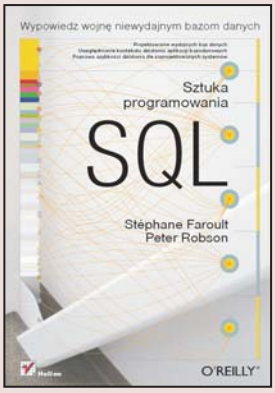
SQL. Sztuka programowania



Autorzy: Stéphane Faroult, Peter Robson
Tłumaczenie: Marek Pętlicki
ISBN: 978-83-246-0895-9
Tytuł oryginału: The Art of SQL
Format: B5, stron: 472

Wypowiedz wojnę niewydajnym bazom danych

- Projektowanie wydajnych baz danych
- Uwzględnianie kontekstu działania aplikacji bazodanowych
- Poprawa szybkości działania źle zaprojektowanych systemów

Twoje bazy danych działają zbyt wolno? Pora to zmienić! Wraz ze wzrostem wielkości korporacyjnych baz danych czas dostępu do nich ma coraz większe znaczenie. Napisanie poprawnie działającego kodu w języku SQL nie jest trudne, jednak tworzenie wydajnych aplikacji bazodanowych jest prawdziwą sztuką. Jak możesz zgłębić jej tajniki i stać się lepszym programistą? Zdaniem autora tej książki nauka wydajnej pracy z bazami danych przypomina poznawanie zasad prowadzenia wojny, dlatego wzorem klasycznej pozycji „Sztuka wojny" autorstwa Sun Tzu prowadzi Cię on przez poszczególne etapy kampanii przeciwko nieefektywnie zaprojektowanym i napisanym aplikacjom bazodanowym.

„SQL. Sztuka programowania" to praktyczny podręcznik, dzięki któremu szybko poszerzysz swą wiedzę w zakresie efektywnego stosowania języka SQL. Nauczysz się dbać o wydajność aplikacji już na etapie ich projektowania, a także myśleć o pracy z bazami danych w kategoriach procesów, wykraczając poza same zapytania języka SQL. Dowiesz się, jak poprawnie używać indeksów oraz jak monitorować szybkość działania bazy. Poznasz standardowe scenariusze zwiększania wydajności, które pozwolą Ci zastosować sprawdzone fortele we własnych projektach oraz w bazach zaprojektowanych przez innych programistów.

- Projektowanie pod kątem wydajności
- Efektywne korzystanie z baz danych w programach
- Poprawne stosowanie indeksów
- Projektowanie optymalnych zapytań SQL
- Praca z dużymi zbiorami danych
- Korzystanie ze struktur drzewiastych
- Monitorowanie wydajności
- Obsługa współbieżności
- Radzenie sobie z niewydajnymi projektami

Poznaj praktyczne techniki poprawy wydajności baz danych

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

SPIS TREŚCI

ROZDZIAŁ DRUGI

Prowadzenie wojny

Wydajne wykorzystanie baz danych

Il existe un petit nombre de principes fondamentaux de la guerre,
dont on ne saurait s'écarter sans danger, et dont l'application au contraire
a été presque en tous temps couronnée par le succès.

Istnieje niewielka liczba fundamentalnych zagadnień związanych z prowadzeniem wojny, których nie wolno lekceważyć: zaprawdę, stosowanie się do nich prawie niezawodnie prowadzi do sukcesu.

— Generał Antoine-Henri de Jomini (1779 – 1869)
Zarys sztuki wojennej

Każdy, kto był zaangażowany w proces przejścia projektu z fazy rozwoju w fazę produkcyjną, przyzna z pewnością, że prawie czuł tumult i wrzawę bitwy. Bardzo często zdarza się, że na kilka tygodni przed sądnym dniem przeprowadzone testy wydajności ujawniają smutny fakt: system nie będzie działał tak płynnie, jak to zakładano. Zaprasza się ekspertów, optymalizuje zapytania SQL, w kryzysowych burzach mózgów biorą udział administratorzy baz danych i systemów. W końcowym rozrachunku na sprzęcie dwukrotnie bardziej kosztownym osiąga się wydajność jedynie teoretycznie zbliżoną do zakładanej.

Często w zastępstwie działań strategicznych stosuje się działania taktyczne. Strategia wymaga zastosowania architektury i modelu przystosowanych do wymagań projektu. Podstawowych zasad stosowanych w czasie wojny jest zaledwie kilka, ale zadziwiająco często bywają one ignorowane. Błędy w architekturze okazują się niezwykle kosztowne, a programista SQL-a musi wkraczać na pole bitwy dobrze przygotowany do walki, musi wiedzieć, gdzie chce dotrzeć i którą drogą. W tym rozdziale przeanalizujemy podstawowe cele zwiększające szanse na sukces w pisaniu programów w sposób efektywny wykorzystujących bazy danych.

Identyfikacja zapytań

Przez stulecia jedynym sposobem, w jaki generał mógł śledzić losy bitwy, była obserwacja oddziałów na podstawie kolorów umundurowania i niesionych przez nich proporców. Gdy jakiś proces w środowisku bazy danych zużywa nadmierną ilość mocy procesora, często istnieje możliwość zidentyfikowania zapytania SQL odpowiedzialnego za to zadanie. Nierzadko jednak trudne bywa odkrycie tego, która część aplikacji wywołała problematyczne zapytanie, szczególnie w środowisku, w którym wykorzystywane są zapytania budowane w sposób dynamiczny. Mimo tego że wiele solidnych produktów wyposażonych jest w mechanizmy monitorujące, często zdumiewająco trudno jest znaleźć odniesienie między zapytaniem SQL a środowiskiem, w którym ono działa. Z tego powodu dobrze jest nabrać nawyku oznaczania programów i krytycznych modułów przez włączanie komentarzy w kodzie SQL w taki sposób, aby łatwo było zidentyfikować źródło kłopotów. Na przykład:

```
/* REJESTRACJA KLIENTA */ select ...
```

Tego typu komentarze identyfikujące mogą być pomocne w śledzeniu błędnie działającego kodu. Mogą być również pomocne przy określaniu obciążenia serwera bazy danych powodowanego przez każdą korzystającą z niego aplikację, szczególnie w przypadku, gdy spodziewamy się, że wprowadzane zmiany mogą spowodować zwiększenie obciążenia, i musimy oszacować, czy posiadany sprzęt ma szansę sprostać większym wymaganiom.

Niektóre produkty posiadają specjalizowane mechanizmy rejestrujące, które pozwalają uniknąć komentowania każdego wyrażenia. Na przykład pakiet `dbms_application_info` serwera Oracle pozwala oznakować program za pomocą 48-znakowej nazwy modułu, 32-znakowej nazwy akcji i 64-znakowego pola informacji o kliencie. Zawartość tych pól jest kontrolowana przez twórcę aplikacji. W środowisku Oracle można użyć tego pakietu do śledzenia tego, jaka aplikacja wykorzystuje bazę danych w danej chwili, jak również tego, co dana aplikacja robi. Do tego służą dynamiczne perspektywy `V$`, za pomocą których można śledzić stan pamięci.

Identyfikowanie wyrażeń ułatwia śledzenie zależności obciążenia.

Trwałe połączenia do bazy danych

Nowe połączenie do bazy danych tworzy się szybko i łatwo, lecz ta prostota czasem przesłania fakt, że szybkie, cykliczne wywołania połączeń wiążą się z konkretnym, niemałym kosztem. Dlatego połączenia z bazą danych warto traktować z rozwagą. Konsekwencje wywoływania wielu cyklicznych połączeń, być może ukrytych w ramach aplikacji, mogą być znaczące, co zademonstruje kolejny przykład.

Jakiś czas temu trafiłem na aplikację, która przetwarzała dużą liczbę niewielkich plików tekstowych o rozmiarach do stu wierszy. Każdy wiersz zawierał dane oraz identyfikator bazy danych, do której dane te miały być załadowane. W tym przypadku był wykorzystywany tylko jeden serwer bazy danych, ale prezentowana zasada obowiązuje również w przypadku setek baz danych.

Przetwarzanie każdego z plików odbywało się zgodnie z następującą procedurą:

```
Otwarcie pliku
Aż do napotkania końca pliku
    Odczytaj wiersz
    Połącz się z serwerem zdefiniowanym w treści
    Wpisz dane
    Zamknij połączenie
Zamknij plik
```

Opisany proces działał zadowalająco z wyjątkiem przypadków, gdy do załadowania trafiała duża liczba niewielkich plików w odstępach czasu przekraczających możliwość ich przetworzenia przez aplikację. To powodowało znaczny przyrost dziennika zaległości w bazie danych (ang. *backlog*), który następnie musiał być przetworzony przez bazę danych, co zajmowało dodatkowy czas.

