, operacja skanowania indeksu będzie mniej wydajna. Dlatego możemy jednoznacznie stwierdzić, że fragmentacja zewnętrzna prowadzi do obniżenia wydajności skanowania indeksu.

Jeśli jednak wykonujemy operację przeszukiwania, to fakt, że strony na poziomie liści są nieuporządkowane, nie ma wpływu na wydajność kwerendy. Dlatego błędne jest przekonanie, że fragmentacja zewnętrzna wpływa na wydajność operacji przeszukiwania.

To samo dotyczy sprawdzania odwołań do indeksu klastrowego w indeksie nieklastrowym lub sprawdzania w indeksie nieklastrowym kluczy odwołujących się do sterty. Poziom liści indeksu nieklastrowego zawiera wskaźnik do powiązanej strony danych w tabeli. Dlatego fragmentacja nie generuje dodatkowych operacji wejścia-wyjścia podczas takiego sprawdzania.

Wyraźnie widać, że fragmentacja zewnętrzna nie wpływa na wszystkie kwerendy. Jeśli SQL Server wykonuje niewielką liczbę operacji skanowania indeksu, to fragmentacja prawdopodobnie nie jest główną przyczyną problemu. W takiej sytuacji lepiej skupić się na zbadaniu innych potencjalnych źródeł problemów z wydajnością.

Możemy szybko ocenić stosunek skanowań indeksu do innych operacji na indeksach, korzystając z dynamicznego widoku administracyjnego (ang. *dynamic management view*, DMV) `sys.dm_db_index_usage_stats`. Obiekt ten zwraca wiersz

dla każdego indeksu w instancji, który był używany od ostatniego uruchomienia usługi Database Engine. Każdy wiersz zawiera liczbę operacji na indeksie oraz czas ich ostatniego wykonania.

Kwerenda z listingu 11.2 zwraca informacje o indeksach w bazie danych Marketing. Łączy ona wyniki z widokiem katalogowym `sys.indexes`, aby wyświetlić nazwę i typ indeksu.

Listing 11.2. Badanie użycia indeksów według operacji na indeksach

```
SELECT
    OBJECT_NAME(ius.object_id) AS TableName
  , i.name AS IndexName
  , i.type_desc
  , ius.user_seeks
  , ius.user_scans
  , ius.last_system_lookup
  , ius.last_user_seek
  , ius.last_user_scan
  , ius.last_user_lookup
FROM sys.dm_db_index_usage_stats ius
INNER JOIN sys.indexes i
    ON ius.index_id = i.index_id
    AND ius.object_id = i.object_id
WHERE DB_NAME(database_id) = 'Marketing' ;
```

WSKAZÓWKA Widok `sys.dm_db_index_usage_stats` nie zawiera informacji o indeksach zoptymalizowanych pod kątem danych przechowywanych w pamięci. Chociaż ten typ indeksów wykracza poza ramy niniejszej książki, warto wspomnieć, że statystyki ich użycia można znaleźć w widoku `sys.dm_db_xtp_index_stats`.

Nie zakładaj, że fragmentacja wpływa na wszystkie operacje indeksowania. Ma ona negatywny wpływ na skanowanie indeksów, ale nie na operacje przeszukiwania. Dlatego jeśli indeks charakteryzuje się wysokim stosunkiem operacji przeszukiwania, fragmentacja prawdopodobnie nie jest główną przyczyną problemu. Lepiej poświęcić czas na zbadanie innych możliwych przyczyn.

11.3. Numer 69 — reorganizowanie indeksów w celu poprawienia gęstości stron

W SQL Serverze mamy dwie opcje konserwacji indeksów: reorganizację i przebudowę. Podczas przebudowy indeksu SQL Server tworzy nową strukturę B-drzewa i usuwa pierwotną. Pozwala to na niemal idealne usunięcie fragmentacji, ale wiąże się z wysokim zużyciem zasobów. Z drugiej strony, reorganizacja indeksu ma znacznie niższy koszt, ale ograniczone możliwości. Zamiast tworzyć nową

strukturę B-drzewa, zagęszcza strony na poziomie liści do poziomu określonego przez `FILLFACTOR`, próbując usunąć fragmentację wewnętrzną, oraz próbuje zmienić układ stron na poziomie liści, aby usunąć fragmentację zewnętrzną.

Wyobraźmy sobie indeks z celowo niskim współczynnikiem `FILLFACTOR` wynoszącym 60%. W miarę dodawania danych do indeksu strony stopniowo się zapełniają. Jednak ze względu na niską gęstość zapisu na stronach fragmentacja zewnętrzna jest bardzo niewielka.

Często spotykanym błędem popełnianym przez administratorów baz danych w tym scenariuszu jest skupianie się wyłącznie na fragmentacji zewnętrznej i przeprowadzanie reorganizacji indeksu. Problem z takim podejściem polega na tym, że reorganizacja indeksu zagęszcza strony do poziomu określonego przez parametr `FILLFACTOR`, ale nie zmniejsza gęstości stron do tego poziomu.

W rezultacie pomimo niedawnej reorganizacji indeksu wkrótce ulegnie on znacznej fragmentacji zewnętrznej. Dzieje się tak, ponieważ strony pozostają zapełnione po reorganizacji, co doprowadzi do niekorzystnych ich podziałów w najbliższej przyszłości.

Wtedy będziemy musieli przebudować indeks, aby naprawić fragmentację zewnętrzną i zmniejszyć liczbę podziałów stron. Oznacza to, że w krótkim czasie wykonamy zarówno reorganizację, jak i przebudowę indeksu, podczas gdy moglibyśmy po prostu od razu go przebudować.

Poziom fragmentacji wewnętrznej i zewnętrznej możemy sprawdzić za pomocą dynamicznej funkcji administracyjnej (ang. *dynamic management function*, DMF) `sys.dm_db_index_physical_stats`. Funkcja ta przyjmuje parametry opisane w tabeli 11.1.

Tabela 11.1. Parametry funkcji `sys.dm_db_index_physical_stats`

| Parametr | Opis |
|-------------------------------|--|
| <code>database_id</code> | Identyfikator bazy danych, dla której chcemy uzyskać wyniki. NULL zwraca informacje o wszystkich bazach danych. |
| <code>object_id</code> | Identyfikator obiektu tabeli, dla której chcemy uzyskać wyniki. NULL zwraca informacje o wszystkich tabelach. |
| <code>index_id</code> | Identyfikator indeksu, dla którego chcemy uzyskać wyniki. NULL zwraca informacje o wszystkich indeksach. |
| <code>partition_number</code> | Numer partycji, dla której chcemy uzyskać wyniki. NULL zwraca informacje o wszystkich partycjach. |
| <code>mode</code> | Może być ustawiony na <code>LIMITED</code> , <code>SAMPLED</code> lub <code>DETAILED</code> (ograniczony, próbkowany lub szczegółowy). <code>LIMITED</code> skanuje tylko poziom korzenia i poziomy pośrednie B-drzewa. <code>SAMPLED</code> skanuje 1% stron w tabeli*, a <code>DETAILED</code> skanuje każdą stronę indeksu. |

* Jeśli tabela ma mniej niż 10 000 stron, zostanie użyty tryb szczegółowy.

Funkcja zwraca jeden wiersz dla każdego poziomu każdego indeksu w zakresie. Kluczowe kolumny dla określenia poziomu fragmentacji to `avg_fragmentation_in_percent`, która pokazuje poziom fragmentacji zewnętrznej, oraz `avg_page_space_used_in_percent`, która informuje o gęstości stron.

WSKAZÓWKA Choć kolumny `avg_fragmentation_in_percent` i `avg_page_space_used_in_percent` mają kluczowe znaczenie dla określenia poziomu fragmentacji, inne kolumny również są ważne dla prawidłowej interpretacji danych. Omówimy to szerzej przy okazji następnej pomyłki.

Kwerenda przedstawiona na listingu 11.3 pokazuje, jak uzyskać informacje o wewnętrznej i zewnętrznej fragmentacji dla wszystkich indeksów w tabeli `Marketing.Impressions` bazy danych `Marketing`. Kolumna `index_level` odnosi się do poziomu w strukturze B-drzewa, gdzie 0 oznacza poziom liści.

Listing 11.3. Określanie poziomu fragmentacji indeksu

```
DECLARE @object_id BIGINT
SET @object_id = (
    SELECT object_id
    FROM sys.objects
    WHERE name = 'Impressions'
) ;
```

← Przepisuje zmiennej wartość `object_id` tabeli `Impressions`.

```
SELECT
    OBJECT_NAME(ips.Object_id) AS TableName
    , i.name AS IndexName
    , ips.avg_fragmentation_in_percent
    , ips.avg_page_space_used_in_percent
    , index_level
FROM sys.dm_db_index_physical_stats(
    DB_ID('Marketing'),
    @object_id,
    NULL,
    NULL,
    'DETAILED'
) ips
```

← Zwraca poziomy fragmentacji z `sys.dm_db_index_physical_stats`.

```
INNER JOIN sys.indexes i
    ON ips.index_id = i.index_id
    AND ips.object_id = i.object_id
```

← łączy z widokiem systemowym `sys.indexes`, aby zwrócić nazwę indeksu. Złączenie musi być wykonane zarówno na `object_id`, jak i `index_id` (unikatowym w obrębie tabeli).

```
ORDER BY
    ips.object_id
    , ips.index_id
    , index_level DESC ;
```

Reorganizacja indeksu może być skutecznym sposobem na zmniejszenie niewielkiej fragmentacji zewnętrznej, a także wewnętrznej, w sposób mało obciążający system. Jeśli jednak wymagana jest niska gęstość stron w celu uniknięcia ich podziałów, ważne jest, aby przy planowaniu konserwacji indeksów brać pod uwagę zarówno gęstość stron, jak i fragmentację zewnętrzną. Poziom fragmentacji indeksu można sprawdzić za pomocą dynamicznego widoku administracyjnego `sys.dm_db_index_physical_stats`.

11.4. Numer 70 — błędne interpretowanie statystyk fragmentacji

Administratorom baz danych zdarza się narzekać, że „przebudowa indeksu nie usuwa fragmentacji”. Co gorsza, niektórzy używają tego jako wymówki, by w ogóle nie przebudowywać indeksów. W rzeczywistości przebudowa indeksów z pewnością usuwa fragmentację, a jeśli ktoś myśli inaczej, to dlatego, że błędnie interpretuje wyniki funkcji `sys.dm_db_index_physical_stats`. Weźmy na przykład kwerendę z listingu 11.4, która ilustruje błąd popełniany przez administratorów. Kwerenda ta pobiera dane dla tabeli `Impression` z funkcji `sys.dm_db_index_physical_stats`. Ponieważ wyniki zawierają jeden wiersz dla każdego poziomu B-drzewa, kwerenda oblicza średnią wewnętrzną fragmentację dla każdego indeksu w tabeli.

Listing 11.4. Pomyłkowe agregowanie statystyk fragmentacji wewnętrznej

```
SELECT
    i.name AS IndexName
    , AVG(ips.avg_page_space_used_in_percent)
      AS AveragePageDensity
FROM sys.dm_db_index_physical_stats(
    DB_ID('Marketing')
    , OBJECT_ID('marketing.Impressions')
    , NULL
    , NULL
    , 'DETAILED'
) ips
INNER JOIN sys.indexes i
    ON ips.index_id = i.index_id
    AND ips.object_id = i.object_id
GROUP BY i.name
ORDER BY IndexName;
```

Uśrednia dane dla poziomu korzenia, poziomu pośredniego i liści struktury B-drzewa.

Jeśli założymy, że indeksy zostały właśnie przebudowane, a ich współczynnik `FILLFACTOR` ustawiono na 0 (co oznacza 100% wypełnienia — wystarczająco dużo wolnego miejsca na jeden wiersz), wyniki mogą Cię zaskoczyć. Moje rezultaty znajdziesz na rysunku 11.7, ale Twoje mogą się nieco różnić.

| | IndexName | AveragePageDensity |
|---|--------------------------------|--------------------|
| 1 | ImpressionUID | 66.2356972242814 |
| 2 | PK__Impressi__AE00637B967C6A3E | 58.5592661230541 |

Rysunek 11.7. Średnia agregacja zewnętrzna w przypadku agregacji wszystkich poziomów B-drzewa

Jak widać, średnia gęstość stron dla indeksów 1 i 2 wynosi, odpowiednio, 66% i 58%, podczas gdy docelowa wartość to 100%. Na pierwszy rzut oka mogłoby się wydawać, że fragmentacja wewnętrzna nie została usunięta, ale wynika to z tego, że zadajemy „niewłaściwe pytanie”.

Przyjrzyjmy się temu dokładniej, skupiając się na indeksie ImpressionUID i wykonując kwerendę przedstawioną na listingu 11.5. Ta kwerenda nie agreguje danych. Zwraca jeden wiersz dla każdego poziomu indeksu, a tym razem dodaliśmy do złączenia nazwę indeksu, aby ograniczyć liczbę zwracanych wierszy (moglibyśmy też użyć klauzuli WHERE). Zauważ, że uwzględniliśmy również kolumny index_level, page_count i record_count z sys.dm_db_index_physical_stats. Dzięki temu uzyskamy znacznie więcej informacji kontekstowych.

Listing 11.5. Zwracanie szczegółowych informacji wraz z kontekstem

```
SELECT
    i.name AS IndexName
    , ips.avg_page_space_used_in_percent
      AS PageDensity
    , ips.index_level
    , ips.page_count
    , ips.record_count
FROM sys.dm_db_index_physical_stats(
    DB_ID('Marketing')
    , OBJECT_ID('marketing.Impressions')
    , NULL
    , NULL
    , 'DETAILED'
) ips
INNER JOIN sys.indexes i
ON ips.index_id = i.index_id
AND ips.object_id = i.object_id
AND i.name = 'ImpressionUID';
```

Pobiera surowe dane bez agregacji. W rezultacie zwracany jest jeden wiersz dla każdego poziomu indeksu.

Wyniki, jakie uzyskałem, przedstawiłem na rysunku 11.8, choć Twoje rezultaty mogą się różnić.

| | IndexName | PageDensity | index_level | page_count | record_count |
|---|---------------|------------------|-------------|------------|--------------|
| 1 | ImpressionUID | 99.922942920682 | 0 | 3832 | 999999 |
| 2 | ImpressionUID | 91.8777983691623 | 1 | 17 | 3832 |
| 3 | ImpressionUID | 6.90635038299975 | 2 | 1 | 17 |

Rysunek 11.8. Szczegóły fragmentacji wewnętrznej pokazujące fragmentację na każdym poziomie

Jak widać, poziom 0 (czyli poziom liści) ma docelową gęstość stron. Zawiera też 3832 strony. Poziom 1 (poziom pośredni) ma gęstość stron wynoszącą 91%, ale liczy tylko 17 stron. Poziom 2 (poziom główny) ma gęstość stron wynoszącą zaledwie 6%, ale składa się z pojedynczej strony.

Poziom główny ma tak niską gęstość stron, bo nie zawiera wystarczającej liczby wierszy, aby wypełnić więcej niż 6% strony. Mała liczba stron wpływa

również na średnią gęstość stron na poziomie pośrednim. Przy uśrednianiu jedna niepełna strona daje tu znacznie wyższy procent, niż kiedy niepełna jest 1 strona na 3832.

Oznacza to, że jeśli zagregujemy średnie dla każdego poziomu indeksu, wyniki będą zniekształcone i może się wydawać, że średnia gęstość strony spadła do 66%, podczas gdy w rzeczywistości znacząca średnia gęstość wynosi 99%, co jest całkowicie akceptowalne.

Zachowaj ostrożność przy interpretacji wyników zwracanych przez `sys.dm_db_index_physical_stats`. Podwójne agregowanie rezultatów może w wielu przypadkach prowadzić do błędnych wniosków dotyczących naszych indeksów. To z kolei może prowadzić do problemów, takich jak niepotrzebna konserwacja indeksów lub nawet błędne przekonanie, że możemy całkowicie zrezygnować z ich konserwacji. W większości przypadków powinniśmy skupić się jedynie na fragmentacji na poziomie 0, który zawsze jest poziomem liści indeksu.

11.5. Numer 71 — nieprzebudowywanie indeksów

Użytkownicy SQL Servera, którzy nie przebudowują indeksów, zwykle usprawiedliwiają to tym, że baza danych jest używana przez całą dobę i nie ma okna czasowego na konserwację. W związku z tym konserwacja indeksów nie jest możliwa — SQL Server blokuje obiekty na czas przebudowy, uniemożliwiając użytkownikom dostęp do tabeli. W niektórych przypadkach, gdy administratorzy baz danych kierują się taką filozofią, stosują jako zamiennik reorganizację indeksów. W innych sytuacjach konserwacja indeksów po prostu nie jest wykonywana.

Takie podejście jest pomyłką i prawie zawsze prowadzi do pogorszenia wydajności z powodu fragmentacji wewnętrznej i zewnętrznej, co z kolei skutkuje nieoptymalną pracą systemu podczas skanowania indeksów. Znacznie lepszym rozwiązaniem jest korzystanie z przebudowy indeksów online. Funkcja ta została wprowadzona już w 2005 roku, ale wielu administratorów baz danych nadal jej nie zna.

Podczas wykonywania operacji indeksowania online, zamiast utrzymywania restrykcyjnych blokad przez cały czas trwania operacji, na początku zakładana jest blokada stabilności schematu, a na końcu blokada modyfikacji schematu. Obie te blokady są utrzymywane tylko przez krótki czas. Warto jednak zauważyć, że jeśli blokada schematu zostanie zablokowana przez inne transakcje, sama zablokuje transakcje oczekujące za nią w kolejce. W głównej fazie operacji

indeksowania jedyną utrzymywaną blokadą jest intencyjna współdzielona blokada tabeli, która nie blokuje innych transakcji.

Warto również zwrócić uwagę, że przy przebudowie indeksów online powinniśmy ustawić parametr MAXDOP na 1. MAXDOP określa liczbę rdzeni procesora, które będą wykorzystane do wykonania operacji. Ustawienie MAXDOP na 1 oznacza, że zostanie użyty tylko jeden rdzeń. Ze względu na sposób przydzielania fragmentów indeksu do procesorów podczas przebudowy online, jeśli opcja ALLOW_PAGE_LOCKS jest włączona (co jest ustawieniem domyślnym), istnieje ryzyko zwiększenia fragmentacji zewnętrznej zamiast jej usunięcia. Z kolei wyłączenie ALLOW_PAGE_LOCKS spowoduje, że użytkownicy indeksu nie będą mogli przechodzić z blokad wierszy na blokady stron, co oznacza, że zostanie nałożonych znacznie więcej blokad na obiekty.

UWAGA Przebudowa indeksu online trwa dłużej niż odpowiadająca jej przebudowa offline.

Poniższe polecenie (listing 11.6) pokazuje, jak przeprowadzić przebudowę indeksu ImpressionUID w trybie online.

Listing 11.6. Przebudowa indeksu w trybie online

```
ALTER INDEX ImpressionUID
ON marketing.Impressions REBUILD
WITH(
    ONLINE = ON,
    MAXDOP = 1
) ;
```

UWAGA W dalszej części tego rozdziału będę używać terminu „okno konserwacji” w odniesieniu do okresu zmniejszonej aktywności użytkowników, w przeciwieństwie do całkowitego wyłączenia systemu.

Od wersji SQL Server 2017 możliwe jest wstrzymywanie i wznowianie przebudowy indeksów online, jeśli wykracza ona poza okno konserwacji. Polecenie przedstawione na listingu 11.7 wstrzyma operację, jeśli jej wykonywanie zajmie dłużej niż 1 minutę.

Listing 11.7. Przebudowa indeksu w trybie online z możliwością wznowienia

```
ALTER INDEX ImpressionUID
ON marketing.Impressions REBUILD
WITH(
    ONLINE = ON,
    RESUMABLE = ON,
    MAX_DURATION = 1,
    MAXDOP = 1
) ;
```

Umożliwi to wznowienie przebudowy indeksu w następnym oknie konserwacji za pomocą polecenia przedstawionego na listingu 11.8.

Listing 11.8. Wznawianie przebudowy Indeksu w trybie online

```
ALTER INDEX ImpressionUID
ON marketing.Impressions RESUME
WITH(
    MAXDOP = 1
) ;
```

Jeśli nie ma wstrzymanej przebudowy indeksu oczekującej na wznowienie, polecenie zakończy się niepowodzeniem. Dlatego zalecam użycie bloku TRY...CATCH w tym procesie, jak pokazano na listingu 11.9. Skrypt ten spróbuje wznowić przebudowę indeksu. Jeśli nie ma dostępnej wstrzymanej przebudowy, rozpocznie nową operację przebudowy.

Listing 11.9. Wznawianie wstrzymanej przebudowy lub rozpoczynanie nowej

```
BEGIN TRY
    ALTER INDEX ImpressionUID
    ON marketing.Impressions RESUME
    WITH(
        MAXDOP = 1
    ) ;
END TRY
BEGIN CATCH
    ALTER INDEX ImpressionUID
    ON marketing.Impressions REBUILD
    WITH(
        ONLINE = ON,
        RESUMABLE = ON,
        MAX_DURATION = 1,
        MAXDOP = 1
    ) ;
END CATCH
```

Zaniedbywanie przebudowy indeksów prowadzi do fragmentacji zewnętrznej i powstawania stron o gęstości wyższej niż docelowa. Oba te problemy mogą negatywnie wpływać na wydajność kwerend. Aby temu zaradzić, warto rozważyć przebudowę indeksów w trybie online. Takie operacje trwają dłużej niż przebudowa offline, ale nie blokują transakcji użytkowników przez cały czas trwania procesu. Można je również wstrzymać pod koniec okna konserwacji i wznowić w kolejnym.

11.6. Numer 72 — bezkrytyczne przebudowywanie wszystkich indeksów

Konserwacja indeksów jest istotna, ale wiąże się ze zwiększonym wykorzystaniem zasobów. Aby więc zminimalizować to obciążenie, powinniśmy przebudowywać tylko te indeksy, które tego wymagają. Pomyłką, którą czasem popełniają administratorzy baz danych, jest przebudowywanie wszystkich indeksów bez względu na poziom ich fragmentacji. Przykładem takiego podejścia może być prosty skrypt przedstawiony na listingu 11.10, który mógłby być zaplanowany do uruchomienia jako zadanie w narzędziu SQL Server Agent. Skrypt ten tworzy listę poleceń `ALTER INDEX...REBUILD` w formacie XML, a następnie przekształca ją w jeden skrypt w formacie `NVARCHAR(MAX)` przed jego wykonaniem.

WSKAZÓWKA Więcej informacji na temat unikania użycia kursorów w działaniach administratora bazy danych znajdziesz w rozdziale 9.

Listing 11.10. Bezkrytyczne przebudowywanie wszystkich indeksów

```
DECLARE @SQL NVARCHAR(MAX) ;
```

```
SET @SQL = (
    SELECT ' ALTER INDEX ' +
        i.name +
        ' ON ' + s.name +
        '.' +
        o.name +
        ' REBUILD ;'
    FROM sys.indexes i
    INNER JOIN sys.objects o
        ON i.object_id = o.object_id
    INNER JOIN sys.schemas s
        ON s.schema_id = o.schema_id
    WHERE i.type_desc <> 'HEAP'
        AND o.type_desc = 'USER_TABLE'
    FOR XML PATH('')
) ;
```

Użyto tylko filtrów `HEAP` i `USER_TABLE`, co oznacza, że zostaną przebudowane wszystkie indeksy we wszystkich tabelach użytkownika.

```
EXEC(@SQL) ;
```

Takie podejście może być szczególnie problematyczne w przypadku dużych baz danych z wieloma indeksami, ponieważ powoduje długotrwałe i intensywne wykorzystanie zasobów. Zamiast tego zaleca się przebudowywanie tylko indeksów o wysokim stopniu fragmentacji.

Skrypt przedstawiony na listingu 11.11 rozszerza poprzedni przykład, dodając do kwerendy widok `sys.dm_db_index_physical_stats`, co pozwala na analizę poziomów fragmentacji indeksów oraz liczby stron tworzących indeks. Dzięki temu możemy zastosować dodatkowe filtry dla liczby stron, fragmentacji

zewnętrznej i gęstości stron. Filtr gęstości stron sprawdza, czy średnia gęstość strony jest większa niż docelowy współczynnik wypełnienia indeksu, ponieważ tego problemu nie da się rozwiązać poprzez reorganizację indeksu.

Listing 11.11. Przebudowa indeksów w zależności od fragmentacji

```

DECLARE @SQL NVARCHAR(MAX) ;

SET @SQL = (
    SELECT
        ' ALTER INDEX ' +
            i.name +
            ' ON ' +
            s.name +
            '.' + o.name +
            ' REBUILD ; '
    FROM sys.dm_db_index_physical_stats(
        DB_ID(),
        NULL,
        NULL,
        NULL,
        'DETAILED'
    ) ips
    INNER JOIN sys.indexes i
        ON i.object_id = ips.object_id
        AND i.index_id = ips.index_id
    INNER JOIN sys.objects o
        ON i.object_id = o.object_id
    INNER JOIN sys.schemas s
        ON s.schema_id = o.schema_id
    WHERE i.type_desc <> 'HEAP'
        AND o.type_desc = 'USER_TABLE' ←——— Nadal odfiltrowujemy tabele systemowe i sterty.
        AND (
            ips.avg_fragmentation_in_percent > 20 OR ←——— Odfiltrowujemy indeksy,
            ips.avg_page_space_used_in_percent >      które nie mają problemów
            CASE                                         z fragmentacją.
                WHEN i.fill_factor = 0
                THEN 100
                ELSE i.fill_factor
            END #B
        )
        AND ips.page_count > 1000
        AND ips.index_level = 0 ←——— Odfiltrowujemy małe indeksy i poziomy inne niż liście.
    FOR XML PATH('')
) ;

EXEC(@SQL) ;

```

UWAGA Poziom fragmentacji zewnętrznej, który powinien wywoływać przebudowę, zależy od profilu obciążenia konkretnej aplikacji. Użyta tutaj wartość 20% ma charakter jedynie poglądowy i nie należy jej traktować jako zalecanej najlepszej praktyki.

Nie powinniśmy przebudowywać wszystkich indeksów, ponieważ niepotrzebnie zużywa to zasoby serwera. Zamiast tego powinniśmy skupić się na przebudowie tylko tych indeksów, które tego wymagają, biorąc pod uwagę poziom ich fragmentacji i gęstość stron.

11.7. Numer 73 — aktualizowanie statystyk po przebudowaniu indeksów

Wiele lat temu napisałem na blogu wpis o najgorszej procedurze konserwacyjnej na świecie. Była ona zmodyfikowaną wersją procedury odkrytej właśnie przeze mnie w firmie, w której wówczas pracowałem. Uruchamiała się codziennie w nocy i wykonywała następujące zadania:

1. Przebudowa wszystkich indeksów.
2. Aktualizacja wszystkich statystyk.
3. Zmniejszenie rozmiaru bazy danych.

Wyjaśniłem już, że rutynowe zmniejszanie baz danych nie jest dobrym pomysłem, ponieważ prowadzi do niemal idealnej fragmentacji. Łatwo więc zrozumieć, dlaczego wykonywanie tej operacji zaraz po przebudowie wszystkich indeksów byłoby wyjątkowo złym rozwiązaniem. W tym podrozdziale przyjrzymy się jednak, dlaczego aktualizacja wszystkich statystyk tuż po przebudowie indeksów również nie jest dobrą praktyką.

Aby zrozumieć, dlaczego tak jest, musisz najpierw poznać sposób aktualizacji statystyk. Domyślnie, gdy uruchomimy polecenie `UPDATE STATISTICS` na wybranej tabeli, zaktualizuje ono wszystkie statystyki utworzone dla jej kolumn i indeksów. Aktualizacja ta wykorzysta domyślną wielkość próbki. Wielkość ta jest określana według następujących reguł:

- Jeśli rozmiar tabeli jest mniejszy niż 8 MB: skanuj 100% wierszy.
- Jeśli rozmiar tabeli przekracza 8 MB: pobierz próbkę od 10% do 30% wierszy, w zależności od całkowitej liczby wierszy.

Jeśli chcemy zaktualizować wszystkie statystyki w bazie danych, możemy użyć systemowej procedury przechowywanej o nazwie `sp_updatestats`. Uruchomienie procedury `sp_updatestats` bez parametrów spowoduje wykonanie kursora, który przejdzie przez wszystkie tabele i wykona polecenie `UPDATE STATISTICS` z wartościami domyślnymi. W rezultacie zostaną zaktualizowane statystyki wszystkich kolumn i indeksów dla tabel większych niż 8 MB z użyciem próbki o wielkości od 10% do 30% danych.

Musimy też wziąć pod uwagę, że podczas przebudowy indeksu jego statystyki są automatycznie aktualizowane na podstawie pełnej próbki (100% wierszy). Oznacza to, że po przebudowie indeksu jego statystyki zależą od wszystkich wierszy. Jeśli później zaktualizujemy statystyki tego indeksu, używając wartości domyślnych, w rzeczywistości obniżymy ich jakość, jednocześnie niepotrzebnie zużywając zasoby systemu.

Co powinniśmy zrobić inaczej? Cóż, w większości przypadków automatyczna aktualizacja statystyk jest akceptowalnym rozwiązaniem. Istnieje opcja automatycznego aktualizowania statystyk w razie potrzeby, a także dodatkowa opcja aktualizacji asynchronicznej, która zapobiega blokowaniu kwerendy wywołującej aktualizację statystyk. Chociaż opcja asynchroniczna oznacza, że kwerenda, która wywołała aktualizację, nie skorzysta z zaktualizowanych statystyk, to kolejne kwerendy już z nich skorzystają. Zapobiega to również blokowaniu kwerendy, która wywołała aktualizację, do czasu zakończenia aktualizacji statystyk.

Automatyczne aktualizowanie statystyk jest domyślnie włączone, ale jeśli zostało wyłączone, możemy je ponownie aktywować za pomocą polecenia z listingu 11.12.

Listing 11.12. Włączanie automatycznej aktualizacji statystyk

```
ALTER DATABASE Marketing
SET AUTO_UPDATE_STATISTICS ON ;
```

Jeśli chcemy, aby statystyki były aktualizowane asynchronicznie (co oczywiście ma tę wadę, że kwerenda, która wywołała aktualizację, nie skorzysta z nowych statystyk), możemy to zrobić za pomocą polecenia przedstawionego na listingu 11.13. Należy jednak pamiętać, że opcja ta wymaga również włączenia funkcji `AUTO_UPDATE_STATISTICS`. W przeciwnym razie nie będzie miała żadnego efektu.

Listing 11.13. Włączanie asynchronicznej aktualizacji statystyk

```
ALTER DATABASE Marketing
SET AUTO_UPDATE_STATISTICS_ASYNC ON ;
```

Jeśli mamy konkretny problem z wydajnością i musimy zadbać o to, aby kwerendy zawsze korzystały z najnowszych statystyk, możemy użyć polecenia `UPDATE STATISTICS` dla określonego zestawu tabel. Jeśli działanie to ma być zsynchronizowane z przebudową indeksów, możemy zastosować słowo kluczowe `COLUMNS`, aby ograniczyć aktualizację tylko do statystyk kolumn z pominięciem statystyk indeksów. Odpowiedni przykład dla tabeli `marketing.Impressions` przedstawiono na listingu 11.14. Słowo kluczowe `FULLSCAN` spowoduje użycie 100% danych do aktualizacji statystyk.

Listing 11.14. Aktualizacja statystyk kolumn ograniczona do wybranej tabeli

```
UPDATE STATISTICS marketing.Impressions
WITH
    FULLSCAN
    , COLUMNS ;
```

OSTRZEŻENIE Przed ręczną aktualizacją statystyk należy rozważyć kompromis między korzyściami wynikającymi z aktualnych statystyk a kosztem ponownej kompilacji kwerend.

Jeśli znajdziemy się w sytuacji, w której musimy zaktualizować statystyki kolumn dla wszystkich tabel w bazie danych, możemy zastosować podobne podejście jak przy przebudowie indeksów. Pokazano to na listingu 11.15.

Listing 11.15. Aktualizowanie statystyk wszystkich kolumn w bazie danych

```
DECLARE @SQL NVARCHAR(MAX) ;

SET @SQL = (
    SELECT
        ' UPDATE STATISTICS ' +
        s.name +
        '.' +
        t.name +
        ' WITH FULLSCAN, COLUMNS ; '
    FROM sys.tables t
    INNER JOIN sys.schemas s
        ON t.schema_id = s.schema_id
    FOR XML PATH('')
) ;

EXEC(@SQL) ;
```

WSKAZÓWKĄ W przypadku dużej liczby tabel kwerendę tę można ulepszyć przez dodanie filtrowania według schematu lub wzorca nazwy tabeli.

Powinniśmy unikać ręcznej aktualizacji statystyk, chyba że mamy konkretny scenariusz, który tego wymaga. Jeśli ręczna aktualizacja statystyk jest konieczna, nie powinniśmy aktualizować statystyk nowo przebudowanych indeksów, ponieważ może to prowadzić do pogorszenia jakości statystyk.

11.8. Numer 74 — nieoptymalizowanie konserwacji indeksów zgodnie z własnymi potrzebami

W miarę rozwoju SQL Servera wprowadzono szereg ulepszeń związanych z indeksami, które pozwalają dostosować ich konserwację do specyfiki środowiska. Mimo to rzadko spotyka się administratorów baz danych, którzy w pełni wykorzystują te możliwości. Mniej doświadczeni lub przypadkowi administratorzy zazwyczaj stosują najbardziej podstawowe metody przebudowy indeksów. Jest to jednak pomyłka, ponieważ pomijają w ten sposób optymalizacje, które mogłyby

znacząco wpłynąć na wydajność operacji konserwacyjnych lub wykorzystanie indeksów.

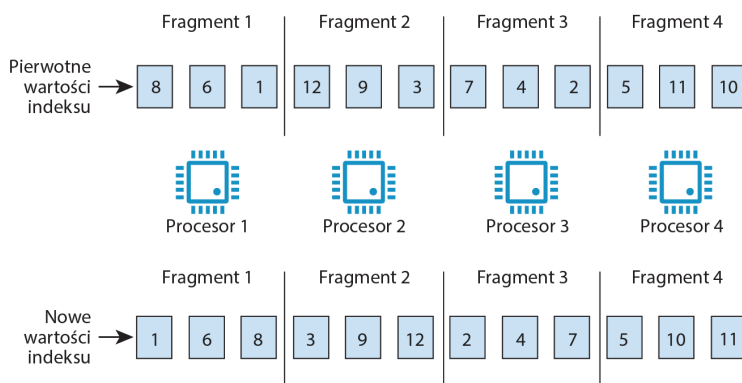
Omówiliśmy już niektóre z tych ulepszeń, takie jak przebudowa indeksów online czy możliwość wznowiania operacji na indeksach. Istnieją jednak jeszcze inne, rzadziej stosowane udoskonalenia, które warto rozważyć. W tym podrozdziale przyjrzymy się ustawieniu MAXDOP, które jest rzadko określane przy przebudowie indeksów. Rozważymy również wykorzystanie opcji SORT_IN_TEMPDB. Na koniec omówimy opcję OPTIMIZE_FOR_SEQUENTIAL_KEY.

11.8.1. Parametr MAXDOP

Wielu administratorów baz danych zna parametr MAXDOP, który określa maksymalną liczbę procesorów, jaką SQL Server może wykorzystać do wykonania kwerendy, która według algorytmicznych obliczeń przekracza próg kosztu zrównoleglenia. Jednak wielu z nich nie zdaje sobie sprawy z konsekwencji tego ustawienia w kontekście przebudowy indeksów.

Jeśli nie określimy parametru MAXDOP dla operacji przebudowy indeksu, SQL Server użyje domyślnego ustawienia MAXDOP dla bazy danych. Pomijanie tego parametru przy przebudowie indeksów jest pomyłką, ponieważ należy wziąć pod uwagę kompromisy wydajnościowe.

Aby to zrozumieć, spójrzmy na diagram z rysunku 11.9. Przedstawia sytuację przy parametrze MAXDOP ustawionym na 4 i pokazuje, jak indeks zostaje podzielony na fragmenty, z których każdy jest przypisywany do jednego z czterech procesorów. Następnie fragmenty te są łączone z powrotem w celu utworzenia nowego indeksu.