Użytkownikowi bazy danych wyjaśniłem przyczynę opóźnień, którą była nadmierna liczba otwieranych i zamykanych połączeń. Jako demonstrację problemu przygotowałem prosty program (napisany w C) emulujący aplikację oraz proponowane przeze mnie rozwiązania. Wyniki działania demonstracji przedstawia tabela 2.1.

UWAGA

Program generujący wyniki z tabeli 2.1 wykorzystywał proste instrukcje wstawiające wiersze do tabel. Klientowi wspomniałem również o technikach bezpośredniego ładowania danych do tabel, które działają jeszcze szybciej.

TABELA 2.1. Wyniki testu nawiązywania i kończenia połączeń

Test	Wynik
Nawiązanie i zakończenie połączenia dla każdego wstawianego wiersza z pliku	7,4 wiersza na sekundę
Jedno połączenie, każdy wiersz wstawiany indywidualnie	1681 wierszy na sekundę
Jedno połączenie, wiersze wstawiane w porcjach po 10	5914 wierszy na sekundę
Jedno połączenie, wiersze wstawiane w porcjach po 100	9190 wierszy na sekundę

Ta demonstracja pokazała, jak istotne jest, aby minimalizować liczbę osobnych połączeń z bazą danych. Z tego powodu kluczową rolę odgrywało tu proste sprawdzenie, czy kolejne wprowadzenie danych ma odbyć się do tej samej bazy, do której połączenie już zostało otwarte. Analizę można by

poprowadzić dalej, ponieważ liczba możliwych baz danych była oczywiście ograniczona. Teoretycznie można było osiągnąć dalszy zysk wydajności, wykorzystując tablicę uchwytów połączeń, po jednym dla każdej z możliwych baz danych, i nawiązując połączenie dopiero wtedy, gdy będzie to konieczne. W ten sposób wykorzystywanych byłoby maksymalnie tyle uchwytów, ile istnieje baz danych. Jak widać w tabeli 2.1, prosta technika pojedynczego połączenia (lub minimalizacji liczby nawiązywanych połączeń) usprawnia wydajność do dwustu razy. I to wszystko dzięki niewielkiej modyfikacji kodu.

Oczywiście przy tej okazji mogłem zademonstrować również znaczące korzyści wynikające z ograniczenia liczby wywołań między aplikacją a bazą danych, wykorzystując ładowanie danych za pomocą tablic. Przez wstawianie wielu wierszy w pojedynczej operacji całe zadanie można przyspieszyć jeszcze pięciokrotnie. Wyniki z tabeli 2.1 pokazują przyspieszenie o tysiąc dwieście razy w stosunku do pierwotnej wydajności operacji.

Skąd bierze się tak znaczne przyspieszenie?

Przyczyną pierwszego przyspieszenia jest fakt, że nawiązanie połączenia z bazą danych jest operacją „ciężką", to znaczy wykorzystującą dużo zasobów systemowych.

We wciąż popularnym środowisku klient-serwer nawiązanie połączenia to prosta operacja, co powoduje, że niewiele osób ma świadomość kryjącego się za nią procesu. W pierwszym etapie klient musi nawiązać połączenie z modułem nasłuchującym, wchodzącym w skład serwera, po czym moduł nasłuchujący uruchamia osobny proces lub wątek serwera bazy danych albo przekazuje (bezpośrednio lub pośrednio) żądanie do istniejącego, oczekującego procesu serwera.

Niezależnie od liczby operacji (wywoływania nowych procesów lub wątków i wywoływania obsługi zapytania) system bazy danych musi utworzyć nowe środowisko dla każdej sesji, dzięki czemu będzie ona mogła śledzić realizowane w niej zadania. System obsługi bazy danych musi sprawdzić poprawność hasła dla konta, za pomocą którego zostało nawiązane połączenie i utworzona nowa sesja. System obsługi bazy danych często musi również wykonać kod związany z procedurą zalogowania do bazy danych (zaimplementowany w postaci wyzwalacza). To samo dotyczy również kodu inicjalizacyjnego niezbędnego do działania procedur osadzonych i pakietów. Na tym tle standardowy

protokół nawiązania połączenia między klientem a serwerem ma niewielki wpływ na powstałe opóźnienia. Z powodu tej „zasobożerności" procesu nawiązania połączenia tak wielkie znaczenie z punktu widzenia wydajności mają rozmaite techniki pozwalające na utrzymanie raz nawiązanego połączenia z bazą danych, jak pule połączeń (ang. *connection pooling*).

Druga przyczyna przyspieszenia wiąże się ze zmniejszeniem liczby cykli przesyłania danych między aplikacją a bazą danych (tzw. round-trips), co, jak widać, również ma niebanalny udział w czasochłonności operacji.

Nawet w przypadku, gdy zostało nawiązane tylko jedno połączenie i jest ono wykorzystywane do realizacji wszystkich kolejnych operacji, przełączanie kontekstu między programem a jądrem systemu obsługi bazy danych również ma wpływ na dalsze opóźnienia. Jeśli zatem system obsługi bazy danych umożliwia wykorzystanie jakiegoś mechanizmu ładowania danych większymi porcjami (jak na przykład tablice), warto wziąć pod uwagę jego użycie. W rozwiązaniach, które wykorzystują rzeczywiste tablice, warto sprawdzić, jaki jest domyślny rozmiar tablicy, i dostosować go do potrzeb aplikacji. Każdy rodzaj operacji wykorzystujących przetwarzanie pojedynczymi wierszami wiąże się oczywiście z analogicznymi konsekwencjami, na co będziemy mieli sporo dowodów w tym rozdziale.

Połączenia z bazami danych i przełączenia kontekstu są jak Chińskie Mury — im ich więcej, tym dłużej trwa przekazanie wiadomości.

Strategia przed taktyką

To strategia definiuje taktykę, nie odwrotnie. Dobry programista nie postrzega procesu w kategoriach drobnych kroczków, lecz z perspektywy ostatecznego wyniku. Najefektywniejszy sposób uzyskania wyniku nie musi wynikać z procesów biznesowych, często sprawdza się mniej bezpośrednie podejście. Następny przykład pokaże, w jaki sposób nadmierne skupienie się na procesach proceduralnych może odwrócić uwagę od najbardziej efektywnych rozwiązań.

Kilka lat temu otrzymałem zapytanie z prośbą, abym spróbował je zoptymalizować. „Spróbował" to było słowo kluczowe tego kontraktu. Wcześniej odbyły się dwa podejścia do tej optymalizacji, pierwsze podejmowane przez autorów, drugie przez eksperta Oracle. Mimo tych prób każde wywołanie tego zapytania trwało około dwudziestu minut, co zdaniem autorów było nie do przyjęcia.

Celem tej procedury było wyliczanie ilości materiałów zamawianych przez centralną jednostkę fabryki. Obliczenia wykorzystywały istniejące zapasy i zamówienia pochodzące z różnych źródeł. Dane były odczytywane z kilku identycznych tabel, a następnie agregowane w jednej tabeli zbiorczej. Procedura składała się z sekwencji wyrażeń o następującej filozofii działania: najpierw dane ze wszystkich tabel były kopiowane do tabeli zbiorczej, następnie wywoływane było zapytanie agregujące dane dotyczące materiałów, następnie z tabeli były usuwane nadmiarowe dane niemające znaczenia dla procedury. Ta sekwencja była powtarzana dla każdej tabeli źródłowej. Żadne z zapytań SQL nie było szczególnie skomplikowane, żadnego z nich z osobna nie można też było określić jako szczególnie nieefektywnego.

Większą część dnia zajęło mi zrozumienie procesu, co w końcu zaowocowało postawieniem pytania: „Dlaczego procedurę wykonywano w kilku etapach?". Wystarczyłoby podzapytanie z operatorem unii (`UNION`), za pomocą którego można połączyć wszystkie wyniki z tabel źródłowych. Następnie pojedyncza instrukcja `SELECT` pozwoliłaby za jednym zamachem zapełnić tabelę wynikową. Różnica w wydajności była imponująca: z dwudziestu minut czas wykonania zadania skrócił się do dwudziestu sekund. Skuteczność modyfikacji była tak zdumiewająca, że sporą część czasu zajęła mi weryfikacja, czy aby nowa procedura generuje dokładnie takie same wyniki jak poprzednia.

Nie były tu potrzebne nadzwyczajne umiejętności, wystarczyła umiejętność myślenia poza schematami. Poprzednie próby optymalizacji procesu zakończyły się niepowodzeniem, ponieważ były za bardzo zbliżone do sedna problemu. Często warto zdobyć się na świeże spojrzenie, zrobić krok wstecz, aby poszerzyć perspektywę. Warto było zadać dwa pytania: „Co mamy dostępnego przed wykonaniem procedury?" oraz „Jakie wyniki chcemy uzyskać z procedury?". W połączeniu ze świeżym spojrzeniem odpowiedzi na te pytania doprowadziły do tak wielkiego udoskonalenia procesu.

Spójrz na problem z szerszej perspektywy, zanim zagłębisz się w drobniejsze szczegóły rozwiązania.

Najpierw definicja problemu, potem jego rozwiązanie

Brak wiedzy może być niebezpieczny. Często zdarza się, że ludzie słyszeli lub czytali o nowych czy nietypowych technikach, w niektórych przypadkach nawet dość ciekawych, które natychmiast chcieli wdrożyć w ramach rozwiązywania swoich problemów. Programiści i architekci systemów nader często zachwycają się tego typu „rozwiązaniami", które w gruncie rzeczy jedynie przyczyniają się do powstawania nowych problemów.

Na szczycie listy „gotowych rozwiązań" z reguły można znaleźć denormalizację. Nieświadomi praktycznych zagrożeń związanych z modyfikacją nadmiarowych danych zwolennicy denormalizacji często sugerują ją jako pierwsze podejście przy „optymalizacji" wydajności. Niestety, zdarza się to często na etapie rozwoju aplikacji, gdy jeszcze nie jest za późno na zmiany projektu (lub przynajmniej naukę konstruowania wydajnych złączeń tabel). Szczególnie popularnym panaceum na problemy wydaje się specjalna forma denormalizacji, jaką jest zmaterializowana perspektywa. Zmaterializowane perspektywy czasem określa się nazwą **migawek** (ang. *snapshot*). Jest to mniej spektakularna nazwa, ale za to bardziej zbliżona do nagiej prawdy o tym zjawisku: chodzi bowiem ni mniej, ni więcej o kopie danych wykonania w określonym punkcie czasu. Nie chcę tu sugerować, że nigdy nie zdarza się, że od czasu do czasu, przyparty do muru, nie jestem zmuszony do zastosowania kontrowersyjnych technik. Jak stwierdził Franz Kafka: „Logiki nie da się podważyć, ale nie ma ona szans w obliczu człowieka, który po prostu próbuje przetrwać".

Jednak przytłaczającą większość problemów daje się rozwiązać dzięki inteligentnemu zastosowaniu dość tradycyjnych technik. Warto nauczyć się wyciągać wszystko, co dobre, z takich prostych rozwiązań. Gdy już się je opanuje, człowiek uczy się doceniać ich ograniczenia. Dopiero potem można próbować osądzić potencjalne zalety (o ile istnieją) nowych rozwiązań technicznych.

Wszystkie technologiczne rozwiązania są zaledwie środkiem do osiągnięcia celu. Wielkim zagrożeniem dla niedoświadczonego programisty jest pociąg do najnowszej technologii, który szybko zamienia się w cel dla samego siebie. A zagrożenie jest tym większe w przypadku osób ciekawych, obdarzonych dużym entuzjazmem, o technicznym zacięciu.

Fundamenty są ważniejsze od mody: naucz się podstaw fachu, zanim zaczniesz bawić się nowinkami techniki.

Stabilny schemat bazy danych

Wykorzystanie języka DDL (ang. *data definition language*) do tworzenia, modyfikowania czy usuwania obiektów bazy danych jest bardzo naganną praktyką, która powinna zostać oficjalnie zakazana. Nie ma powodu, aby dynamicznie tworzyć, modyfikować i usuwać obiekty bazy. Z wyjątkiem partycji, o czym wspomnę w rozdziale 5., oraz tabel tymczasowych, z zaznaczeniem, że **mają być zadeklarowane jako tymczasowe w systemie zarządzania bazami danych** (istnieje jeszcze kilka ważnych wyjątków od tej reguły, o czym wspomnę w rozdziale 10.).

Zastosowania języka DDL wykorzystują podstawowy słownik danych bazy. Ten słownik jest również centralnym obiektem wszystkich operacji w bazie danych, a wykorzystanie go prowadzi do wywoływania globalnych blokad, które powodują znaczne konsekwencje w przypadku wydajności bazy. Jedyną dopuszczalną operacją DDL jest przycinanie tabeli (`TRUNCATE`), które jest bardzo szybką metodą usuwania danych z tabeli (ale należy pamiętać, że w przypadku tej operacji nie mamy możliwości jej wycofania za pomocą instrukcji `ROLLBACK`!).

Tworzenie, modyfikacja i usuwanie obiektów baz danych to zadania etapu projektowego, a nie codzienne operacje wykonywane przez aplikację kliencką.

Operacje na rzeczywistych danych

Wielu programistów chętnie posługuje się tymczasowymi tablicami roboczymi, do których na przykład ładują dane do dalszego przetwarzania. Tego typu podejście z reguły uznaje się za niewłaściwe, ponieważ może ono prowadzić do zamknięcia percepcji w zakresie procesów biznesowych i uniemożliwić szersze spojrzenie. Należy pamiętać, że tabele tymczasowe nie dają możliwości zastosowania pewnych zaawansowanych technik dostępnych w przypadku rzeczywistych tabel (część z tego typu opcji omówię w rozdziale 5.). Indeksowanie tabel tymczasowych (o ile w ogóle jest dostępne) może na przykład być mniej optymalne. W wyniku tych ograniczeń zapytania wykorzystujące tabele tymczasowe mogą działać mniej wydajnie od prawidłowo napisanych zapytań na rzeczywistych tabelach. Wadą zastosowania tabel tymczasowych jest ponadto konieczność wykonania dodatkowego zapytania wypełniającego tabelę tymczasową danymi.

Nawet w przypadku, gdy zastosowanie tabel tymczasowych jest uzasadnione, nie należy nigdy wykorzystywać w tym charakterze rzeczywistych tabel udających tabele tymczasowe, szczególnie gdy liczba zapisywanych w nich wierszy jest duża. Jeden z problemów leży tu w mechanizmie gromadzenia statystyk: jeśli statystyki dotyczące tabel nie są gromadzone w czasie rzeczywistym, to system zarządzania bazą danych wykorzystuje do tego chwile mniejszego obciążenia. Natura tabel roboczych polega na tym, że z reguły w takich przestojach bywają puste, co powoduje, że optymalizator uzyskuje zupełnie błędne informacje. W efekcie system podejmuje błędne decyzje w wyniku nieodpowiednich planów wykonawczych przygotowywanych przez optymalizator zapytań, co prowadzi bezpośrednio do obniżonej wydajności. Jeśli ktoś jest naprawdę zmuszony do użycia tabel tymczasowych, powinien używać do tego celu tabel, które system zarządzania bazami danych jest w stanie zidentyfikować jako tymczasowe.

Wykorzystanie tabel tymczasowych oznacza przepychanie danych do mniej optymalnego zasobu.

Przetwarzanie zbiorów w SQL-u

SQL przetwarza dane w postaci zbiorów. W przypadku operacji modyfikacji (`UPDATE`) lub usuwania danych (pod warunkiem, że te modyfikacje nie są dokonywane na całych tabelach) użytkownik musi zdefiniować zbiór wierszy, których dana modyfikacja dotyczy. W ten sposób definiuje się pewną **granularność** procesu, który można określić jako **zgrubną**, jeśli zmiany dotyczą większej liczby wierszy, lub **drobną**, gdy przetwarzamy jedynie kilka wierszy.

Każda próba modyfikacji dużej liczby wierszy tabeli, ale małymi porcjami jest z reguły złym pomysłem i zwykle okazuje się bardzo niewydajna. Takie podejście jest dopuszczalne jedynie w przypadku, gdy na bazie danych będą dokonywane bardzo rozległe zmiany, które na czas transakcji mogą spowodować tymczasowe zajęcie bardzo dużej ilości zasobów oraz zajmują bardzo dużo czasu w przypadku wycofania transakcji (`ROLLBACK`). Istnieją również opinie, że w przypadku modyfikacji dużych porcji danych należy w ramach kodu DML (ang. *data manipulation code*) umieszczać co jakiś czas instrukcje zatwierdzające zmiany (`COMMIT`). Tego typu podejście może jednak nie sprawdzić się w przypadku, gdy dane są ładowane z pliku i procedura zostanie przerwana w trakcie. Z czysto praktycznego punktu widzenia często o wiele prościej i szybciej jest wznowić proces od początku, niż próbować zlokalizować miejsce w danych wejściowych, do którego zostały już załadowane, i od niego wznawiać proces.

Natomiast biorąc pod uwagę rozmiar loga transakcji (ang. *transaction log*) wykorzystywanego do wycofania zmian transakcji, również istnieją opinie, że fizyczna implementacja bazy danych powinna być przygotowana do przyjmowania zmian wykonywanych przez aplikację, a aplikacja nie powinna być zmuszona do omijania ograniczeń fizycznej implementacji. Jeśli wymagana ilość zasobów niezbędnych do zapisania loga transakcji jest rzeczywiście bardzo duża, zapewne należy zastanowić się, czy częstotliwość dokonywania tego typu zmian jest odpowiednia do konstrukcji bazy. Może się bowiem okazać, że zmiana strategii z miesięcznych, gigantycznych aktualizacji danych na kilkakrotnie mniejsze, tygodniowe, albo zupełnie niewielkie, dzienne modyfikacje spowoduje, że problem zupełnie przestanie istnieć.