Rysunek 11.9. Równoległa przebudowa indeksu powoduje utworzenie fragmentu indeksu na każdym procesorze

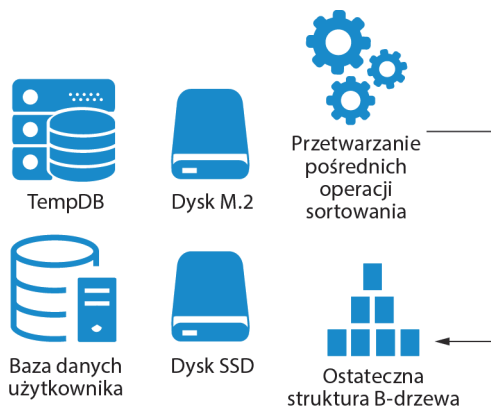
Jak widać, choć wartości zostały poprawnie uporządkowane w obrębie każdego fragmentu, nadal występuje fragmentacja w indeksie, ponieważ nie uwzględniono kolejności wartości między fragmentami. Dlatego powinniśmy

świadomie określać optymalną wartość MAXDOP do wykorzystania podczas przebudowy indeksów, biorąc pod uwagę potrzeby naszego środowiska. Jeśli mamy bardzo krótkie okno czasowe na konserwację, a nasze indeksy szybko ulegają fragmentacji, użycie wielu procesorów do przebudowy pozwoli nam skrócić czas jej wykonania. Jeśli jednak nie mamy ograniczeń czasowych, przeprowadzenie przebudowy jako operacji jednowątkowej zmniejszy końcowy poziom fragmentacji, a tym samym poprawi wydajność operacji skanowania indeksu.

11.8.2. Opcja SORT_IN_TEMPDB

`SORT_IN_TEMPDB` to opcja przebudowy indeksu, której potencjał rzadko jest w pełni wykorzystywany. Gdy jest ona aktywna, pośrednie operacje sortowania są wykonywane w bazie tymczasowej TempDB, a nie w bazie danych użytkownika, w której sortowanie odbywa się domyślnie. W nielicznych przypadkach, gdy spotykałem się z zastosowaniem bazy TempDB, administratorzy baz danych używali jej do obejścia problemu niewystarczającej ilości miejsca na dysku z danymi, uniemożliwiającej przebudowę większych indeksów. W takiej sytuacji, gdy TempDB znajduje się na innym woluminie, administratorzy przenoszą tymczasowe dane na inny dysk. To rozwiązanie może mieć sens w przypadku fizycznych serwerów SQL Servera, ale dla maszyn wirtualnych lub chmurowych lepszym wyjściem jest po prostu zwiększenie przestrzeni dyskowej na woluminie, który ma za mało miejsca.

Jednak często pomijaną zaletą opcji `SORT_IN_TEMPDB` jest jej wpływ na wydajność operacji konserwacyjnych. W środowiskach, w których baza TempDB znajduje się na szybkim nośniku, takim jak lokalny dysk M.2, a pliki danych przechowywane są na wolniejszych dyskach SSD, opcja `SORT_IN_TEMPDB` może skrócić czas operacji przebudowy dużych indeksów. Proces ten zilustrowano na rysunku 11.10.



Rysunek 11.10. *Optymalizacja wydajności pośrednich sortowań przez wykonywanie ich na szybszym nośniku pamięciowym*

Kompromis, który musimy tutaj rozważyć, dotyczy przestrzeni dyskowej. Z jednej strony zmniejszamy wymagania w zakresie miejsca na bazę danych użytkownika, ale z drugiej strony zwiększamy ogólne zapotrzebowanie na pamięć masową. Podczas wykonywania pośrednich operacji sortowania w bazie danych użytkownika obszary używane do sortowania są zwalniane mniej więcej w tym samym tempie, w jakim są przydzielane do nowej struktury indeksu. Gdy jednak sortujemy wyniki w TempDB, potrzebujemy wystarczająco dużo miejsca w TempDB do przechowywania całego pośredniego sortowania, a jednocześnie wystarczająco dużo miejsca w bazie danych użytkownika do przechowywania końcowej struktury B-drzewa.

11.8.3. Opcja **OPTIMIZE_FOR_SEQUENTIAL_KEY**

Klucze indeksów mogą być sekwencyjne lub niesekwencyjne. Przykładem sekwencyjnego klucza indeksu jest klucz klastrowy zbudowany na kolumnie typu `IDENTITY`. Wartości w takiej kolumnie (o ile nie zresetujemy licznika) będą stale rosnać. Z kolei przykładem klucza niesekwencyjnego może być klucz indeksu klastrowego oparty na kolumnie typu `GUID`. Nowe wartości wstawiane do takiego indeksu mogą wymagać umieszczenia w dowolnym miejscu struktury w celu zachowania porządku indeksu.

Gdy mamy do czynienia z indeksem sekwencyjnym, w którym występuje duża liczba operacji wstawiania, możemy napotkać problemy z wydajnością spowodowane rywalizacją o zatraski stron na ostatniej stronie indeksu. Może dojść do sytuacji, w której liczba operacji wstawiania jest tak duża, że tworzy się kolejka zatrasków stron, a wszystkie operacje są wykonywane w tempie najwolniejszego żądania. Na przykład, jeśli jedna operacja wstawiania zostanie opóźniona z powodu konieczności podziału strony, wszystkie operacje ustawione za nią będą musiały czekać na przyznanie zatrasku strony. Zjawisko to nazywane jest *rywalizacją o ostatnią stronę podczas wstawiania*.

Wielu administratorów baz danych nie wie, że w wersji SQL Server 2019 wprowadzono nową funkcję o nazwie `OPTIMIZE_FOR_SEQUENTIAL_KEY`. Jeśli utworzymy indeks z tą opcją, SQL Server zastosuje optymalizacje mające na celu złagodzenie problemów wydajnościowych.

Kiedy opcja ta jest włączona, SQL Server wykorzystuje mechanizm kontroli przepływu, który analizuje wątki przed momentem, w którym żądają one blokady strony. Mechanizm ten ocenia stan wątku oraz procesora, na którym wątek jest uruchomiony. Na podstawie tych informacji przyznaje priorytet wątkom, które prawdopodobnie zakończą się w pojedynczym cyklu procesora, co zwiększa przepustowość systemu.

Bardzo ważne jest, aby korzystać z tej opcji tylko wtedy, gdy mamy konkretny przypadek użycia, który tego wymaga. W sytuacji, do której została zaprojektowana

— kiedy indeksy mają klucz sekwencyjny, a liczba wątków wymagających blokad znacznie przekracza dostępną liczbę procesorów — opcja ta może przynieść znaczną poprawę wydajności. Jednak w przypadku indeksów, które nie borykają się z problemami ograniczonej przepustowości, użycie tej opcji może prowadzić do pogorszenia wydajności, nawet jeśli klucz jest sekwencyjny.

Polecenie przedstawione na listingu 11.16 pokazuje, jak włączyć opcję dla indeksu `ImpressionUID` w tabeli `marketing.Impressions`. Zwróć uwagę, że instrukcja `ALTER INDEX` wykorzystuje klauzulę `SET`. Opcja `SET` jest stosowana natychmiast i nie ma potrzeby przebudowywania indeksu.

Listing 11.16. Włączanie opcji `OPTIMIZE_FOR_SEQUENTIAL_KEY`

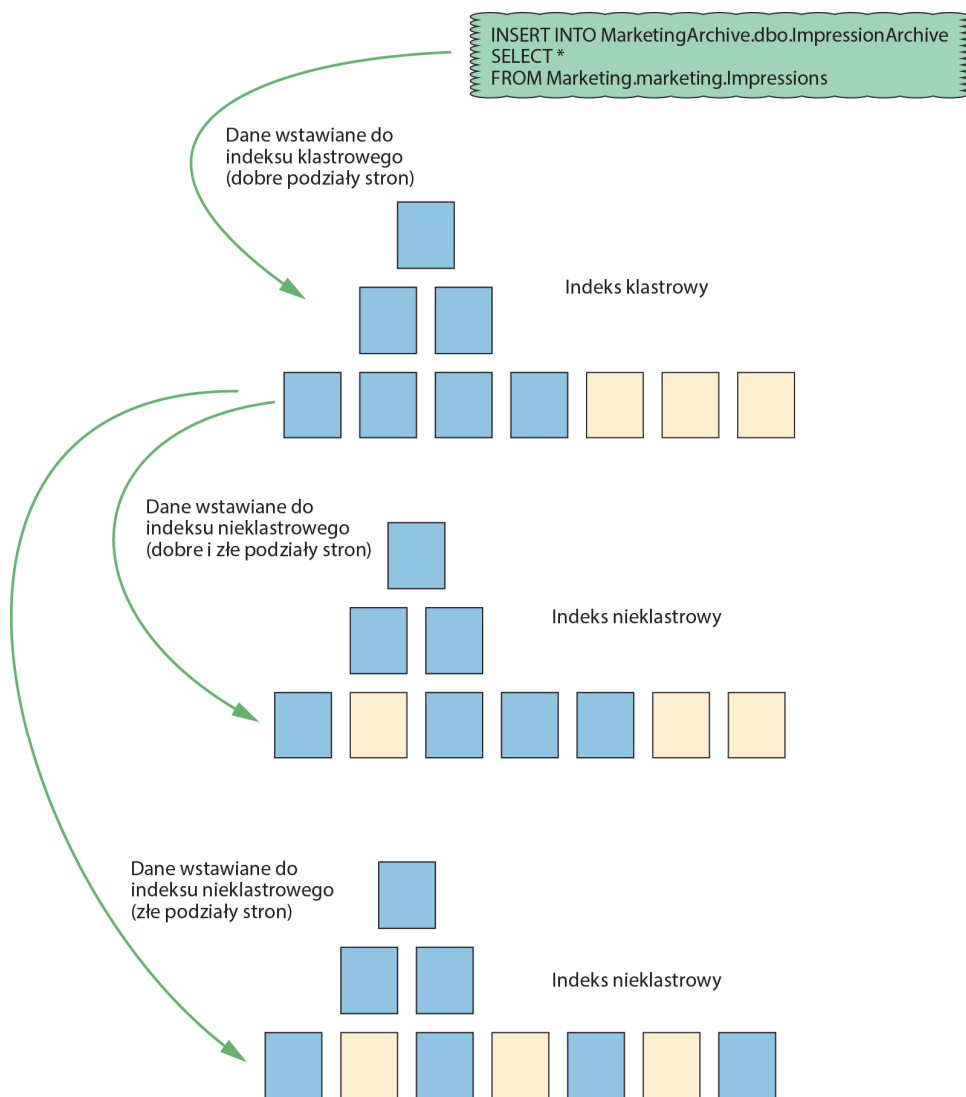
```
ALTER INDEX ImpressionUID
ON marketing.Impressions
SET (
    OPTIMIZE_FOR_SEQUENTIAL_KEY = ON
) ;
```

11.9. Numer 75 — niewyłączanie indeksów podczas hurtowego wczytywania danych

Dobrze utrzymane indeksy i odpowiednia strategia indeksowania są kluczowe dla zapewnienia optymalnej wydajności odczytu danych. Wiąże się z tym jednak pewna wada — negatywny wpływ na wydajność operacji zapisu. Częstoą pomyłką jest hurtowe wczytywanie danych do tabeli mającej wiele indeksów nieklastrowych.

Rozważmy tabelę `ImpressionArchive` w bazie danych `MarketingArchive`. Może się zdarzyć, że do tej tabeli zostaną wstawione miliony wierszy podczas jednego wczytywania. Wczytywanie to będzie wymagać tysięcy operacji wejścia-wyjścia w celu przydzielenia nowych stron dla tabeli i zapisania danych.

Rozważmy jednak sytuację, w której utworzono dwa indeksy nieklastrowe na potrzeby raportowania. W takim przypadku zaktualizowane muszą zostać nie tylko strony danych tabeli, ale także każdy z indeksów nieklastrowych. Rozmiar wiersza w indeksie nieklastrowym jest mniejszy niż w klastrowym, ponieważ zawiera on tylko klucz indeksu nieklastrowego, ewentualne dodatkowe kolumny oraz wskaźnik do indeksu klastrowego, w przeciwieństwie do pełnego wiersza danych zawartego w indeksie klastrowym. Mimo to przy dużej operacji wstawiania danych podziały stron są nieuniknione i znacznie bardziej prawdopodobne jest, że będą to niekorzystne podziały stron. Proces ten zilustrowano na rysunku 11.11.



Rysunek 11.11. Wstawienie nowych wierszy do tabeli z indeksami nieklastrowymi

Możemy sobie wyobrazić, ile dodatkowych operacji wejścia-wyjścia (i czasu) będą wymagały aktualizacje indeksów nieklastrowych. Zamiast stosować to podejście, które prawdopodobnie zwiększy również fragmentację zewnętrzną indeksów nieklastrowych, często bardziej wydajne jest wyłączenie indeksów, a następnie ich przebudowanie po zakończeniu wczytywania danych.

WSKAZÓWKA Warto również zauważyć, że wymagane aktualizacje indeksów nieklastrowych będą miały również wpływ na operacje wejścia-wyjścia potrzebne do zapisywania dziennika transakcji podczas wykonywania operacji INSERT.

Skrypt z listingu 11.17 pokazuje, jak wyłączyć wszystkie indeksy nieklastrowe w tabeli ImpressionsArchive, wykonać operację INSERT, a następnie przebudować te indeksy. Warto zauważyć, że w razie wyłączenia indeksu klastrowego tabela stałaby się niedostępna. Dlatego zamiast używać słowa kluczowego ALL, wykorzystujemy dynamiczny skrypt oparty na metadanych z sys.indexes do wyłączenia tylko indeksów nieklastrowych. Przy przebudowie indeksów użyjemy jednak słowa kluczowego ALL, co oznacza, że przebudowany zostanie również indeks klastrowy. Gdybyśmy nie chcieli przebudowywać indeksu klastrowego, musielibyśmy zastosować to samo podejście co przy wyłączaniu indeksów, czyli skrypt oparty na metadanych.

Listing 11.17. Wyłączanie i ponowne włączanie indeksów

```
DECLARE @SQL NVARCHAR(MAX) ;

SET @SQL = (
    SELECT
        ' ALTER INDEX ' + name + ' ON ImpressionsArchive DISABLE ; '
    FROM sys.indexes
    WHERE object_id = OBJECT_ID('ImpressionsArchive')
        AND type > 1
    FOR XML PATH ('')
) ;

EXEC(@SQL) ;
GO

INSERT INTO MarketingArchive.dbo.ImpressionsArchive (
    ImpressionUID,
    ReferralURL,
    CookieID,
    CampaignID,
    RenderingID,
    CountryCode,
    StateID,
    BrowserVersion,
    OperatingSystemID,
    BidPrice,
    CostPerMille,
    EventTime
)
SELECT
    ImpressionUID
    , ReferralURL
    , CookieID
    , CampaignID
    , RenderingID
    , CountryCode
    , StateID
    , BrowserVersion
    , OperatingSystemID
    , BidPrice
    , CostPerMille
```

```
, EventTime
FROM Marketing.marketing.Impressions ;
GO

ALTER INDEX ALL ON dbo.ImpressionsArchive REBUILD ;
GO
```

Choć indeksy nieklastrowe poprawiają wydajność operacji odczytu, mogą negatywnie wpływać na wydajność operacji INSERT i UPDATE. W przypadku hurtowego wczytywania danych korzystne może być wyłączenie i ponowne włączenie indeksów zamiast przeprowadzania wymaganych aktualizacji indeksów podczas procesu wczytywania.

11.10. Numer 76 — nadmierne poleganie na narzędziu Database Engine Tuning Advisor

Database Engine Tuning Advisor (DTA) to graficzne narzędzie dostępne z menu *Tools* w SQL Server Management Studio, które może pomóc administratorom baz danych w tworzeniu i usuwaniu indeksów, indeksowanych widoków, strategii partycjonowania oraz potencjalnych zmian w fizycznym projekcie bazy danych. W tym podrozdziale skupimy się na wskazówkach dotyczących strategii indeksowania.

Częstą pomyłką, którą popełniają mniej doświadczeni administratorzy baz danych, jest uruchomienie narzędzia DTA dla danego obciążenia, a następnie wdrożenie wszystkich zaleceń bez głębszej analizy. Problem z takim podejściem polega na tym, że nie bierze ono pod uwagę obciążeń nieuwzględnionych w dostarczonych próbkach, a co ważniejsze, nie odzwierciedla wiedzy biznesowej.

Uruchamiając DTA, wybieramy bazę danych (lub bazy), dla której chcemy otrzymać rekomendacje, a następnie przekazujemy obciążenie do analizy. Obciążenie to może mieć formę pliku (lub tabeli) śledzenia SQL, możemy też wskazać magazyn planów, a jeśli zaimplementowano tabelę Query Store, możemy przeprowadzić analizę na jej podstawie. Oczywiście kwerendy, które nie znajdują się w dostarczonym obciążeniu, nie będą brane pod uwagę. Oznacza to, że istnieje ryzyko pominięcia kluczowych dla biznesu kwerend.

Ponadto narzędzia tego typu nigdy nie będą dysponować wiedzą biznesową, jaką posiada doświadczony administrator bazy danych lub programista. Wyobraźmy sobie na przykład krytyczny proces wykonywany na koniec miesiąca, który trwa długo i ma krótkie okno czasowe na przetwarzanie ETL. Jeśli dostarczymy

plik śledzenia SQL, który nie obejmuje nocy, kiedy uruchomiono proces na koniec miesiąca, narzędzie DTA może zalecić usunięcie indeksu, co spowoduje, że proces przekroczy swoje następne okno czasowe.

Praktyczny przykład

Najbardziej uderzający przykład tej pomyłki, z jakim się spotkałem i jaki widziałem, miał miejsce około dziesięciu lat temu. Nowy administrator baz danych w firmie, w której pracowałem, uruchomił DTA i wdrożył wszystkie jego zalecenia bez ich przejrzenia czy dogłębnego zrozumienia.

Problem polegał na tym, że jako obciążenie administrator przekazał pamięć podręczną planów kilka godzin po ponownym uruchomieniu usługi Database Engine. Nie był świadomy, że pamięć podręczna planów jest opróżniana przy każdym restarcie instancji.

W efekcie usunął on około 90% indeksów ze wszystkich baz danych na serwerze. Co gorsza, działo się to w czasach, gdy obiekty SQL Servera nie były jeszcze rutynowo przechowywane w systemie kontroli wersji. Ostatecznie naprawa tego problemu zajęła cały weekend!

Narzędzie DTA przydaje się do generowania pomysłów odnośnie do tego, jak możemy ulepszyć naszą strategię indeksowania. Ma ono jednak pewne ograniczenia i jego zalecenia nigdy nie powinny być wdrażane bezkrytycznie. Zawsze należy oceniać wyniki i wykorzystywać wiedzę biznesową, aby ustalić, jak wdrożenie tych zaleceń może wpłynąć na procesy, które nie zostały uwzględnione w analizowanym obciążeniu.

11.11. Numer 77 – nieużywanie indeksów kolumnowych

Do tej pory skupialiśmy się na tradycyjnych indeksach wykorzystujących strukturę B-drzewa do organizacji danych. A gdybym powiedział, że istnieje inny rodzaj indeksu, zwany indeksem kolumnowym (ang. *columnstore*), który może zapewnić stukrotną poprawę wydajności kwerend typowych dla hurtowni danych? Brzmi imponująco, prawda? A gdybym dodał, że programiści i administratorzy hurtowni danych nadal rzadko korzystają z indeksów kolumnowych? Co jest tego przyczyną? Przecież wszyscy chcielibyśmy, żeby nasze kwerendy wykonywały się 100 razy szybciej, prawda? Owszem, i właśnie dlatego niewykorzystywanie ich może być dużym błędem. Zanim zagłębimy się w przyczyny tego stanu rzeczy, przyjrzyjmy się bliżej, czym właściwie jest indeks kolumnowy.

Komplikacje związane z przechowywaniem danych

Optymalizacje przechowywania danych w SQL Serverze oznaczają, że niektóre elementy mogą być zapisywane poza wierszem. Na przykład ciąg znaków dłuższy niż 8000 bajtów zostanie przeniesiony do innej jednostki alokacji przeznaczonej dla wartości przekraczających limit. Ponadto, jeśli zastosowano funkcje takie jak strumienie plikowe (ang. *filestream*), dane mogą być przechowywane nawet poza silnikiem bazy danych. Dla uproszczenia przyjmiemy jednak, że wszystkie elementy danych są przechowywane w ramach stron danych tabeli.

Mając na uwadze to istotne zastrzeżenie, możemy powiedzieć, że w tradycyjnym indeksie dane są organizowane poprzez przechowywanie wierszy na stronach. W związku z tym minimalną zawartością pojedynczej strony danych jest cały wiersz danych. Na poziomie liści indeksu klastrowego znajdują się więc wiersze danych. W korzeniu i na poziomach pośrednich indeksu klastrowego, a także na wszystkich poziomach indeksu nieklastrowego, strony danych nadal zawierają wiersze danych, ale te wiersze po prostu wskazują na inne strony danych.

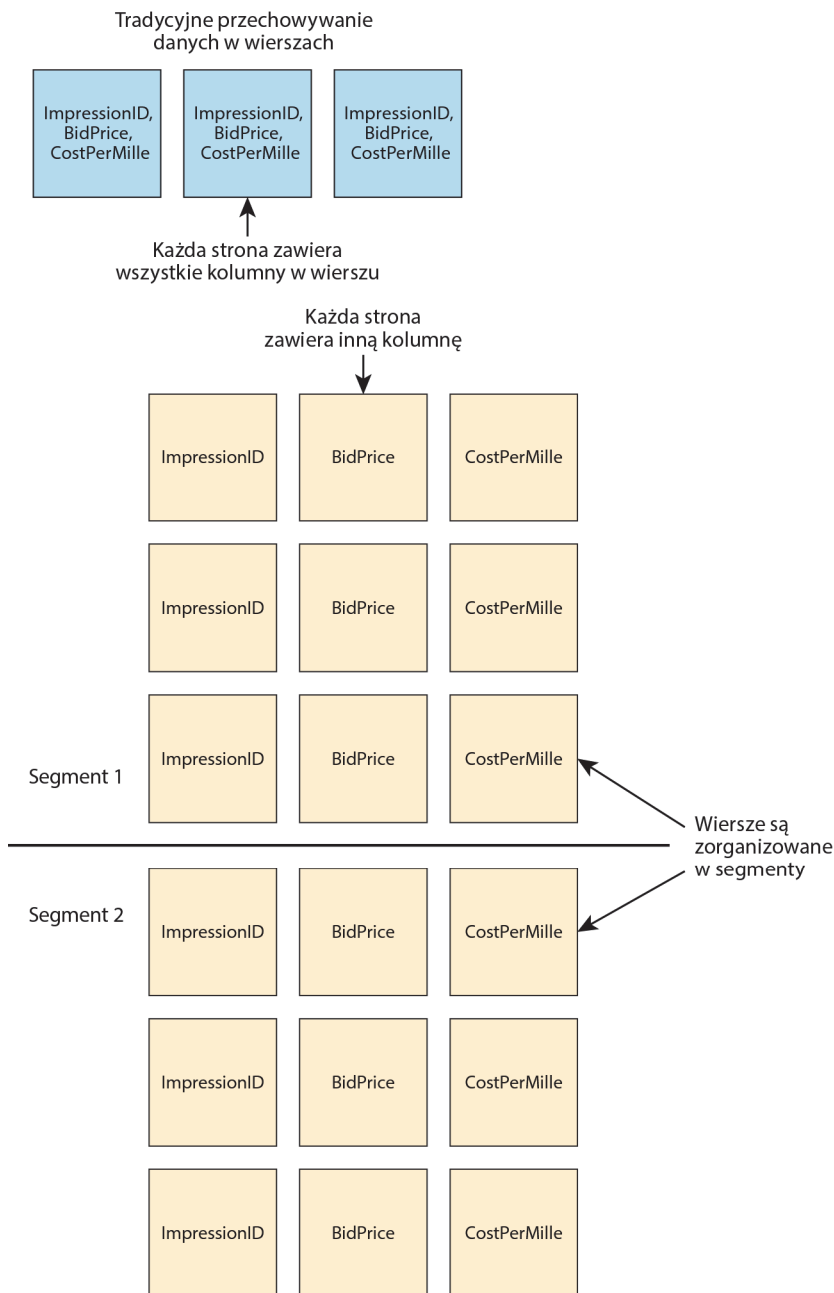
W indeksie kolumnowym organizacja danych jest odwrócona. Dane nie są zorganizowane w wiersze, ale w kolumny. Strony bazy danych przechowują zbiory wartości z poszczególnych kolumn, a nie zbiory wierszy.

Indeks kolumnowy jest podzielony na segmenty, z których każdy zawiera od 102 400 do 1 048 576 wierszy. Każdy segment zawiera metadane opisujące przechowywane w nim dane, co pozwala silnikowi kwerend pomijać segmenty nieistotne dla danej kwerendy. W każdym segmencie dane są rozdzielane na kolumny, a każda kolumna jest przechowywana na osobnym zestawie stron. Następnie indeks jest znacząco kompresowany, co dodatkowo zmniejsza liczbę stron, które muszą zostać odczytane w celu obsłużenia kwerendy. Różnicę w organizacji danych przedstawiono na rysunku 11.12.

Struktura indeksów kolumnowych sprawia, że są one niezwykle efektywne w przypadku kwerend typowych dla hurtowni danych. Utworzenie indeksów kolumnowych w tabelach faktów może znacząco poprawić wydajność kwerend wymagających agregacji danych z tych tabel.

To prowadzi z powrotem do pytania, dlaczego indeksy kolumnowe są tak mało popularne. Szczerze mówiąc, jest to kwestia pierwszego wrażenia. Pierwsza wersja indeksów kolumnowych wprowadzona w wersji SQL Server 2012 miała spore ograniczenia. Dostępne były tylko nieklastrowe indeksy kolumnowe i można było utworzyć tylko jeden na tabelę. Indeksy kolumnowe nie były obsługiwane w indeksowanych widokach i nie można było ich używać w połączeniu ze śledzeniem zmian lub przechwytywaniem zmian danych.

Największym ograniczeniem było jednak to, że po utworzeniu indeksu kolumnowego tabelę można było tylko odczytywać. Oznaczało to konieczność usunięcia indeksu przed wykonaniem jakichkolwiek operacji INSERT, UPDATE lub DELETE. Co więcej, budowanie indeksów kolumnowych trwa dłużej niż w przypadku tradycyjnych indeksów opartych na strukturze B-drzewa.



Rysunek 11.12. Organizacja danych w indeksie kolumnowym

Ogólnie rzecz biorąc, pierwsza wersja indeksów kolumnowych nie sprawdziła się w większości scenariuszy. Problem polega na tym, że mimo znaczących ulepszeń na przestrzeni ostatnich 12 i więcej lat, ich reputacja się nie zmieniła.

Wielu administratorów baz danych na samo wspomnienie indeksu kolumnowego natychmiast odpowiada: „To się u mnie nie sprawdzi!”. Jest to po prostu pomyłka.

W nowoczesnym świecie indeksów kolumnowych wciąż istnieją pewne niewygodne ograniczenia. Zaawansowane typy danych, takie jak HIERARCHYID, GEOGRAPHY i GEOMETRY, nie są obsługiwane, podobnie jak duże typy danych, na przykład (N) VARCHAR(MAX) czy XML.

WSKAZÓWKA Nadal jesteśmy ograniczeni do tylko jednego indeksu kolumnowego w tabeli, ale ze względu na naturę indeksów kolumnowych ma to sens. Możemy po prostu uwzględnić wszystkie istotne kolumny w jednym indeksie.

Warto jednak zaznaczyć, że obecnie obsługiwane są zarówno klastrowe, jak i nieklastrowe indeksy kolumnowe. Po utworzeniu klastrowego indeksu kolumnowego nadal możemy tworzyć nieklastrowe indeksy o strukturze B-drzewa w tej samej tabeli. Oznacza to, że indeksy kolumnowe stały się naprawdę przydatne w przypadku obciążeń analitycznych i hurtowni danych. Warto rozważyć ich zastosowanie, gdy mamy do czynienia z dużymi tabelami faktów, na których wykonywane są złożone agregacje.

Aby zobaczyć, jaką różnicę może zrobić indeks kolumnowy, porównajmy wydajność kwerendy agregującej z tradycyjnym indeksem klastrowym o strukturze B-drzewa i klastrowym indeksem kolumnowym. Skrypt w listingu 11.18 wykonuje kwerendę, która oblicza średnią cenę oferty (BidPrice) i średni koszt za tysiąc wyświetleń (CostPerMille) na podstawie tabeli ImpressionsArchive w bazie danych MarketingArchive.

Listing 11.18. Wykonywanie kwerendy agregującej

```
SET STATISTICS TIME ON ;

SELECT
    AVG(BidPrice)
    , AVG(CostPerMille)
FROM dbo.ImpressionsArchive ;
```

Jeśli sprawdzimy statystyki wykonania w zakładce *Messages* w wynikach kwerendy, dowiemy się, jak długo trwało jej wykonanie. Na moim stanowisku testowym łączny czas wykonania wyniósł 1067 ms. To szybko, ale pamiętajmy, że tabela jest stosunkowo mała. W rzeczywistej bazie danych możemy mieć do czynienia z miliardami szerokich wierszy.

WSKAZÓWKA Pamiętaj, że czas wykonania kwerendy może się różnić w zależności od wielu czynników, takich jak specyfikacja komputera czy inne procesy zużywające zasoby systemowe.

Skrypt przedstawiony na listingu 11.19 pokazuje, jak utworzyć klastrowy indeks kolumnowy w tabeli ImpressionsArchive w bazie danych MarketingArchive.

Listing 11.19. Tworzenie klastrowego indeksu kolumnowego z nieklastrowym indeksem o strukturze B-drzewa

```
CREATE CLUSTERED COLUMNSTORE INDEX ImpressionsArchiveCCSI  
ON dbo.ImpressionsArchive ;  
GO
```

Teraz wykonajmy ponownie kwerendę z listingu 11.18 i przeanalizujemy statystyki czasu wykonania. Na moim stanowisku testowym całkowity czas wykonania wyniósł 45 ms. Oznacza to, że kwerenda została zrealizowana w zaledwie 4% czasu potrzebnego na wykonanie tej samej kwerendy z użyciem tradycyjnego indeksu klastrowego o strukturze B-drzewa. Łatwo sobie wyobrazić, jak ogromne korzyści wydajnościowe można osiągnąć, stosując to rozwiązanie w przypadku kwerend wykonywanych na bardzo dużych tabelach produkcyjnych.

Polecenie z listingu 11.20 pokazuje, jak utworzyć nieklastrowy indeks o strukturze B-drzewa, który będzie obsługiwać kwerendy filtrujące po CampaignID i CostCode, a jednocześnie zwracać BidPrice na liście SELECT.

Listing 11.20. Tworzenie nieklastrowego indeksu o strukturze B-drzewa

```
CREATE NONCLUSTERED INDEX CampaignIDCountryCodeWithBidPrice  
ON dbo.ImpressionsArchive(CampaignID, CountryCode)  
INCLUDE (BidPrice) ;  
GO
```

Podsumowanie

- B-drzewo jest podstawową organizacją indeksów w bazach danych. Składa się ono z poziomu korzenia, który zawiera pojedynczą stronę, zera lub wielu poziomów pośrednich oraz pojedynczego poziomu liści.
- Poziom liści indeksu klastrowego zawiera faktyczne strony danych tabeli, podczas gdy poziomy liści indeksu nieklastrowego zawiera wskaźniki do stron danych tabeli.
- Fragmentacja wewnętrzna odnosi się do niskiego wypełnienia stron, co powoduje, że odczytywanych jest więcej stron niż to konieczne.
- Istnieje kompromis między niską gęstością stron a ryzykiem ich podziałów spowodowanych aktualizacjami bardzo gęsto wypełnionych stron.
- Korzystne podziały stron występują wtedy, gdy strony są alokowane na końcu indeksu.
- Niekorzystne podziały stron występują wtedy, gdy nowe strony są alokowane w środku indeksu, a dane muszą zostać przeniesione na nową stronę. Takie podziały stron powodują zwiększenie liczby operacji wejścia-wyjścia i negatywnie wpływają na wydajność systemu.

- Fragmentacja zewnętrzna odnosi się do nieuporządkowania stron, które może negatywnie wpływać na wydajność.
- Fragmentacja zewnętrzna wpływa negatywnie na wydajność tylko w przypadku skanowania indeksów. Nie ma ona wpływu na bezpośrednie przeszukiwanie indeksów.
- Przeszukiwanie indeksu rozpoczyna się od korzenia B-drzewa i przechodzi przez wszystkie poziomy, aż do znalezienia poszukiwanego wiersza.
- Skanowanie indeksu polega na odczytywaniu poziomu liści indeksu aż do momentu dotarcia do końca poszukiwanych danych.
- Pętla indeksowa wykorzystuje indeks nieklastrowy do filtrowania lub agregacji danych, a następnie pobiera dodatkowe informacje z indeksu klastrowego lub sterty.
- Widok DMV `sys.dm_db_index_usage_stats` pozwala zbadać statystyki dotyczące operacji przeszukiwania, skanowania i sprawdzania indeksów.
- Nie należy reorganizować indeksów w celu poprawy gęstości stron. Reorganizacja indeksów wypełnia strony jedynie do poziomu określonego przez `FILLFACTOR`. Nie zmniejsza ona gęstości do poziomu `FILLFACTOR`. Zamiast tego należy przebudowywać indeksy.
- Należy unikać agregowania statystyk fragmentacji z widoku `sys.dm_db_index_physical_stats` i zamiast tego skupić się na danych na poziomie liści.
- Nieprzebudowywanie indeksów ma wpływ na ich skanowanie, a co za tym idzie, na wydajność kwerend. W przypadku baz danych działających w trybie ciągłym (24/7) zaleca się przebudowywanie indeksów w trybie online.
- Nie przebudowuj bezkrytycznie wszystkich indeksów. Lepiej przebudowywać tylko te indeksy, które tego wymagają, na podstawie statystyk fragmentacji.
- Nie aktualizuj statystyk po przebudowie indeksów, ponieważ może to prowadzić do pogorszenia jakości statystyk.
- W większości przypadków automatyczna aktualizacja statystyk jest wystarczająca.
- Jeśli postanowisz zaktualizować statystyki, weź pod uwagę kompromis związany z ponowną kompilacją planów kwerend.
- W zależności od kompromisu między wydajnością kwerend a dostępnym oknem konserwacji rozważ użycie parametru `MAXDOP` podczas przebudowy indeksów.

- Jeśli borykasz się z problemem współzawodnictwa przy wstawianiu na ostatniej stronie, rozważ użycie opcji `OPTIMIZE_FOR_SEQUENTIAL_KEY`.
- Jeśli musisz wykonać operację hurtowego wczytywania danych, rozważ wyłączenie indeksów nieklastrowych w docelowej tabeli, aby poprawić wydajność zapisu.
- Nie polegaj zbyttnio na narzędziach takich jak Database Engine Tuning Advisor. Ich wskazówki bywają użyteczne, ale uzupełnij je własną wiedzą biznesową.
- Indeksy kolumnowe organizują strony według kolumn, a nie wierszy.
- Rozważ użycie indeksów kolumnowych w dużych tabelach faktów, aby poprawić wydajność kwerend analitycznych.

Kopie zapasowe

W tym rozdziale:

- Jak dostosować strategię backupu do wymagań organizacji
- Znaczenie testowania kopii zapasowych i jak robić to w praktyce
- Jak migawki baz danych i migawki pamięci masowej mogą uzupełniać strategię backupu
- Znaczenie uwzględniania procesów ekstrakcji, transformacji i wczytywania danych przy planowaniu harmonogramów backupu
- Trzy modele przywracania w SQL Serverze i ich zastosowania
- Kwestie bezpieczeństwa przy tworzeniu kopii zapasowych bazy danych

W tym rozdziale omówimy typowe błędy popełniane przez administratorów baz danych podczas planowania i wdrażania backupu. Na przestrzeni lat zaobserwowałem tendencję do traktowania strategii backupu jako czegoś drugorzędnego. Wartość kopii zapasowych bywa trudna do oszacowania — oczywiście do momentu, gdy pilnie potrzebujemy odzyskać bazę danych.

Przykład z życia wzięty

Kilka lat temu, pracując jako architekt rozwiązań i główny programista w pewnym projekcie, poprosiłem zespół administratorów baz danych o wykonanie kopii zapasowej bazy produkcyjnej i przywrócenie jej w miejsce bazy deweloperskiej w celu odświeżenia środowiska. Niestety, administrator baz danych przez pomyłkę wykonał kopię zapasową bazy deweloperskiej i przywrócił ją w miejsce bazy produkcyjnej.

W związku z tym poprosiłem administratora bazy danych o pilne przywrócenie kopii zapasowej bazy produkcyjnej z poprzedniej nocy. Okazało się jednak, że tworzenie kopii zapasowych nie działało prawidłowo, a najnowsza dostępna kopia pochodziła sprzed trzech tygodni.

Na szczęście udało mi się odtworzyć dane z ostatnich trzech tygodni, korzystając z danych źródłowych, choć zajęło to strasznie dużo czasu. W niektórych przypadkach brak kopii zapasowych może spowodować poważne problemy dla firmy, w tym utratę przychodów, szkody wizerunkowe lub niezgodność z przepisami. Istnieją nawet przykłady organizacji, które zbankrutowały z powodu braku kopii zapasowych.