Pracowite zapytania SQL

SQL nie jest językiem proceduralnym. Choć w zapytaniach SQL istnieje możliwość zastosowania logiki języków proceduralnych, należy traktować je z rozwagą. Problemy z rozróżnieniem logiki proceduralnej od przetwarzania deklaratywnego najlepiej widać w przypadkach, gdy ktoś próbuje wydobyć dane z bazy, dokonać na nich modyfikacji, po czym ponownie zapisać w bazie. Gdy program lub procedura działająca w ramach programu otrzyma dane wejściowe, często zdarza się, że są one wykorzystywane do odczytu innych danych z bazy, po czym następuje pętla lub inna forma logiki funkcyjnej (z reguły pętla *if…then…else*) zastosowana do wydobywania kolejnych danych z bazy. W większości przypadków jest to efekt głęboko zakorzenionych nawyków lub słabej znajomości SQL-a w połączeniu z niewolniczym oddaniem w stosunku do specyfikacji funkcjonalnej. Wiele stosunkowo skomplikowanych operacji można wykonać za pomocą pojedynczego zapytania w SQL-u. Jeśli użytkownik poda wartość, często można podać interesujący go wynik bez konieczności rozbijania logiki na poszczególne instrukcje o niewielkim związku z wynikiem końcowym.

Istnieją dwa główne powody, aby unikać stosowania logiki proceduralnej w SQL-u:

Każdy dostęp do bazy danych wiąże się z operacjami na wielu różnych warstwach programowych, włączając w to operacje sieciowe.

Nawet w przypadku, gdy sieć nie jest wykorzystywana, wchodzą w grę operacje wymiany danych między procesami. Więcej operacji dostępu oznacza więcej wywołań funkcji, więcej przepustowości, a co się z tym wiąże, dłuższe oczekiwanie na odpowiedź. Gdy tego typu wywołania są często powtarzane, opóźnienia stają się wyraźnie zauważalne.

„Proceduralny" znaczy, że wydajność i utrzymanie zależą od programu, nie od bazy danych.

Większość systemów baz danych do wykonywania operacji, jak na przykład złączenia, wykorzystuje zaawansowane algorytmy przekształcające zapytania w ich inne formy tak, aby wykonywały się w wydajniejszy sposób. Optymalizatory oparte na koszcie (ang. *cost-based optimizers*, CBO) to skomplikowane moduły, które pokonały długą drogę. Na początku swojego istnienia mechanizmy tego typu były praktycznie

bezużyteczne, ale z czasem nabrały dojrzałości i obecnie dają doskonałe wyniki. Dobry mechanizm CBO bywa bardzo skuteczny w znajdowaniu najbardziej odpowiedniego planu wykonania. Jednakże CBO analizuje każde zapytanie SQL, nic ponad to. Wrzucając jak największą liczbę operacji w pojedyncze wyrażenie, przerzucamy na bazę danych odpowiedzialność za znalezienie najbardziej optymalnego wykonania. Dzięki temu program może wykorzystać również przyszłe usprawnienia w działaniu silnika systemu zarządzania bazą danych. W ten sposób również przyszłe usprawnienia aplikacji są częściowo przerzucane na dostawcę bazy danych.

Jak to zwykle bywa, również od zasady unikania logiki proceduralnej istnieją dobrze uzasadnione wyjątki. Dotyczy to sytuacji, gdy tego typu podejście pozwala na znaczące przyspieszenie w uzyskiwaniu wyników. Rozbudowane, monstrualne zapytania SQL nie zawsze stanowią wzorzec wydajności. Jednak należy pamiętać, że proceduralny kod sklejający kolejno wykonywane zapytania SQL operujące na tych samych danych (tabelach i wierszach) to dobry materiał na pojedyncze zapytanie SQL. Mechanizm CBO analizuje pojedyncze zapytanie skonstruowane w zgodzie z regułami modelu relacyjnego jako jedną całość i jest w stanie opracować efektywny sposób jego wykonania.

Jak najwięcej pracy zrzucaj na optymalizator zapytań, aby skorzystać z jego możliwości.

Maksymalne wykorzystanie dostępu do bazy danych

Gdy planujemy wizytę w wielu sklepach, w pierwszym kroku musimy zdecydować się na to, co chcemy kupić w każdym z nich. Dzięki temu można zaplanować trasę podróży i zminimalizować konieczność przechodzenia między sklepami tam i z powrotem. Odwiedzamy pierwszy sklep, dokonujemy zakupów, po czym odwiedzamy kolejny sklep. To zdrowy rozsądek, a jednak zasada obowiązująca w tym przykładzie wydaje się obca wielu praktycznym zastosowaniom baz danych.

Gdy z pojedynczej tabeli chcemy odczytać kilka różnych informacji, nawet niezbyt powiązanych (co rzeczywiście wydaje się dość powszechnym przypadkiem), nieoptymalne jest nawiązywanie wielu połączeń, po jednym dla odczytania każdej porcji danych. Na przykład nie należy odczytywać pojedynczych kolumn, jeśli potrzebujemy wartości z kilku z nich. Należy do tego wykorzystać pojedynczą operację. Niestety, dobre praktyki programowania zorientowanego obiektowo z zasady pojedynczego odczytu wartości atrybutów zrobiły zaletę, nie wadę. Nie wolno jednak mylić metod obiektowych z przetwarzaniem danych w bazach. Mieszanie tych pojęć należy do najpoważniejszych błędów, tabel nie wolno ślepo traktować jako klas, a kolumn jako atrybutów.

Każdą wizytę w bazie danych wykorzystuj do wykonania jak największej ilości pracy.

Zbliżenie do jądra systemu DBMS

Im bliżej jądra systemu zarządzania bazą danych może działać kod, tym będzie działał wydajniej. To właśnie sedno siły baz danych. Na przykład niektóre systemy zarządzania bazami danych pozwalają na rozszerzanie swoich możliwości za pomocą nowych funkcji, które można pisać w niskopoziomowych językach programowania, jak na przykład C. Języki niskiego poziomu operujące na wskaźnikach mają istotną cechę negatywną: jeśli w procedurze zostanie popełniony błąd, można doprowadzić do uszkodzenia danych w pamięci. Problem byłby poważny już w przypadku, gdyby z programu korzystał tylko jeden użytkownik. Kłopot z bazami danych polega jednak na tym, że mogą obsługiwać dużą liczbę użytkowników. W przypadku uszkodzenia danych w pamięci, można uszkodzić dane innego, zupełnie „niewinnego" programu. W praktyce bywa tak, że solidne systemy zarządzania bazami danych wykonują kod w pewnej izolacji (technika określana mianem piaskownicy, ang. *sandbox*), dzięki czemu w przypadku awarii procesu nie pociąga on za sobą całej bazy danych. Przykładem jest tu system Oracle, który do wymiany danych między procesami a bazą danych wykorzystuje specjalny mechanizm komunikacji. Ten proces jest podobny do łączy między bazami danych pracujących na różnych serwerach. Jeśli zysk uzyskany z wydajności zewnętrznych funkcji

w C w stosunku do osadzonych procedur PL/SQL rekompensuje koszt skonstruowania zewnętrznego środowiska i przełączeń kontekstu, warto wykorzystać funkcje zewnętrzne. Jednak nie warto ich stosować w przypadku, gdy mają być wykorzystywane do wywoływania pojedynczo dla każdego wiersza tabeli. Jak to zwykle bywa, decyzja jest kwestią równowagi i znajomości wszystkich konsekwencji stosowania różnych strategii rozwiązywania tego samego problemu.

Jeśli funkcje mają być jednak wykorzystane, warto w pierwszej kolejności rozważyć wykorzystanie funkcji standardowych, dostępnych w mechanizmie zarządzania bazą danych. Nie chodzi tu wyłącznie o kwestię unikania „ponownego wynajdowania koła", lecz przede wszystkim o to, że funkcje wbudowane działają znacznie bliżej jądra systemu zarządzania bazą danych niż zewnętrzny kod, a w związku z tym są bardziej wydajne.

Oto prosty przykład wykorzystania kodu SQL w bazie Oracle, za pomocą którego zademonstruję wydajność uzyskaną dzięki zastosowaniu funkcji wbudowanych. Załóżmy, że mamy dane tekstowe wprowadzone ręcznie przez użytkownika i że te dane zawierają zbędne ciągi znaku spacji. Potrzebna jest nam funkcja, która zastąpi takie ciągi wielu znaków spacji pojedynczym znakiem. Przyjmijmy, że nie będziemy korzystać z obsługi wyrażeń regularnych, dostępnej w Oracle Database 10g, a zamiast tego napiszemy własną funkcję:

```
create or replace function squeeze1(p_string in varchar2)
return varchar2
is
  v_string varchar2(512)  := '';
  c_char   char(1);
  n_len    number := length(p_string);
  i        binary_integer := 1;
  j        binary_integer;
begin
  while (i <= n_len)
  loop
    c_char := substr(p_string, i, 1);
    v_string := v_string || c_char;
    if (c_char = ' ')
    then
      j := i + 1;
      while (substr(p_string || 'X', j, 1) = ' ')
      loop
        j := j + 1;
```

```
      end loop;
      i := j;
    else
      i := i + 1;
    end if;
    end loop;
  return v_string;
end;
/
```

Uwaga na marginesie: na końcu ciągu znaków dopisywany jest znak `X`, aby uniknąć porównania wartości `j` z długością tego ciągu.

Istnieją różne metody eliminacji ciągów spacji, w których można wykorzystać funkcje udostępniane w ramach bazy Oracle. Oto jedna z alternatyw:

```
create or replace function squeeze2(p_string in varchar2)
return varchar2
is
  v_string varchar2(512) := p_string;
  i        binary_integer := 1;
begin
  i := instr(v_string, ' ');
  while (i > 0)
  loop
    v_string := substr(v_string, 1, i)
                || ltrim(substr(v_string, i + 1));
    i := instr(v_string, ' ');
  end loop;
  return v_string;
end;
/
```

Trzecia z metod może być następująca:

```
create or replace function squeeze3(p_string in varchar2)
return varchar2
is
  v_string varchar2(512) := p_string;
  len1     number;
  len2     number;
begin
  len1 := length(p_string);
  v_string := replace(p_string, '  ', ' ');
  len2 := length(v_string);
  while (len2 < len1)
  loop
    len1 := len2;
```

```
    v_string := replace(v_string, '  ',' ');
    len2 := length(v_string);
  end loop;
  return v_string;
end;
/
```

Gdy te trzy alternatywne metody zostaną przetestowane na prostym przykładzie, każda, zgodnie z oczekiwaniem, da dokładnie takie same wyniki i nie będzie znaczącej różnicy w wydajności:

```
SQL> select squeeze1('azeryt   hgfrdt     r')
  2  from dual
  3  /
azeryt hgfrdt r

Elapsed: 00:00:00.00
SQL> select squeeze2('azeryt   hgfrdt     r')
  2  from dual
  3  /
azeryt hgfrdt r

Elapsed: 00:00:00.01
SQL> select squeeze3('azeryt   hgfrdt     r')
  2  from dual
  3  /
azeryt hgfrdt r

Elapsed: 00:00:00.00
```

Załóżmy jednak, że operacja oczyszczania ciągów znaków z wielokrotnych spacji będzie wywoływana tysiące razy dziennie. Poniższy kod można zastosować do załadowania bazy danych danymi testowymi, za pomocą których można w nieco bardziej realistycznych warunkach przetestować wydajność trzech przedstawionych wyżej funkcji oczyszczających ciągi znaków z wielokrotnych spacji:

```
create table squeezable(random_text varchar2(50))
/

declare
  i        binary_integer;
  j        binary_integer;
  k        binary_integer;
  v_string varchar2(50);
```

```
begin
  for i in 1 .. 10000
  loop
    j := dbms_random.value(1, 100);
    v_string := dbms_random.string('U', 50);
    while (j < length(v_string))
    loop
      k := dbms_random.value(1, 3);
      v_string := substr(substr(v_string, 1, j) || rpad(' ', k)
                  || substr(v_string, j + 1), 1, 50);
      j := dbms_random.value(i, 100);
    end loop;
    insert into squeezable values(v_string);
  end loop;
  commit;
end;
/
```

Ten skrypt tworzy tabelę testową złożoną z dziesięciu tysięcy wierszy (to dość niewiele, biorąc pod uwagę średnią liczbę wywołań zapytań SQL). Test wywołuje się następująco:

```
select squeeze_func(random_text)
from squeezable;
```

Gdy wywoływałem ten test, wyłączyłem wyświetlanie wyników na ekranie. Dzięki temu upewniłem się, że czasy działania algorytmów oczyszczających wielokrotne spacje nie są zafałszowane przez czas niezbędny na wyświetlenie wyników. Testy były wywołane wielokrotnie, aby upewnić się, że nie wystąpił efekt przyspieszenia wykonania dzięki zbuforowaniu danych.

Czasy działania tych algorytmów prezentuje tabela 2.2.

TABELA 2.2. Czas wykonania funkcji oczyszczających ciągi spacji na danych testowych (10 000 wierszy)

Funkcja	Mechanizm	Czas wykonania
`squeeze1`	PL/SQL pętla po elementach ciągu znaków	0,86 sekund
`squeeze2`	`instr() + ltrim()`	0,48 sekund
`squeeze3`	`replace()` wywołana w pętli	0,39 sekund

Choć wszystkie z tych funkcji wykonują się dziesięć tysięcy razy w czasie poniżej jednej sekundy, `squeeze2()` jest 1,8 razy szybsza od `squeeze1()`, a `squeeze3()` jest ponad 2,2 razy szybsza. Jak to się dzieje? Po prostu

PL/SQL nie jest tak „blisko jądra", jak funkcja SQL-a. Różnica wydajności może wyglądać na niewielką w przypadku sporadycznego wywoływania funkcji, lecz w programie wsadowym lub na obciążonym serwerze OLTP różnica może już być poważna.

Kod uwielbia jądro SQL-a — im jest bliżej, tym jest gorętszy.

Robić tylko to, co niezbędne

Programiści często wykorzystują instrukcję `count(*)` do implementacji testu istnienia. Z reguły dochodzi do tego w celu implementacji następującej procedury:

```
Jeśli istnieją wiersze spełniające warunek
Wykonaj na nich działanie
```

Powyższy schemat jest implementowany za pomocą następującego kodu:

```
select count(*)
into counter
from tabela
where <warunek>
if (counter > 0) then
```

Oczywiście w 90% tego typu wywołania instrukcji `count(*)` są zupełnie zbędne, dotyczy to również powyższego przykładu. Jeśli jakieś działanie musi być wykonane na podzbiorze wierszy tabeli, dlaczego go po prostu nie wykonać od razu? Jeśli nie zostanie zmodyfikowany ani jeden wiersz, jaki w tym problem? Nikomu nie stanie się żadna krzywda. Ponadto w sytuacji, gdy operacja wykonywana w bazie danych składa się z wielu zapytań, po wywołaniu pierwszego z nich liczbę zmodyfikowanych wierszy można odczytać ze zmiennej systemowej (`@@ROWCOUNT` w Transact-SQL, `SQL%ROWCOUNT` w PL/SQL itp.), specjalnego pola SQL Communication Area (SQLCA) w przypadku wykorzystania osadzonego SQL (embedded SQL) lub za pośrednictwem specjalizowanych API, jak na przykład funkcji `mysql_affected_rows()` języka PHP. Liczba przetworzonych wierszy jest czasem zwracana z funkcji, która wykonuje operację w bazie danych, jak metoda `executellpdate()` biblioteki JDBC. Zliczanie wierszy bardzo często

nie służy niczemu oprócz zwiększenia ilości pracy, jaką musi wykonać baza, ponieważ wiąże się ono z dwukrotnym przetworzeniem (a raczej: odczytaniem, a potem przetworzeniem) tych samych danych.

Ponadto nie należy zapominać, że jeśli naszym celem jest aktualizacja lub wstawienie wierszy (częsty przypadek: wiersze są zliczane po to, by stwierdzić istnienie klucza), niektóre systemy zarządzania bazami danych oferują specjalne instrukcje (jak MERGE w Oracle 9i Database), które działają bardziej wydajnie niż w przypadku zastosowania osobnych zapytań zliczających.

Nie ma potrzeby, aby jawnie kodować to, co baza danych wykonuje w sposób niejawny.

Instrukcje SQL-a odwzorowują logikę biznesową

Większość systemów baz danych udostępnia mechanizmy monitorujące, za pomocą których można sprawdzać stan wykonywanych aktualnie instrukcji, a nawet śledzić liczbę ich wywołań. Przy okazji można uświadomić sobie liczbę przetwarzanych jednocześnie „jednostek biznesowych": zamówień lub innych zgłoszeń, klientów z wystawionymi fakturami lub dowolnych innych zdarzeń istotnych z punktu widzenia biznesowego. Można zweryfikować, czy istnieje sensowne (a nawet absolutnie precyzyjne) przełożenie między dwoma klasami aktywności. Innymi słowy: czy dla zadanej liczby klientów liczba odwołań do bazy danych za każdym razem jest taka sama? Jeśli zapytanie do tabeli klientów jest wywoływane dwadzieścia razy częściej, niż wskazywałaby na to liczba przetwarzanych klientów, to z pewnością wskazuje na jakiś błąd. Taka sytuacja sugerowałaby, że zamiast jednorazowego odczytu danych z tabeli, program dokonuje dużej ilości zbędnych odczytów tych samych danych z tabeli.