Zacznijmy ten rozdział od omówienia strategii backupu, uwzględniając przy tym wymagania biznesowe dotyczące docelowego punktu przywracania (ang. *recovery point objective*, RPO) i docelowego czasu przywracania (ang. *recovery time objective*, RTO). Następnie przejdziemy do analizy błędów, które popełniają administratorzy baz danych, polegając na migawkach jako strategii odtwarzania danych.

Istnieje stare powiedzenie, że nie masz kopii zapasowej, dopóki jej nie przywrócisz, a my przyjrzymy się konsekwencjom nietestowania kopii zapasowych. Następnie zajmiemy się planowaniem harmonogramów backupu. Skupimy się szczególnie na konfliktach między planowaniem okien konserwacji a czasem przeznaczonym na procesy ekstrakcji, transformacji i wczytywania danych (ang. *extract, transform, load*, ETL).

W dalszej kolejności przyjrzymy się różnym modelom przywracania danych, które można zastosować w SQL Serverze, oraz wpływowi, jaki niewłaściwy wybór modelu może mieć na wydajność i zajętość dysku. Omówimy również konsekwencje niewykonania kopii zapasowej po zmianie modelu przywracania. Przedyskutujemy, jak należy tworzyć doraźne kopie zapasowe w sposób, który nie zakłóca sekwencji odtwarzania. Na koniec przyjrzymy się kwestiom bezpieczeństwa związanym z przyjętą strategią backupu.

12.1. Numer 78 — nieuwzględnianie RPO i RTO

RPO określa, ile danych godzimy się utracić w przypadku awarii. *RTO* określa natomiast, ile czasu zajmie usunięcie skutków awarii. Oba te wskaźniki mają kluczowe znaczenie w planowaniu strategii backupu bazy danych.

Gdy pytamy użytkowników biznesowych, ile danych godzą się utracić w przypadku awarii, ich odpowiedź nieuchronnie brzmi: „Żadnych”. Podobnie, kiedy pytamy o akceptowalny czas trwania przestoju w razie awarii, zazwyczaj słyszymy: „Potrzebuję natychmiastowego przywrócenia systemu”.

Jeśli te założenia są prawdziwe, to powinniśmy skonfigurować wysoką dostępność i odzyskiwanie po awarii dla odpowiednich aplikacji, co zostanie omówione w rozdziale 13. Jednak jeśli porozmawiamy z osobą odpowiedzialną za aplikację i wyjaśnimy koszty związane z utrzymaniem redundantnych, geograficznie rozproszonych serwerów wymaganych w takiej topologii, okaże się prawdopodobnie, że wymagania nie są aż tak wysokie, jak początkowo się wydawało. Na przykład, jeśli aplikacja jest uznawana za P4 (gdzie P1 to aplikacja o krytycznym znaczeniu dla firmy), to mało prawdopodobne, aby tworzenie topologii wysokiej dostępności miało sens biznesowy, a w praktyce można polegać na strategii tworzenia i przywracania kopii zapasowych.

Konwersacje te mogą być czasem dość uciążliwe, w zależności od umiejętności i osobowości rozmówcy. Z tego powodu wielu administratorów baz danych popełnia błąd, przypisując wszystkie bazy danych do tego samego domyślnego harmonogramu i unikając trudnych rozmów z osobami odpowiedzialnymi za aplikację.

Jest to jednak pomyłka, której należy unikać. Bezkrytyczne stosowanie domyślnego harmonogramu bez zrozumienia wymagań dotyczących RPO i RTO może prowadzić do problemów. Na przykład, jeśli nasz domyślny harmonogram dopuszcza utratę godziny danych, ale utrata więcej niż 30 minut danych miałaby poważny wpływ na działalność firmy, to narażamy się na kłopoty w przyszłości.

Z drugiej strony, jeśli mamy domyślny harmonogram pozwalający na utratę 30 minut danych, ale tylko bardzo niewielki procent aplikacji ma RPO krótszy niż 4 godziny, to niepotrzebnie zużywamy zasoby na tworzenie zbędnych kopii zapasowych. Wykorzystujemy też więcej pamięci masowej, niż faktycznie potrzebujemy. Wiąże się to z dodatkowymi kosztami, a jeśli pamięć masowa znajduje się w chmurze, jest to koszt bezpośrednio mierzalny.

Jeśli nie bierzemy pod uwagę RTO, narażamy się na znaczne zakłócenie działalności biznesowej spowodowane niemożnością przywrócenia baz danych w akceptowalnym czasie.

Aby zrozumieć, jak możemy wpłynąć na RPO bazy danych, przypomnijmy sobie różne rodzaje kopii zapasowych, które możemy wykonać w SQL Serverze. Dostępne typy kopii zapasowych zostały szczegółowo opisane w tabeli 12.1.

Tabela 12.1. Rodzaje kopii zapasowych

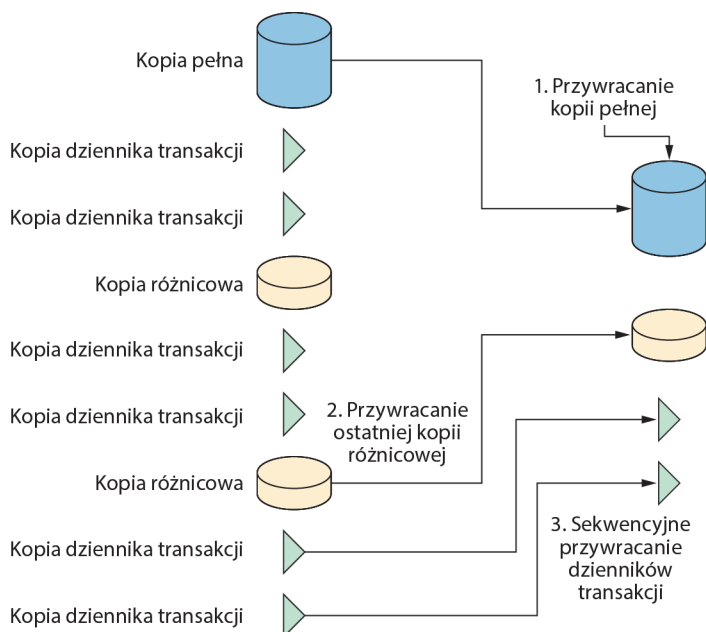
| Rodzaje kopii zapasowych | Opis |
|--------------------------|--|
| Pełna | Tworzy kopię wszystkich danych w bazie*. |
| Różnicowa | Tworzy kopię stron danych, które uległy zmianie od czasu wykonania ostatniej pełnej kopii zapasowej*. |
| Dziennika transakcji | Tworzy kopię dziennika transakcji, zawierającego zapis wszystkich operacji od czasu wykonania ostatniej kopii dziennika. |

* Istnieje możliwość ograniczenia kopii zapasowej do konkretnych plików danych lub grup plików.

Pełny backup ma największy wpływ na zasoby systemu i może powodować problemy z wydajnością na obciążonym systemie. Dlatego zazwyczaj wykonuje się go w oknie konserwacji. Wpływ backupu różnicowego jest zmienny i zależy od tego, ile stron w plikach danych uległo zmianie od ostatniego pełnego backupu. Choć zwykle jest znacznie mniej obciążający niż backup pełny, może nadal wpływać na wydajność systemów transakcyjnych (OLTP).

Backup dziennika transakcji nie powoduje odczytania plików danych, a jedynie tworzenie kopii samego dziennika. Ponieważ dziennik transakcji rejestruje wszystkie operacje od czasu ostatniego backupu dziennika, w niektórych systemach jest równoważny backupowi przyrostowemu. Podczas przywracania bazy transakcje te są odtwarzane w bazie danych, co oznacza, że możemy przeprowadzić *przywracanie do punktu w czasie*, czyli zatrzymać proces odtwarzania bazy danych w określonym momencie, w środku pliku dziennika transakcji. Jest to bardzo przydatne w sytuacjach, gdy doszło do błędu użytkownika, który spowodował nieprawidłową aktualizację lub usunięcie danych. Możemy wtedy zatrzymać odtwarzanie tuż przed wystąpieniem błędnej operacji. Nie jest to możliwe w przypadku pełnej lub różnicowej kopii zapasowej, ponieważ zawierają one kopie stron danych. Rysunek 12.1 przedstawia *łańcuch przywracania*, czyli kolejność, w jakiej należy odtwarzać kopie zapasowe, aby przywrócić bazę danych.

W typowym harmonogramie backupu możemy zaplanować backup pełny na noc, gdy aplikacja nie jest używana. Możemy również zaplanować backup różnicowy na porę lunchu, gdy użycie systemu jest minimalne. Jednak kluczowe dla spełnienia wymaganego RPO są kopie zapasowe dziennika transakcji. Należy zaplanować je zgodnie z ustalonym RPO. Na przykład, jeśli RPO wynosi 1 godzinę, należy ustawić wykonywanie kopii zapasowych dziennika transakcji co godzinę.



Rysunek 12.1. Łańcuch przywracania

Najważniejszą kwestią związaną z RTO jest miejsce przechowywania kopii zapasowych i szybkość ich odzyskiwania. W najgorszym przypadku, z jakim się spotkałem, kopie zapasowe baz danych były tworzone przez korporacyjne narzędzie do backupu, które kompresowało i deduplikowało dane przed wysłaniem ich bezpośrednio na taśmę. Umowa SLA przewidywała 6 godzin na odzyskanie danych z robota taśmowego.

Częstszym scenariuszem jest sytuacja, w której korporacyjne narzędzie do backupu przechowuje kopie na dysku przez pewien czas, zazwyczaj kilka dni, zanim prześle je na taśmę. W środowisku chmurowym odpowiednikiem jest system, w którym narzędzie do backupu przechowuje backupy przez kilka dni w pamięci masowej z natychmiastowym dostępem, takiej jak S3 w AWS lub Blob Storage w Azure. Następnie dane są przenoszone do magazynu archiwalnego, takiego jak Azure Deep Archive lub Azure Storage Archive Access Tier. Dostęp do tych magazynów nie jest natychmiastowy. Na przykład w przypadku Deep Archive umowa SLA gwarantuje odzyskanie danych w ciągu 24 godzin, choć w rzeczywistości zazwyczaj odbywa się to znacznie szybciej.

Zdarza się też, że zespół odpowiedzialny za kopie zapasowe reaguje powoli. Byłem świadkiem, jak przywracanie kopii zapasowej z poprzedniej nocy opóźniło się o wiele godzin tylko dlatego, że ktoś z zespołu backupu zwlekał z pobraniem zadania ze swojej kolejki.

W takich sytuacjach, gdy mamy krótki RTO, możemy zdecydować się na rezygnację z tworzenia kopii zapasowych baz danych przez narzędzie korporacyjne

wykorzystujące SQL Server Agent. Bardziej odpowiednie może być zaplanowanie własnego, natywnego backupu SQL Servera, a następnie wykorzystanie narzędzia korporacyjnego jedynie do przenoszenia plików kopii zapasowych.

Decydując się na takie podejście, musimy jednak pamiętać, że to na nas spoczywa odpowiedzialność za prawidłowe wykonywanie kopii zapasowych i rozwiązywanie ewentualnych problemów. Należy też mieć na uwadze koszty związane z duplikowaniem danych. W przypadku wielu dużych baz danych dodatkowa przestrzeń dyskowa wymagana na serwerach może szybko się powiększać. Jeśli korzystamy z serwerów w chmurze, może to prowadzić do znacznego, bezpośredniego wzrostu kosztów.

Jeśli zatem musimy przechowywać kopie zapasowe lokalnie, powinniśmy zadbać o wdrożenie procesów usuwania starych plików backupu. Skrypt Powershella przedstawiony na listingu 12.1 pokazuje, jak można usunąć pliki kopii zapasowych starsze niż trzy dni, przy założeniu, że kopie znajdują się w lokalizacji *D:\Backups*. Możemy zaplanować uruchamianie tego skryptu za pomocą narzędzia SQL Agent lub harmonogramu zadań Windows.

Listing 12.1. Usuwanie starych kopii zapasowych

```
$Path = 'D:\Backups\'
$Days = 3
$CutoffDate = (Get-Date).AddDays(-$Days)

Get-ChildItem -Path $Path -Recurse | Where-Object {$_.CreationTime -lt $CutoffDate} |
Remove-Item -Force -Recurse -Verbose -Confirm:$false
```

Planując strategię backupu dla bazy danych, zawsze powinniśmy brać pod uwagę wymagania biznesowe. Choć może się wydawać, że łatwiej jest stosować jeden harmonogram kopii zapasowych dla wszystkich baz danych, takie podejście może prowadzić do problemów w przyszłości. Zamiast tego powinniśmy dokładnie zrozumieć wskaźniki RPO i RTO dla każdej bazy danych i opracować indywidualną strategię tworzenia kopii zapasowych, która spełni te wymagania bez niepotrzebnego zużywania zasobów.

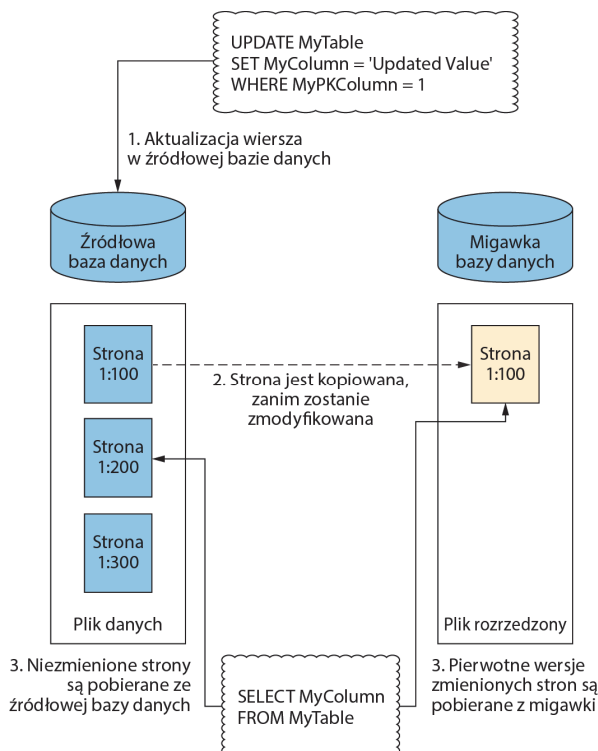
12.2. Numer 79 — stosowanie migawek bazy danych jako strategii przywracania

Migawki baz danych wykorzystują technologię kopiowania przy zapisie do tworzenia przeznaczonych tylko do odczytu kopii bazy z określonego punktu w czasie. Takie migawki mogą być niezwykle przydatne w wielu zastosowaniach. Na przykład świetnie nadają się do generowania raportów z danych z konkretnego punktu

w czasie. Są również doskonałe, gdy potrzebujemy porównać aktualne dane ze stanem z przeszłości. Co więcej, można ich użyć do ochrony danych przed błędami ludzkimi. Jeśli ktoś przypadkowo usunie dane, łatwiej i szybciej odzyskać je z migawki lub nawet cofnąć całą bazę danych do stanu z migawki niż odtworzyć dane w innej lokalizacji i skopiować brakujące informacje. Migawki można również wykorzystać do wycofania nieudanego wdrożenia, pod warunkiem że źródłowa baza danych jest nadal w dobrej kondycji.

Nie ulega wątpliwości, że migawki mogą stanowić uzupełnienie strategii backupu. Często pomyłką jest jednak traktowanie migawek jako zamiennika strategii backupu. To sytuacja, której bezwzględnie należy unikać. Aby zrozumieć dlaczego, musisz najpierw poznać zasadę działania migawek bazy danych.

Proces tworzenia migawki pokazano na rysunku 12.2. Nowo utworzona migawka bazy danych jest całkowicie pusta. Jest to tzw. *plik rozrzedzony*, zaimplementowany z użyciem mechanizmów systemu plików NTFS. Za każdym razem, kiedy modyfikowana jest strona w źródłowej bazie danych, zostaje ona skopiowana do migawki przed wprowadzeniem zmian. Gdy użytkownik wykonuje kwerendę na migawce, SQL Server sprawdza, czy wymagane strony danych istnieją w migawce. Jeśli tak, są one zwracane z migawki. Jeśli nie, dane są pobierane ze źródłowej bazy danych.



Rysunek 12.2. Proces tworzenia migawki

Jak widać, migawka jest całkowicie zależna od źródłowej bazy danych. W przypadku uszkodzenia lub usunięcia tej bazy migawka również staje się niedostępna. Co więcej, nawet gdyby migawka była dostępna, jej zależność od źródłowej bazy danych oznacza, że nie można jej przenieść na inną instancję. W rezultacie, gdyby instancja lub serwer stały się niedostępne, nie byłibyśmy w stanie odzyskać danych z migawki.

Choć migawki baz danych są bardzo przydatne w wielu przypadkach, w tym jako uzupełnienie strategii backupu, nie mogą one zastąpić strategii backupu. Decyzja o wdrożeniu migawek baz danych nie powinna mieć wpływu na naszą ogólną strategię tworzenia kopii zapasowych.

12.3. Numer 80 — używanie migawek zachowujących spójność w razie awarii jako strategii przywracania danych

Większość rozwiązań pamięci masowej klasy korporacyjnej oferuje funkcję tworzenia migawek, która umożliwia wykonanie kopii zapasowej dysku lub całej maszyny wirtualnej. Implementacja technologii migawek różni się w zależności od producenta, więc nie będziemy się tym szczegółowo zajmować. Istnieją jednak pewne wspólne koncepcje, które warto zrozumieć.

Po pierwsze, migawki są *kopiami zapasowymi na poziomie bloków*, co oznacza, że zapisywane są fizyczne bloki na dysku, a nie pliki dostępne na poziomie systemu operacyjnego. Po drugie, istnieją różne rodzaje migawek, a każdy z nich zapewnia inny poziom spójności danych.

Najbardziej podstawowym poziomem spójności jest *migawka zachowująca spójność w razie awarii*. W tym modelu zachowana jest kolejność zapisu danych, a wszystkie bloki na dysku są archiwizowane na podstawie tego samego, konkretnego znacznika czasowego. Dzięki temu pliki pozostają w spójnym stanie.

Problem w tym, że SQL Server działa inaczej niż system plików. Podczas transakcji część danych może znajdować się w pamięci podręcznej buforów i oczekiwać na zapis na dysku, podczas gdy inne dane zostały już zapisane. Oznacza to, że po przywróceniu dysku lub maszyny wirtualnej baza danych może znaleźć się w niespójnym stanie, jeśli w momencie backupu trwały jakieś transakcje.

Niezrozumienie tego prowadzi do błędu polegającego na tym, że zespoły odpowiedzialne za pamięć masową i chmurę próbują chronić maszyny wirtualne

hostujące bazy danych poprzez konfigurowanie migawek zachowujących spójność w razie awarii. Nie zdają sobie sprawy, że te migawki mogą powodować uszkodzenia danych, które ujawnią się dopiero po przywróceniu maszyny wirtualnej.

Migawki są często wykorzystywane do szybkiego przywracania maszyny wirtualnej w przypadku wprowadzenia niepożądanych zmian. Dlatego wiele planów wprowadzania zmian zawiera strategię wycofania obejmującą przywracanie serwera z migawki. Jeśli jednak używane są migawki zachowujące spójność w razie awarii, taka strategia może okazać się niewystarczająca.

Kolejnym poziomem ochrony jest *migawka spójna na poziomie aplikacji*. W przypadku tego typu migawki bazy danych są wyciszane, a dane znajdujące się w pamięci są zapisywane na dysk przed wykonaniem migawki. Zapewnia to spójność bazy danych i sprawia, że ten rodzaj migawki jest bezpiecznym rozwiązaniem.

Spójne na poziomie aplikacji migawki dysków hostujących serwery Windows wykonuje się z użyciem usługi Volume Shadow Copy Service (VSS) w celu obsługi oczekujących operacji wejścia-wyjścia. Obsługa migawek spójnych na poziomie aplikacji w Linuksie pozostawia jednak nieco do życzenia. Na przykład w czasie pisania tej książki VMware obsługuje migawki spójne na poziomie aplikacji tylko poprzez implementację skryptów wykonywanych przed zamrożeniem i po odmrózeniu systemu, które trzeba napisać ręcznie. Podobne ograniczenie występuje w Azure, ale Microsoft udostępnia plik szablonu konfiguracji, który możemy wypełnić. Znajduje się on w repozytorium na GitHubie pod adresem <https://mng.bz/GNzO>.

Pomyłka na odwrót

Spotkałem się również z odwrotną sytuacją. W niektórych organizacjach zespoły w ogóle nie korzystają z migawek do ochrony serwerów baz danych. Wynika to z faktu, że są świadome ryzyka uszkodzenia danych, jakie mogą powodować migawki zachowujące spójność w razie awarii, i dlatego całkowicie unikają stosowania tej metody.

W praktyce migawki spójne na poziomie aplikacji mogą stanowić bardzo przydatny mechanizm szybkiego odzyskiwania systemu po nieudanym wdrożeniu. Oczywiście powinny one jedynie uzupełniać właściwą strategię backupu bazy danych, a nie ją zastępować, niemniej jednak mają swoją wartość.

Chociaż konfiguracja migawek pamięci masowej zazwyczaj nie należy do obowiązków administratora bazy danych, bardzo ważne jest, aby administratorzy rozumieli konsekwencje stosowania migawek i sposób ich konfiguracji. Dzięki temu administrator może służyć radą i wsparciem zespołowi odpowiedzialnemu za infrastrukturę chmurową lub pamięć masową, któremu powierzono zadanie konfiguracji migawek.

Migawki mogą zapewnić szybkie przywracanie danych i stanowić dobre uzupełnienie strategii backupu baz danych. Należy jednak zalecić zespołom

odpowiedzialnym za przechowywanie danych, aby dla serwerów baz danych konfigurowały tylko migawki spójne na poziomie aplikacji. Nie powinno się stosować migawek zachowujących spójność w razie awarii, ponieważ mogą one prowadzić do przywracania uszkodzonych baz danych.

12.4. Numer 81 — nietestowanie kopii zapasowych

Wróćmy do starego powiedzenia: „Nie masz kopii zapasowej, dopóki jej nie odtworzyłeś”. Jest w nim sporo prawdy. Tworzenie kopii zapasowej to operacja wejścia-wyjścia intensywnie wykorzystująca dysk, często przez sieć. Zakłócenia mogą pojawić się w wielu miejscach — w sieci, w pamięci masowej albo w konkretnym uszkodzonym sektorze. Każda z tych sytuacji może spowodować błędny zapis, a my się nie dowiemy, że nasza kopia zapasowa jest uszkodzona, dopóki nie spróbujemy jej odtworzyć i nie odkryjemy, że jest to niemożliwe.

Mogę sobie wyobrazić, jak przewracasz oczami, czytając powyższy akapit i myślisz: „Przecież nie mogę przywracać wszystkich moich kopii zapasowych. Nie mam na to czasu ani miejsca na dysku!”. Masz rację, ale nigdy nie propagowałbym nierealistycznych oczekiwań. Istnieje jednak złoty środek, który pozwala zwiększyć pewność co do naszych kopii zapasowych bez ponoszenia ogromnych kosztów w postaci czasu i zasobów. Niestety, podejście to jest często pomijane przez administratorów baz danych. Nieznalezienie tego złotego środka to pomyłka, która może się później boleśnie zemścić.

Czym więc jest ów złoty środek? Cóż, składa się on z kilku aspektów. Po pierwsze, musimy sprawdzić, czy nasze kopie zapasowe zostały pomyślnie utworzone. Brzmi to jak truizm, ale zaskakująco często administratorzy baz danych w ogóle tego nie sprawdzają albo po prostu ignorują wszelkie błędy. Po drugie, powinniśmy zweryfikować integralność naszych kopii zapasowych. Omówimy każdą z tych kwestii w kolejnych punktach.

12.4.1. Weryfikacja poprawności wykonania kopii zapasowych

Jeśli kopie zapasowe baz danych są tworzone za pomocą korporacyjnego narzędzia do tworzenia kopii zapasowych, prawdopodobnie codziennie wysyłany jest raport zawierający informacje o nieudanych kopiach. Jako administratorzy baz danych powinniśmy poprosić o dodanie nas do listy odbiorców tego raportu. Warto przeglądać go przy pierwszej porannej kawie i kontaktować się z zespołem odpowiedzialnym za kopie zapasowe, aby naprawić wszelkie błędy poprzez ponowne uruchomienie zadania w odpowiednim oknie czasowym. Oczywiście,

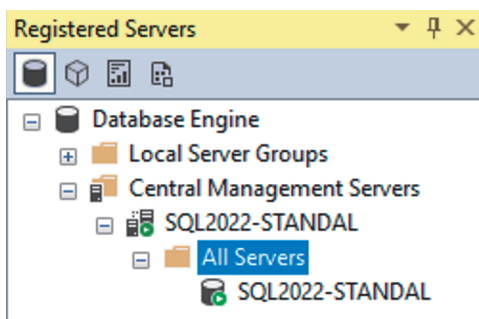
im dłużej baza danych pozostaje bez kopii zapasowej, tym większe ryzyko. Jeśli tworzenie kopii zapasowej nie powiedzie się wielokrotnie, powinniśmy zbadać przyczynę problemu we współpracy z zespołem ds. kopii zapasowych.

Jeśli kopie zapasowe są tworzone przy użyciu natywnych mechanizmów SQL Servera, powinniśmy sprawdzać, czy proces ten kończy się sukcesem, i samodzielnie rozwiązywać ewentualne problemy. Może się to wydawać trudnym zadaniem, szczególnie w przypadku dużych środowisk. Dlatego ważne jest, aby wdrożyć system powiadomień lub raportowania, który pomoże nam monitorować ten proces.

Jeśli mamy narzędzie do monitorowania instancji SQL Servera, możemy stworzyć niestandardowy test, który będzie weryfikować powodzenie zadań tworzenia kopii zapasowych. Jeśli jednak nie mamy takiej możliwości, to warto wiedzieć, że istnieje kilka innych sposobów osiągnięcia tego samego celu z użyciem wbudowanych funkcji SQL Servera.

Pierwszym sposobem jest dodanie alarmów narzędzia SQL Agent do zadań SQL Agent wykonujących kopie zapasowe. Problem z tym podejściem polega na tym, że musielibyśmy również włączyć funkcję Database Mail w każdej instancji SQL Servera. Jest to dość kłopotliwe i niezgodne z poziomem 1 standardu bezpieczeństwa CIS dla SQL Servera.

Drugim sposobem jest skonfigurowanie centralnego serwera zarządzania. Można to zrobić w programie SQL Server Management Studio przez przejście do menu *View* i wybranie opcji *Registered Servers*. Spowoduje to wyświetlenie okna *Registered Servers*. W tym miejscu możemy utworzyć centralny serwer zarządzania. Pod nowo utworzonym węzłem możemy utworzyć grupę serwerów zawierającą wszystkie instancje w naszym środowisku. Na koniec w grupie tej możemy zarejestrować wszystkie nasze instancje. Okno *Registered Servers* z centralnym serwerem zarządzania przedstawiono na rysunku 12.3.



Rysunek 12.3. Okno *Registered Servers*

Umożliwia to wykonanie kwerendy na grupie instancji. Po wykonaniu tej operacji do wyników kwerendy zostanie dodana kolumna o nazwie *Server*, która poinformuje nas, z której instancji pochodzi dany wiersz. Każdego ranka możemy

uruchomić kwerendę z listingu 12.2 dla grupy serwerów, aby zobaczyć listę wszystkich nieudanych kopii zapasowych z ostatniego uruchomienia. Kwerenda wykorzystuje funkcję `ROW_NUMBER()` do przypisywania rosnącego numeru do każdego wiersza. Klauzula `PARTITION BY` zapewnia, że numery wierszy będą obliczane osobno dla każdego serwera, a klauzula `ORDER BY` — że najnowsze wykonanie otrzyma wartość 1. Następnie kwerenda zewnętrzna odfiltrowuje wszystkie wiersze, które nie mają numeru wiersza równego 1 (czyli najnowsze wykonanie).

OSTRZEŻENIE Aby skorzystać z tego skryptu, zastąp nazwę zadania swoją własną. Skrypt zakłada, że używasz jednakowej nazwy dla zadań kopii zapasowych we wszystkich instancjach.

Listing 12.2. Wyświetlanie nieudanych zadań backupu wszystkich serwerów

```
SELECT
    Server
    , name
    , message
    , run_status
FROM (
    SELECT
        Server
        , j.name
        , jh.message
        , jh.run_status
        , ROW_NUMBER() OVER(PARTITION BY Server ORDER BY run_date, run_time DESC) AS
        ↳ RowNumber
    FROM msdb.dbo.sysjobhistory jh
    INNER JOIN msdb.dbo.sysjobs j
        ON j.job_id = jh.job_id
    WHERE jh.run_status = 0
        AND j.name = 'Backups'
) Results
WHERE RowNumber = 1 ;
```

Trzecią i najbardziej złożoną metodą jest utworzenie inwentaryzacyjnej bazy danych, która przechowuje szczegółowe informacje o wszystkich instancjach SQL Servera w danym środowisku. Byłoby najlepiej, gdyby ta baza danych była uzupełniana w ramach zautomatyzowanego procesu tworzenia instancji. Można wykorzystywać ją do wielu celów, więc właściwie nie ma żadnych ograniczeń, jeśli chodzi o dane, jakie są w niej przechowywane. Jednak w tym konkretnym przypadku musielibyśmy zapisywać nazwę serwera i instancji, nazwę i zaszyfrowane hasło konta z odpowiednimi uprawnieniami, bazy danych znajdujące się w danej instancji oraz harmonogram backupu wymagany dla każdej bazy danych.

Pozwoli nam to utworzyć mechanizm harmonogramowania, będący w stanie określić, które bazy danych powinny być archiwizowane na poszczególnych serwerach w danym momencie. Następnie możemy zaplanować uruchamianie skryptu PowerShella co minutę za pomocą narzędzia SQL Agent. Ten skrypt PowerShella będzie pobierał listę baz danych wymagających kopii zapasowej

z mechanizmu harmonogramowania. Skrypt może następnie pobrać dane logowania dla odpowiedniej instancji i wykonać kopię zapasową każdej z wymaganych baz danych. Po zakończeniu pracy skrypt może zapisać wyniki z powrotem w bazie danych. Takie podejście umożliwia nam łatwe zarządzanie kopiami zapasowymi z jednego miejsca, włącznie ze zmianą harmonogramów backupu. Pozwala również na proste raportowanie błędów backupu z centralnej lokalizacji.

Osobiście preferuję korzystanie z inwentaryzacyjnej bazy danych połączonej z mechanizmem harmonogramowania. Utworzyłem takie rozwiązanie w kilku miejscach pracy na przestrzeni lat i sprawdziło się ono bardzo dobrze. Zwykle używam go do planowania wszystkich typowych zadań konserwacyjnych bazy danych, nie tylko tworzenia kopii zapasowych. Należy jednak pamiętać, że gdy po raz pierwszy tworzyłem taki system, napisanie i skonfigurowanie wszystkiego zgodnie z wymaganiami mojej firmy zajęło mi prawie trzy miesiące. Chociaż kolejne wdrożenia przebiegały znacznie szybciej, ponieważ miałem już gotowy kod bazowy, to nadal potrzeba sporo czasu, aby wszystko odpowiednio skonfigurować i przetestować. Z tego powodu nie jest to odpowiednie rozwiązanie dla wielu organizacji.

Jeśli niestandardowy mechanizm harmonogramowania nie jest odpowiedni dla Twojego środowiska, zalecam użycie centralnego serwera zarządzania. Nie jest to najlepsza opcja, ponieważ opiera się na funkcjach SQL Server Management Studio, a nie tylko na podstawowych funkcjach SQL Servera. Jest jednak lepsza niż próba zarządzania Database Mail w każdej instancji osobno.

12.4.2. Weryfikacja integralności kopii zapasowej

SQL Server umożliwia sprawdzenie poprawności kopii zapasowej bez konieczności jej faktycznego przywracania. Proces ten polega na odczytaniu całego zestawu kopii zapasowej w celu upewnienia się, że jest on kompletny i dostępny. Wykonuje się również ograniczone sprawdzanie danych, starając się wykryć jak najwięcej potencjalnych problemów. Jeśli kopia zapasowa została utworzona z sumą kontrolną, zostanie ona zweryfikowana. Ponadto sprawdzone zostanie, czy na dysku jest wystarczająco dużo miejsca do przeprowadzenia ewentualnego przywrócenia danych.

Podczas tworzenia kopii zapasowej można włączyć sumy kontrolne stron. Powoduje to obliczenie sumy kontrolnej dla każdej strony i zapisanie jej w nagłówku strony przed zapisem na dysk. Gdy strona jest odczytywana z dysku, suma kontrolna może zostać ponownie obliczona w celu sprawdzenia, czy jest zgodna z wcześniej zapisaną. Polecenie z listingu 12.3 pokazuje, jak włączyć tę opcję dla bazy danych Marketing.

Listing 12.3. Włączanie sum kontrolnych stron

```
ALTER DATABASE Marketing  
SET PAGE_VERIFY CHECKSUM ;
```

Jeśli ta opcja jest włączona, sumy kontrolne będą weryfikowane podczas odczytu stron z dysku. Gdy użyjemy opcji `WITH CHECKSUM` przy tworzeniu kopii zapasowej, SQL Server wygeneruje sumę kontrolną dla całej bazy danych. Podczas odtwarzania kopii zapasowej lub — w tym przypadku — weryfikacji kopii SQL Server sprawdzi, czy suma kontrolna bazy danych jest prawidłowa.

UWAGA Użycie opcji `WITH CHECKSUM` jest najlepszym sposobem na zapewnienie integralności kopii zapasowej, ale będzie miało wpływ na wydajność procesu tworzenia kopii. Prawdopodobnie zauważysz spadek wydajności tworzenia kopii zapasowej, szczególnie w przypadku dużych baz danych.

Polecenie z listingu 12.4 wykonuje kopię zapasową bazy danych `Marketing` z użyciem klauzuli `WITH CHECKSUM` do wygenerowania sumy kontrolnej. Dzięki temu będziemy mogli później zweryfikować poprawność kopii zapasowej, sprawdzając tę sumę kontrolną.

WSKAZÓWKA Podczas wykonywania przykładów z tego rozdziału pamiętaj, aby zmienić lokalizację kopii zapasowej na odpowiadającą Twojemu systemowi.

Listing 12.4. Tworzenie kopii zapasowej bazy danych z sumą kontrolną

```
BACKUP DATABASE Marketing  
TO DISK = 'D:\Backup\MarketingFull127122023.bak'  
WITH  
    NAME = 'Marketing-Full Database Backup'  
    , CHECKSUM ;
```

Poprawność kopii zapasowej możemy zweryfikować za pomocą polecenia `RESTORE VERIFYONLY`. Na listingu 12.5 pokazano, jak to zrobić dla bazy danych `Marketing`.

Listing 12.5. Weryfikacja kopii zapasowej

```
RESTORE VERIFYONLY  
FROM DISK = 'D:\Backup\MarketingFull127122023.bak'  
WITH CHECKSUM ;
```

Jeśli tworzymy kopie zapasowe za pomocą natywnych mechanizmów SQL Servera, po kroku zadania wykonującym backup możemy dodać krok, który uruchomi polecenie `RESTORE VERIFYONLY`. Następnie powinniśmy się upewnić, że ten krok zadania kończy się pomyślnie, korzystając z jednej z technik omówionych w poprzedniej sekcji.

Jeśli kopie zapasowe są tworzone za pomocą korporacyjnego narzędzia do backupu, sytuacja staje się nieco bardziej skomplikowana. Większość narzędzi

obsługuje funkcję `RESTORE VERIFYONLY` przy użyciu własnego agenta do backupu SQL Servera, ale niekoniecznie pozwalają one łatwo uruchomić to polecenie dla wszystkich baz danych w całym środowisku. W takim przypadku konieczna będzie współpraca z zespołem odpowiedzialnym za kopie zapasowe w celu utworzenia rozwiązania wykorzystującego API narzędzia do backupu.