Należy sprawdzić, czy działania w bazie danych są spójne z realizowanymi funkcjami biznesowymi aplikacji.

Programowanie logiki w zapytaniach

Istnieje kilka sposobów implementacji logiki biznesowej w aplikacji wykorzystującej bazę danych. Część logiki proceduralnej można zaimplementować w ramach instrukcji SQL-a (choć ze swej natury SQL mówi o tym, **co** należy zrobić, a nie **jak**). Nawet w przypadku dobrej integracji SQL-a w innych językach programowania zaleca się, aby jak najwięcej logiki biznesowej ujmować w SQL-u. Taka strategia pozwala na uzyskanie wyższej wydajności przetwarzania danych niż w przypadku implementacji logiki w aplikacji. Języki proceduralne to takie, w których można definiować iteracje (pętle) oraz stosować logikę warunkową (konstrukcje *if...then...else*). SQL nie potrzebuje pętli, ponieważ ze swojej natury operuje na zbiorach danych. Potrzebuje jedynie możliwości określania warunków wykonania określonych działań.

Logika warunkowa wymaga obsługi dwóch elementów: IF i ELSE. Obsługa IF w SQL-u to prosta sprawa: warunek `WHERE` zapewnia dokładnie taką semantykę. Natomiast z obsługą logiki ELSE jest pewien problem. Na przykład mamy za zadanie pobrać z tabeli zbiór wierszy, po czym wykonać różne typy operacji, w zależności od typów zbiorów. Fragment tej logiki można zasymulować z użyciem wyrażenia `CASE` (Oracle od dawna obsługuje odpowiednik tej operacji w postaci funkcji `decode()`[1]). Między innymi można modyfikować w locie wartości zwracane w ramach zbioru wynikowego w zależności od spełnienia określonych warunków. W pseudokodzie można to zapisać następująco[2]:

```
CASE
  WHEN warunek THEN <zwrócenie określonej wartości>
  WHEN warunek THEN <zwrócenie innej wartości>
  WHEN warunek THEN <zwrócenie jeszcze innej wartości>
  ELSE <wartość domyślna>
END
```

Porównywanie wartości liczbowych i dat to operacje intuicyjne. W przypadku ciągów znaków mogą być przydatne funkcje znakowe, jak `greatest()` czy `least()` znane z Oracle, czy `strcmp()` z MySQL-a. Czasem też bywa

[1] Funkcja `decode()` jest nieco bardziej „surowa" w stosunku do konstrukcji `CASE`. Do uzyskania tych samych efektów może być konieczne wykorzystanie dodatkowych funkcji, na przykład `sign()`.

[2] Istnieją dwa warianty konstrukcji `CASE`, przedstawiona wersja jest bardziej zaawansowana.

możliwe zastosowanie pewnej formy logiki w instrukcjach za pomocą wielokrotnych i logicznych operacji wstawiania do tabel oraz za pomocą wstawiania łączącego[3] (`merge insert`). Nie należy unikać takich instrukcji, o ile są dostępne w posiadanym systemie zarządzania bazami danych. Innymi słowy, polecenia SQL-a można wyposażyć w dużą ilość elementów kontrolnych. W przypadku pojedynczej operacji korzyść z tych mechanizmów być może nie jest wielka, lecz z zastosowaniem instrukcji `CASE` i wielu instrukcji wykonywanych warunkowo jest już o co walczyć.

O ile to możliwe, warto implementować logikę aplikacji w zapytaniach SQL zamiast w wykorzystującej je aplikacji.

Jednoczesne wielokrotne modyfikacje

Moje główne założenie w tym podrozdziale opiera się na stwierdzeniu, że kolejne modyfikacje danych w pojedynczej tabeli są dopuszczalne pod warunkiem że dotyczą rozłącznych zbiorów wierszy. W przeciwnym razie należy łączyć je w ramach pojedynczego zapytania. Oto przykład z rzeczywistej aplikacji[4]:

```
update tbo_invoice_extractor
set pga_status = 0
where pga_status in (1, 3)
  and inv_type = 0;
update tbo_invoice_extractor
   set rd_status = 0
where  rd_status in (1, 3)
  and  inv_type = 0;
```

W tej samej tabeli dokonywane są dwie kolejne operacje modyfikujące. Czy te same wiersze będą wykorzystywane dwukrotnie? Nie ma możliwości, aby to stwierdzić. Zasadniczym pytaniem jest tu jednak, jak wydajne są kryteria wyszukiwania? Atrybuty o nazwach `type` (typ) lub `status` z dużym prawdopodobieństwem gwarantują słabą dystrybucję wartości. Jest zatem całkiem możliwe, że najefektywniejszym sposobem odczytu tych danych będzie pełne, sekwencyjne przeszukiwanie tabeli.

---

[3] Dostępny na przykład w Oracle od wersji 9.2.

[4] Nazwy tabel zostały zmienione.

Może też być tak, że jedno z zapytań wykorzysta indeks, a drugie będzie wymagało pełnego przeszukiwania. W najkorzystniejszym przypadku obydwa zapytania skorzystają z wydajnego indeksu. Niezależnie jednak od tego, nie mamy prawie nic do stracenia, aby nie spróbować połączyć obydwu zapytań w jedno:

```
update tbo_invoice_extractor
set pga_status = (case pga_status
                    when 1 then 0
                    when 3 then 0
                    else pga_status
                  end),
       rd_status = (case rd_status
                      when 1 then 0
                      when 3 then 0
                      else rd_status
                    end)
where (pga_status in (1, 3)
       or rd_status in (1, 3))
and inv_type = 0;
```

Istnieje prawdopodobieństwo wystąpienia niewielkiego narzutu spowodowanego aktualizacją kolumn o wartości już przez nie posiadanej. Jednak w większości przypadków jedna złożona aktualizacja danych jest o wiele szybsza niż składowe wywołane osobno. Warto zauważyć zastosowanie logiki warunkowej z użyciem instrukcji `CASE`. Dzięki temu przetworzone zostaną tylko te wiersze, które spełniają kryteria, niezależnie od tego, jak wiele kryteriów będzie zastosowanych w zapytaniu.

Operacje modyfikujące warto wykonywać w pojedynczej, złożonej operacji, aby zminimalizować wielokrotne odczyty tej samej tabeli.

Ostrożne wykorzystanie funkcji użytkownika

Gdy w zapytaniu jest wykorzystana funkcja użytkownika, istnieje możliwość, że będzie wywoływana wielokrotnie. Jeśli funkcja występuje w liście `SELECT`, będzie wywoływana dla każdego zwróconego wiersza. Jeśli wystąpi w instrukcji `WHERE`, będzie wywoływana dla każdego sprawdzonego

wiersza, który spełnia kryteria sprawdzone wcześniej. To może oznaczać bardzo wiele wywołań w przypadku, gdy wcześniej sprawdzane kryteria nie są bardzo mocno selektywne.

Warto się zastanowić, co się stanie, gdy taka funkcja wywołuje inne zapytanie. To zapytanie będzie wywoływane przy każdym wywołaniu funkcji. W praktyce jej wynik będzie taki sam jak w przypadku wywołania podzapytania, z tą różnicą, że w przypadku zapytania ukrytego w funkcji optymalizator nie ma możliwości lepszego zoptymalizowania zapytania głównego. Co więcej, procedura osadzona jest wykonywana na osobnej warstwie abstrakcji w stosunku do silnika SQL, więc będzie działać mniej wydajnie niż bezpośrednie podzapytanie.

Zaprezentuję przykład demonstrujący zagrożenia wynikające z ukrywania kodu SQL w funkcjach użytkownika. Weźmy pod uwagę tabelę `flights` opisującą loty linii lotniczych. Tabela ta zawiera kolumny: `flight_number`, `departure_time`, `arrival_time` i `iata_airport_codes`[5]. Słownik kodów (około dziewięć tysięcy pozycji) jest zapisany w osobnej tabeli zawierającej nazwę miasta (lub lotniska, jeśli w jednym mieście znajduje się kilka lotnisk), nazwę kraju itp. Oczywiście każda informacja o locie wyświetlana użytkownikom powinna zawierać nazwę miasta i lotniska docelowego zamiast nic niemówiącego kodu IATA.