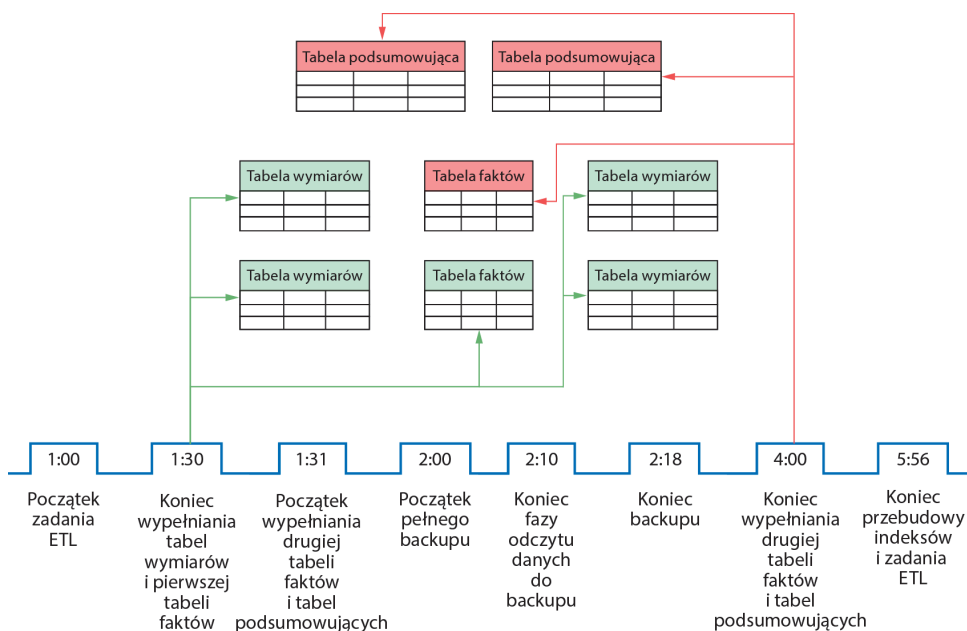
Pełne przywracanie każdej kopii zapasowej bazy danych jest zazwyczaj niepraktyczne. Można jednak znaleźć złoty środek, sprawdzając poprawność wykonania zadań tworzenia kopii zapasowych oraz weryfikując ich integralność. Jeśli dysponujesz odpowiednią infrastrukturą, warto rozważyć skonfigurowanie automatycznego przywracania kopii. Nawet przy użyciu opcji `CHECKSUM` i `RESTORE VERIFYONLY` istnieje ryzyko, że kopia zapasowa będzie uszkodzona.

12.5. Numer 82 — tworzenie kopii zapasowych podczas okna ETL

Dobrze zaprojektowany proces ETL powinien być wielokrotnie uruchamialny, innymi słowy — zapewniać ostateczną zbieżność. Oznacza to, że jeśli proces zostanie przerwany w trakcie wykonywania, można go uruchomić ponownie, przetwarzając wszystkie dane bez powielania informacji. Aby proces ETL mógł być wielokrotnie uruchamialny, musi spełniać kilka warunków. Po pierwsze, muszą być dostępne dane źródłowe w ich pierwotnej formie. Można to osiągnąć poprzez retencję danych, jeśli pochodzą one z plików płaskich, lub przez konfigurowalny interfejs API. Sytuacja jest bardziej skomplikowana, gdy dane źródłowe pochodzą z bazy transakcyjnej — wtedy zazwyczaj konieczne jest buforowanie danych. Drugim warunkiem jest to, że proces ETL musi mieć sparametryzowane daty. Jeśli proces zaplanowany na godzinę 22:00 ma na stałe zakodowane wyrażenie `WHERE DataDate = GETDATE()`, nie można uruchomić go ponownie. Technicznie rzecz biorąc, można by zmodyfikować kod i uruchomić prace ręcznie, ale idea wielokrotnej uruchamialności zakłada, że można to zrobić bez zmian w kodzie. Wreszcie, co prawdopodobnie najważniejsze, proces musi scalać dane w tabelach, a nie tylko je wstawiać. Jeśli proces wykorzystuje instrukcje `INSERT`, istnieje duże prawdopodobieństwo, że ponowne uruchomienie go spowoduje pewnego rodzaju duplikację danych w złożonym przebiegu ETL. Dzieje się tak, bo choć w przypadku awarii transakcja zostanie zakończona lub wycofana, duże i skomplikowane procesy ETL wymagają wielu transakcji. W rezultacie, choć dane będą spójne z punktu widzenia transakcji, mogą nie być spójne z perspektywy logiki biznesowej.

Niestety musimy pogodzić się z tym, że nie wszystkie procesy ETL są dobrze zaprojektowane. Wyobraźmy sobie, że mamy proces ETL, który nie jest wielokrotnie uruchamialny. Wykorzystuje on instrukcję INSERT w wielu transakcjach, generując nowe wiersze w różnych tabelach. Tabele te mają kolumny typu IDENTITY, więc każdy wiersz jest unikatowy z punktu widzenia SQL Servera, ale pozwala na duplikaty kluczy biznesowych. Taki scenariusz jest dość powszechny. Załóżmy, że proces jest skonfigurowany do uruchamiania się o godzinie pierwszej w nocy i trwa 5 godzin. Przyjmijmy również, że harmonogram kopii zapasowych dla tego serwera przewiduje wykonanie pełnej kopii o godzinie drugiej w nocy.

W ciągu pierwszej godziny działania w ramach procesu ETL wyodrębniono wszystkie dane z różnych rozproszonych źródeł i wczytano je do tabel przejściowych w hurtowni danych. Rozpoczęło się również wypełnianie tabel w schemacie gwiazdowym bazy danych, co wymaga wielu poziomów złożonych transformacji. O godzinie 2:00 rozpoczyna się tworzenie kopii zapasowej, a faza odczytu kończy się 10 minut później — jest to punkt spójności kopii zapasowej, czyli moment w czasie, do którego przywrócilibyśmy bazę w przypadku odtwarzania z tej kopii. Tworzenie kopii kończy się o 2:18, a proces ETL zostaje ukończony o 5:56. Jednak wtedy ma miejsce katastrofa. Tuż po godzinie 6:00 następuje całkowita awaria pamięci masowej, która powoduje uszkodzenie bazy danych. Nie mamy innego wyjścia, jak przywrócić ją z kopii zapasowej. Problem ten został zilustrowany na rysunku 12.4.



Rysunek 12.4. Tworzenie kopii zapasowej podczas procesu ETL

Przywracamy bazę danych z kopii zapasowej wykonanej o drugiej w nocy, a zespół wsparcia aplikacji ponownie uruchamia proces ETL. Niestety, gdy dział biznesowy przeprowadza kontrolę odzyskanego systemu, okazuje się, że dane są niespójne. Część danych została zduplikowana, ale nie wszystkie. Przywracamy więc kopię zapasową z poprzedniej nocy, a zespół wsparcia aplikacji ponownie uruchamia dwa ostatnie nocne procesy ETL. Dział biznesowy sprawdza dane i stwierdza, że teraz pojawiły się duplikaty z poprzedniego dnia.

Przy założeniu, że proces ETL trwa około 5 godzin, i zakładając, że przywracanie danych zajmuje 30 minut, a weryfikacja kolejne 30, incydent o najwyższym priorytecie trwa już 18 godzin, a usługa nadal nie została przywrócona. Wszyscy są bardzo zmęczeni i zestresowani, a kolejny cykl ETL ma się rozpocząć za godzinę.

Tę sytuację można by śmiało określić jako koszmar. Wiem to, ponieważ jest ona luźno oparta na scenariuszu, z którym zetknąłem się kilka lat temu w krytycznym dla firmy systemie, który odziedziczyłem. W tym przypadku istnieją tylko dwa rozwiązania. Pierwsze polega na tym, że programista lub zespół wsparcia aplikacji spędzi [wstaw przerażająco dużą liczbę] godzin na ręcznym uruchamianiu kwerend w celu usunięcia duplikatów danych w tabelach bazowych i kasowaniu danych z tabel przechowujących złożone obliczenia. Następnie musi ręcznie przeprowadzić niektóre części procesu ETL, aby wykonać skomplikowane obliczenia.

Drugą opcją byłoby przywrócenie wersji bazy danych z dwóch ostatnich kopii zapasowych. Następnie programista lub zespół wsparcia aplikacji spędziłby [wstaw równie przerażająco dużą liczbę] godzin, próbując połączyć dane z obu baz w jeden spójny zestaw. Prawdopodobnie musiałby potem usunąć dane z tabel przechowujących złożone obliczenia. Na koniec musiałby ręcznie przeprowadzić niektóre części procesu ETL, aby ponownie wykonać te skomplikowane obliczenia.

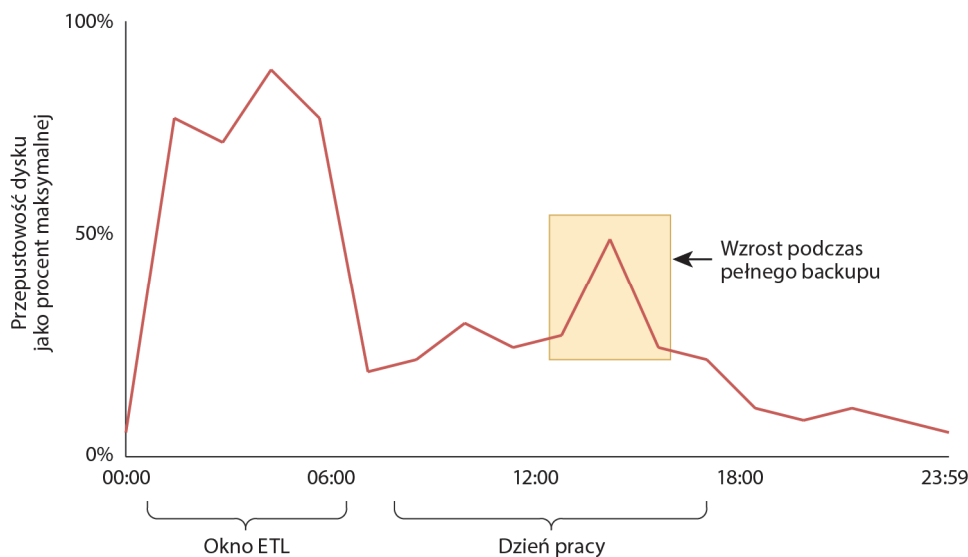
Obie metody są równie ryzykowne i czasochłonne. Istnieje duże prawdopodobieństwo, że dane pozostaną w niedoskonałym stanie, chyba że osoba naprawiająca bazę będzie miała tyle samo szczęścia co umiejętności. W zależności od rodzaju naprawianych danych może to potencjalnie prowadzić do podejmowania niewłaściwych decyzji biznesowych, a nawet do niezgodności z przepisami. Prosty morał z tej historii jest taki, że tworzenie kopii zapasowych w tym samym czasie, gdy działa proces ETL, to pomyłka. Administratorzy baz danych powinni zawsze zakładać najgorszy możliwy scenariusz.

Kolejnym aspektem wartym uwagi, nawet w przypadku dobrze zaprojektowanych procesów ETL, jest wydajność. Złożone procesy ETL często wymagają dużych zasobów i wydajnego dysku. Jeśli uruchomimy backup równocześnie z procesami ETL, najprawdopodobniej spowoduje to znaczne spowolnienie działania systemu.

Co zatem powinniśmy zrobić inaczej? Spotkałem się ze scenariuszami, w których długotrwałe procesy ETL sprawiają, że nie ma czasu na wykonanie pełnej kopii zapasowej bazy danych poza godzinami pracy — ani przed procesem ETL, ani po nim.

Odpowiedź w dużej mierze zależy od rodzaju danych, ale istnieje kilka możliwych rozwiązań. Pierwszą opcją, którą zwykle biorę pod uwagę, jest coś, co nazywam *kreatywnym planowaniem*. Często hurtownia danych jest zaprojektowana tak, żeby procesy ETL mogły efektywnie działać w określonym przedziale czasowym. Oznacza to, że serwer ma w rzeczywistości nadmiar mocy w stosunku do jego wykorzystania w ciągu dnia roboczego. Jeśli tak jest, możemy po prostu wykonać pełną kopię zapasową w godzinach pracy, gdy dane są odczytywane, ale nie modyfikowane.

Scenariusz ten został zilustrowany na rysunku 12.5, który przedstawia przepustowość dysku w ciągu 24 godzin, wyrażoną jako procent maksymalnej możliwej przepustowości. W godzinach pracy przepustowość jest znacznie niższa od maksymalnej, nawet gdy uwzględnimy pełną kopię zapasową wykonywaną w środku dnia.

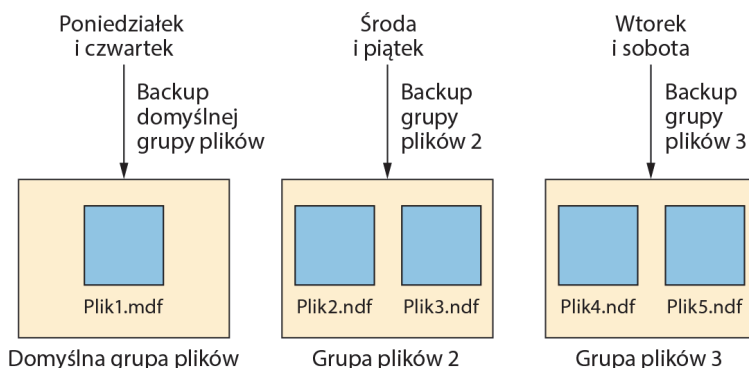


Rysunek 12.5. Przepustowość dysku w ciągu 24 godzin

Jeśli kreatywne planowanie nie jest możliwe, inną opcją jest wykorzystanie backupu różnicowego. W tym scenariuszu wykonywalibyśmy pełną kopię zapasową bazy danych w weekendy, a kopię różnicową na koniec codziennego procesu ETL. Może to być dobre, pragmatyczne rozwiązanie, ale nie zawsze się sprawdzi. Kopia różnicowa zawiera wszystkie strony, które zostały zmodyfikowane od czasu wykonania ostatniej pełnej kopii zapasowej. Jeśli każdej nocy

uruchamiamy duży proces ETL, a także wykonujemy zadania takie jak przebudowa indeksów (patrz rozdział 11.), to pod koniec tygodnia kopia różnicowa może być tak duża, że również nie uda się jej ukończyć w wyznaczonym czasie.

Możemy wreszcie rozważyć backup grup plików. Jest to opcja, po którą sięgam w ostateczności, ponieważ wprowadza zarówno złożoność, jak i ryzyko. W tym scenariuszu tworzylibyśmy kopie zapasowe różnych grup plików w różne dni tygodnia, jak pokazano na rysunku 12.6. Złożoność polega na konieczności łączenia wielu plików kopii zapasowych podczas odtwarzania danych. Ryzyko polega na tym, że możemy nie być w stanie spełnić wymagań w zakresie RPO.



Rysunek 12.6. Strategia backupu grup plików

OSTRZEŻENIE Jeśli nasz wskaźnik RPO wynosi 24 godziny (co jest typowe dla tego rodzaju systemów), to przy takim podejściu nie spełnilibyśmy wymagań. Przed wdrożeniem takiego rozwiązania firma musiałaby zaakceptować to ryzyko.

Podczas tworzenia harmonogramu backupu zawsze powinniśmy brać pod uwagę procesy ETL i dbać o to, aby nie nakładały się na backup. Czasami można to osiągnąć po prostu przez wykonywanie kopii zapasowych w innym oknie czasowym. Jeśli nie jest to możliwe, warto rozważyć codzienne kopie różnicowe i cotygodniowe kopie pełne. W ostateczności można pomyśleć o kopiach zapasowych grup plików, ale to nie tylko zwiększa złożoność, lecz także wpływa negatywnie na RPO.

12.6. Numer 83 — stosowanie wyłącznie modelu przywracania FULL w hurtowniach danych i systemach deweloperskich

Model przywracania bazy danych określa, w jaki sposób jest ona chroniona. Istnieją trzy modele odzyskiwania: pełny (FULL), prosty (SIMPLE) i z hurtowym rejestrowaniem (BULK LOGGED). Gdy baza danych jest ustawiona na model SIMPLE, możliwe jest wykonywanie tylko pełnych i różnicowych kopii zapasowych. Kopie zapasowe dziennika transakcji nie są dostępne. Pełne kopie zapasowe są niezbędne do odtworzenia bazy danych, natomiast kopie różnicowe są opcjonalne. W modelu SIMPLE SQL Server automatycznie przycina dziennik transakcji podczas tworzenia punktu kontrolnego. Proces ten służy do zapisywania zmodyfikowanych (brudnych) stron z pamięci podręcznej na dysk. Punkt kontrolny jest wyzwalany na podstawie określonego interwału przywracania bazy danych oraz gdy wykonywane są pewne operacje, takie jak tworzenie kopii zapasowej. Może być również wyzwolony ręcznie przez administratora bazy danych za pomocą polecenia CHECKPOINT. Modele przywracania opisano w tabeli 12.2.

Tabela 12.2. Modele przywracania

| Model odtwarzania | Pełne kopie zapasowe | Kopie różnicowe | Kopie dziennika transakcji | Przycinanie dziennika | Poziom ochrony |
|-------------------|----------------------|-----------------|----------------------------|-----------------------|---|
| SIMPLE | Tak | Opcjonalnie | Nie | W punkcie kontrolnym | Spójność transakcyjna |
| BULK LOGGED | Tak | Opcjonalnie | Tak | Przy kopii dziennika | Przywracanie do końca kopii zapasowej |
| FULL | Tak | Opcjonalnie | Tak | Przy kopii dziennika | Przywracanie do wybranego punktu w czasie w kopii zapasowej |

UWAGA W modelach odzyskiwania innych niż SIMPLE punkt kontrolny nie przycina dziennika transakcji.

W modelu FULL konieczne jest tworzenie zarówno pełnych kopii zapasowych, jak i kopii dziennika transakcji. Kopie różnicowe znów są opcjonalne. Kopia

zapasowa dziennika transakcji umożliwia przywrócenie bazy do określonego punktu w czasie, w przeciwieństwie do pełnych i różnicowych kopii zapasowych, gdzie musimy odtworzyć cały zestaw kopii. W modelu FULL kluczowe znaczenie ma regularne tworzenie kopii dziennika transakcji. Wynika to stąd, że w modelu tym dzienniki transakcji nie są automatycznie przycinane. Zamiast tego są one przycinane w ramach procesu tworzenia kopii zapasowej dziennika. Jeśli więc nie będziemy tworzyć kopii zapasowych dziennika, będzie on stale rósł, aż zajmie całą dostępną przestrzeń dyskową.

BULK LOGGED to wyspecjalizowany model, który nie pozwala na przywracanie kopii do dowolnego punktu w czasie. Można go wykorzystać do zwiększenia wydajności operacji hurtowego wstawiania danych. Zazwyczaj stosuje się go tylko przez krótki czas, podczas intensywnego procesu ETL, a normalnym modelem przywracania dla bazy danych jest FULL.

Istnieje powszechne błędne przekonanie, że należy używać modelu FULL dla wszystkich baz danych. Jeszcze bardziej rozpowszechnione jest mylne przesądzenie, że modelu tego należy używać dla wszystkich baz produkcyjnych, ale nie dla baz nieprodukcyjnych. Kierowanie się tymi błędnymi założeniami byłoby poważną pomyłką.

Wybór modelu powinien opierać się przede wszystkim na wymaganiach dotyczących przywracania bazy danych. W przypadku bazy rozwojowej dane zazwyczaj zmieniają się rzadko. Znacznie częściej modyfikowany jest schemat bazy, a jeśli programiści stosują nowoczesne praktyki programistyczne, zmiany te są regularnie zapisywane w systemie kontroli wersji.

WSKAZÓWKA Więcej informacji na temat kontroli wersji znajdziesz w rozdziale 7.

Oznacza to, że dla takiej bazy danych wskaźnik RPO będzie znacznie wyższy niż dla bazy produkcyjnej i prawdopodobnie nie będzie konieczności przywracania jej do konkretnego punktu w czasie. W związku z tym nie ma potrzeby tworzenia kopii zapasowych dziennika transakcji. Wystarczy wykonywać codzienną lub nawet cotygodniową kopię zapasową bazy. Gdybyśmy zastosowali model FULL dla takich baz rozwojowych, musielibyśmy również zarządzać kopiami zapasowymi dziennika transakcji, co oznaczałoby dodatkowy wysiłek bez żadnych korzyści.

Wyjaśnia to, dlaczego model SIMPLE jest często odpowiednim wyborem dla bazy danych w środowisku rozwojowym. Ale co z hurtowniami danych działającymi w środowisku produkcyjnym? Skoro są to bazy produkcyjne, to czy nie powinniśmy mieć znacznie niższego RPO i możliwości przywracania do konkretnego punktu w czasie?

W praktyce niekoniecznie. Weźmy pod uwagę typową hurtownię danych lub bazę analityczną, która jest zasilana danymi w nocy przez proces ETL, a w ciągu dnia wykorzystywana do raportowania. W takim scenariuszu wskaźnik RPO dla bazy danych wynosi faktycznie 24 godziny. Jedynym wymogiem jest możliwość

przywrócenia bazy do stanu po ostatnim uruchomieniu ETL. Dlatego model odzyskiwania SIMPLE jest często najbardziej odpowiednim wyborem.

Spotkałem się z kilkoma dużymi hurtowniami danych, które borykały się z problemami wydajnościowymi podczas nocnych procesów ETL z powodu niewłaściwego wyboru modelu przywracania. Problem polega na tym, że w modelu FULL w dzienniku transakcji zapisywanych jest znacznie więcej informacji. SQL Server musi być w stanie przywrócić bazę danych do dowolnego punktu w czasie. W związku z tym rejestrowanie jest szczegółowe i obejmuje nawet informacje o konkretnych stronach przydzielonych do realizacji transakcji.

Natomiast w modelu SIMPLE SQL Server wykorzystuje dziennik transakcji jedynie do wycofywania transakcji i utrzymywania spójności bazy danych. Dzięki temu zapisywana jest znacznie mniejsza ilość danych. W przeciwieństwie do modelu FULL, który rejestruje każdą zaalokowaną stronę, model SIMPLE zapisuje tylko informacje o alokacji ekstentów. Ponieważ jeden ekstent składa się z ośmiu stron, ilość zapisywanych danych może być nawet osiem razy mniejsza.

Zawsze powinniśmy podejmować najlepszą możliwą decyzję dla konkretnego przypadku użycia. Oznacza to, że należy rozważyć najbardziej odpowiedni model przywracania dla danej bazy danych, zamiast stosować ogólną regułę.

12.7. Numer 84 — używanie modelu SIMPLE dla baz OLTP

W poprzednim podrozdziale omówiliśmy konsekwencje niewłaściwego stosowania modelu przywracania FULL. Tutaj zajmiemy się odwrotną sytuacją i przeanalizujemy skutki nieodpowiedniego wykorzystania modelu SIMPLE.

Spotkałem się z sytuacjami, w których administratorzy baz danych stosowali model odzyskiwania SIMPLE jako domyślny dla wszystkich baz, włącznie z bazami obsługującymi obciążenia OLTP w środowisku produkcyjnym. Jako powód podawali problemy z zapełnianiem się dziennika transakcji oraz dużym obciążeniem systemu, które uniemożliwiało tworzenie więcej niż jednej kopii zapasowej dziennie. Te argumenty są jednak często błędne. Przyjrzyjmy się im bliżej.

Jeśli chodzi o zapełnianie dysku przez dziennik transakcji, omówiliśmy wcześniej przycinanie dziennika i to, że w modelu FULL dzienniki transakcji są przycinane w momencie tworzenia ich kopii zapasowej. Dlatego możemy uniknąć ciągłego rozrastania się dziennika transakcji i zajmowania całej przestrzeni dyskowej przez regularne wykonywanie kopii zapasowych dziennika. Jeśli po backupie dziennik nie zostanie przycięty, należy zbadać przyczynę tego problemu. Kwerenda z listingu 12.6 pozwoli ustalić, dlaczego dziennik nie został przycięty podczas poprzedniej próby.

Listing 12.6. Analizowanie przyczyny nieprzycięcia dziennika

```
SELECT
    name
    , log_reuse_wait_desc
FROM sys.databases ;
```

Wiele przyczyn nieprzycięcia dziennika ma charakter przejściowy i wynika z niefortunnego zbiegu okoliczności. Może to być brak punktu kontrolnego od czasu ostatniej kopii zapasowej, trwający proces tworzenia lub przywracania kopii zapasowej lub trwające tworzenie migawki bazy danych.

Istnieją również inne powody, zwykle przejściowe, które jednak należy zbadać, jeśli występują regularnie i powodują duży przyrost dziennika transakcji. Na przykład częstym powodem jest występowanie aktywnej transakcji. Zazwyczaj taki stan trwa krótko, ale jeśli nasza aplikacja bazodanowa ma wiele długo działających transakcji, może to powodować problemy, nie tylko z kopiami zapasowymi dziennika, ale także z rywalizacją o blokady czy zakleszczeniami. Może programiści będą musieli zająć się tym problemem, aby zapobiec długotrwałym transakcjom.

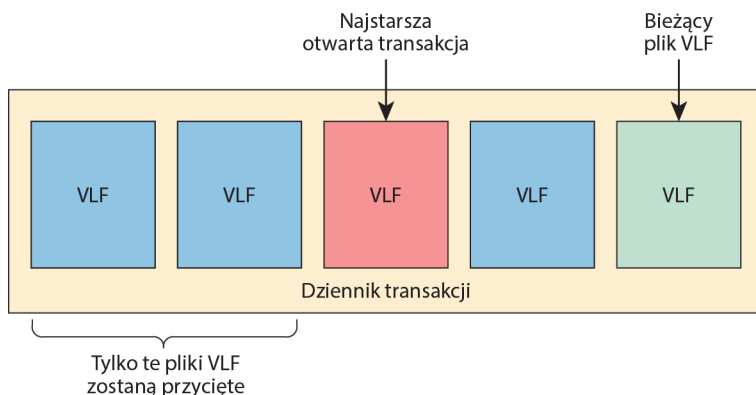
Jeśli przyczyną nieprzycięcia dziennika jest REPLICATION lub AVAILABILITY_↪REPLICA, oznacza to, że transakcje są stosowane do baz wtórnych. Jeśli sytuacja się utrzymuje, może to wskazywać na problem z dostarczaniem lub stosowaniem tych transakcji. Warto sprawdzić narzędzia monitorujące i pulpity nawigacyjne, aby upewnić się, że topologia replikacji lub grupy dostępności AlwaysOn działa prawidłowo, oraz zrozumieć opóźnienia w stosowaniu transakcji.

Typ oczekiwania XTP_CHECKPOINT może wystąpić, gdy korzystamy z tabel przechowywanych w pamięci. Punkty kontrolne tych tabel zapisują na dysku strumienie przechowywanych w nich danych. Problem z przycinaniem dziennika związany z punktami kontrolnymi tabel pamięciowych może wystąpić z kilku powodów. Po pierwsze, w starszych wersjach SQL Servera, takich jak 2014 lub 2016, występowały błędy powodujące problemy z punktami kontrolnymi tabel pamięciowych. W takim przypadku należy zainstalować najnowszy pakiet serwisowy i aktualizację zbiorczą.

Innym powodem jest to, że w przypadku tabel pamięciowych automatyczne punkty kontrolne są wyzwalane dopiero wtedy, gdy dziennik transakcji zawiera 1,5 GB transakcji od ostatniego punktu kontrolnego. W mniejszych systemach może to prowadzić do sytuacji, w której nie jest zapisywana wystarczająca ilość danych, aby kiedykolwiek wyzwolić punkt kontrolny, a w konsekwencji nie dochodzi do przycinania dziennika. Jeśli tak się dzieje, może być konieczne zaplanowanie ręcznego punktu kontrolnego przed wykonaniem kopii zapasowej dziennika transakcji. Jeśli planujemy kopie zapasowe w narzędziu SQL Agent, możemy po prostu dodać krok zadania przed krokiem wykonującym kopię dziennika. Jeśli kopie zapasowe są wykonywane przez korporacyjne narzędzie do backupu, będziemy musieli współpracować z administratorami kopii zapasowych, aby dodać krok poprzedzający tworzenie kopii do zasad backupu.

WSKAZÓWKA Jeśli serwer ma 16 rdzeni i co najmniej 128 GB pamięci RAM, włączone zostaną duże punkty kontrolne. Oznacza to, że punkt kontrolny jest tworzony dopiero po zapisaniu 12 GB danych w dzienniku od czasu ostatniego punktu kontrolnego.

Należy pamiętać, że chociaż mówimy o przycinaniu dziennika, w rzeczywistości przycinane są wirtualne pliki dziennika (ang. *virtual log file*, VLF). Przycinanie może zostać wykonane tylko do pierwszego aktywnego obecnie pliku VLF. Jeśli którykolwiek VLF poza obecnie używanym nie może zostać przycięty, przyczynę pokaże kolumna `log_reuse_wait_desc`. Dziennik transakcji przedstawiony na rysunku 12.7 zawierałby wartość `ACTIVE_TRANSACTION` w kolumnie `log_reuse_wait_desc`, choć dwa pliki VLF zostałyby przycięte.



Rysunek 12.7. Przycinanie pliku VLF

Oznacza to, że wszystkie pliki VLF oprócz jednego mogły zostać przycięte i nie ma żadnego problemu. Aby sprawdzić liczbę używanych plików VLF, możemy uruchomić kwerendę z listingu 12.7. Zwraca ona całkowitą liczbę plików VLF, liczbę używanych plików VLF oraz przyczynę ostatniego nieudanego przycięcia (która jest taka sama jak wartość `log_reuse_wait_desc`) dla bazy danych Marketing. Informacje te pochodzą z funkcji DMF `sys.dm_db_log_stats`.

Listing 12.7. Określanie, ile plików VLF jest aktywnych

```
SELECT
    total_vlf_count
    , active_vlf_count
    , log_truncation_holdup_reason
FROM sys.dm_db_log_stats(DB_ID('Marketing')) ;
```

Jeśli nie używamy modelu FULL, bo uważamy, że system jest zbyt obciążony, by tworzyć więcej niż jedną kopię zapasową dziennie, powinniśmy się zastanowić nad konsekwencjami takiego podejścia. Powinniśmy wziąć pod uwagę dwa główne aspekty.

Najpierw musimy rozważyć wskaźnik RPO dla bazy danych. Jeśli baza jest tak obciążona, że trudno zaplanować wykonywanie wielu kopii zapasowych, prawdopodobnie oznacza to, że dane w niej zmieniają się bardzo szybko. W takiej sytuacji baza może mieć bardzo krótki RPO. Jeśli tak jest, wykonywanie tylko jednej pełnej kopii zapasowej dziennie może nie wystarczyć, aby spełnić wymagania dotyczące przywracania danych.

Drugim aspektem jest wpływ backupu dziennika transakcji. Pełny backup może mocno wpływać na wydajność bazy danych i najlepiej zaplanować go w okresie niskiego lub zerowego obciążenia, ale backup dziennika transakcji ma stosunkowo niewielki wpływ na wydajność. Dziennik transakcji zwykle nie przekracza 20% całkowitego rozmiaru bazy danych i niekoniecznie musi być zapełniony w momencie tworzenia kopii. W rzeczywistości, im częściej tworzymy kopie dziennika transakcji, tym mniej plików VLF musi być uwzględnionych w kopii zapasowej, co dodatkowo zmniejsza obciążenie systemu.

Jeśli backup dziennika transakcji powoduje problemy z wydajnością aplikacji bazodanowej, możemy rozważyć kilka rozwiązań. Jednym z nich jest częstsze wykonywanie kopii dziennika, co skróci czas trwania pojedynczej operacji. W przypadku korzystania z grup dostępności AlwaysOn możemy zaplanować backup na serwerze pomocniczym, co wyeliminuje wpływ na wydajność serwera głównego. Wreszcie, jeśli backup dziennika transakcji nadal powoduje problemy z wydajnością, musimy zastanowić się, czy serwer ma odpowiednią moc obliczeniową. Chociaż zazwyczaj nie jestem zwolennikiem rozwiązywania problemów przez „dorzucanie sprzętu”, serwer musi spełniać wymagania środowiska, w którym jest używany, w tym uwzględniać aspekty pozafunkcjonalne, takie jak RPO.

Jeśli nie używamy modelu FULL dla baz danych z obciążeniem typu OLTP, musimy wziąć pod uwagę ograniczenia funkcjonalności, jakie się z tym wiążą. Technologie zapewniające wysoką dostępność i przywracanie po awarii, takie jak grupy dostępności AlwaysOn czy przekazywanie dzienników, działają wyłącznie w modelu FULL. Replikacja transakcyjna, będąca technologią dystrybucji danych, również wymaga tego modelu do poprawnego funkcjonowania.

Stosowanie modelu odzyskiwania SIMPLE dla bazy danych obsługującej zadania typu OLTP w środowisku produkcyjnym jest zazwyczaj pomyłką. Problemy związane z używaniem modelu FULL można rozwiązać przez zapewnienie odpowiedniej strategii backupu dziennika transakcji lub analizę długotrwałych transakcji i tabel pamięciowych.

Najważniejszym powodem stosowania modelu odzyskiwania FULL jest zapewnienie takiego poziomu RPO, który spełnia wymagania biznesowe. Należy jednak wziąć pod uwagę również technologie wysokiej dostępności i dystrybucji danych, które opierają się na modelu FULL.

12.8. Numer 85 — nietworzenie kopii zapasowej po zmianie modelu przywracania

Częstym błędem podczas promowania bazy danych z etapu przedprodukcyjnego do roli pełnoprawnej bazy produkcyjnej jest zmiana modelu odzyskiwania z SIMPLE na FULL oraz zaplanowanie kopii zapasowych dziennika transakcji obok pełnych kopii zapasowych. Brzmi to całkiem rozsądnie, prawda? Dlaczego więc stanowi to problem?

Wyobraźmy sobie, że wdrażamy naszą bazę danych na produkcji o godzinie 14:00. Mamy zaplanowaną pełną kopię zapasową o północy, a kopie dziennika transakcji są tworzone co 30 minut. Oznacza to, że po wdrożeniu, a przed następną pełną kopią zapasową, zostanie wykonanych 19 kopii dziennika transakcji.

Zobaczmy, jak działa to w praktyce. Skrypt z listingu 12.8 tworzy nową bazę danych o nazwie PromotionDB z modelem odzyskiwania SIMPLE, a następnie wykonuje jej kopię zapasową.

Listing 12.8. Tworzenie bazy danych i wykonywanie pełnej kopii zapasowej

```
CREATE DATABASE PromotionDB ; ← Tworzy bazę danych.
GO

ALTER DATABASE PromotionDB SET RECOVERY SIMPLE ; ← Ustawia model SIMPLE.
GO

USE PromotionDB ;
GO

CREATE TABLE dbo.Incidental (
    ID INT
) ;
GO

BACKUP DATABASE PromotionDB
TO DISK = 'D:\Backups\PromotionDBSimpleBackup.bak' ;
GO ← Wykonuje pełną kopię zapasową bazy danych.
```

Wyobraźmy sobie teraz, że nadszedł czas na promocję bazy danych. Skrypt z listingu 12.9 zmienia model odzyskiwania na FULL. Następnie symuluje aktywność użytkownika poprzez wstawianie pewnych danych do tabeli.

Listing 12.9. Promocja bazy danych

```
ALTER DATABASE PromotionDB SET RECOVERY FULL ;
GO

INSERT INTO dbo.Incidental
SELECT object_id
FROM sys.all_objects ;
```


Następnie za pomocą polecenia z listingu 12.10 możemy zasymulować działanie zadania SQL Server Agent, które wykonuje kopię zapasową dziennika transakcji.

Listing 12.10. Tworzenie kopii zapasowej dziennika transakcji

```
BACKUP LOG PromotionDB
TO DISK = 'D:\Backups\PromotionDBLogBackupInFull.trn' ;
```

Ojej! Próba wykonania tego polecenia kończy się następującym błędem:

```
Msg 4214, Level 16, State 1, Line 1
BACKUP LOG cannot be performed because there is no current database backup.
Msg 3013, Level 16, State 1, Line 1
BACKUP LOG is terminating abnormally.
```

Ale przecież polecenie z listingu 12.8 wykonało pełną kopię zapasową bazy danych. Na czym więc polega problem?

Problem wynika z mało znanego faktu: po zmianie modelu odtwarzania bazy danych z SIMPLE na FULL należy zainicjować nowy model poprzez wykonanie pełnej kopii zapasowej. Jeśli tego nie zrobimy, baza danych w praktyce pozostanie w modelu SIMPLE do momentu utworzenia pełnej kopii zapasowej.

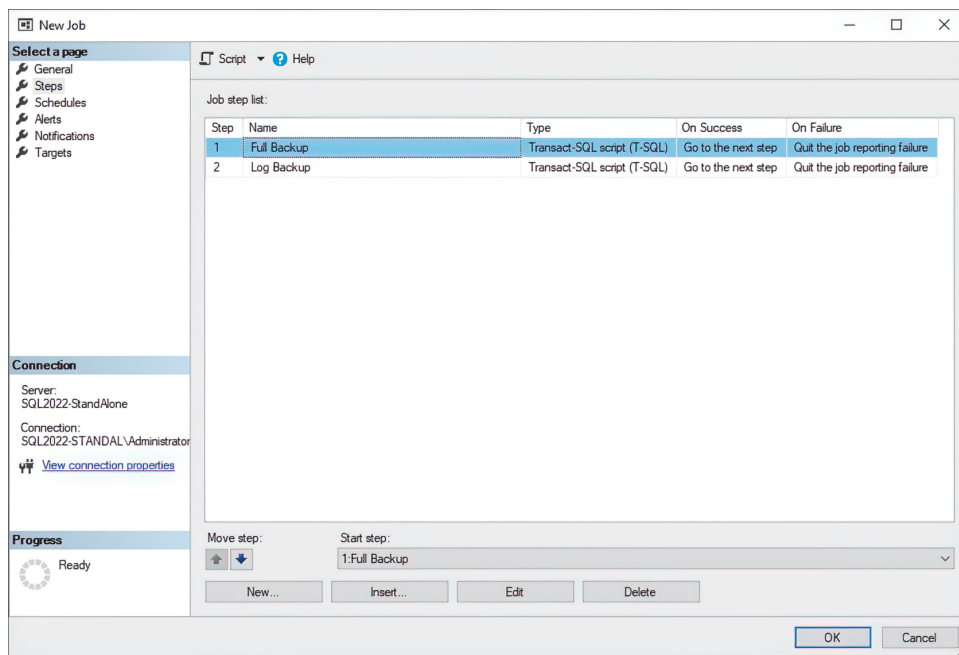
W naszym przykładzie przed następną pełną kopią zapasową bazy danych wystąpi 19 nieudanych prób utworzenia kopii dziennika transakcji. Problem rozwiąże się sam, gdy o północy zostanie wykonana zaplanowana pełna kopia zapasowa, a kolejne kopie dziennika będą tworzone poprawnie. Niestety, nie pomoże to w okresie przejściowym. Jeśli pechowo problem wystąpi pierwszego dnia i zajdzie potrzeba przywrócenia bazy, nie będziemy w stanie tego zrobić.

Co zatem powinniśmy zrobić inaczej? Cóż, to dość proste. Musimy jedynie pamiętać, że po zmianie modelu przywracania bazy danych należy również wykonać pełną kopię zapasową bazy. Powinniśmy zadbać o to, aby było to częścią naszego procesu wdrożeniowego podczas promowania bazy danych.

12.9. Numer 86 — planowanie backupu dziennika natychmiast po backupie pełnym

Niejednokrotnie widziałem, jak przypadkowy administrator bazy danych, wiedząc, że musi wykonywać kopie zapasowe dziennika transakcji, aby umożliwić przywracanie do określonego punktu w czasie, tworzy w narzędziu SQL Agent zadanie, które przeprowadza backup zgodnie z harmonogramem przedstawionym

na rysunku 12.8. Zadanie to wykonuje pełną kopię zapasową bazy danych, a następnie natychmiast tworzy kopię zapasową dziennika transakcji.



Rysunek 12.8. Błędny harmonogram backupu

Tworzenie takiego harmonogramu backupu jest pomyłką, ale przyjrzyjmy się dlaczego. Administrator bazy danych zaplanował backup dziennika transakcji, aby umożliwić odtwarzanie danych do określonego punktu w czasie. Technicznie rzecz biorąc, to rozwiązanie działa. Dziennik transakcji przechowuje wszystkie operacje od momentu utworzenia ostatniej kopii zapasowej dziennika.

Wyobraź sobie, że to zadanie SQL Agent jest zaplanowane do działania co-dziennie o północy. Jeśli o godzinie 10:00 dojdzie do incydentu, w którym ktoś przypadkowo usunie część danych, a my zostaniemy poproszeni o przywrócenie bazy danych do stanu z godziny 9:55, to jesteśmy w stanie to zrobić. Wystarczy wykonać dodatkową kopię zapasową dziennika transakcji, a następnie przywrócić dane dożądanego punktu w czasie.

Problem pojawia się w przypadku katastrofalnej awarii. O godzinie 10:00 uszkodzeniu ulega macierz dyskowa, na której przechowywana jest baza danych, i musimy odtworzyć bazę na nowym serwerze. W tej sytuacji nie możemy wykonać kopii zapasowej dziennika transakcji. Jedyną opcją jest przywrócenie pełnej kopii zapasowej. W takim scenariuszu kopia zapasowa dziennika, którą wykonaliśmy zaraz po pełnej kopii zapasowej bazy, staje się bezużyteczna.

Zamiast tego powinniśmy planować wykonywanie kopii zapasowych dziennika transakcji zgodnie z wymaganym wskaźnikiem RPO dla bazy danych. Jeśli możemy pozwolić sobie na utratę danych z maksymalnie jednej godziny, powinniśmy zaplanować tworzenie kopii dziennika co godzinę. Choć zaplanowanie backupu dziennika bezpośrednio po backupie pełnym umożliwia technicznie przywrócenie bazy danych do dowolnego punktu w czasie, to nie pomaga w osiągnięciu wymaganego RPO.

12.10. Numer 87 — nieużywanie backupu COPY_ONLY do tworzenia doraźnych kopii zapasowych

Czasami zachodzi potrzeba wykonania kopii zapasowej bazy danych poza ustalonym harmonogramem. Często dzieje się tak, gdy chcemy utworzyć punkt przywracania tuż przed wprowadzeniem zmian lub gdy zamierzamy odświeżyć środowisko deweloperskie najnowszymi danymi z systemu produkcyjnego. Jednak wykonanie standardowego backupu pełnego może zakłócić naszą sekwencję przywracania. Wyobraźmy sobie, że mamy zaplanowane następujące kopie zapasowe:

- 1:00 — backup pełny,
- 6:00 — backup różnicowy,
- 12:00 — backup różnicowy,
- 18:00 — backup różnicowy,
- backup dziennika transakcji co godzinę.

W normalnych okolicznościach, w przypadku awarii o godzinie 13:30, nasza sekwencja przywracania wyglądałaby tak:

- Przywróć pełną kopię zapasową z godziny 1:00.
- Przywróć kopię różnicową z godziny 12:00.
- Przywróć kopię dziennika transakcji z godziny 13:00.

Wyobraźmy sobie teraz, że wprowadziliśmy zmianę o godzinie 10:00 i w ramach tej zmiany wykonaliśmy pełną kopię zapasową bazy danych. Zmiana zakończyła się sukcesem, więc usunęliśmy tę kopię.

Problem polega na tym, że gdy wykonaliśmy pełną kopię zapasową o godzinie 10:00, stała się ona bazą dla kopii różnicowych tworzonych o 12:00 i 18:00.

Ponieważ usunęliśmy tę pełną kopię, nie jesteśmy już w stanie przywrócić danych z tych kopii różnicowych.

W związku z tym sekwencja przywracania w naszym scenariuszu wygląda tak:

- Przywróć pełną kopię z godziny 1:00.
- Przywróć kopię różnicową z godziny 6:00.
- Przywróć siedem kopii zapasowych dziennika transakcji, od godziny 7:00 do 13:00.

Jeśli nie wiemy, że kopia zapasowa została wykonana o godzinie 10:00, prawdopodobnie spróbujemy standardowej procedury przywracania i stracimy cenny czas na próby zdiagnozowania, dlaczego nie możemy przywrócić kopii różnicowej z godziny 12:00.

Co powinniśmy zrobić inaczej? Kopie zapasowe w SQL Serverze mają funkcję o nazwie `COPY_ONLY`. Zaprojektowano ją specjalnie z myślą o sytuacjach, w których trzeba wykonać doraźną kopię zapasową poza normalnym harmonogramem. Co istotne, nie wpływa ona na podstawę kopii różnicowych, a tym samym nie zakłóca sekwencji przywracania danych.

Polecenie z listingu 12.11 tworzy kopię zapasową typu `COPY_ONLY` bazy danych Marketing.

Listing 12.11. Wykonywanie kopii zapasowej typu `COPY_ONLY`

```
BACKUP DATABASE Marketing  
TO DISK = 'D:\Backups\MarketingFullCopyOnly.bak'  
WITH COPY_ONLY ;
```

W razie korzystania z kopii różnicowych należy wykonywać doraźne kopie zapasowe z opcją `COPY_ONLY`, aby uniknąć wpływu na podstawę kopii różnicowych i zapobiec zmianie sekwencji przywracania. Osobiście zawsze używam funkcji `COPY_ONLY` dla doraźnych kopii zapasowych, nawet gdy nie są wykonywane kopie różnicowe. Nie ma to żadnych negatywnych skutków, a nie zmusza mnie do sprawdzania harmonogramów backupu.

12.11. Numer 88 — zapominanie, że kopie zapasowe są częścią systemu bezpieczeństwa

Zazwyczaj rozpatrujemy kopie zapasowe z perspektywy odzyskiwania danych. Niestety żyjemy w świecie, w którym musimy brać pod uwagę również zagrożenia, takie jak ataki ransomware. Kopie zapasowe są kluczowym elementem ochrony naszych organizacji przed tego typu atakami, ale tylko wtedy, gdy

uwzględnimy to podczas planowania strategii backupu. Co więc może się stać, jeśli nie weźmiemy pod uwagę zagrożenia ransomware przy opracowywaniu naszej strategii?

Wyobraź sobie, że tworzymy kopie zapasowe przy użyciu wbudowanych funkcji SQL Servera i przechowujemy je na udostępnionym zasobie sieciowym. Nagle atakuje haker i okazuje się, że nasze bazy danych zostały skasowane. Próbujemy je odtworzyć z backupów, ale odkrywamy, że zasób sieciowy został zaszyfrowany i nie mamy do niego dostępu. W najlepszym wypadku będzie to bardzo kosztowne. W najgorszym — grożą nam problemy prawne i utrata reputacji. W tym momencie staje się jasne, że nasza strategia backupu była błędna.

Aby zapobiec skutecznemu atakowi, powinniśmy rozważyć zastosowanie fizycznej izolacji naszych kopii zapasowych. Jeśli tworzymy kopie zapasowe za pomocą korporacyjnego narzędzia do backupu, to w zależności od środowiska prawdopodobnie będą one przechowywane w bibliotece taśmowej, gdzie taśmy można fizycznie wyjąć i tym samym uchronić przed atakiem. Alternatywnie możemy skorzystać z ochrony przed ransomware oferowanej przez dostawcę usług backupu w chmurze. Zazwyczaj polega to na przesyłaniu kopii zapasowych do magazynu w chmurze, który należy do dostawcy usług backupu i jest przez niego zarządzany, w związku z czym również nie jest narażony na atak.

Jeśli jednak tworzymy kopie zapasowe za pomocą natywnych narzędzi SQL Servera, musimy sami wziąć odpowiedzialność za zapewnienie ich ochrony. Najlepszym sposobem na osiągnięcie tego celu jest przeniesienie kopii zapasowych do chmurowego magazynu danych, gdzie możemy wdrożyć odpowiednie zasady przechowywania i zarządzania cyklem życia danych. Na przykład usługa Azure Blob Storage umożliwia zastosowanie zasad retencji opartych na czasie, które obejmują przechowywanie danych w kontenerze w trybie jednokrotnego zapisu i wielokrotnego odczytu, co uniemożliwia modyfikację obiektów binarnych wewnątrz kontenera przez cały okres retencji.

Dodatkowym zabezpieczeniem jest szyfrowanie kopii zapasowych bazy danych. Aby to zrobić, musimy najpierw upewnić się, że w głównej bazie danych istnieje klucz główny. Konieczne jest również utworzenie w tej bazie certyfikatu, który posłuży do szyfrowania kopii zapasowych. Skrypt z listingu 12.12 pokazuje, jak utworzyć niezbędne obiekty kryptograficzne w głównej bazie danych.

Listing 12.12. Tworzenie obiektów kryptograficznych wymaganych do szyfrowania kopii zapasowych

```
USE master ;
GO
```

```
CREATE MASTER KEY ← Tworzy klucz główny bazy danych.
```

```
    ENCRYPTION BY PASSWORD = 'Ha$10' ;
```

```
GO
```

```
CREATE CERTIFICATE CertForBackupEncryption ← Tworzy certyfikat.
```

```
WITH SUBJECT = 'Certyfikat szyfrowania backupu' ;  
GO
```

Zanim zaczniemy korzystać z certyfikatu, musimy koniecznie wykonać jego kopię zapasową i przechowywać ją w bezpiecznym miejscu. Jeśli stracimy certyfikat, nie będzie możliwości odzyskania zaszyfrowanych kopii zapasowych. W praktyce oznaczałoby to, że przeprowadziliśmy atak ransomware sami na sobie! Na listingu 12.13 pokazano, jak wykonać kopię zapasową certyfikatu.

Listing 12.13. Tworzenie kopii zapasowej certyfikatu

```
BACKUP CERTIFICATE CertForBackupEncryption  
TO FILE = 'c:\Certs\CertForBackupEncryptionCert'  
WITH PRIVATE KEY (  
    FILE = 'c:\certs\CertForBackupEncryptionPK' ,  
    ENCRYPTION BY PASSWORD = 'Pa$$w0rd'  
) ;
```

Po utworzeniu tych obiektów możemy użyć polecenia z listingu 12.14 do wykonania zaszyfrowanej kopii zapasowej. Korzystając z opcji `WITH ENCRYPTION`, podajemy pożądany algorytm szyfrowania (w naszym przypadku AES 256) oraz certyfikat, który ma być użyty do zaszyfrowania danych.

Listing 12.14. Wykonywanie zaszyfrowanej kopii zapasowej

```
BACKUP DATABASE Marketing  
TO DISK = 'D:\Backups\MarketingFull256.bak'  
WITH ENCRYPTION (  
    ALGORITHM = AES_256  
    , SERVER CERTIFICATE = CertForBackupEncryption  
) ;
```

Pełna lista obsługiwanych algorytmów szyfrowania w SQL Server 2022 jest następująca:

- AES 128,
- AES 192,
- AES 256,
- Triple DES.

WSKAZÓWKA Należy pamiętać, że im dłuższy algorytm, tym większy jego wpływ na wydajność.

Rozważając kwestię kopii zapasowych, powinniśmy brać pod uwagę zarówno bezpieczeństwo, jak i przywracanie danych po awarii. Musimy zadbać o to, by nasze kopie zapasowe były przechowywane na nośnikach niezmiennych, najlepiej odłączonych od naszej sieci. Powinniśmy również ograniczyć ryzyko poprzez zaszyfrowanie naszych kopii zapasowych.

Podsumowanie

- Przywracanie do punktu w czasie oznacza odtworzenie bazy danych do określonego momentu w trakcie tworzenia kopii zapasowej dziennika transakcji.
- Łańcuch przywracania to sekwencja plików kopii zapasowych, które muszą zostać przywrócone w odpowiedniej kolejności.
- Wskaźnik RPO określa akceptowalną ilość danych, które można utracić w przypadku wystąpienia awarii.
- Wskaźnik RTO określa akceptowalny czas przestoju w przypadku awarii.
- Planując strategię backupu, uwzględnij zarówno RPO, jak i RTO oraz odpowiednio dostosuj swoją strategię.
- Migawki baz danych wykorzystują technologię kopiowania przy zapisie, aby tworzyć „stopklatkę” bazy danych z określonego momentu w czasie.
- Migawka bazy danych może uzupełniać strategię tworzenia kopii zapasowych, ale nie może jej zastąpić, ponieważ opiera się ona na stronach danych w bazie źródłowej.
- Migawki pamięci masowej tworzą kopie zapasowe danych na poziomie bloków i umożliwiają szybkie odtworzenie woluminu.
- Jeśli w SQL Serverze stosowane są migawki pamięci masowej, ważne jest, aby były to migawki spójne na poziomie aplikacji.
- Migawki spójne w razie awarii mogą powodować uszkodzenie bazy danych, ponieważ nie wykorzystują mechanizmu VSS Writer do zapisywania oczekujących operacji wejścia-wyjścia na dysku.
- Ważne jest, aby sprawdzać poprawność wykonanych kopii zapasowych. Można to zrobić za pomocą polecenia `RESTORE WITH VERIFY ONLY`.
- Unikaj tworzenia kopii zapasowych podczas okna ETL, ponieważ nie wszystkie procesy ETL nadają się do ponownego uruchomienia. Może to sprawić, że wykonane kopie zapasowe okażą się bezużyteczne.
- Model przywracania FULL umożliwia wykonywanie kopii zapasowych dziennika transakcji.
- W modelu FULL dziennik transakcji jest przycinany tylko podczas procesu tworzenia jego kopii zapasowej.
- Model przywracania SIMPLE umożliwia wykonywanie tylko pełnych i różnicowych kopii zapasowych.
- W modelu odzyskiwania SIMPLE dziennik transakcji jest automatycznie przycinany przy każdym utworzeniu punktu kontrolnego.

- Punkt kontrolny zapisuje na dysku dane, które zostały zmodyfikowane w pamięci.
- Nie zawsze konieczne jest stosowanie modelu FULL dla nieprodukcyjnych baz danych oraz hurtowni danych, nawet używanych produkcyjnie. Wymagany wskaźnik RPO może tego nie uzasadniać.
- Ogólnie rzecz biorąc, w przypadku produkcyjnych baz danych typu OLTP należy stosować model odzyskiwania FULL. Niezastosowanie tego modelu może negatywnie wpłynąć na wskaźnik RPO i uniemożliwić korzystanie z mechanizmów wysokiej dostępności.
- Po zmianie modelu przywracania bazy danych zawsze należy wykonać pełną kopię zapasową. Jeśli zmienisz model z SIMPLE na FULL, zmiana ta nie zacznie obowiązywać, dopóki nie zostanie wykonana pełna kopia zapasowa bazy danych.
- Nie planuj wykonania pojedynczej kopii zapasowej dziennika transakcji po wykonaniu pełnej kopii zapasowej bazy danych. Umożliwi to odtworzenie do określonego punktu w czasie, ale nie pomoże w spełnieniu wymagań w zakresie RPO.
- Jeśli musisz wykonać doraźną kopię zapasową poza ustalonym harmonogramem, użyj opcji WITH COPY_ONLY. W ten sposób unikniesz wpływu na podstawę kopii różnicowych i nie zakłócisz sekwencji odtwarzania danych.
- Pamiętaj, że kopie zapasowe odgrywają ważną rolę w bezpieczeństwie. W szczególności mogą pomóc w ochronie przed atakami ransomware.
- Upewnij się, że Twoje kopie zapasowe są fizycznie odizolowane od sieci albo przechowywane na nośnikach niezmiennych, z odpowiednią polityką retencji danych.
- Pamiętaj o szyfrowaniu kopii zapasowych, aby zapewnić dodatkową warstwę ochrony przed cyberatakami.

13 Dostępność

W tym rozdziale:

- Różnica między wysoką dostępnością a przywracaniem po awarii
- Wymagania dotyczące dostępności
- Testowanie przywracania po awarii
- Kiedy warto stosować grupy dostępności AlwaysOn
- Przeciążanie klastrów

Wysoka dostępność (ang. *high availability*, HA) i przywracanie po awarii (ang. *disaster recovery*, DR) to kluczowe koncepcje, które powinien dobrze zrozumieć każdy administrator baz danych, aby podejmować właściwe decyzje wdrożeniowe i chronić swoje środowiska przed pojedynczymi punktami podatności na awarię. Temat ten jest zarówno szeroki, jak i głęboki, dlatego w tym rozdziale nie będziemy skupiać się na szczegółach implementacyjnych. Zamiast tego omówimy najczęstsze pomyłki popełniane przez administratorów podczas planowania strategii HA i DR. Jeśli szukasz wskazówek dotyczących wdrażania tych rozwiązań w SQL Serverze, polecam moją książkę *SQL Server 2019 AlwaysOn* dostępną pod adresem <https://mng.bz/znaQ>.

W miarę rozwoju SQL Servera znacznie poszerzył się zakres pojęć i technologii, które muszą poznać administratorzy baz danych, aby wdrażać rozwiązania HA i DR. Na przykład, aby zaimplementować grupy dostępności AlwaysOn (które od teraz będziemy nazywać po prostu grupami dostępności), administratorzy baz

danych muszą rozumieć zasady działania klastrów Windows. Nawet jeśli wdrożenie odbywa się we współpracy z zespołem administratorów Windows, administratorzy baz danych powinni znać te koncepcje.

W tym rozdziale skupimy się najpierw na koncepcjach HA i DR, ponieważ wciąż spotykam się z sytuacjami, w których administratorzy baz danych mylą te pojęcia, co może prowadzić do zagrożeń i ostatecznie do przestojów krytycznych aplikacji biznesowych. Następnie omówimy konsekwencje niedostosowania strategii HA/DR do wymagań biznesowych.

Przyjrzymy się zagrożeniom związanym z brakiem testowania strategii DR, co zdarza się zaskakująco często. Następnie wyjaśnię, dlaczego grupy dostępności nie zawsze są właściwym rozwiązaniem, co pozwoli Ci uniknąć typowej pułapki. Na koniec przeanalizujemy konsekwencje przeciążenia klastra i omówimy, dlaczego może to prowadzić do dłuższych przestojów w razie incydentu.

13.1. Numer 89 — mylenie HA z DR

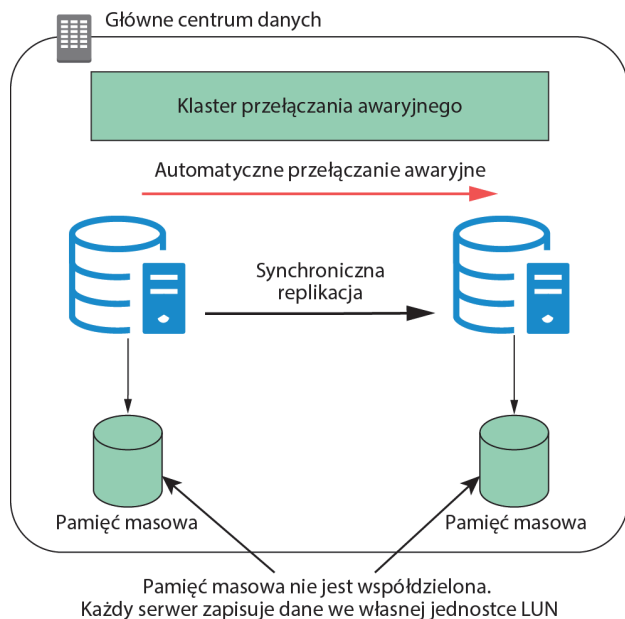
Przywracanie po awarii (ang. *disaster recovery*, DR) odnosi się do mechanizmu używanego do odzyskiwania bazy danych w przypadku poważnego incydentu. Taki incydent może obejmować zarówno uszkodzenie bazy danych, jak i awarię serwera. Może to być nawet utrata całego centrum danych lub strefy dostępności w chmurze. Najprostszą formą DR jest strategia tworzenia i przywracania kopii zapasowych, ale wiele krytycznych aplikacji wymaga rozwiązania zapewniającego szybsze odzyskiwanie w razie katastrofy. Osiąga się to poprzez redundantny sprzęt, zwykle w innej lokalizacji geograficznej, który jest synchronizowany z systemem produkcyjnym. Jest to znane jako *ciepła rezerwa*.

Wysoka dostępność (ang. *high availability*, HA) odnosi się do mechanizmu umożliwiającego automatyczne przełączanie awaryjne i odzyskanie bazy danych w przypadku wystąpienia problemu na serwerze produkcyjnym. Wymaga to redundantnego sprzętu, który często znajduje się w tym samym miejscu co serwer produkcyjny. Bliska lokalizacja zmniejsza opóźnienia sieciowe podczas synchronizacji, a także opóźnienia między klientem a serwerem po przełączeniu awaryjnym. Redundantna infrastruktura w tym scenariuszu nazywana jest *gorącą rezerwą*.

Błędem, który często popełniają administratorzy baz danych, jest mylenie tych dwóch koncepcji. Choć są one podobne i częściowo się pokrywają, służą różnym celom. Dlatego zdarza się, że administratorzy konfiguruje topologię HA, a następnie informują kierownictwo, że wdrożyli system DR. Jeśli rzeczywiście wykonują kopie zapasowe baz danych objętych topologią HA i przechowują je poza siedzibą firmy, to formalnie mają rację. Jednak takie podejście może wprowadzać kierownictwo w błąd co do rzeczywistego stanu zabezpieczeń.

Dlaczego to jest ważne? Odpowiedź jest taka, że firma pozostaje z błędnym przekonaniem, że może szybko przywrócić działanie systemów po awarii o znacznie szerszym zakresie, niż ma to miejsce w rzeczywistości. Doskonałym przykładem jest utrata centrum danych. Może to nastąpić z wielu powodów, w tym z powodu awarii zasilania, błędnej zmiany konfiguracji w głównym przełączniku sieciowym, a nawet klęski żywiołowej.

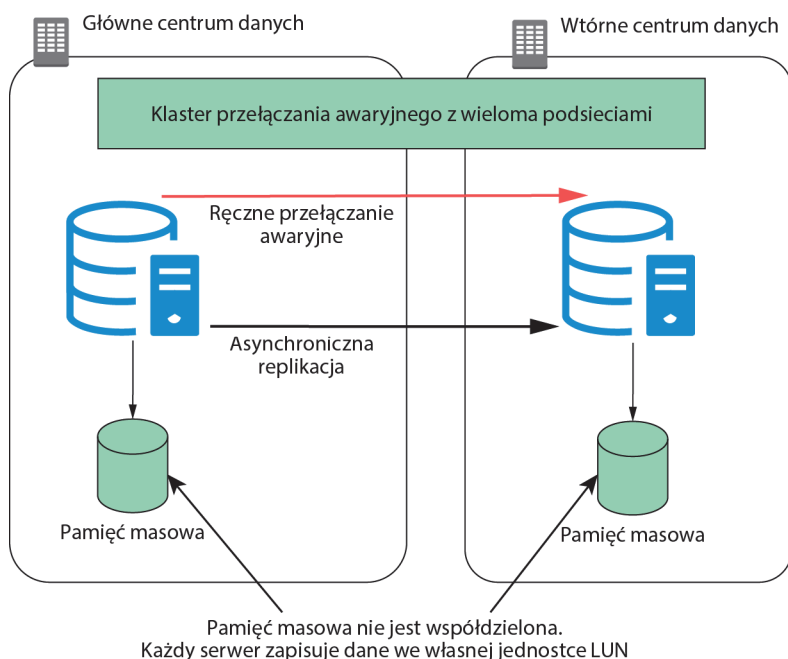
Diagram z rysunku 13.1 przedstawia topologię wysokiej dostępności skonfigurowaną przy użyciu grup dostępności. Topologia składa się z dwóch serwerów, z których każdy znajduje się w innej szafie serwerowej w tym samym centrum danych. Oznacza to, że serwer, szafa i przełącznik sieciowy szafy przestały być pojedynczymi punktami podatności na awarię. To skłoniło administratora baz danych do założenia, że dysponuje systemem DR.



Rysunek 13.1. Topologia HA z grupami dostępności zapewniająca automatyczne przełączanie awaryjne w jednym obiekcie

Problem z traktowaniem tej topologii jako rozwiązania DR polega na tym, że nadal istnieje wiele pojedynczych punktów podatności na awarię, a przywrócenie działania systemu we wszystkich scenariuszach awarii nie jest możliwe. Wyobraźmy sobie na przykład, że w centrum danych nastąpiła przerwa w dostawie prądu lub doszło do zalania. W zależności od topologii centrum danych awaria kluczowego elementu sieciowego, takiego jak przełącznik szkieletowy, mogłaby nawet unieruchomić całe środowisko. W takiej sytuacji administrator bazy danych musiałby uruchomić serwer w innym centrum danych lub w chmurze, a następnie przywrócić dane z najnowszej kopii zapasowej.

Teraz przyjrzyjmy się topologii przedstawionej na rysunku 13.2. Jest to topologia DR, która nie zapewnia HA. Innymi słowy, zapasowy serwer znajduje się w drugim centrum danych i umożliwia szybkie przywrócenie usługi w przypadku katastrofy. Nie zapewnia jednak wysokiej dostępności. Definicja HA zakłada automatyczne przełączanie awaryjne, a ta konfiguracja opiera się na ręcznym procesie przełączania. W tej konfiguracji klaster jest znany jako *klaster przełączania awaryjnego z wieloma podsieciami*, ponieważ ma wirtualne adresy IP przypisane do każdej podsieci, co umożliwia przełączanie awaryjne między lokalizacjami.



Rysunek 13.2. Topologia DR z grupami dostępności umożliwiającą ręczne przełączanie awaryjne między lokalizacjami

WSKAZÓWKA Podsieć to wydzielona część większej sieci. Zazwyczaj każda lokalizacja ma własną podsieć lub podsieci. Takie rozwiązanie zmniejsza ilość ruchu przesyłanego przez łącze WAN, ponieważ zapobiega przesyłaniu między lokalizacjami niepotrzebnego ruchu sieciowego, takiego jak rozgłoszenia.

Dlaczego więc nie skonfigurować HA między dwoma centrami danych i nie upiec dwóch pieczeni na jednym ogniu? W niektórych środowiskach, gdzie istnieje bardzo szybkie połączenie o niskim opóźnieniu między obiema lokalizacjami, może to być możliwe. Jednak ogólnie rzecz biorąc, wyzwaniem jest synchronizacja redundantnej infrastruktury.

Technologie takie jak grupy dostępności zapewniają synchroniczną replikację danych w celu zapewnienia wysokiej dostępności. Oznacza to, że w przypadku

nieoczekiwanej awarii nie dochodzi do utraty danych. Wiąże się to jednak z tym, że każda operacja zapisu musi zostać ukończona na serwerze wtórnym, zanim zostanie zatwierdzona na serwerze głównym. Jeśli serwery nie znajdują się blisko siebie, opóźnienia sieciowe będą większe, co może mieć negatywny wpływ na wydajność systemu.

Grupy dostępności a tradycyjne klastry

Jeśli do zapewnienia wysokiej dostępności wykorzystuje się tradycyjne klastry, to zamiast synchronizacji danych przechowywana jest tylko jedna kopia danych, zazwyczaj w sieci pamięci masowej (ang. *storage area network*, SAN). Oznacza to, że nie występuje opóźnienie sieciowe związane z synchronizacją danych, a co za tym idzie, nie ma problemu z wydajnością.

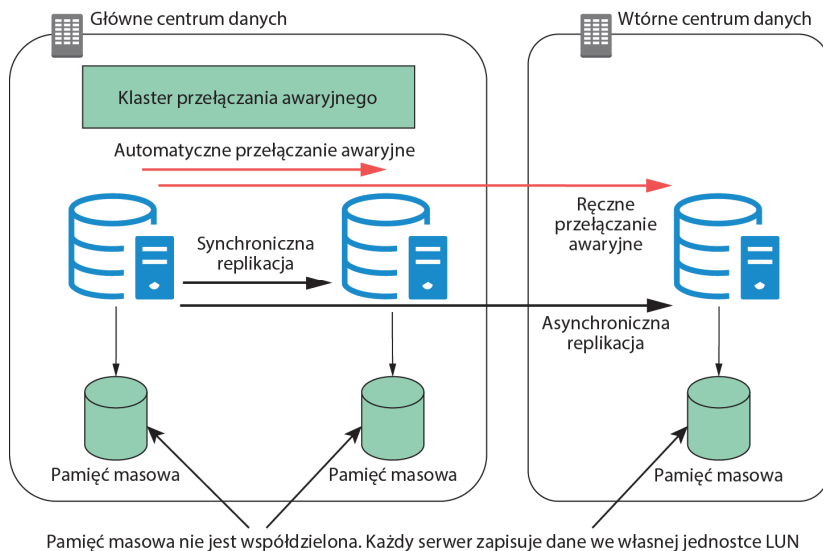
Oznacza to również, że pamięć masowa wprowadza do topologii dodatkowe pojedyncze punkty podatności na awarię — na przykład połączenie z siecią SAN, kontroler pamięci masowej czy samą pamięć masową. W praktyce jednak te punkty podatności na awarię istnieją głównie na papierze. W rzeczywistości klastry zwykle mają redundantne połączenie sieciowe z siecią SAN. Sama sieć SAN składa się z wielu węzłów, a pamięć masowa jest skonfigurowana jako macierz RAID.

Prawdziwą zaletą grup dostępności w porównaniu z tradycyjnymi klastrami jest ich elastyczność. Można je wykorzystać do wdrożenia HA, DR, a nawet skalowania odczytu danych. Tradycyjny klastr może zapewnić jedynie wysoką dostępność, ponieważ znajduje się w pojedynczym centrum danych. Choć możliwe jest skonfigurowanie geograficznie rozproszonego klastra opartego na replikacji SAN między lokalizacjami, często nie jest to zalecane rozwiązanie ze względu na liczne komplikacje występujące podczas przełączania awaryjnego.

Grupy dostępności umożliwiają również bardziej precyzyjną implementację. Klastry działają na poziomie instancji, natomiast w przypadku grup dostępności implementacja odbywa się na poziomie bazy danych. Oznacza to, że można skonfigurować wysoką dostępność tylko dla najważniejszych baz danych, a dzięki temu zmniejszyć obciążenie systemu.

Kiedy technologie takie jak grupy dostępności są używane jako rozwiązanie DR, dane są replikowane asynchronicznie. Oznacza to, że każda operacja zapisu jest najpierw wykonywana na serwerze podstawowym, a następnie replikowana na serwer wtórny. Ponieważ serwer podstawowy nie czeka na potwierdzenie zakończenia zapisu, możemy uniknąć problemów z wydajnością spowodowanych opóźnieniami sieciowymi. Wiąże się to jednak z ryzykiem utraty niewielkiej ilości danych. Ilość utraconych danych będzie zależać zarówno od wydajności sieci, jak i wydajności serwerów. Jeśli serwery mają za małą moc, może powstać kolejka transakcji oczekujących na synchronizację.

Wiele systemów krytycznych wymaga skonfigurowania zarówno HA, jak i DR. Można to osiągnąć na różne sposoby. Możliwe jest łączenie różnych technologii lub po prostu wykorzystanie grup dostępności w całej implementacji. Rozważmy na przykład topologię przedstawioną na rysunku 13.3. W topologii tej grupy dostępności zostały wykorzystane do wdrożenia zarówno HA, jak i DR.



Rysunek 13.3. Topologia HA/DR wykorzystująca grupy dostępności, która zapewnia lokalne automatyczne przełączanie awaryjne oraz ręczne przełączanie awaryjne między lokalizacjami

Administratorzy baz danych powinni rozumieć różnicę między HA a DR. Nieznajomość tych koncepcji może prowadzić do sytuacji, w której firma błędnie zakłada, że jej poziom ochrony jest wyższy od rzeczywistego.

13.2. Numer 90 — nieprojektowanie architektury pod kątem wymagań

Aby określić odpowiednią architekturę HA i DR dla danej aplikacji, musimy zrozumieć jej wymagania dotyczące dostępności. Problem polega na tym, że gdy pytamy osobę odpowiedzialną za aplikację: „Jak długo twoja aplikacja może być niedostępna?”, odpowiedź niezmiennie brzmi: „Wcale nie może!”.

Na pierwszy rzut oka wydaje się to logiczne. Teoretycznie moglibyśmy zbudować pełną topologię HA i DR dla każdej aplikacji w naszej organizacji. Problem w tym, że szybko staje się to bardzo kosztowne. Jeśli mamy szczęście i w projekt zaangażowani są architekt rozwiązań oraz analityk biznesowy, to jest to kwestia mniejszej wagi, ponieważ to oni będą odpowiedzialni za zrozumienie wymagań i zdefiniowanie odpowiedniego rozwiązania. Jako administratorzy baz danych będziemy po prostu wdrażać ich projekt.

Niestety nie zawsze mamy szczęście dysponować zasobami architektonicznymi lub analitycznymi. Oznacza to, że administratorzy baz danych czasami muszą

wypełnić tę lukę. Często pomyłką, którą popełniają administratorzy w takiej sytuacji, jest bezkrytyczne akceptowanie wymagań dotyczących dostępności określanych przez osoby odpowiedzialne za aplikację. Kiedy jednak osoby te podejmują decyzje w oderwaniu od kontekstu, często nie zdają sobie sprawy z konsekwencji swoich żądań.

Aby zrozumieć to zagadnienie, musimy najpierw poznać poziomy dostępności. Być może słyszałeś, jak ludzie mówią o wymaganej dostępności na poziomie czterech czy pięciu dziewiątek, ale co to właściwie oznacza w praktyce?

Poziom dostępności odnosi się do procentowego czasu, w którym aplikacja jest dostępna dla użytkowników. Liczba dziewiątek w wymaganiach określa procent czasu, w którym aplikacja powinna poprawnie działać. Na przykład cztery dziewiątki oznaczają, że aplikacja jest dostępna przez 99,99% czasu, a pięć dziewiątek oznacza dostępność przez 99,999% czasu.

Dopuszczalne czasy przestoju dla każdego z tych poziomów dostępności zostały szczegółowo przedstawione w tabeli 13.1.