W tym miejscu trafiamy na jedną ze sprzeczności w inżynierii nowoczesnego oprogramowania. Do „dobrych praktyk" programowania zaliczana jest między innymi modularność, polegająca w uproszczeniu na opracowaniu kilku odosobnionych warstw logiki. Ta zasada sprawdza się doskonale w ogólnym przypadku, lecz w kontekście baz danych, w których kod stanowi element wspólny między programistą a bazą danych, potrzeba zastosowania modularności kodu jest znacznie mniej wyraźna. Zastosujmy jednak zasadę modularności, tworząc niewielką funkcję zwracającą pełną nazwę lotniska na podstawie kodu IATA:

```
create or replace function airport_city(iata_code in char)
return varchar2
is
  city_name varchar2(50);
begin
  select city
```

[5] IATA: International Air Transport Association.

```
  into city_naine
  from iata_airport_codes
  where code = iata_code;
  return(city_name);
end;
/
```

Dla czytelników niezaznajomionych ze składnią Oracle: wywołanie `trunc(sysdate)` zwraca dzisiejszą datę, godzinę 00:00, arytmetyka dat jest oparta na dniach. Warunek dotyczący czasów odlotu odnosi się zatem do czasów między 8:30 a 16:00 dnia dzisiejszego. Zapytania wykorzystujące funkcję `airport_city()` mogą być bardzo proste, na przykład:

```
select flight_number,
       to_char(departure_time, 'HH24:MI') DEPARTURE,
       airport_city(arrival) "TO"
from flights
where departure_time between trunc(sysdate) + 17/48
                         and trunc(sysdate) + 16/24
order by departure_time
/
```

To zapytanie wykonuje się z zadowalającą prędkością. Z zastosowaniem losowej próbki na mojej maszynie siedemdziesiąt siedem wierszy jest zwracanych w czasie 0,18 sekundy (średnia z kilku wywołań). Taka wydajność jest do przyjęcia. Statystyki informują jednak, że podczas wywołania zostały odczytane trzysta trzy bloki w pięćdziesięciu trzech operacjach odczytu z dysku. A należy pamiętać, że ta funkcja jest wywoływana rekurencyjnie dla każdego wiersza.

Alternatywą dla funkcji pobierającej dane z tabeli (słownika) może być złączenie tabel. W tym przypadku zapytanie nieco się skomplikuje:

```
select f.flight_number,
       to_char(f.departure_time, 'HH24:MI') DEPARTURE,
       a.city "TO"
from flights f,
     iata_airport_codes a
where a.code = f.arrival
  and departure_time between trunc(sysdate) + 17/48
                         and trunc(sysdate) + 16/24
order by departure_time
/
```

To zapytanie wykonuje się w czasie 0,05 sekundy (te same statystyki, ale nie mamy do czynienia z rekurencyjnymi wywołaniami). Takie oszczędności mogą wydać się niewiele warte — trzykrotne przyspieszenie zapytania w wersji nieoptymalnej trwającego ułamek sekundy. Jednak dość powszechne jest, że w rozbudowanych systemach (między innymi na lotniskach) niektóre zapytania są wywoływane setki tysięcy razy dziennie. Załóżmy, że nasze zapytanie musi być wywoływane pięćdziesiąt tysięcy razy dziennie. Gdy zostanie użyta wersja zapytania wykorzystująca funkcję, całkowity czas wykonania tych zapytań wyniesie około dwie godziny i trzydzieści minut. W przypadku złączenia będą to czterdzieści dwie minuty. Oznacza to usprawnienie rzędu 300%, co w środowiskach o dużej liczbie zapytań oznacza znaczące przyspieszenie, które może przekładać się na konkretne oszczędności finansowe. Bardzo często zastosowanie funkcji powoduje niespodziewany spadek wydajności zapytania. Co więcej, wydłużenie czasu wykonania zapytań powoduje, że mniej użytkowników jest w stanie korzystać z bazy danych jednocześnie, o czym więcej piszę w rozdziale 9.

Kod funkcji użytkownika nie jest poddawany analizie optymalizatora.

Oszczędny SQL

Doświadczony programista baz danych zawsze stara się wykonać jak najwięcej pracy za pomocą jak najmniejszej liczby instrukcji SQL-a. Klasyczny programista natomiast stara się dostosować swój program do ustalonego schematu funkcyjnego. Na przykład:

```
-- Odczyt początku okresu księgowego
select closure_date
into dtPerSta
from tperrslt
where fiscal_year=to_char(Param_dtAcc,'YYYY')
  and rslt_period='1' || to_char(Param_dtAcc,'MM');
-- Odczyt końca okresu na podstawie daty początku
select closure_date
into dtPerClosure
from tperrslt
where fiscal_year=to_char(Param_dtAcc,'YYYY')
  and rslt_period='9' || to_char(Param_dtAcc,'MM');
```

To jest przykład kodu o bardzo niskiej jakości, mimo tego że szybkość jego wykonania jest zadowalająca. Niestety, taka jakość jest typowa dla większości kodu, z którym muszą mierzyć się specjaliści od optymalizacji. Dlaczego dane są odczytywane z zastosowaniem dwóch osobnych zapytań? Ten przykład był uruchamiany na bazie Oracle, w której łatwo zaimplementować zapytanie zapisujące odpowiednie wartości w tabeli wynikowej. Wystarczy odpowiednio zastosować instrukcję `ORDER BY` na kolumnie `rslt_period`:

```
select closure_date
bulk collect into dtPerStaArray
from tperrslt
where fiscal_year=to_char(Param_dtAcc,'YYYY')
  and rslt_period in ('1' || to_char(Param_dtAcc,'MM'),
                      '9' || to_char(Param_dtAcc,'MM'))
order by rslt_period;
```

Dwie odczytane daty są zapisywane odpowiednio w pierwszej i drugiej komórce macierzy. Operacja `bulk collect` jest specyficzna dla języka PL/SQL, lecz w pozostałych językach obsługujących pobieranie danych do macierzy obowiązuje podobna zasada.

Warto zauważyć, że macierz nie jest tu niezbędna, a te dwie wartości można pobrać do zmiennych skalarnych, wystarczy zastosować następującą sztuczkę[6]:

```
select max(decode(substr(rslt_period, 1, 1), -- sprawdzenie pierwszego znaku
           '1', closure_date,
                -- jeśli to '1', zwracamy datę
                to_date('14/10/1066', 'DD/MM/YYYY'))),
                -- w przeciwnym razie
       max(decode(substr(rslt_period, 1, 1),
           '9', closuredate, -- o tę datę chodzi
                to_date('14/10/1066', 'DD/MM/YYYY'))),
into dtPerSta, dtPerClosure
from tperrslt
where fiscal_year=to_char(Param_dtAcc, 'YYYY')
  and rslt_period in ('1' || to_char(Param_dtAcc,'MM'),
                      '9' || to_char(Param_dtAcc,'MM'));
```

[6] Funkcja `decode()` baz danych Oracle działa jak instrukcja `CASE`. Dane porównywane podaje się w pierwszym argumencie. Jeśli wartość jest równa drugiemu argumentowi, zwracany jest trzeci. Jeśli nie zostanie podany piąty argument, w takim przypadku czwarty jest traktowany jako wartość `ELSE`; w przeciwnym razie, jeśli pierwszy argument jest równy czwartemu, zwracany jest piąty i tak dalej w odpowiednich parach wartości.

W tym przykładzie wynik będzie dwuwierszowy, a oczekujemy wyniku jednowierszowego zawierającego dwie kolumny (tak, jak w przykładzie z macierzą). Dokonamy tego, sprawdzając za każdym razem wartość w kolumnie rozróżniającej wartości z każdego wiersza, czyli `rslt_period`. Jeśli odnaleziony wiersz jest tym, którego szukamy, zwracana jest odpowiednia data. W przeciwnym razie zwracana jest dowolna data (w tym przypadku data bitwy pod Hastings), znacznie starsza (z punktu widzenia porównania „**mniejsza**") od jakiejkolwiek daty w tej tabeli. Wybierając maksimum, mamy pewność, że otrzymamy odpowiednią datę. Ten trik jest bardzo praktyczny i można go z powodzeniem stosować do danych znakowych lub liczbowych. Więcej tego typu technik omówię w rozdziale 11.

SQL jest językiem deklaratywnym, zatem należy zachować dystans do proceduralności zastosowań biznesowych.

Ofensywne kodowanie w SQL-u

Programistom często doradza się programowanie defensywne polegające między innymi na sprawdzaniu poprawności wszystkich parametrów przed ich zastosowaniem w wywołaniu. Przy korzystaniu z baz danych większe zalety ma jednak kodowanie ofensywne, polegające na wykonywaniu kilku działań równolegle.

Dobrym przykładem jest mechanizm obsługi kontroli poprawności polegający na wykonywaniu serii sprawdzeń i zaprojektowany w ten sposób, że w przypadku wystąpienia choć jednego wyniku negatywnego wywoływany jest wyjątek. Załóżmy, że mamy przetworzyć płatność kartą płatniczą. Kontrola takiej transakcji składa się z kilku etapów. Należy sprawdzić, że poprawny jest identyfikator klienta i numer karty oraz że są prawidłowo ze sobą powiązane. Należy również zweryfikować datę ważności karty. No i oczywiście bieżący zakup nie może spowodować przekroczenia limitu karty. Gdy wszystkie testy zakończą się pomyślnie, może zostać przeprowadzona operacja obciążenia konta karty.