Tabela 13.1. *Poziomy dostępności*

| Poziom dostępności | Przestój tygodniowy | Przestój miesięczny |
|--------------------|--------------------------------|-------------------------------|
| 99% | 1 godzina, 40 minut, 48 sekund | 7 godzin, 18 minut, 17 sekund |
| 99,9% | 10 minut, 4 sekundy | 43 minuty, 49 sekund |
| 99,99% | 1 minuta | 4 minuty, 23 sekundy |
| 99,999% | 6 sekund | 26 sekund |

Analizując te dane, możemy zauważyć, że osiągnięcie dostępności na poziomie pięciu dziewiątek nie jest łatwym zadaniem. Sześciusekundowa przerwa w działaniu może być spowodowana czymś tak prostym, jak chwilowe zakłócenie w sieci. Aby dodać nieco więcej kontekstu, warto zauważyć, że gdybyśmy chcieli uruchomić serwer na jednej z popularnych platform chmurowych, takiej jak Azure, w momencie pisania tej książki otrzymalibyśmy umowę SLA gwarantującą dostępność na poziomie 95% miesięcznie dla pojedynczego serwera.

Aby osiągnąć pełną dostępność na poziomie pięciu dziewiątek, konieczne jest zainwestowanie znacznych środków finansowych zarówno w lokalną, jak i geograficznie rozproszoną infrastrukturę nadmiarową, wraz z redundantnymi prywatnymi łączami między lokalizacjami. W większości przypadków takie rozwiązanie po prostu nie jest opłacalne.

Aby naprawdę zrozumieć wymagania, musimy poznać koszt przestoju danej aplikacji. Wymaga to zrozumienia zarówno wymiernych, jak i niewymiernych kosztów związanych z niedostępnością aplikacji. Koszty wymierne są stosunkowo proste do obliczenia. Na przykład, jeśli mamy aplikację sprzedażową, wymiernym kosztem jest utrata przychodów spowodowana niemożnością składania zamówień przez klientów. Koszt godzinowy można obliczyć, biorąc miesięczne dane sprzedażowe, mnożąc je przez 12, a następnie dzieląc tę liczbę przez 8760 (liczba godzin w roku).

Koszty niematerialne są znacznie trudniejsze do oszacowania, ale mogą być w rzeczywistości wyższe. Na przykład jeśli klient nie może złożyć zamówienia, może skorzystać z usług konkurencji i już nigdy nie wrócić. Inne koszty niematerialne mogą obejmować spadek morale zespołu sprzedażowego, prowadzący do zwiększonej rotacji pracowników, a nawet utratę reputacji firmy. W niektórych sytuacjach może to nawet prowadzić do niezgodności z przepisami. Ponieważ koszty niematerialne można jedynie szacować, w branży przyjmuje się zasadę mnożenia kosztów materialnych przez 3.

Po obliczeniu godzinowego kosztu przestoju wartość tę można odnieść do całego cyklu życia aplikacji. Następnie można porównać ten koszt z wydatkami na infrastrukturę niezbędną do zapewnienia wymaganego poziomu dostępności. Wyobraźmy sobie na przykład, że właściciel aplikacji ustalił, iż koszt przestoju wynosi 1000 zł na godzinę, a przewidywany cykl życia aplikacji to trzy lata. Całkowity koszt przestoju przy dostępności na poziomie dwóch dziewiątek (99%) wyniósłby 87 650 zł. Można to obliczyć za pomocą poniższego wzoru.

$$(\text{Liczba godzin przestoju rocznie} \cdot \text{Lata cyklu życia aplikacji}) \cdot \text{Koszt godziny przestoju}$$

Następnie możemy porównać to z kosztem infrastruktury potrzebnej do spełnienia tych wymagań. Przykład takiego porównania znajduje się w tabeli 13.2. W obliczeniach przyjęto, że koszt przestoju wynosi 1000 zł na godzinę.

UWAGA Koszty w tych obliczeniach są jedynie przykładowe. Oczywiście będą się one różnić w zależności od konkretnej aplikacji.

Tabela 13.2. Porównanie kosztów przestojów

| Poziom dostępności | Łączny koszt przestojów | Koszt rozwiązania HA/DR | Koszt przestojów + koszt rozwiązania HA/DR |
|--------------------|-------------------------|-------------------------|--|
| 99% | 262 810 zł | 0 zł* | 262 810 zł |
| 99,9% | 26 290 zł | 89 193 zł | 115 483 zł |
| 99,99% | 2630 zł | 100 389 zł | 103 019 zł |
| 99,999% | 259 zł | 301 167 zł | 301 696 zł |

* Możliwy do osiągnięcia bez architektury odpornej na awarie, pod warunkiem dostępności kopii zapasowych.

W tym przykładzie koszt osiągnięcia dostępności na poziomie pięciu dziewiątek wynosi 301 696 zł. Jest to więcej niż koszt przestoju bez redundantnej infrastruktury, który wynosi 262 810 zł. Dlatego takie rozwiązanie nie jest ekonomiczne. Osiągnięcie poziomu czterech dziewiątek przyniosłoby oszczędność 159 791 zł w porównaniu z brakiem redundantnej infrastruktury (262 810 – 103 019 dolarów). Z kolei osiągnięcie poziomu trzech dziewiątek dałoby oszczędność jedynie 147 327 zł (262 810 – 115 483 dolarów). W związku z tym w tym scenariuszu najbardziej odpowiednim poziomem dostępności byłyby cztery

dziwiątki. Gdy przedstawimy osobie odpowiedzialnej za aplikację finansowe implikacje jego wymagań, będzie w stanie zrozumieć, dlaczego zerowy poziom przestojów nie jest rozsądnym oczekiwaniem.

W przypadku infrastruktury lokalnej najlepsze rozwiązanie umożliwiające osiągnięcie wymaganego poziomu dostępności będzie zależeć od infrastruktury organizacji. Warto omówić to z zespołem odpowiedzialnym za infrastrukturę. Zazwyczaj jednak, aby uzyskać dostępność na poziomie czterech dziewiątek, konieczne będzie zastosowanie topologii HA i DR obejmującej wiele lokalizacji, podobnej do projektu przedstawionego na rysunku 13.3.

Dostawcy usług chmurowych oferują różne umowy o gwarantowanym poziomie świadczenia usług (ang. *service level agreement*, SLA). Jednak w czasie pisanie tej książki, w przypadku Azure, w celu osiągnięcia dostępności na poziomie czterech dziewiątek (99,99%) wymagane są dwie maszyny wirtualne rozdzielone między dwie strefy dostępności dla rozwiązania typu infrastruktura jako usługa (IaaS) lub baza danych Azure SQL skonfigurowana na poziomie Business Critical lub Premium.

Przed przystąpieniem do konfiguracji HA/DR niezwykle istotne jest zrozumienie wymagań. Jeśli nie mamy dostępu do architekta rozwiązań lub analityka biznesowego, będziemy musieli wspierać firmę w zrozumieniu wymagań dla danej aplikacji. Zawsze powinniśmy zachęcać osoby odpowiedzialne za aplikację do opierania wymagań na analizie kosztów i korzyści.

13.3. Numer 91 — nietestowanie strategii DR

Bardzo częstą pomyłką popełnianą przez przypadkowych administratorów baz danych jest nietestowanie strategii DR. Ich rozumowanie opiera się na tym, że widzą zsynchronizowane dane, więc co może pójść nie tak? Niestety odpowiedź na to pytanie brzmi — wiele rzeczy. Zawsze lepiej jest odkryć problemy podczas kontrolowanego testu niż o drugiej w nocy, gdy konieczne jest rzeczywiste odzyskanie danych.

Zacznijmy od rozważenia niektórych codziennych problemów, które mogą wystąpić podczas przełączania awaryjnego. Najczęstszym z nich jest wydajność węzła zapasowego. Wyobraźmy sobie, że przenosimy grupę dostępności z repliki podstawowej na wtórną w innym centrum danych. Przełączenie działa, ale użytkownicy natychmiast zaczynają narzekać, że wydajność jest tak słaba, że nie mogą pracować. Może to być spowodowane wieloma czynnikami, od opóźnień sieciowych w łączności z lokalizacją zapasową po zbyt słabą specyfikację repliki zapasowej, która nie radzi sobie z obciążeniem.

Innym częstym powodem niemożności przełączenia awaryjnego są uprawnienia. Aby przełączenie się powiodło, konto SYSTEM musi mieć uprawnienia VIEW SERVER STATE, CONNECT SQL oraz ALTER ANY AVAILABILITY GROUP w replice wtórnej. Spotkałem się z sytuacjami, w których administratorzy baz danych, dbający o bezpieczeństwo, usunęli uprawnienia z konta SYSTEM, nie zdając sobie sprawy, jakie będzie to miało konsekwencje dla grup dostępności.

Trzeci najczęstszy problem, z jakim się spotykam, dotyczy szyfrowania. Jeśli w replice wtórnej skonfigurowana jest opcja FORCE PROTOCOL ENCRYPTION, ale sama replika wtórna nie jest skonfigurowana do szyfrowania danych, monitorowanie stanu repliki wtórnej nie będzie mogło połączyć się z repliką lokalną, co jest częścią procesu przełączania awaryjnego.

To tylko niewielki przykład problemów, które mogą wystąpić podczas próby przełączenia awaryjnego grup dostępności. Oczywiście inne technologie HA/DR, takie jak klastry czy przesyłanie dzienników, mają swoje własne potencjalne problemy. W rozdziale 12. wspomnieliśmy o powiedzeniu: „Nie masz kopii zapasowej, dopóki jej nie odtworzysz”. Podobna filozofia głosi: „Nie masz DR, dopóki nie przeprowadzisz przełączenia awaryjnego”.

Należy zatem regularnie testować strategię DR. Częstotliwość tych testów będzie zależeć od liczby obsługiwanych aplikacji i zasad obowiązujących w organizacji. Typowo przyjmuje się, że każda aplikacja powinna przechodzić test DR raz w roku. Zazwyczaj rozsądne jest wykonywanie go jako zadania cyklicznego i testowanie niewielkiej liczby aplikacji co tydzień lub miesiąc. Pozwala to na włączenie tego procesu do standardowych, rutynowych działań.

Organizowanie testu DR wymaga, aby ustalić z osobą odpowiedzialną za aplikację okno czasowe na przestój. Należy zaplanować dłuższe okno niż przewidywany czas przełączania awaryjnego, na wypadek wystąpienia problemów wymagających rozwiązania. Ważne jest również zapewnienie dostępności jednego lub kilku użytkowników, którzy przetestują możliwość połączenia z aplikacją i sprawdzą, czy działa ona z akceptowalną wydajnością podczas testu. W zależności od wykorzystywanych technologii i umiejętności zespołu administratorów baz danych może być konieczne zaangażowanie innych zespołów. Jeśli na przykład w zespole brakuje specjalistów od systemu Windows, a korzystamy z grup dostępności lub klastrów przełączania awaryjnego jako technologii HA/DR, warto poprosić o pomoc inżyniera Windows. Z kolei w przypadku geoklastra prawdopodobnie będziemy potrzebować inżyniera SAN do przeprowadzenia przełączenia awaryjnego pamięci masowej oraz wsparcia w rozwiązywaniu ewentualnych problemów.

Przeprowadzanie testów DR może wydawać się uciążliwym obowiązkiem zarówno dla administratorów baz danych, jak i dla biznesu, ale jest niezwykle istotne. Znacznie lepiej jest odkryć, że nie możemy przełączyć aplikacji podczas kontrolowanego testu z zaplanowanym przestojem, niż dowiedzieć się o tym w środku poważnego incydentu.

13.4. Numer 92 — zakładanie, że grupy dostępności zawsze są właściwym rozwiązaniem

Od czasu wprowadzenia grup dostępności stały się one preferowaną technologią HA/DR administratorów baz danych. Prowadzi to jednak do błędnego przekonania, że grupy dostępności są zawsze najlepszym wyborem dla wszystkich aplikacji bazodanowych. To nieporozumienie może prowadzić do problemów, jeśli nie zrozumiemy ograniczeń grup dostępności oraz innych dostępnych opcji.

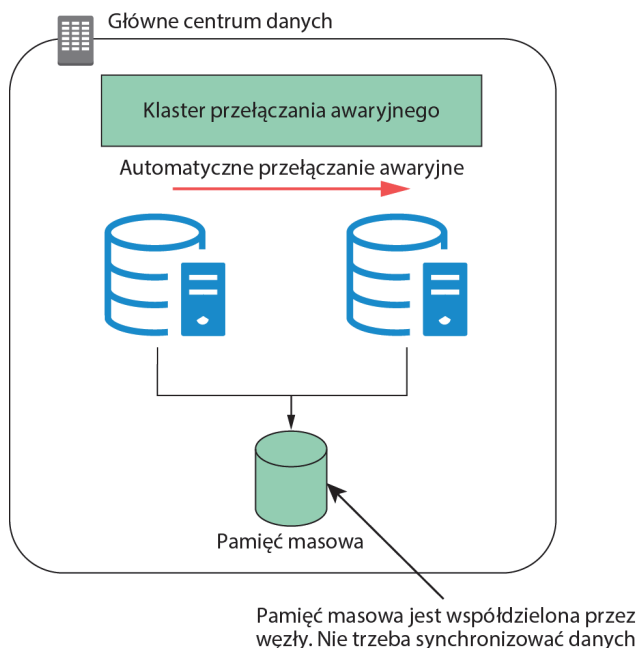
Wyobraź sobie, że mamy dużą hurtownię danych, która każdej nocy przetwarza ogromne ilości informacji. Raporty generowane przez tę hurtownię są kluczowe dla działalności firmy, dlatego niezbędne jest zapewnienie wysokiej dostępności systemu. Konfigurujemy grupy dostępności w trybie synchronicznym z automatycznym przełączaniem awaryjnym. Niestety zespół aplikacyjny zgłasza, że procesy ekstrakcji, transformacji i wczytywania danych nie mieszczą się już w wyznaczonym oknie czasowym.

Po przeprowadzeniu analizy stwierdziliśmy, że prawdopodobną przyczyną problemu jest synchroniczna replikacja między kopiami bazy danych. Bardzo duża liczba transakcji, która występuje podczas procesu ETL, powoduje wydłużanie się kolejki zatwierdzeń w replice wtórnej, a replika podstawowa nie może zatwierdzić transakcji, dopóki nie otrzyma potwierdzenia, że zatwierdzenie powiodło się w replice wtórnej. To właśnie ten mechanizm zapewnia, że nie dojdzie do utraty danych. Aby rozwiązać ten problem, zmieniamy konfigurację grupy dostępności na zatwierdzanie asynchroniczne. Oczywiście oznacza to, że musimy również przejść na ręczne przełączanie awaryjne. Nie spełnia to wymogu automatycznego przełączania awaryjnego, ale mamy nadzieję, że będzie to kompromisowe rozwiązanie, które okaże się wystarczające.

Niestety zespół aplikacji informuje, że mimo znacznej poprawy wydajności proces ETL nadal nie kończy się w wyznaczonym oknie czasowym. Musimy więc wrócić do analizy i zbadać, co jeszcze może powodować ten problem.

Po dokładniejszym zbadaniu sprawy okazuje się, że w celu wdrożenia grupy dostępności musieliśmy zmienić model odzyskiwania bazy z hurtownią danych z SIMPLE na FULL. Powoduje to znaczne nasilenie operacji wejścia-wyjścia, ponieważ każda transakcja jest w pełni rejestrowana w dzienniku. Niestety nie ma innego wyjścia. Albo pogodzimy się z tym, że proces ETL będzie trwał znacznie dłużej, co nie spełnia potrzeb biznesowych, albo zaakceptujemy brak rozwiązania HA czy nawet DR, co również nie spełnia wymagań biznesowych.

To doskonały przykład sytuacji, w której grupy dostępności nie są odpowiednią technologią HA/DR dla bazy danych. Spotkałem się z takim przypadkiem niejednokrotnie. Aby spełnić wymagania biznesowe w tym scenariuszu, najlepiej będzie zastosować klastery przełączania awaryjnego. W rozwiązaniu tym dane są przechowywane tylko w jednym miejscu, a w przypadku awarii pamięć masowa jest odłączana od uszkodzonego węzła i podłączana do węzła zapasowego. Pokazano to na rysunku 13.4.



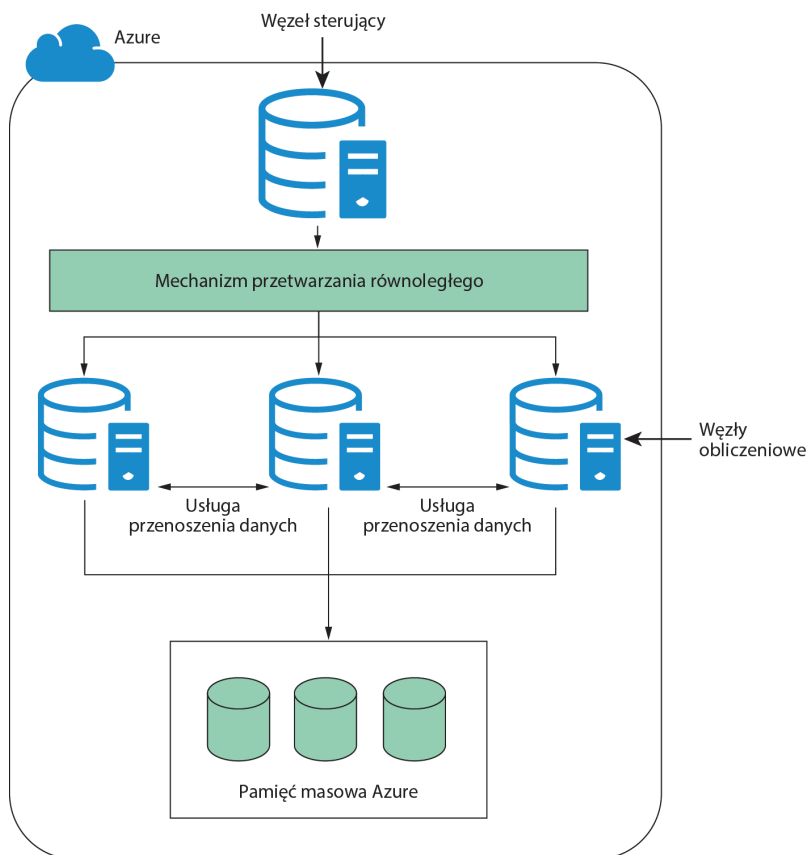
Rysunek 13.4. HA z klastrem przełączania awaryjnego zapewniającym automatyczne przełączanie awaryjne w jednej lokalizacji

Klaster z przełączaniem awaryjnym obecnie wyszły z mody, ale jeśli masz lokalne hurtownie danych lub inne aplikacje z dużą liczbą operacji zapisu może to nadal być rozwiązanie lepsze od grup dostępności. Ponieważ dane nie muszą być synchronizowane, nie trzeba używać modelu przywracania FULL, więc można uniknąć pełnego rejestrowania transakcji w dzienniku.

WSKAZÓWKA Więcej informacji na temat wyboru modelu przywracania znajdziesz w rozdziale 12.

Jeśli planujesz hostować hurtownię danych w chmurze i potrzebujesz wysokiej dostępności, dobrym rozwiązaniem może być dedykowana pula SQL w Azure Synapse. Ta technologia, wcześniej znana jako Parallel Data Warehouse, wykorzystuje masowe przetwarzanie równoległe do realizacji rozproszonych kwerend. W tej architekturze kwerendy są wykonywane w węźle sterującym, który następnie

dystrybuuje je do węzłów obliczeniowych. Dane są przechowywane w magazynie Azure, a specjalna usługa koordynuje przenoszenie danych między węzłami obliczeniowymi, gdy jest to konieczne do realizacji kwerendy. Pokazano to na rysunku 13.5.



Rysunek 13.5. Dedykowana pula SQL w Azure Synapse

Czy kiedykolwiek warto stosować przesyłanie dzienników lub replikację?

Przesyłanie dzienników (ang. *log shipping*) to stara metoda DR, która polega na tworzeniu kopii zapasowych dzienników transakcji, przesyłaniu ich do serwera pomocniczego i przywracaniu w bazie danych. Podobnie jak grupy dostępności, wymaga ustawienia modelu odzyskiwania bazy danych na FULL. Ma jednak znacznie mniej zalet. Można ją stosować tylko do celów DR, a wtórnej bazy danych nie można używać do momentu przełączenia. Oznacza to, że nie da się jej wykorzystać do skalowania odczytu ani przenoszenia zadań administracyjnych. Ponadto po przełączeniu, jeśli nie chcemy wrócić do pierwotnej konfiguracji, musimy ponownie skonfigurować przesyłanie dzienników w przeciwnym kierunku. Po przełączeniu aplikacje muszą też zostać przekierowane na instancję pomocniczą, ponieważ to rozwiązanie nie korzysta z klastrów.

Istnieje tylko jedna sytuacja, w której wciąż warto rozważyć przesyłanie dzienników, choć sam od dawna tego nie stosowałem. Jest to scenariusz, w którym firma chce opóźnić stosowanie danych w bazie wtórnej, aby mieć czas na cofnięcie przypadkowej pomyłki użytkownika.

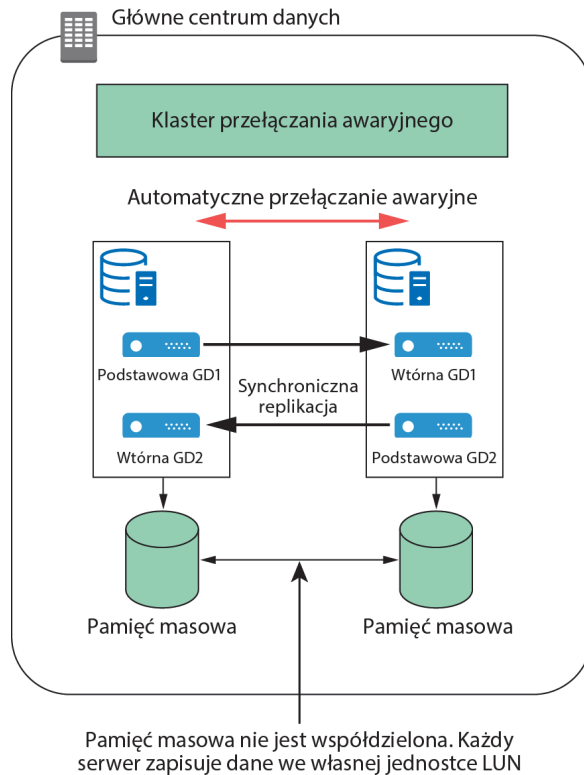
Scenariusz ten nie może być obsługiwany przez grupy dostępności, ale w przypadku przesyłania dzienników wystarczy zwiększyć częstotliwość, z jaką dzienniki są ponownie stosowane w bazie wtórnej.

Replikacja to technologia służąca do rozpraszania danych. Istnieje kilka różnych rodzajów replikacji, m.in. transakcyjna, ze scalaniem i migawkowa. Wiele lat temu, zanim wprowadzono technologie takie jak grupy dostępności, niektórzy używali jej jako narzędzia DR, aby obejść pewne ograniczenia przesyłania dzienników. Nie dość, że jest to niezgodne z przeznaczeniem replikacji, to jeszcze narzędzie to jest bardzo złożone, a zapewnienie jego niezawodności często wymaga dużego nakładu pracy. W dzisiejszych czasach nie ma przypadków, w których replikacja byłaby odpowiednim narzędziem DR.

Przy wdrażaniu rozwiązania HA/DR kluczowe jest wybranie odpowiedniej technologii do danego zadania. Nie ma wątpliwości, że grupy dostępności są zazwyczaj właściwym wyborem. Nie dotyczy to jednak wszystkich scenariuszy, więc musimy zrozumieć ich ograniczenia oraz wiedzieć, kiedy powinniśmy ich unikać. Jeśli mamy dużą hurtownię danych lub nawet aplikację OLTP z ogromną liczbą operacji zapisu, grupy dostępności mogą nie być najlepszym wyborem — szczególnie jeśli wymagana jest wysoka dostępność. W takim przypadku nadal warto rozważyć klastrer przełączania awaryjnego. Możemy również zdecydować się na wykorzystanie odpowiedniej technologii chmurowej, takiej jak dedykowana pula SQL w Azure Synapse.

13.5. Numer 93 — przeciążanie klastra

Większość organizacji stara się ograniczać koszty, a nadmiarowy sprzęt jest drogi. Dlatego powszechną praktyką jest stosowanie klastrów aktywny-aktywny zarówno w topologiach przełączania awaryjnego, jak i w grupach dostępności. W klastrze aktywny-aktywny wszystkie węzły i repliki mogą przechowywać produkcyjne bazy danych. W przypadku awarii któregośkolwiek węzła w klastrze bazy danych znajdujące się w tym węźle są automatycznie przełączane na jeden z pozostałych węzłów. Zilustrowano to na rysunku 13.6 przedstawiającym klastrer dwuwęzłowy, w którym każda replika hostuje produkcyjną grupę dostępności.



Rysunek 13.6. *Klaster aktywny-aktywny wykorzystujący grupy dostępności do zapewnienia automatycznego przełączania awaryjnego w jednej lokalizacji*

Konfiguracja typu aktywny-aktywny to nie równoważenie obciążenia

Należy pamiętać, że klaster typu aktywny-aktywny nie jest rozwiązaniem służącym do równoważenia obciążenia. W danym momencie tylko jedna kopia bazy danych może być aktywna dla operacji zapisu. Jedyną technologią umożliwiającą wykorzystanie jakiegokolwiek formy równoważenia obciążenia (dla operacji zapisu) jest dedykowana pula SQL w Azure Synapse. Wykorzystuje ona technologię przetwarzania masowo równoległego do koordynowania transakcji i synchronizacji danych.

Może słyszałeś też, że do równoważenia obciążenia można wykorzystać technologię replikacji peer-to-peer, ale nie jest to do końca prawda. Replikacja peer-to-peer opiera się na replikacji transakcyjnej i umożliwia aktualizację wielu wersji bazy danych jednocześnie.

W tym przypadku przed bazami danych nie ma żadnego klastra ani kontrolera. Aplikacja musi łączyć się bezpośrednio z jednym z węzłów. Co więcej, obsługa konfliktów jest co najwyżej podstawowa. W przypadku wystąpienia konfliktu agent dystrybucji zatrzyma się i zostanie wygenerowane powiadomienie. Administrator bazy danych musi wtedy ponownie zainicjować replikę albo ręcznie zsynchronizować zmiany. W praktyce oznacza to, że w większości przypadków tę technologię można skutecznie wykorzystać jedynie do aktualizacji różnych obszarów bazy danych.

Projekt przedstawiony na rysunku 13.6 umożliwia rozłożenie obciążenia na wiele węzłów w klastrze. Dzięki temu wykorzystywana jest cała infrastruktura, a jednocześnie zachowana zostaje możliwość przełączania awaryjnego. Wiąże się to z ryzykiem, że w przypadku awarii wydajność może być chwilowo obniżona, ale będzie to tylko krótkotrwały problem, ponieważ administrator bazy danych przywróci bazy w pierwotnym węźle, gdy tylko usterka zostanie usunięta. Takie rozwiązanie zapewnia dobry kompromis między niezawodnością a optymalizacją kosztów.

Pomyłka, którą czasem obserwuję, polega na tym, że obciążenie w tej topologii zaczyna rosnąć. Niekiedy wynika to z organicznego rozwoju aplikacji, a innym razem z dodawania kolejnych grup dostępności lub instancji klastra przełączania awaryjnego do każdego węzła.

Może to doprowadzić do sytuacji, w której w przypadku przełączenia awaryjnego węzeł będący podstawowym dla wszystkich baz danych po prostu nie ma wystarczającej przepustowości, aby obsłużyć wszystkie kwerendy. W rezultacie może dojść do spadku wydajności, przekroczenia limitów czasu i utraty monitorowania. Spotkałem się nawet z przypadkiem, w którym system Windows w ogóle nie był w stanie uruchomić przełączonej awaryjnie instancji.

Jako administratorzy baz danych musimy za wszelką cenę unikać takiej sytuacji. Istnieją dwie metody, których połączenie może pomóc w osiągnięciu tego celu. Pierwszą jest po prostu planowanie wydajności aplikacji bazodanowych. Powinniśmy to robić jeszcze przed wdrożeniem aplikacji, a następnie aktywnie monitorować trendy, aby upewnić się, że jeden węzeł ma wystarczające zasoby do obsługi wszystkich aplikacji, nawet w przypadku ich intensywnego działania. Więcej informacji na temat planowania wydajności można znaleźć w rozdziale 9.

Druga metoda polega na regularnym przeprowadzaniu testów DR. Daje to możliwość przećwiczenia scenariusza przełączenia awaryjnego i upewnienia się, że wydajność systemu nadal spełnia wymagania biznesowe. Temat ten jest szerzej omówiony w podrozdziale 13.3.

Podsumowanie

- Wysoka dostępność (ang. *high availability*, HA) odnosi się do automatycznego przełączania awaryjnego bazy danych w przypadku awarii.
- Odzyskiwanie po awarii (ang. *disaster recovery*, DR) odnosi się do możliwości przywrócenia bazy danych w przypadku wystąpienia poważnego incydentu.
- HA zwykle konfiguruje się między serwerami, które są fizycznie blisko siebie i charakteryzują się niskim opóźnieniem.

- DR zazwyczaj konfiguruje się między serwerami znajdującymi się w różnych lokalizacjach geograficznych, aby aplikacja mogła funkcjonować nawet w przypadku awarii jednego z centrów danych.
- Poziom dostępności aplikacji odnosi się do procentowego czasu, w którym jest ona gotowa do użycia. Często wyraża się go „liczbą dziewiątek”.
- Należy upewnić się, że rozwiązanie HA/DR jest opłacalne. Koszt redundantnego systemu nigdy nie powinien przekraczać kosztu przestoju aplikacji.
- Zawsze testuj strategię DR. Sam fakt, że baza danych jest zsynchronizowana, nie oznacza jeszcze, że faktycznie możemy przeprowadzić przełączenie awaryjne.
- Powszechną praktyką jest coroczne testowanie strategii odtwarzania po awarii (DR) dla każdej aplikacji. Najlepiej realizować to poprzez cykliczny harmonogram testów DR rozłożonych na cały rok.
- Zaangażuj użytkowników w testy DR, aby mogli oni sprawdzić, czy wydajność systemu po przełączeniu awaryjnym jest do przyjęcia.
- Grupy dostępności to doskonała i bardzo elastyczna technologia, ale nie zawsze są one właściwym rozwiązaniem.
- Klastry przełączania awaryjnego obecnie wyszły z mody, ale nie bój się ich używać, jeśli jest to odpowiednia technologia w danym scenariuszu.
- Grupy dostępności mogą nie być odpowiednim rozwiązaniem w przypadku baz danych o bardzo dużej liczbie operacji zapisu, gdy wymagana jest wysoka dostępność.
- Grupy dostępności mogą również nie być odpowiednie dla dużych hurtowni danych, nawet przy zastosowaniu replikacji asynchronicznej.
- Unikaj przeciążania klastra, ponieważ może to prowadzić do problemów podczas przełączania awaryjnego. Możesz temu zapobiec przez planowanie wydajności i testy odzyskiwania po awarii.

14

Bezpieczeństwo

W tym rozdziale:

- Praca z uprawnieniami
- Bezpieczeństwo kont usługowych
- Używanie `xp_cmdshell`
- Audyt uprzywilejowanych działań
- Ataki na mechanizmy szyfrowania
- Ataki typu SQL injection

W tym rozdziale omówimy typowe pomyłki, które wpływają na bezpieczeństwo SQL Servera. Zaczniemy od przyjrzenia się zasadzie najmniejszych przywilejów. Zgodnie z tą zasadą użytkownicy powinni mieć tylko tyle uprawnień, ile potrzebują do codziennej pracy, jednak często nie jest ona przestrzegana. Wyjaśnimy, dlaczego tak się dzieje i dlaczego jest to tak ważne. Przyjrzymy się również kontu `sa`, które jest wbudowanym kontem administratora SQL Servera. Wielu przypadkowych administratorów baz danych pozostawia to konto aktywne. Dowiesz się, dlaczego nie jest to dobry pomysł i jak można temu zaradzić.

Następnie skupimy się na kontach usługowych. Przyjrzymy się typowym błędom popełnianym w definiowaniu strategii dotyczącej kont usługowych. Powiemy również, jak zastosować nowoczesne podejście do wdrażania kont usługowych i jak może to zwiększyć bezpieczeństwo oraz ułatwić zarządzanie środowiskiem.

Przyjrzymy się kontrowersyjnemu narzędziu `xp_cmdshell`, które jest rozszerzoną procedurą składowaną umożliwiającą administratorom (i napastnikom) interakcję z systemem operacyjnym z poziomu instancji SQL Servera. Omówimy

i obalimy mity, które skłaniają administratorów baz danych do niepotrzebnego narażania swoich systemów na ataki.

Następnie zajmiemy się bezpieczeństwem pasywnym, a konkretnie audytem. Przyjrzymy się bardzo powszechnemu błędowi, jakim jest niewdrażanie audytu działań administracyjnych. Zbadamy również, jak skonfigurować audyt w sposób, który zapewni niezaprzeczalność.

Kolejnym tematem, którym się zajmiemy, są ataki polegające na podstawianiu całych wartości. W tego typu atakach osoby o złych zamiarach manipulują zaszyfrowanymi danymi na swoją korzyść. Omówimy, w jaki sposób to robią oraz jak możemy temu zapobiec.

Na koniec zajmiemy się atakami typu SQL injection. Dowiesz się, w jaki sposób cyberprzestępcy wykorzystują wstrzykiwanie kodu SQL do atakowania firm oraz jak możemy się przed tym bronić. Ostatecznie to administratorzy baz danych stanowią ostatnią linię obrony przed tego typu zagrożeniami.

Aby w pełni skorzystać z tego rozdziału, powinieneś znać podstawową terminologię dotyczącą bezpieczeństwa. Tabela 14.1 zawiera słowniczek kluczowych pojęć z zakresu bezpieczeństwa, z którymi należy zapoznać się przed dalszą lekturą.

Tabela 14.1. Terminologia bezpieczeństwa

| Termin | Definicja |
|------------------------|--|
| Podmiot zabezpieczeń | Jednostka, taka jak osoba lub aplikacja, której nadano uprawnienia dostępu do zasobu. |
| Obiekt zabezpieczany | Element, do którego można zastosować uprawnienia. W SQL Serverze są to często obiekty takie jak tabele lub procedury składowane. |
| Login | Podmiot zabezpieczeń na poziomie instancji, umożliwiający osobie dostęp do instancji. |
| Użytkownik bazy danych | Podmiot zabezpieczeń na poziomie bazy danych, który (zazwyczaj) jest powiązany z loginem i umożliwia osobie dostęp do zasobów w bazie danych*. |
| Rola serwerowa | Grupa zawierająca jeden lub więcej loginów, której można nadać uprawnienia do obiektów zabezpieczanych na poziomie instancji. Jeśli rola na poziomie serwera jest „predefiniowana”, oznacza to wbudowaną rolę skonfigurowaną z zestawem uprawnień odpowiadającym typowemu przypadkowi użycia. |
| Rola bazodanowa | Grupa zawierająca jednego lub więcej użytkowników bazy danych, której można nadać uprawnienia do obiektów zabezpieczanych na poziomie bazy danych. Jeśli rola bazy danych jest „predefiniowana”, oznacza to wbudowaną rolę skonfigurowaną z zestawem uprawnień odpowiadającym typowemu przypadkowi użycia. |
| Active Directory (AD) | Usługa katalogowa hostowana w systemie Windows, umożliwiająca centralne zarządzanie uprawnieniami. |

Tabela 14.1. Terminologia bezpieczeństwa – ciąg dalszy

| Termin | Definicja |
|---------------------------|---|
| Domena | Zbiór obiektów AD, takich jak użytkownicy, grupy i komputery. Informacje te są przechowywane w bazie danych synchronizowanej między kontrolerami domeny. Kontrolery domeny to serwery, w których działa usługa kontrolera domeny i które służą do zarządzania AD. |
| Użytkownik AD | Konto użytkownika Windows utworzone i zarządzane w AD. |
| Grupa AD | Grupa utworzona i zarządzana w AD. Zawiera użytkowników AD i może otrzymywać uprawnienia do obiektów zabezpieczanych w całej domenie. |
| Konto usługowe | Konto użytkownika, zazwyczaj w Active Directory, przeznaczone wyłącznie do uruchamiania usług systemu Windows. Konta te są zwykle skonfigurowane tak, aby uniemożliwić logowanie interaktywne, i często ustawione tak, aby ich hasła nie wygasły automatycznie**. |
| Uwierzytelnianie Kerberos | Bezpieczny protokół uwierzytelniania używany przez Active Directory, zapewniający metodę wzajemnego uwierzytelniania poprzez wymianę kluczy. |
| Nazwa główna usługi | Unikatowy identyfikator dla instancji usługi. Jest to funkcja uwierzytelniania Kerberos. |

* Użytkownik ograniczony nie jest powiązany z loginem. W związku z tym dostęp jest możliwy tylko w obrębie jednej bazy danych.

** Jeśli wyłączono automatyczne wygasanie hasła dla konta usługowego, hasło to nadal powinno podlegać rotacji.

14.1. Numer 94 – niestosowanie zasady najmniejszych przywilejów

Fundamentem strategii bezpieczeństwa IT, na wszystkich poziomach — od aplikacji po centrum danych, jest *zasada najmniejszych przywilejów*. Zgodnie z nią każdy podmiot bezpieczeństwa (taki jak osoba lub usługa) powinien mieć dostęp tylko do tych zasobów, które są niezbędne do wykonywania codziennych obowiązków. Jeśli kiedykolwiek potrzebne są dodatkowe uprawnienia do konkretnego zadania, można je przyznać na czas jego trwania, a następnie odebrać.

Mimo że jest to powszechnie znana i logiczna zasada, często nie stosuje się jej w warstwie SQL Servera. Niepokojące jest to, jak często spotyka się instancje SQL Servera, w których wiele osób spoza zespołu administratorów baz danych ma przypisaną rolę sysadmin. Predefiniowana rola sysadmin daje pełny dostęp do instancji SQL Servera — możliwość wykonywania dowolnych operacji bez ograniczeń.

Najczęstszą tego przyczyną jest niezrozumienie uprawnień w SQL Serverze i sposobu ich konfiguracji. Co ciekawe, często nie dotyczy to administratorów baz danych. Zazwyczaj wynika ono z braku specjalistycznej wiedzy o SQL Serverze wśród innych specjalistów IT. Na przykład często zdarza się, że zespół deweloperski twierdzi, iż potrzebuje uprawnień administratora systemu, ponieważ musi tworzyć i usuwać bazy danych na serwerze nieprodukcyjnym.

Jeszcze częstszym zjawiskiem są żądania od firm zewnętrznych, które twierdzą, że ich oprogramowanie wymaga uprawnień administratora systemu we wszystkich instancjach SQL Servera, aby mogło działać prawidłowo. Najczęściej winowajcami są tu firmy dostarczające narzędzia do tworzenia kopii zapasowych, monitorowania, a także, o ironio, narzędzia zabezpieczające.

Ten scenariusz zawsze budzi poważne obawy, ponieważ spełnienie takiego żądania często oznacza przyznanie jednemu kontu uprawnień administratora systemu we wszystkich instancjach SQL Servera w całej organizacji. Jest to oczywista i poważna luka w zabezpieczeniach. Jeśli ktoś przejmie takie konto, może uzyskać dostęp do wszystkich danych organizacji.

Z perspektywy administratora bazy danych pomyłką jest zgadzanie się na takie prośby. Trzeba przyznać, że może być to trudne, gdy dostawca oprogramowania kategorycznie stwierdza, że wymaga uprawnień administratora systemu. Jednak jako ostatnia linia obrony danych organizacji musimy pozostać nieugięci.

Na przykład narzędzie do backupu, które rzekomo wymaga uprawnień administratora systemu, w rzeczywistości ich nie potrzebuje. Prawdopodobnie wystarczy mu jedynie rola serwerowa `##MS_DatabaseManager##` oraz predefiniowana rola bazodanowa `db_backupoperator` w bazach danych, które mają być archiwizowane.

UWAGA Rola `##MS_DatabaseManager##` została wprowadzona w wersji SQL Server 2022. W poprzednich wersjach SQL Servera można zamiast niej użyć roli `db_creator`. Zaletą nowej roli jest jej spójność z bazami danych Azure SQL.

W zdecydowanej większości przypadków istnieje predefiniowana rola serwerowa lub bazodanowa, która zapewnia uprawnienia wymagane przez użytkownika aplikacji bez narażania organizacji na ryzyko związane z przyznawaniem uprawnień `sysadmin` wielu kontom. Pełną listę predefiniowanych ról serwerowych można znaleźć pod adresem <https://mng.bz/oMV6>, natomiast pełną listę predefiniowanych ról bazodanowych można znaleźć pod adresem <https://mng.bz/aVWX>.

Wsparcie operacyjne

Należy pamiętać, że nadawanie rozszerzonych uprawnień osobom spoza zespołu administratorów baz danych niesie ze sobą nie tylko zagrożenia związane z bezpieczeństwem. Jako zespół DBA jesteśmy odpowiedzialni za niezawodne i wydajne działanie naszego środowiska SQL Servera. Przyznanie dodatkowych uprawnień osobom spoza zespołu umożliwia im wprowadzanie zmian, o których możemy nie wiedzieć. Może to prowadzić do problemów z wydajnością i stabilnością systemu, które później musimy sami rozwiązywać.

Jeśli nie ma odpowiedniej roli serwerowej lub bazodanowej, możemy utworzyć własne niestandardowe role z wymaganymi uprawnieniami. Na przykład skrypt z listingu 14.1 tworzy rolę serwera o nazwie AvailabilityGroupsFailover. Następnie dodaje do tej grupy konto SYSTEM, a potem przyznaje mu uprawnienia niezbędne do wykonania przełączenia awaryjnego w replice wtórnej.

Listing 14.1. Tworzenie niestandardowej roli serwerowej

```
CREATE SERVER ROLE AvailabilityGroupsFailover ;  
GO  
  
ALTER SERVER ROLE AvailabilityGroupsFailover ADD MEMBER [NT AUTHORITY\SYSTEM] ;  
  
GRANT ALTER ANY AVAILABILITY GROUP TO AvailabilityGroupsFailover ;  
  
GRANT CONNECT SQL TO AvailabilityGroupsFailover ;  
  
GRANT CREATE AVAILABILITY GROUP TO AvailabilityGroupsFailover ;  
  
GRANT VIEW SERVER STATE TO AvailabilityGroupsFailover ;
```

Skrypt z listingu 14.2 pokazuje, jak utworzyć niestandardową rolę bazodanową, aby rozwiązać problem często napotykanym przez administratorów DBA, a mianowicie brak roli umożliwiającej wykonywanie procedur składowanych bez nadawania zbędnych uprawnień do bazy danych. Jeśli ten skrypt zostanie wykonany na bazie danych model, będzie on automatycznie generowany we wszystkich nowych bazach danych tworzonych w danej instancji serwera.

Listing 14.2. Rola bazodanowa, która umożliwia wykonywanie procedur składowanych

```
USE model ;  
GO  
  
CREATE ROLE ExecSP ;  
GO  
  
GRANT EXECUTE TO ExecSP ;
```

WSKAZÓWKA Baza danych model bywa bardzo przydatna, ale należy uważać, aby nie umieścić w niej zbyt wielu elementów, ponieważ może to utrudnić zarządzanie bazami danych w skali całego przedsiębiorstwa.

SQL Server ułatwia nam zarządzanie uprawnieniami zgodnie z zasadą najmniejszych przywilejów. Niestety inne zespoły i organizacje nie zawsze podchodzą do tego równie skrupulatnie. Ważne jest jednak, abyśmy trzymali się naszych zasad i pomagali użytkownikom precyzyjnie określić, jakich uprawnień faktycznie potrzebują. Dzięki temu możemy pełnić rolę ostatniej linii obrony danych naszej organizacji.

14.2. Numer 95 — niewyłączanie konta sa

SQL Server oferuje dwa tryby uwierzytelniania: uwierzytelnianie Windows oraz tryb mieszany. Jeśli w danej instancji skonfigurowano uwierzytelnianie Windows, wszyscy użytkownicy muszą korzystać z tego mechanizmu, aby uzyskać dostęp do serwera. W takim przypadku podczas instalacji instancji należy dodać co najmniej jednego użytkownika lub grupę systemu Windows do predefiniowanej roli serwerowej `sysadmin`.

Jeśli skonfigurowano mieszany tryb uwierzytelniania, dostęp do instancji serwera można uzyskać zarówno za pomocą uwierzytelniania Windows, jak i *uwierzytelniania drugiego poziomu*. Uwierzytelnianie drugiego poziomu odnosi się do loginów SQL z przypisanymi hasłami. Użytkownicy mogą wtedy łączyć się z instancją, podając nazwę i hasło loginu. Gdy włączony jest mieszany tryb uwierzytelniania, nadal można dodawać użytkowników do roli `sysadmin` na serwerze, ale konto `sa` zostanie aktywowane i konieczne będzie określenie dla niego hasła podczas instalacji instancji.

WSKAZÓWKA W idealnym świecie zawsze korzystalibyśmy z uwierzytelniania Windows, ale nie zawsze jest to możliwe. Wiele aplikacji korzysta z uwierzytelniania SQL, aby uzyskać dostęp do instancji bazy danych.

Bardzo częstą pomyłką przypadkowych administratorów baz danych jest pozostawienie tej konfiguracji bez zmian. Dlaczego jest to błąd? Problem polega na tym, że konto `sa` jest powszechnie znane i posiada pełne uprawnienia administracyjne do instancji. Oznacza to, że gdyby osoba o złych zamiarach chciała zaatakować naszą instancję, jej pierwszym krokiem byłaby prawdopodobnie próba uzyskania dostępu do konta `sa`.

Co gorsza, ze względu na specyfikę loginów SQL konta te są podatne na ataki typu *brute force*. Atak typu *brute force* polega na tym, że napastnik próbuje zalogować się z użyciem programu, używając znanej nazwy użytkownika i słownika haseł. Sprawdza każde hasło po kolei, aż jedno zadziała. Ponieważ konto `sa` jest powszechnie znane, atakujący nie musi się wysilać, aby odkryć nazwę konta z uprawnieniami administratora — już ją zna. Dzięki temu może od razu spróbować siłowo odgadnąć hasło.

Co zatem powinniśmy zrobić inaczej? Mamy dwie możliwości. Po pierwsze, możemy po prostu wyłączyć konto `sa`. W tym przypadku najpierw upewnijmy się, że mamy inne konta z uprawnieniami administracyjnymi (najlepiej użytkowników Windows w grupie AD) w serwerowej roli `sysadmin`. Te konta administracyjne mogą być używane zamiast konta `sa`. Wyłączanie konta `sa` pokazano na listingu 14.3.

Listing 14.3. Wyłączanie konta sa

```
ALTER LOGIN sa DISABLE ;
```

Możemy też zachować konto sa, ale zmienić jego nazwę. Takie podejście pozwoli zachować uprawnienia administracyjne konta, jednocześnie ukrywając jego powszechnie znaną nazwę. Utrudni to potencjalnym napastnikom przeprowadzenie ataku, ale nie uczyni go niemożliwym. Jeśli uda im się odkryć nową nazwę konta, nadal będzie ono podatne na ataki typu brute force. Oznacza to, że choć ta metoda poprawia bezpieczeństwo, nie jest tak skuteczna jak całkowite wyłączenie konta i korzystanie z uwierzytelniania Windows.

OSTRZEŻENIE Każde konto SQL jest narażone na ataki typu brute force. Jeśli to możliwe, powinniśmy unikać nadawania podwyższonych uprawnień kontom SQL.

Polecenie z listingu 14.4 pokazuje, jak zmienić nazwę konta sa.

Listing 14.4. Zmiana nazwy konta sa

```
ALTER LOGIN sa  
  WITH NAME = InstanceAdmin ;
```

Konfiguracja sa

Podczas pracy z loginami SQL, szczególnie z kontem sa, warto zadbać o ich maksymalne zabezpieczenie. SQL Server oferuje przydatną funkcję, która pozwala na wymuszenie stosowania domenowych zasad tworzenia i wygasania haseł dla loginów SQL. Dzięki temu można zapewnić odpowiednią złożoność haseł oraz wymusić ich regularną zmianę. Poniższe polecenie pokazuje, jak włączyć stosowanie domenowych zasad tworzenia i wygasania haseł:

```
ALTER LOGIN InstanceAdmin  
  WITH CHECK_POLICY = ON  
    , CHECK_EXPIRATION = ON ;
```

Ponieważ konto sa jest powszechnie znane i ma pełne uprawnienia administracyjne do instancji, prawdopodobnie będzie ono pierwszym celem ataku osoby próbującej włamać się do systemu. Aby maksymalnie zwiększyć bezpieczeństwo instancji, należy wyłączyć konto sa. Jeśli nie jest to możliwe, należy przynajmniej zmienić jego nazwę.

14.3. Numer 96 — używanie niewłaściwej ziarnistości konta usługowego

Określenie strategii dla kont usługowych od dawna jest przedmiotem wielu dyskusji. Z punktu widzenia bezpieczeństwa idealnym rozwiązaniem byłoby utworzenie osobnego konta usługowego dla każdej usługi w każdej instancji SQL Servera. Wyobraźmy sobie, że mamy trzy serwery SQL, w których działają usługi Database Engine, SQL Server Integration Services oraz SQL Server Analysis Services. W takim przypadku mielibyśmy łącznie dziewięć kont usługowych.

Problem z tym podejściem, które nazywamy *strategią drobnoziarnistą*, polega na tym, że szybko staje się ono prawdziwym utrapieniem w zarządzaniu. Jeśli mamy środowisko SQL Servera składające się z 200 instancji rozproszonych na 150 serwerach, z których każdy uruchamia średnio dwie usługi, nagle musimy zarządzać 400 kontami na 150 serwerach. Chociaż automatyczne wygasanie haseł jest zwykle wyłączone dla kont usługowych, są to często konta o wysokich uprawnieniach, więc i tak musimy regularnie zmieniać wszystkie hasła.

Co gorsza, często wymaga to synchronizacji zmian w zasobach Active Directory lub SQL Servera, a czasem nawet w plikach konfiguracyjnych aplikacji czy kluczach rejestru. Przy takim poziomie ziarnistości hasła zwykle nie są zmieniane, a chociaż drobnoziarnistość utrudnia atakującym poruszanie się między instancjami i funkcjami, to jednak zwiększa narażenie poszczególnych zasobów. Niektóre z nich mogą być kluczowe dla firmy, a jeśli organizacja podlega regulacjom prawnym, może być konieczne wykazanie, że hasła są regularnie zmieniane.

Często prowadzi to do sytuacji, w której organizacje przechodzą na drugi koniec spektrum, gdzie jedno (lub bardzo niewiele) kont usługowych zarządza ogromną liczbą instancji i funkcji. Jest to znane jako *strategia gruboziarnista*. Rozwiązanie to jest jeszcze gorsze niż strategia drobnoziarnista. Jeśli konto usługowe zostanie przejęte, napastnik uzyskuje dostęp do całego środowiska SQL Servera w organizacji. Co więcej, zazwyczaj oznacza to, że hasło nadal nie jest zmieniane, ponieważ niemożliwe jest skoordynowanie zmiany w tak wielu aplikacjach jednocześnie.

Wielu administratorów baz danych nie zdaje sobie sprawy, że Microsoft rozwiązał ten problem już w wersji Windows Server 2012, wprowadzając konta usługowe zarządzane grupowo (ang. *Group Managed Service Accounts*, gMSA). To specjalny rodzaj kont usługowych, które upraszczają zarządzanie nazwami usług i automatycznie zmieniają hasła. Hasła te są bardzo złożone i przechowywane w Active Directory. Co ważne, hasła te nigdy nie są ujawniane administratorom. Konta te są w pełni samozarządzalne i nie mogą być używane interaktywnie.

Połączenie tych cech sprawia, że są one znacznie bezpieczniejsze niż tradycyjne konta usługowe.

Administrator AD tworzy konto gMSA w Active Directory, a następnie na serwerze hostującym SQL Server uruchamiany jest skrypt, który autoryzuje serwer do korzystania z tego konta. Administratorzy baz danych rzadko używają tych kont po prostu dlatego, że nie wiedzą o ich istnieniu. Nie ma powodu, żeby z nich nie korzystać.

Całościowo rzecz ujmując: główną zaletą kont gMSA w strategii zarządzania kontami usługowymi jest to, że pozwalają one na znacznie bardziej „gruboziarniste” podejście bez wystawiania organizacji na zwiększone ryzyko. Każda firma będzie miała własne standardy dotyczące poziomu ziarnistości kont gMSA, ale ja zwykle zaczynam od zgrupowania ich według ekosystemu i działu biznesowego. Nie ma żadnego powodu, by nie stosować kont gMSA w SQL Serverze, więc należy zawsze używać ich zamiast tradycyjnych kont usługowych.

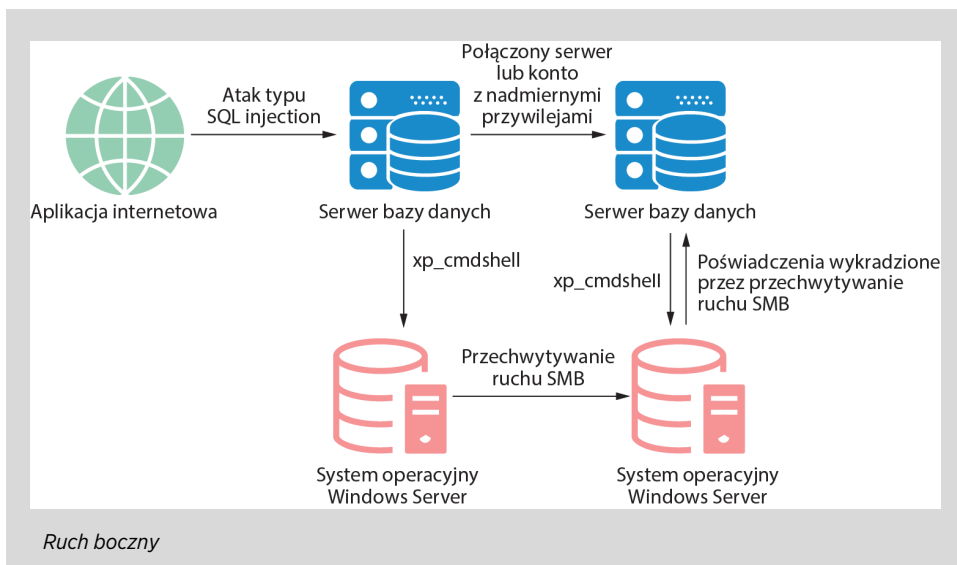
14.4. Numer 97 — włączanie procedury xp_cmdshell

xp_cmdshell to rozszerzona procedura składowana, która umożliwia administratorom wykonywanie poleceń w systemie operacyjnym. Od dawna jest uznawana za niebezpieczną i domyślnie wyłączona. Zagrożenie wynika z tego, że umożliwia ona przejście z instancji SQL Servera do systemu operacyjnego. Co więcej, działa w kontekście konta usługowego SQL Servera, które zwykle ma bardzo wysokie uprawnienia. Jeśli korzystasz z konta usługowego o szerokich uprawnieniach (patrz podrozdział 14.3), procedura może również umożliwić przejście do innych instancji SQL Servera.

Ruch boczny

Ruch boczny to technika wykorzystywana przez cyberprzestępców do wykorzystania luki w zabezpieczeniach jednego serwera w celu uzyskania dostępu do innych serwerów. Jeśli na przykład napastnik uzyskał dostęp do jednej instancji SQL Servera, może uzyskać dostęp do innych instancji poprzez podszywanie się pod uprzywilejowanych użytkowników albo poprzez wykorzystanie powiązanych serwerów.

Napastnik może też wykorzystać procedurę xp_cmdshell, aby uzyskać dostęp do systemu operacyjnego serwera z przejętej instancji SQL Servera. Następnie może zastosować inne techniki ataku, takie jak przechwytywanie ruchu SMB, w celu pozyskania poświadczeń z podwyższonymi uprawnieniami do innych serwerów. Schemat tego procesu przedstawiono na rysunku poniżej.



Niestety, od niedawna zauważalny jest trend, w którym mniej doświadczeni administratorzy baz danych włączają funkcję `xp_cmdshell`. Argumenty przemawiające za takim działaniem są dwa. Po pierwsze, twierdzi się, że tylko członkowie grupy `sysadmin` mają uprawnienia do korzystania z tej funkcji, a administratorom można zaufać, że będą z niej korzystać odpowiedzialnie. Drugi argument mówi, że wyłączenie `xp_cmdshell` nie poprawia bezpieczeństwa, ponieważ członkowie grupy `sysadmin` mogą ją po prostu ponownie włączyć. Oznacza to, że jeśli napastnik przejmie kontrolę nad kontem administracyjnym, może bez problemu aktywować tę funkcję, jeśli będzie chciał jej użyć.

Omówmy teraz szczegółowo każdy z tych punktów. Najpierw zajmijmy się argumentem, że tylko członkowie grupy `sysadmin` mogą korzystać z tej funkcji i że można im zaufać, iż będą jej używać odpowiedzialnie. Po pierwsze, należy zaznaczyć, że to nieprawda, jakoby tylko członkowie grupy `sysadmin` mogli korzystać z tej funkcji. W rzeczywistości wymagania dotyczące uruchamiania procedury składowanej są następujące:

- Członek roli serwerowej `sysadmin`.
- Przyznane uprawnienie `CONTROL SERVER` w instancji SQL Servera.
- Przyznane uprawnienia `EXEC` dla `xp_cmdshell` (wymaga proxy `xp_cmdshell`).
- Przyznane uprawnienie do impersonacji użytkownika spełniającego dowolne z powyższych wymagań.

Uzyskanie dostępu w sposób inny niż przez konto `sysadmin` byłoby nietypową sytuacją, więc dlaczego to ma znaczenie? Cóż, właśnie dlatego, że jest to nietypowe.

A ponieważ jest nietypowe, jest też mało przejrzyste. Zapewne uważnie monitorujemy, kto należy do grupy `sysadmin`, ale czy naprawdę śledzimy, komu bezpośrednio przyznano uprawnienie `CONTROL SERVER` lub komu nadano uprawnienie `IMPERSONATE` względem innego podmiotu zabezpieczeń?

Rozważając to zagrożenie, należy pamiętać, że ponad 70% ataków jest przeprowadzanych wewnętrznie przez osoby działające na szkodę organizacji, które są jej pracownikami lub były nimi w przeszłości. To właśnie te osoby mogą mieć okazję do utworzenia tylnych drzwi przez wykorzystanie nieprzejrzystych uprawnień.

Musimy również się zastanowić, dlaczego w ogóle ktoś miałby używać procedury `xp_cmdshell`. Jeśli administrator bazy danych ma uprawnienia dostępu do systemu operacyjnego, to czemu nie używa go bezpośrednio? Z mojego doświadczenia wynika, że administratorzy baz danych, którzy opowiadają się za stosowaniem procedury `xp_cmdshell`, to zazwyczaj osoby, którym zasady firmy nie zezwalają na dostęp do systemu operacyjnego. W takich przypadkach to sami administratorzy używają `xp_cmdshell` do ruchu bocznego w systemie operacyjnym. Choć nie musi to być złośliwy atak, powinniśmy traktować każdą praktykę naruszającą zasady bezpieczeństwa organizacji jako niewłaściwą, niezależnie od tego, jak niewinne są intencje.

Rozważmy zatem drugi argument, który mówi, że jeśli masz uprawnienia do korzystania z `xp_cmdshell`, to masz również uprawnienia do jego włączenia. Jest to prawda we wszystkich przypadkach, z wyjątkiem sytuacji, gdy skonfigurowano proxy `xp_cmdshell`, a użytkownikowi przyznano uprawnienia `EXECUTE` do tej procedury składowanej.

Na pierwszy rzut oka ten argument wydaje się słuszny. Nawet jeśli zabezpieczymy się przed użyciem `xp_cmdshell` poprzez zastosowanie wyzwalacza DDL lub zasady administracyjnej SQL Servera, administrator i tak będzie miał uprawnienia, by wyłączyć ten wyzwalacz lub zasadę przed wykonaniem procedury.

Jednak patrząc głębiej, powinniśmy rozważyć kwestię warstw zabezpieczeń. Jeśli procedura `xp_cmdshell` jest włączona, można jej łatwo użyć. Jeśli jest wyłączona, atakujący musi ją najpierw aktywować. Jeśli mamy wyzwalacz zapobiegający włączeniu tej funkcji, trzeba go najpierw usunąć, a następnie włączyć procedurę, zanim będzie można wyrządzić szkodę. Pokonanie każdej z tych warstw zabezpieczeń wymaga czasu, co zwiększa prawdopodobieństwo wykrycia i powstrzymania ataku.

Dlatego prawdopodobnie najważniejszą rzeczą, jaką możemy zrobić jako administratorzy baz danych, jest zapewnienie, że system alarmowania jest skonfigurowany tak, aby generował powiadomienie o najwyższym priorytecie w przypadku próby włączenia funkcji `xp_cmdshell`.

Rozważmy skrypt z listingu 14.5. Na początku skrypt tworzy niestandardowy komunikat o błędzie, który ostrzega, że ktoś próbował włączyć `xp_cmdshell`. Następnie tworzy wyzwalacz, który będzie przechwytywać to zdarzenie. Wyzwalacze DDL

są tworzone na poziomie instancji i uruchamiane w reakcji na wykonanie poleceń DDL. Ten wyzwalacz jest ustawiony tak, aby reagować na zdarzenia ALTER_INSTANCE. Gdy wystąpi jakiekolwiek zdarzenie ALTER_INSTANCE, wyzwalacz zostanie uruchomiony. Wyzwalacz wczytuje tekst wykonanego polecenia do zmiennej. Następnie używa funkcji CHARINDEX(), aby sprawdzić, czy instrukcja zawiera ciągi znaków sp_configure i xp_cmdshell. Jeśli tak, to za pomocą instrukcji WITH LOG generowany jest niestandardowy komunikat o błędzie, który utworzyliśmy na początku skryptu. Zapewnia to wysłanie alarmu do dziennika SQL Servera oraz do dziennika zdarzeń aplikacji systemu Windows. Na koniec skrypt wycofuje transakcję i uniemożliwia włączenie xp_cmdshell.

Listing 14.5. Tworzenie wyzwalacza, który zapobiega wykonaniu procedury xp_cmdshell

```
EXEC sp_addmessage 50001, 16, 'Próba wyłączenia xp_cmdshell' ;
GO

CREATE TRIGGER prevent_xp_cmdshell
ON ALL SERVER
FOR ALTER_INSTANCE
AS
BEGIN
    DECLARE @Statement NVARCHAR(4000) ;

    SET @Statement = (SELECT EVENTDATA().value(
        '/EVENT_INSTANCE/TSQLCommand/CommandText)[1]', 'nvarchar(4000)') ;

    IF (CHARINDEX('sp_configure', @Statement) > 0)
        AND (CHARINDEX('xp_cmdshell', @Statement) > 0)
    BEGIN
        RAISERROR(50001, 16, 1, 'Próba wyłączenia xp_cmdshell') WITH LOG ;
        ROLLBACK ;
    END
END ;
GO
```

Możemy teraz poprosić nasz zespół ds. obserwowalności o utworzenie alarmu w przypadku wystąpienia błędu 50001 w dzienniku zdarzeń. Jeśli nasza organizacja nie posiada zaawansowanego narzędzia do monitorowania, możemy osiągnąć podobny efekt z użyciem alarmów SQL Agent. Jednak lepszym rozwiązaniem jest zastosowanie wyspecjalizowanego narzędzia do monitorowania, co pozwoli uniknąć uruchamiania usługi Database Mail.

WSKAZÓWKA Alarmowanie w przypadku wystąpienia błędu omówiono w rozdziale 7.

Szybka reakcja na ten alarm bezpieczeństwa daje szansę na powstrzymanie ataku, zanim wyrządzi on jakiekolwiek szkody. Choć to podejście nie jest idealne i wiąże się z pewnym ryzykiem, jest znacznie lepsze niż pozostawienie włączonej funkcji xp_cmdshell, co pozwoliłoby napastnikom na swobodne przemieszczanie się w sieci i atakowanie innych obszarów infrastruktury. Dodatkową

korzyścią jest zadowolenie zespołu ds. cyberbezpieczeństwa i audytorów, co zaoszczędzi nam czas na pisanie raportów o wyjątkach.

Każdy zespół ds. bezpieczeństwa ma własne procedury i zasady postępowania w przypadku wykrycia aktywnego ataku. Moje osobiste podejście polega jednak na wyłączeniu zaatakowanej instancji na czas prowadzenia dochodzenia. Spowoduje to przerwę w działaniu systemu, ale jest to najlepszy sposób na zabezpieczenie danych firmy.

Niepokojące jest, że procedura `xp_cmdshell` znów wraca do łask przypadkowych administratorów baz danych. Jest to bardzo niebezpieczne narzędzie, które naraża nasze organizacje na ryzyko. Niestety nie ma sposobu na całkowite wyeliminowanie tego zagrożenia, ale powinniśmy minimalizować ryzyko poprzez wprowadzenie warstw zabezpieczeń. Trzeba zadbać, aby jak najmniej osób miało podwyższone uprawnienia w instancji bazy danych, przez stosowanie zasady najmniejszych przywilejów. Trzeba wyłączyć niebezpieczne funkcje, które nie są (lub nie powinny być) używane, takie jak `xp_cmdshell`, oraz wprowadzić niezawodny system alarmowania, aby mieć szansę na zapobieżenie skutecznemu atakowi. Należy wreszcie korzystać z audytu, aby zapewnić niezaprzeczalność działań kont uprzywilejowanych. To doskonale wprowadzenie do naszego następnego tematu, który dotyczy właśnie tej kwestii.

14.5. Numer 98 — nieaudytowanie działań administracyjnych

Jak wspomniano wcześniej w tym rozdziale, choć nie lubimy o tym myśleć, większość cyberataków pochodzi z wewnątrz organizacji. Trzeba pamiętać, że w dużych firmach może być wiele osób posiadających uprawnienia administracyjne do instancji SQL Servera. Byłoby najlepiej, gdyby dostęp ten miał tylko DBA, ale jak już to omówiono, w niektórych środowiskach może występować nieoptymalna konfiguracja zabezpieczeń, w wyniku czego również inne osoby, takie jak programiści aplikacji czy zespoły wsparcia, będą miały uprawnienia administracyjne. Nawet jeśli uda nam się ograniczyć członkostwo w grupie `sysadmin` wyłącznie do administratorów baz danych, w wielu firmach będzie to obejmować kombinację pracowników etatowych, kontraktowych i usług zarządzanych przez zewnętrznych dostawców. Musimy także wziąć pod uwagę, że nawet w przypadku ataku z zewnątrz, konto administracyjne może zostać przejęte i wykorzystane przeciwko nam.

Z tych powodów zapewnienie niezaprzeczalności ma kluczowe znaczenie. Każda osoba posiadająca uprzywilejowany dostęp musi ponosić odpowiedzialność za swoje działania. Ponadto, jeśli konto zostało przejęte i jest wykorzystywane do przeprowadzenia ataku, musimy być w stanie szybko to wykryć

i podjąć odpowiednie kroki (takie jak zablokowanie konta), aby zapobiec kontynuacji ataku.

SQL Server pozwala zapewnić niezaprzeczalność poprzez audytowanie działań administracyjnych za pomocą narzędzia o nazwie SQL Server Audit. Narzędzie to można skonfigurować tak, aby zapisywało działania audytowe w dzienniku zabezpieczeń systemu Windows, co uniemożliwia administratorom SQL Servera ingerencję w te zapisy. Niestety w większości środowisk, które widuję, SQL Server Audit nie jest skonfigurowany. Jest to błąd, który każdy dbający o bezpieczeństwo administrator baz danych powinien niezwłocznie naprawić.

Aby rejestrować działania administracyjne w sposób odporny na manipulacje, należy najpierw spełnić kilka warunków wstępnych. W wielu organizacjach administratorzy baz danych będą musieli poprosić zespół odpowiedzialny za systemy Windows o pomoc w przygotowaniu tych wstępnych ustawień:

- Ustawienie audytu dostępu do obiektów musi być skonfigurowane tak, aby umożliwić rejestrowanie zdarzeń *generowanych przez aplikacje*. Można to osiągnąć za pomocą narzędzia wiersza poleceń `auditpol`.
- Konto usługowe SQL Servera musi mieć przyznane uprawnienie do *generowania audytów bezpieczeństwa*. Można to skonfigurować w przystawce `secpol.msc`.
- Zasady audytu muszą być skonfigurowane tak, aby umożliwić *inspekcję obiektów* zarówno dla operacji zakończonych sukcesem, jak i niepowodzeniem. Można to również wykonać w przystawce `secpol.msc`.

WSKAZÓWKA Po wprowadzeniu tych zmian należy ponownie uruchomić serwer.

Te wstępne kroki umożliwiają SQL Serverowi zapisywanie zdarzeń audytu w dzienniku zabezpieczeń systemu Windows, który nie może być modyfikowany przez członków grupy `sysadmin`. Po ich wykonaniu możemy utworzyć audyt w SQL Serverze. Ten obiekt audytu określa, gdzie chcemy przechowywać wyniki audytu oraz jakie działanie powinno zostać podjęte w przypadku nieudanego zapisu do dziennika audytu. Działanie to może być ustawione na kontynuację, przerwanie operacji lub wyłączenie serwera. Polecenie przedstawione na listingu 14.6 pokazuje, jak utworzyć audyt, który będzie zapisywał zdarzenia w dzienniku zabezpieczeń systemu Windows.

Listing 14.6. Tworzenie obiektu audytu

```
USE master ;
GO

CREATE SERVER AUDIT AdminActivityAudit
TO SECURITY_LOG WITH (
    ON_FAILURE = CONTINUE
) ;
```

Teraz musimy utworzyć specyfikację obiektu audytu. W tym miejscu definiujemy, jakie działania chcemy monitorować. Lista typów działań podlegających audytowi jest obszerna i obejmuje prawie wszystkie operacje, które można wykonać w SQL Serverze. Na poziomie instancji obejmuje to wszystko, od operacji tworzenia i przywracania kopii zapasowych, przez próby logowania, aż po tworzenie lub usuwanie baz danych. Na poziomie bazy danych można rejestrować wszystkie operacje, od rozpoczęcia wykonywania wsadowego po zmianę właściciela schematu. Pełna lista typów działań jest zbyt obszerna, aby ją tu przedstawić, ale można ją znaleźć pod adresem <https://mng.bz/9on1>.

W tabeli 14.2 przedstawiono typy działań, które będziemy monitorować w naszym przykładowym scenariuszu.

Tabela 14.2. Typy akcji audytu

| Typy zdarzeń audytu | Opis |
|---------------------------------------|--|
| AUDIT_CHANGE_GROUP | Zdarzenie występuje podczas tworzenia, usuwania lub modyfikowania audytu. Dodanie tego typu zapewni niezaprzeczalność w przypadku, gdy użytkownik z uprawnieniami administratora próbuje wyłączyć audyt, wykonać niepożądane działanie, a następnie ponownie go włączyć. |
| DBCC_GROUP | Zdarzenie występuje podczas wykonywania polecenia DBCC. |
| SERVER_OBJECT_CHANGE_GROUP | Zdarzenie występuje podczas tworzenia, usuwania lub modyfikowania obiektu na poziomie instancji |
| SERVER_OBJECT_PERMISSION_CHANGE_GROUP | Zdarzenie występuje podczas przyznawania lub odbierania uprawnień do obiektu powiązanego ze schematem. |
| SERVER_OBJECT_OWNERSHIP_CHANGE_GROUP | Zdarzenie występuje podczas zmiany właściciela obiektu na poziomie instancji. |
| SERVER_OPERATION_GROUP | Zdarzenie występuje podczas dokonywania zmian w konfiguracji instancji. |
| SERVER_PERMISSION_CHANGE_GROUP | Zdarzenie występuje podczas przyznawania lub odbierania uprawnień na poziomie instancji. |
| SERVER_PRINCIPAL_CHANGE_GROUP | Zdarzenie występuje podczas tworzenia, usuwania lub modyfikowania kont użytkowników na poziomie instancji. |
| SERVER_PRINCIPAL_IMPERSONATION_GROUP | Zdarzenie występuje podczas impersonacji użytkownika na poziomie instancji. |
| SERVER_ROLE_MEMBER_CHANGE_GROUP | Zdarzenie występuje podczas zmiany członkostwa w roli serwerowej. |
| SERVER_STATE_CHANGE_GROUP | Zdarzenie występuje podczas modyfikowania stanu instancji. |

Aby utworzyć tę specyfikację audytu, możemy użyć polecenia przedstawionego na listingu 14.7. Zwróć uwagę, że klauzula `FOR SERVER AUDIT` łączy specyfikację audytu z naszym obiektem audytu.

Listing 14.7. Tworzenie specyfikacji audytu

```
CREATE SERVER AUDIT SPECIFICATION AdminActivitySpecification
FOR SERVER AUDIT AdminActivityAudit
    ADD (SERVER_ROLE_MEMBER_CHANGE_GROUP),
    ADD (AUDIT_CHANGE_GROUP),
    ADD (DBCC_GROUP),
    ADD (SERVER_OBJECT_PERMISSION_CHANGE_GROUP),
    ADD (SERVER_PERMISSION_CHANGE_GROUP),
    ADD (SERVER_PRINCIPAL_IMPERSONATION_GROUP),
    ADD (SERVER_OBJECT_CHANGE_GROUP),
    ADD (SERVER_PRINCIPAL_CHANGE_GROUP),
    ADD (SERVER_OPERATION_GROUP),
    ADD (SERVER_STATE_CHANGE_GROUP),
    ADD (SERVER_OBJECT_OWNERSHIP_CHANGE_GROUP) ;
```

Specyfikacje audytu baz danych

Oprócz obserwowania aktywności na poziomie instancji za pomocą specyfikacji audytu serwera możemy również monitorować działania na poziomie bazy danych z użyciem specyfikacji audytu bazy danych. Te obiekty są również powiązane z obiektem audytu, ale umożliwiają szczegółowe śledzenie każdej aktywności, włącznie z instrukcjami `SELECT`. Jeśli chcemy monitorować wiele baz danych, musimy utworzyć specyfikację audytu bazy danych dla każdej z nich osobno.

Musimy jednak uważać na wydajność. Choć audyt SQL Servera jest „lekkim” narzędziem, to monitorowanie zbyt wielu niepotrzebnych zdarzeń może mieć negatywny wpływ na wydajność intensywnie używanego serwera. Ogólnie rzecz biorąc, szczegółowy audyt stosuje się głównie w celu zachowania zgodności z przepisami albo w specyficznych scenariuszach.

Ze względu na charakter kont z uprawnieniami administratora systemu należy zawsze prowadzić audyt działań administracyjnych na poziomie instancji, aby zapewnić niezaprzeczalność w przypadku ataku wewnętrznego. Pomaga to również w izolowaniu konta, które zostało przejęte w wyniku ataku zewnętrznego. Możemy monitorować uprzywilejowane działania za pomocą mechanizmu audytu SQL Servera, a także zadbać o to, aby dzienniki nie zostały zmodyfikowane, przez zapisywanie rekordów audytu w dzienniku zabezpieczeń systemu Windows.

14.6. Numer 99 — narażanie firmy na ataki polegające na podstawianiu całych wartości

SQL Server oferuje wiele sposobów szyfrowania danych chroniących przed różnymi scenariuszami ataków. Jednym z nich jest technologia TDE (ang. *Transparent Data Encryption*), która zabezpiecza bazę danych przed próbami jej „kradzieży” i podłączenia jej lub przywrócenia w innej instancji serwera. Innym rozwiązaniem jest mechanizm Always Encrypted, który uniemożliwia odszyfrowywanie i przeglądanie danych nawet członkom serwerowej roli sysadmin. Jest to szczególnie przydatne w środowiskach o wysokim poziomie bezpieczeństwa, gdzie przechowywane są wyjątkowo wrażliwe dane.

W tej części skupimy się na szyfrowaniu na poziomie komórek i częściej pomyłce popełnianej podczas konfigurowania tego typu szyfrowania, które pozwala zabezpieczyć wrażliwe dane, takie jak numery kart kredytowych. Konkretny błąd, który omówimy, polega na pozostawieniu naszej organizacji podatnej na atak polegający na podstawieniu całej zaszyfrowanej wartości. Jest to atak, w którym napastnik zastępuje zaszyfrowaną wartość inną zaszyfrowaną wartością, która według niego przyniesie mu korzyść lub umożliwi przeprowadzenie szkodliwego działania, takiego jak oszustwo z użyciem karty kredytowej.

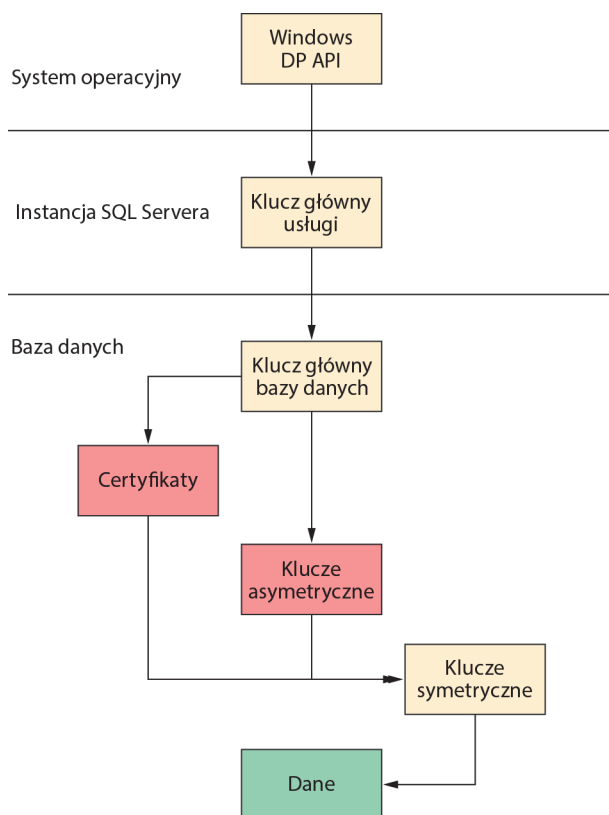
OSTRZEŻENIE Szyfrowanie na poziomie komórek powinno być stosowane z rozważą. Może ono powodować znaczne pogorszenie wydajności, a także zwiększyć objętość danych nawet o 4000%. Zazwyczaj stosuje się je jedynie w celu spełnienia wymogów przepisowych lub w bardzo specyficznych przypadkach użycia.

Aby zrozumieć, na czym polega atak polegający na podstawieniu całej wartości, posłużymy się bazą danych HumanResources, którą utworzyliśmy w rozdziale 3. Baza ta zawiera tabelę o nazwie Employees, w której przechowywane są wszystkie dane pracowników fikcyjnej firmy MagicChoc. Wyobraźmy sobie, że dział kadr uznał kolumnę Salary (wynagrodzenie) w tej tabeli za wysoce wrażliwą i wymagającą szyfrowania.

W kolejnym punkcie przygotujemy zaszyfrowane środowisko, a następnie omówimy, czym jest atak polegający na podstawieniu całych wartości i jak można się przed nim zabezpieczyć.

14.6.1. Przygotowywanie zaszyfrowanego środowiska

Aby spełnić to wymaganie, musimy najpierw stworzyć hierarchię szyfrowania obiektów. Zaczynamy od klucza głównego usługi (ang. *Service Master Key*), który jest szyfrowany przy użyciu Windows Data Protection API i stanowi korzeń szyfrowania w instancji SQL Servera. Klucz ten jest używany do szyfrowania głównego klucza bazy danych, który z kolei jest korzeniem szyfrowania w konkretnej bazie danych. Główny klucz bazy danych jest następnie wykorzystywany do szyfrowania kluczy asymetrycznych i certyfikatów w bazie danych. Na dole hierarchii znajdują się klucze symetryczne. Te klucze symetryczne są szyfrowane z użyciem certyfikatu lub klucza asymetrycznego i służą do szyfrowania danych. Hierarchia ta została zilustrowana na rysunku 14.1.

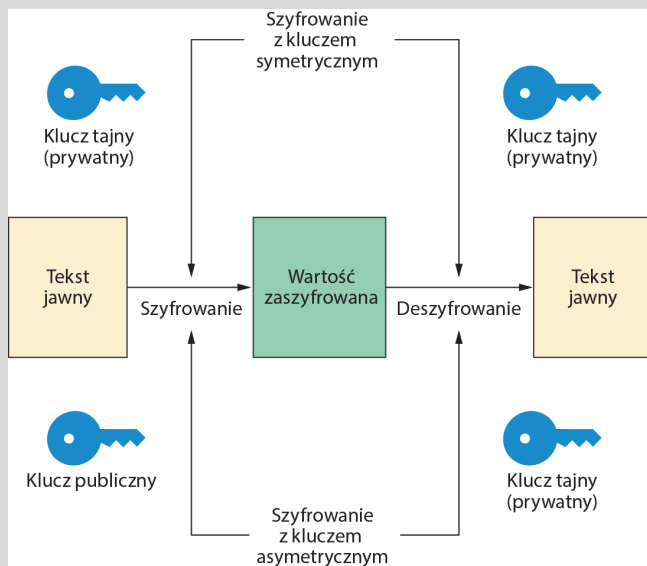


Rysunek 14.1. Hierarchia szyfrowania w SQL Serverze

Klucze symetryczne i asymetryczne

Chociaż pełne wyjaśnienie kryptografii wykracza poza ramy niniejszej książki, warto zrozumieć różnicę między kluczem symetrycznym a asymetrycznym. W szyfrowaniu symetrycznym używa się jednego tajnego klucza zarówno do szyfrowania, jak i deszyfrowania danych. Z kolei w szyfrowaniu asymetrycznym stosuje się dwa różne klucze: prywatny (tajny) do deszyfrowania oraz publiczny do szyfrowania danych.

Ta różnica w podejściu, zilustrowana na rysunku w tej ramce, sprawia, że szyfrowanie asymetryczne jest bezpieczniejsze, ale jednocześnie wymaga dodatkowych zasobów. Dlatego do szyfrowania danych często stosuje się klucz symetryczny.



Klucze symetryczne i asymetryczne

Główny klucz usługi jest tworzony automatycznie przy pierwszym uruchomieniu instancji SQL Servera. Skrypt przedstawiony na listingu 14.8 pokazuje, jak utworzyć klucz główny bazy danych w bazie HumanResources wraz z certyfikatem i kluczem symetrycznym. Certyfikat jest szyfrowany przy użyciu klucza głównego bazy danych. Warto zauważyć, że skrypt wykonuje również kopię zapasową klucza głównego bazy danych, a klucz symetryczny jest szyfrowany z użyciem certyfikatu. Jest to bardzo istotne, ponieważ utrata tego klucza oznaczałaby utratę dostępu do wszystkich danych zaszyfrowanych na niższych poziomach hierarchii.

Listing 14.8. Tworzenie hierarchii szyfrowania

```
USE HumanResources ;
GO
```

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'Pa$$w0rd' ; ← Tworzy klucz główny bazy danych.
```

GO

OPEN MASTER KEY DECRYPTION BY PASSWORD = 'Pa\$\$w0rd';

BACKUP MASTER KEY TO FILE = 'c:\keys\HumanResourcesMasterKey.key' ←

Tworzy kopię zapasową
klucza głównego bazy
danych.

ENCRYPTION BY PASSWORD = 'Ha\$10' ;

GO

CREATE CERTIFICATE SalaryCert ← Tworzy certyfikat.

WITH SUBJECT = 'Wynagrodzenia pracowników' ;

GO

CREATE SYMMETRIC KEY SalaryKey ← Tworzy klucz symetryczny.

WITH ALGORITHM = AES_128

ENCRYPTION BY CERTIFICATE SalaryCert ;

GO

Kolejnym krokiem jest przygotowanie tabeli poprzez dodanie kolumny o nazwie EncryptedSalary, która będzie miała typ danych VARBINARY(8000). Zaszzyfrowane wartości są zawsze przechowywane jako VARBINARY. Ta kolumna będzie służyć do przechowywania zaszzyfrowanych wartości i można ją dodać za pomocą polecenia przedstawionego na listingu 14.9.

Listing 14.9. Dodawanie kolumny EncryptedSalary

ALTER TABLE dbo.Employees

ADD EncryptedSalary VARBINARY(8000) ;

Skrypt z listingu 14.10 wypełni kolumnę EncryptedSalary, wstawiając do niej zaszzyfrowaną wersję wynagrodzenia. W tym celu skrypt najpierw otwiera klucz symetryczny. Następnie wykorzystuje funkcję ENCRYPTBYKEY() do zaszzyfrowania danych. Pierwszym parametrem przekazywanym do tej funkcji jest globalnie unikatowy identyfikator (GUID) klucza symetrycznego. Aby uniknąć konieczności wyszukiwania tego identyfikatora, skrypt wykorzystuje wbudowaną funkcję KEY_GUID(). Przekazujemy do niej nazwę klucza symetrycznego, a funkcja zwraca jego GUID. Drugim parametrem funkcji ENCRYPTBYKEY() są dane, które chcemy zaszzyfrować. W tym przypadku jest to kolumna Salary. Na koniec skrypt zamyka klucz symetryczny, aby zapobiec jego użyciu przez inne kwerendy.

Listing 14.10. Wypełnianie kolumny EncryptedSalary

OPEN SYMMETRIC KEY SalaryKey

DECRYPTION BY CERTIFICATE SalaryCert ;

UPDATE dbo.Employees

SET EncryptedSalary =

ENCRYPTBYKEY(KEY_GUID('SalaryKey'), CAST(Salary AS VARCHAR(10))) ;

CLOSE SYMMETRIC KEY SalaryKey ;

UWAGA W środowisku produkcyjnym należałoby usunąć kolumnę Salary, aby dane nie były już przechowywane w postaci jawnego tekstu.

14.6.2. Zapobieganie atakom polegającym na podstawianiu całych wartości

Aby zrozumieć atak polegający na podstawieniu całej wartości, wyobraź sobie, że firma MagicChoc nieświadomie zatrudniła nieuczciwą osobę. Robin Round jest menedżerem ds. rekrutacji. Pracuje w dziale kadr i posiada uprawnienia SELECT i UPDATE do tabeli Employees. Jeśli Robin wykona kwerendę SELECT na tej tabeli, będzie mógł zobaczyć informacje o pracownikach, takie jak ich imiona i stanowiska. Będzie również mógł odczytać zaszyfrowaną kolumnę z wynagrodzeniami, choć wyniki będą dla niego niezrozumiałymi zaszyfrowanymi wartościami.

Choć Robin nie jest w stanie ustalić, jakie wynagrodzenie otrzymuje Bob Walford (dyrektor generalny), to wie, że jest ono znacznie wyższe niż jego własne. Oznacza to, że gdyby Robin zmienił swoją pensję na taką samą zaszyfrowaną wartość jak pensja Boba Walforda, po prostu ominąłby całe szyfrowanie i przyznał sobie ogromną podwyżkę! Taki atak poprzez podstawienie całej wartości zademonstrowano na listingu 14.11.

Listing 14.11. Atak poprzez podstawienie całej wartości

```
UPDATE dbo.Employees
SET EncryptedSalary =
(
    SELECT EncryptedSalary
    FROM dbo.Employees
    WHERE FirstName = 'Bob'
        AND LastName = 'Walford'
)
WHERE FirstName = 'Robin'
    AND LastName = 'Round' ;
```

Kwerenda z listingu 14.12 wykorzystuje funkcję DECRYPTBYKEY() do odszyfrowania kolumny EncryptedSalary. Wyniki pokazują, że Robin Round i Bob Walford mają teraz takie samo wynagrodzenie.

Listing 14.12. Ocena skutków ataku

```
OPEN SYMMETRIC KEY SalaryKey
DECRYPTION BY CERTIFICATE SalaryCert ;

SELECT
    FirstName
    , LastName
    , Role
    , CAST(CONVERT(VARCHAR(10),DECRYPTBYKEY(EncryptedSalary)) AS MONEY)
FROM dbo.Employees
WHERE (FirstName = 'Bob' AND LastName = 'Walford') OR
    (FirstName = 'Robin' AND LastName = 'Round') ;

CLOSE SYMMETRIC KEY SalaryKey ;
```

Wykonanie tej kwerendy pokaże, że wynagrodzenie obu osób wynosi teraz 96 000 dolarów.

Co zatem powinniśmy zrobić inaczej? Otóż możemy zabezpieczyć się przed tego typu atakiem, dodając tak zwaną *kolumnę uwierzytelniającą*. Polega to na zaszyfrowaniu dodatkowej wartości, często klucza głównego tabeli, wraz z wrażliwymi danymi. Następnie, podczas aktualizowania danych, sprawdza się zgodność tego elementu uwierzytelniającego.

Aby to zademonstrować, użyjmy skryptu z listingu 14.13 do dodania nowej kolumny do tabeli Employees. Kolumna ta nazywa się EncryptedSalaryWithAuth i zostaje wypełniona zaszyfrowanymi danymi z kolumny Salary. Tym razem jednak używamy kolumny EmployeeID jako elementu uwierzytelniającego.

Listing 14.13. Szyfrowanie wynagrodzeń z elementem uwierzytelniającym

```
ALTER TABLE dbo.Employees
    ADD EncryptedSalaryWithAuth VARBINARY(8000) ;
GO

OPEN SYMMETRIC KEY SalaryKey
    DECRYPTION BY CERTIFICATE SalaryCert ;

UPDATE dbo.Employees
    SET EncryptedSalaryWithAuth =
        ENCRYPTBYKEY(
            Key_GUID('SalaryKey'),
            CAST(Salary AS VARCHAR(10)),
            1,
            CAST(EmployeeID AS VARBINARY(8000))
        ) ;

CLOSE SYMMETRIC KEY SalaryKey ;
```

Następnie użyjmy kwerendy z listingu 14.14, aby ponownie zasymulować atak, tym razem na kolumnę EncryptedSalaryWithAuth.

Listing 14.14. Symulacja ataku z wykorzystaniem mechanizmu uwierzytelniania

```
UPDATE dbo.Employees
    SET EncryptedSalaryWithAuth =
    (
        SELECT EncryptedSalaryWithAuth
        FROM dbo.Employees
        WHERE FirstName = 'Bob'
            AND LastName = 'Walford'
    )
    WHERE FirstName = 'Robin'
        AND LastName = 'Round' ;
```

Możemy teraz użyć kwerendy z listingu 14.15, aby ocenić skutki tego ataku.

Listing 14.15. Ocena skutku ataku na kolumnę uwierzytelniającą

```
OPEN SYMMETRIC KEY SalaryKey
DECRYPTION BY CERTIFICATE SalaryCert ;

SELECT
    FirstName
  , LastName
  , Role
  , CAST(
        CONVERT(
            VARCHAR(10),
            DECRYPTBYKEY(
                EncryptedSalaryWithAuth,
                1 ,
                CONVERT(VARBINARY, EmployeeID)
            )
        ) AS MONEY
    )
FROM dbo.Employees
WHERE (FirstName = 'Bob' AND LastName = 'Walford') OR
      (FirstName = 'Robin' AND LastName = 'Round') ;

CLOSE SYMMETRIC KEY SalaryKey ;
```

Wykonanie tej kwerendy pokaże, że pensja Robina jest teraz wyświetlana jako wartość NULL. Choć udało się mu zaktualizować zaszyfrowaną wartość wynagrodzenia, deszyfrowanie już nie działa, ponieważ element uwierzytelniający nie pasuje. Oznacza to, że atak się nie powiódł.

Szyfrowanie może być bardzo przydatnym narzędziem do ochrony poufnych danych. Należy jednak stosować je z umiarem, ponieważ może prowadzić do znacznego rozrostu aplikacji i problemów z wydajnością. Przy szyfrowaniu szczególnie wrażliwych informacji powinniśmy zadbać o to, by nie narażać naszej organizacji na ataki polegające na podstawieniu całej zaszyfrowanej wartości. Takie ataki polegają na tym, że napastnik podmienia jedną zaszyfrowaną wartość na inną. Aby zapobiec tego typu atakom, można dodać kolumnę uwierzytelniającą.

14.7. Numer 100 – narażanie firmy na ataki typu SQL injection

SQL injection (wstrzykiwanie kodu SQL) to atak, w którym napastnik próbuje wprowadzać instrukcje T-SQL w polach aplikacji przeznaczonych na standardowe dane wejściowe użytkownika. W rezultacie aplikacja tworzy poprawne, ale niezamierzone i często szkodliwe kwerendy do bazy danych. Celem tych kwerend jest zazwyczaj wyrządzenie poważnych szkód w środowisku serwera SQL, a często także umożliwienie napastnikowi przedostania się do innych systemów.

Przypadkowi administratorzy baz danych często mylnie uznają, że nie muszą się przejmować takimi atakami. Uważają, że cała odpowiedzialność spoczywa na programistach, którzy powinni zadbać o niezawodne sprawdzanie poprawności wszystkich danych wejściowych w swoich aplikacjach.

Do pewnego stopnia mają rację. Aby jednak zrozumieć, dlaczego jest to błędne podejście, posłużmy się analogią do muzeum, które zatrudnia nocnego stróża. Czy fakt zatrudnienia stróża oznacza, że po zamknięciu muzeum wszystkie drzwi pozostaną otwarte, a alarmy chroniące bezcenne eksponaty zostaną wyłączone? Oczywiście, że nie. Takie postępowanie znacznie zwiększyłoby ryzyko, że złodziej po prostu wślizgnie się do środka, gdy stróż będzie akurat patrolował inne piętro.

Administratorzy baz danych muszą pamiętać, że stanowią ostatnią linię obrony danych organizacji. Oczywiście, programiści aplikacji powinni weryfikować dane wejściowe w swoich programach, ale nie możemy mieć pewności, że faktycznie to robią. Nie możemy też zagwarantować, że programista nie popełnił zwyczajnego błędu, który nie został wykryty podczas testów.

Aby wyjaśnić, dlaczego ochrona przed atakami typu SQL injection jest tak istotna, poświęćmy chwilę na omówienie sposobu przeprowadzania tych ataków i szkód, jakie mogą wyrządzić. W tym celu utwórzmy bardzo prostą, niezabezpieczoną stronę internetową z niezabezpieczoną konfiguracją SQL Servera na zapleczu.

Przykłady w tym podrozdziale zakładają, że zastosowano następujące złe praktyki:

- Konto usługowe należy do grupy Administrators w systemie Windows.
- Konto usługowe jest członkiem serwerowej roli sysadmins.
- Funkcja xp_cmdshell jest włączona.
- Instancja korzysta z uwierzytelniania mieszanego, ale nie wyłącza konta sa ani nie zmienia jego nazwy.
- Loginy SQL nie są skonfigurowane do egzekwowania zasad haseł obowiązujących w domenie.
- Aplikacja internetowa nie weryfikuje danych wejściowych.
- Aplikacja internetowa używa konta sa do łączenia się z bazą danych.
- Aplikacja internetowa przechowuje nazwę użytkownika i hasło bezpośrednio w kodzie, w postaci jawnego tekstu.
- Aplikacja internetowa dynamicznie tworzy kod SQL.

WSKAZÓWKA Dodatkowym problemem związanym z bezpieczeństwem jest to, że hasła użytkowników w tabeli Users są przechowywane w postaci jawnego tekstu. Hasła zazwyczaj powinny być szyfrowane.

Skrypt z listingu 14.16 tworzy bazę danych o nazwie `SQLInjection` zawierającą tabelę `Users`. Tabela ta jest wypełniona dwoma nazwami użytkowników wraz z ich hasłami. Tworzona przez nas witryna zakłada, że baza danych znajduje się w instancji domyślnej. Jeśli chcesz użyć nazwanej instancji, wystarczy odpowiednio zaktualizować ciąg połączenia w pliku `C#`.

Listing 14.16. Tworzenie bazy danych `SQLInjection`

```
CREATE DATABASE SQLInjection ;
GO

USE SQLInjection ;
GO

CREATE TABLE dbo.Users
(
    ID                INT                NOT NULL    IDENTITY    PRIMARY KEY,
    UserName          NVARCHAR(128)     NOT NULL,
    UserPassword      NVARCHAR(512)     NOT NULL
);
GO

INSERT INTO dbo.Users(UserName, UserPassword)
VALUES('Pete', 'Password1'),
      ('Terri', 'MyPassword') ;
GO
```

Skrypt z listingu 14.17 zawiera kod prostej strony logowania oraz podstawowej strony powitalnej. Aby wypróbować ten przykład, należy utworzyć oparty na `C#` projekt `ASPX Web Forms` w `Visual Studio`. Następnie dodaj do projektu dwa formularze internetowe i skopiuj kod dla każdego z nich.

WSKAZÓWKA Jeśli używasz wersji `Visual Studio 2022`, `Microsoft` będzie zachęcać do używania platformy `Razor`. Jednak w tym przykładzie wykorzystamy szablon `ASP.NET Web Application (.NET Framework)`. Aby uzyskać do niego dostęp, upewnij się, że masz zainstalowany komponent *Additional Project Templates (Previous Versions)*. Podczas tworzenia projektu wybierz opcję *Web Forms*.

Listing 14.17. Kod strony logowania i strony powitalnej

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Login.aspx.cs"
Inherits="SQLInjection.Login" %> <----- Kod strony Login.aspx.

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <style type="text/css">
        .auto-style1 {
            width: 503px;
            height: 249px;
            margin-left: 67px;
```

```

    }
</style>
</head>
<body>
    <form id="form1" runat="server">
        <div style="margin-left: 280px">

            <br />
            <br />
            <asp:Login ID="Login1" runat="server" Height="244px" OnAuthenticate="Login1_
            ↳Authenticate" Width="483px" BackColor="#EFF3FB" BorderColor="#B5C7DE"
            ↳BorderPadding="4" BorderStyle="Solid" BorderWidth="1px" Font-Names="Verdana"
            ↳Font-Size="0.8em" ForeColor="#333333" style="margin-left: 64px">
                <InstructionTextStyle Font-Italic="True" ForeColor="Black" />
                <LoginButtonStyle BackColor="White" BorderColor="#507CD1" BorderStyle="Solid"
                ↳BorderWidth="1px" Font-Names="Verdana" FontSize="0.8em" ForeColor="#284E98" />
                <TextBoxStyle Font-Size="0.8em" />
                <TitleTextStyle BackColor="#507CD1" Font-Bold="True" FontSize="0.9em"
                ↳ForeColor="White" />
            </asp:Login>
            <br />
            <br />
            <asp:Label ID="Label1" runat="server"></asp:Label>
            <br />

        </div>
    </form>

</body>
</html>

<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Welcome.aspx.cs"
Inherits="SQLInjection.Welcome" %> ← Kod strony Welcome.aspx.

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <style type="text/css">
        .auto-style1 {
            width: 856px;
            height: 336px;
            margin-left: 235px;
        }
    </style>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <div>
                <br />
                <asp:Label ID="Label2" runat="server" Text="Witaj!"></asp:Label>
            </div>
        </div>
    </form>
</body>
</html>

```

Następnie możemy wykorzystać kod z listingu 14.18 do utworzenia plików C# powiązanych z plikami ASPX. Aby wykonać te kroki, wystarczy skopiować i wkleić podany kod.

UWAGA Kod *Welcome.aspx.cs* jest tylko pustym szablonem. Nie zapewnia żadnej funkcjonalności.

Listing 14.18. Kod plików C# powiązanych z plikami ASPX

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Data;
using System.Data.SqlClient;
using System.Reflection.Emit;

namespace SQLInjection
{
    public partial class Login : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }

        protected void Login1_Authenticate(object sender, AuthenticateEventArgs e)
        {
            SqlConnection con = new SqlConnection(@"Data Source=.;Initial Catalog=
            ↪SQLInjection;Integrated Security=False;Uid=sa;Pwd=Pa$$w0rd");
            string qry = "SELECT * FROM Users WHERE UserName='" + Login1.UserName + "'AND
            ↪UserPassword='" + Login1.Password + "' ";
            SqlDataAdapter adapter = new SqlDataAdapter(qry, con);
            DataTable datatable = new DataTable();
            adapter.Fill(datatable);
            if (datatable.Rows.Count >= 1)
            {
                Label1.Visible = false;
                Session["Parameter"] = datatable.Rows[0].Field<string>(1);

                Response.Redirect("Welcome.aspx");
            }
        }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace SQLInjection
{
```

```

public partial class Welcome : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
}

```

Gdybyśmy korzystali z tej strony zgodnie z jej przeznaczeniem, wprowadzilibyśmy nazwę użytkownika i hasło, a następnie zostalibyśmy przekierowani na stronę powitalną. Jednak gdybyśmy byli napastnikiem chcącym zaatakować organizację, przeprowadzilibyśmy atak typu SQL injection. Aby zrozumieć ten atak, trzeba wiedzieć, że gdybyśmy logowali się jako użytkownik Pete, aplikacja utworzyłaby następującą kwerendę:

```
SELECT * FROM Users WHERE UserName='Pete' AND UserPassword='Haslo1' ;
```

Jeśli jednak zamiast wpisać nazwę użytkownika, atakujący wprowadzi ' OR 1=1-- w polu nazwy użytkownika i jakieś przypadkowe znaki w polu hasła, to kwerenda przyjmie postać:

```
SELECT * FROM Users WHERE UserName='' OR 1=1--' AND UserPassword='przypadkoweznaki' ;
```

Ponieważ wszystko po prawej stronie znaków -- jest traktowane jako komentarz i nie zostaje wykonane, a 1 zawsze jest równe 1, wszyscy użytkownicy z tabeli zostaną przekazani do zbioru danych, a kod C# użyje pierwszego zwróconego wiersza. W rezultacie napastnik skutecznie podszyje się pod tożsamość pierwszego użytkownika w tabeli.

W bardziej zaawansowanym ataku złośliwy użytkownik może wydobyć informacje takie jak nazwa bazy danych, nazwa instancji, a nawet nazwa domeny Active Directory, co daje mu dane niezbędne do przeprowadzenia dalszych ataków. Na przykład, jeśli napastnik użyje ciągu ' or 1 = db_name()--, to wynikowy komunikat o błędzie będzie zawierał następujące informacje:

```
Conversion failed when converting the nvarchar value 'SQLInjection' to data type int
```

Jak widać, ujawniona została nazwa bazy danych — SQLInjection. Gdyby napastnik chciał przeprowadzić atak z ruchem bocznym, to ze względu na naszą niezabezpieczoną konfigurację mógłby po prostu użyć polecenia ' AND 1=1; EXEC xp_cmdshell 'net user hacker \$labeha\$10 /ADD' w polu nazwy użytkownika. W rezultacie w naszym serwerze zostałby utworzony nowy użytkownik o nazwie hacker. Konto to mogłoby następnie posłużyć do przeprowadzenia kolejnych ataków na naszą sieć.

Jak chronić organizację przed takimi atakami? Przede wszystkim należy stosować się do zaleceń przedstawionych w tym rozdziale oraz ogólnych rekomendacji dotyczących bezpieczeństwa. Istnieje jednak dodatkowe rozwiązanie, które możemy wdrożyć — egzekwowanie standardów kodowania.

Jeśli narzucimy wymóg, aby aplikacje internetowe korzystały z procedur składowanych, zamiast dynamicznie budować kwerendy SQL, możemy zapobiec tego rodzaju atakom. Rozważmy na przykład procedurę składowaną przedstawioną na listingu 14.19. Procedura ta przyjmuje zmienne nazwy użytkownika i hasła, które są przekazywane przez aplikację.

Listing 14.19. Tworzenie procedury składowanej do logowania

```
CREATE PROCEDURE dbo.Login
    @UserName NVARCHAR(128)
    , @Password NVARCHAR(128)
AS
BEGIN
    SELECT *
    FROM dbo.Users
    WHERE UserName = @UserName
        AND UserPassword = @Password ;
END
```

Ta prosta procedura wykorzystuje tę samą logikę co dynamicznie konstruowana kwerenda SQL ze strony internetowej. Zobaczmy, co się stanie, gdy uruchomimy skrypt z listingu 14.20. Skrypt ten najpierw symuluje poprawne użycie procedury składowanej, gdzie podawane są prawidłowa nazwa użytkownika i hasło. Następnie symuluje atak złośliwego użytkownika próbującego sfalszować hasło.

Listing 14.20. Symulowanie użycia procedury składowanej do logowania

```
DECLARE @UserName NVARCHAR(128) = 'Pete' ;
DECLARE @Password NVARCHAR(128) = 'Haslo1' ;

EXEC dbo.Login @username, @password ; ← Symuluje próbę logowania zwykłego użytkownika.

SET @UserName = ''' OR 1=1--' ;
SET @Password = 'przypadkoweznaki' ;

EXEC dbo.Login @username, @password ; ← Symuluje próbę logowania złośliwego użytkownika.
```

Jak się przekonasz, pierwsze wywołanie procedury składowanej zwraca oczekiwane wyniki. Natomiast drugie wywołanie tej samej procedury nie zwraca żadnych wyników. W rezultacie atak nie dochodzi do skutku.

WSKAZÓWKĄ Zaletą stosowania procedur składowanych, w przeciwieństwie do ścisłej weryfikacji pól w aplikacji internetowej, jest uniknięcie problemów z obsługą dwuczłonowych nazwisk bez nadmiernego komplikowania walidacji danych wejściowych na stronie.

SQL injection to powszechna forma ataku, w której złośliwy użytkownik próbuje wykorzystać słabe zabezpieczenia systemu do przeprowadzenia szkodliwych działań. Działania te mogą obejmować zarówno podszywanie się pod innego użytkownika, jak i destrukcyjne ataki prowadzące do usunięcia danych

lub całych tabel. Taki atak może nawet posłużyć do przedostania się do innych obszarów naszej sieci. Aby uniknąć tego typu zagrożeń, powinniśmy zawsze stosować najlepsze praktyki w zakresie bezpieczeństwa. Warto również rozważyć egzekwowanie wprowadzenia standardów kodowania, które uniemożliwią przeprowadzanie tego rodzaju ataków.

Podsumowanie

- Zasada najmniejszych przywilejów mówi, że każda osoba powinna posiadać tylko taki poziom uprawnień, jaki jest niezbędny do wykonywania jej codziennych obowiązków.
- Jeśli ktoś musi wykonać zadanie wykraczające poza jego standardowe obowiązki, należy tymczasowo zwiększyć jego uprawnienia, a następnie odebrać je po zakończeniu tego zadania.
- Stawiaj opór dostawcom oprogramowania zewnętrznego, którzy twierdzą, że potrzebują uprawnień administratora systemu.
- Konto sa jest powszechnie znanym kontem o wysokich uprawnieniach. Z tego powodu często staje się celem ataków.
- Jeśli Twoja instancja korzysta z uwierzytelniania mieszanego, wyłącz konto sa lub zmień jego nazwę.
- Zamiast tradycyjnych kont usługowych należy stosować grupowo zarządzane konta usługowe (ang. *Group Managed Service Account*, gMSA).
- Konta gMSA zapewniają wyższy poziom bezpieczeństwa niż tradycyjne konta usługowe, a także upraszczają zarządzanie bezpieczeństwem.
- Procedura `xp_cmdshell` nie powinna być włączona. Najlepszym sposobem zabezpieczenia serwera SQL jest zastosowanie wielu warstw ochrony.
- Procedura `xp_cmdshell` otwiera drogę atakującym do przeprowadzania ataków typu lateral movement i uzyskania dostępu do systemu operacyjnego serwera.
- Skonfiguruj alarm o wysokim priorytecie, który uruchomi się, gdy ktokolwiek spróbuje włączyć `xp_cmdshell`. Daje to wczesne ostrzeżenie, które może zapobiec skutecznemu atakowi.
- Audytuj działania administracyjne w instancjach SQL Servera. Zapewni to niezaprzeczalność i pomoże powstrzymać atak w przypadku przejęcia uprzywilejowanego konta.

- Możesz wykorzystać audyt SQL jako lekki mechanizm do inspekcji działań administracyjnych.
- Szyfrowania na poziomie pojedynczych komórek należy używać oszczędnie, ponieważ może ono powodować znaczny rozrost danych i problemy z wydajnością. Jest to jednak dobre narzędzie do zabezpieczania szczególnie wrażliwych informacji.
- Atak polegający na podstawieniu całej wartości opisuje sytuację, w której atakujący zastępuje zaszyfrowaną wartość inną zaszyfrowaną wartością, która według niego przyniesie mu korzyść.
- Ataki polegające na podstawianiu całych wartości mogą być wykorzystywane do przeprowadzania nieuczciwych działań, takich jak oszustwa związane z kartami kredytowymi.
- Jeśli stosujesz szyfrowanie na poziomie komórek, rozważ użycie kolumny uwierzytelniającej, która zapobiegnie atakom polegającym na podstawieniu całych wartości.
- SQL injection to popularny atak, w którym złośliwy aktor wprowadza szkodliwe polecenia SQL do aplikacji dynamicznie konstruującej kwerendy SQL.
- Ataki typu SQL injection są wykorzystywane do podszywania się pod użytkowników, kradzieży informacji, niszczenia danych oraz do przemieszczania się na boki w celu uzyskania dostępu do innych zasobów.
- Możesz zmniejszyć ryzyko ataków typu SQL injection poprzez utworzenie niezawodnego systemu zabezpieczeń.
- Możesz również chronić się przed atakami typu SQL injection poprzez egzekwowanie standardów kodowania, na przykład wymagając, aby aplikacje uzyskiwały dostęp do danych za pomocą procedur składowanych, zamiast dynamicznie budować kwerendy SQL.

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, 69

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
 nieprzechodność 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

- obiektove, 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
 - wrażenia, 171
 - SSMS, SQL Server Management Studio, 39, 196
 - SSRS, SQL Server Reporting Services, 24, 26
 - standardy kodowania, 52
 - sterta, 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 