Niedoświadczony programista mógłby napisać coś takiego:

```
select count(*)
from customers
where customer_id = id_klienta
```

W tym miejscu następuje sprawdzenie wyniku, a jeśli wynik jest pomyślny, następuje wywołanie:

```
select card_num, expiry_date, credit_limit
from accounts
where customer_id = id_klienta
```

Tutaj również nastąpi sprawdzenie wyniku, po którym (w przypadku powodzenia) wywoływana jest transakcja finansowa.

Doświadczony programista zapewne napisze to nieco inaczej (zakładając, że `today()` to funkcja zwracająca bieżącą datę):

```
update accounts
set balance = balance – wielkosc_zamowienia
where balance >= wielkosc_zamowienia
  and credit_limit >= wielkosc_zamowienia
  and expiry_date > today()
  and customer_id = id_klienta
  and card_num = numer_karty
```

Tutaj następuje sprawdzenie liczby zmodyfikowanych wierszy. Jeśli jest to zero, przyczynę takiej sytuacji można sprawdzić za pomocą jednego zapytania:

```
select c.customer_id, a.card_num, a.expiry_date,
       a.creditlimit, a.balance
from customers c
     left outer join accounts a
          on a.customer_id = c.customer_id
          and a.cardnum = numer_karty
where c.customer_id = id_klienta
```

Jeśli zapytanie nie zwróci żadnego wiersza, oznacza to, że wartość `customer_id` jest błędna, jeśli w wyniku `card_num` jest NULL, oznacza to, że numer karty jest błędny itd. Jednak w większości przypadków to drugie zapytanie nie będzie nawet uruchomione.

UWAGA

Warto zwrócić uwagę na wywołanie `count(*)` w pierwszym fragmencie kodu niedoświadczonego programisty. To doskonała ilustracja błędnego użycia funkcji `count(*)` do sprawdzenia, czy w tabeli istnieją pozycje spełniające warunek.

Zasadniczą cechą programowania ofensywnego jest opieranie swoich założeń na rozsądnym prawdopodobieństwie. Na przykład nie ma większego sensu, by sprawdzać istnienie klienta. Jeśli nie istnieje, w bazie danych nie znajdzie się żaden dotyczący go rekord (zatem w wyniku wywołania zapytania bez wcześniejszej kontroli i tak nie zostaną zmodyfikowane

żadne dane)! Zakładamy, że wszystko zakończy się powodzeniem, a nawet jeśli tak się nie stanie, przygotowujemy mechanizm ratujący nas z opresji w tym jednym punkcie — i tylko w tym jednym. Co interesujące, tego typu strategia przypomina nieco „optymistyczną kontrolę współdzielenia" zastosowaną w niektórych bazach danych. Chodzi o to, że założono z góry, iż z dużym prawdopodobieństwem nie będzie sytuacji konfliktu dostępu do danych, a jeśli jednak się to zdarzy, dopiero wówczas uruchamiane są stosowne konstrukcje kontrolne. W wyniku zastosowania tej strategii wydajność systemu jest znacznie wyższa niż w przypadku systemów stosujących strategię pesymistyczną.

Należy kodować w oparciu o rachunek prawdopodobieństwa. Zakłada się, że najprawdopodobniej wszystko zakończy się pomyślnie, a dopiero w przypadku niepowodzenia uruchamia się plany awaryjne.

Świadome użycie wyjątków

Między odwagą a zapalczywością różnica jest dość subtelna. Gdy zalecam stosowanie agresywnych metod kodowania, nie sugeruję bynajmniej szarży w stylu Lekkiej Brygady pod Bałakławą[7]. Programowanie z użyciem wyjątków również może być konsekwencją brawury, gdy dumni programiści decydują się „iść na całość". Mają bowiem przeświadczenie, że testy i możliwość obsługi wyjątków będą ich tarczą w tym boju. No tak, odważni umierają młodo!

Jak sugeruje nazwa, wyjątki to zdarzenia występujące w niecodziennych sytuacjach. W programowaniu z użyciem baz danych nie wszystkie wyjątki wykorzystują te same zasoby systemowe. Należy poznać te uwarunkowania, aby korzystać z wyjątków w sposób inteligentny. Można wyróżnić dobre wyjątki, wywoływane, zanim zostaje wykonane działanie, oraz złe wyjątki, które są wywoływane dopiero po fakcie wyrządzenia poważnych zniszczeń.

[7] Podczas Wojny Krymskiej, w 1854 roku, odbyła się bitwa między wojskami Anglii, Francji i Turcji a siłami Rosji. W wyniku nieprecyzyjnego rozkazu oraz osobistych animozji między niektórymi z dowódców sił sprzymierzonych doszło do szarży ponad sześciuset żołnierzy kawalerii brytyjskiej wprost na baterię rosyjskiej artylerii. Na skutek starcia zginęło około stu dwudziestu kawalerzystów oraz połowa koni, bez jakiegokolwiek dobrego rezultatu. Odwaga ludzi została wkrótce wysławiona przez wiersz Tennysona, a potem w kilku filmach hollywoodzkich, dzięki czemu zwykła głupota jednej militarnej decyzji obróciła się w mit.

Zapytanie wykorzystujące klucz główny, które nie znajdzie żadnych wierszy, wykorzystuje niewiele zasobów — sytuacja jest identyfikowana już na etapie przeszukiwania indeksu. Jeśli jednak w celu stwierdzenia, że dane spełniające warunek nie występują w tabeli, zapytanie nie może użyć indeksu, zachodzi konieczność dokonania pełnego przeszukiwania tabeli (*full scan*). W przypadku wielkich tabel czas potrzebny do odczytu sekwencyjnego w systemie działającym w danym monecie na granicy swojej wydajności można potraktować jako czynnik katastrofalny.

Niektóre wyjątki są szczególnie kosztowne, nawet przy najbardziej sprzyjających okolicznościach. Weźmy na przykład wykrywanie duplikatów. W jaki sposób w bazie danych jest obsługiwany mechanizm unikalności? Prawie zawsze służy do tego unikalny indeks i gdy wystąpi próba wprowadzenia do tabeli wartości zawierającej klucz występujący już w indeksie, zadziała mechanizm zabezpieczający przed zduplikowaniem klucza, co efektywnie zablokuje zapis duplikatu. Jednakże zanim nastąpi próba zapisu indeksu (weryfikacji duplikatu), w tabeli musi zostać fizycznie zapisana odpowiednia wartość (do procedury indeksującej przesyłany jest fizyczny adres wiersza w tabeli). Z tego wynika, że naruszenie ograniczenia unikalności klucza następuje po fakcie zapisu w tabeli danych, które muszą być wycofane, czemu dodatkowo towarzyszy komunikat informujący o wystąpieniu błędu. Wszystkie te operacje wiążą się z określonym kosztem czasowym. Największym jednak grzechem jest podejmowanie samodzielnych prób działania na poziomie wyjątków. W takim przypadku przejmujemy od systemu zadanie obsługi operacji na poziomie wierszy, nie całych zbiorów danych, czyli sprzeciwiamy się fundamentalnej koncepcji relacyjnego modelu danych. Konsekwencją występowania częstych naruszeń ograniczeń w bazie będzie w takim przypadku stopniowa degradacja jej wydajności.

Przyjrzyjmy się przykładowi opartemu na bazie Oracle. Załóżmy, że pracujemy nad integracją systemów informatycznych dwóch połączonych firm. Adres e-mail został ustandaryzowany w postaci wzorca `<InicjałNazwisko>` i ma zawierać co najwyżej dwanaście znaków, wszystkie spacje i znaki specjalne są zastępowane znakami podkreślenia[8].

[8] Przykład nie uwzględnia obsługi polskich znaków diakrytycznych, niedozwolonych w adresach e-mail — *przyp.red.*

Załóżmy, że nową tabelę pracowników należy wypełnić trzema tysiącami wierszy z tabeli `employees_old`. Chcemy też, żeby każdy pracownik posiadał unikalny adres e-mail. Z tego powodu musimy zastosować określoną zasadę nazewnictwa: Jan Kowalski będzie miał e-mail o postaci `jkowalski`, a Józef Kowalski (żadnego pokrewieństwa) `jkowalski2` itd. W naszych danych testowych znajdziemy trzydzieści trzy potencjalne pozycje konfliktowe, co przy próbie ładowania danych da następujący efekt:

```
SQL> insert into employees(emp_num, emp_name,
                           emp_firstname, emp_email)
  2 select emp_num,
  3        emp_name,
  4        emp_firstname,
  5        substr(substr(EMP_FIRSTNAME, 1, 1)
  6              ||translate(EMP_NAME, ' ''', '_'), 1, 12)
  7 from employees_old;

insert into employees(emp_num, emp_name, emp_firstname, emp_email)
*
ERROR at line 1:
ORA-00001: unique constraint (EMP_EMAIL_UQ) violated


Elapsed: 00:00:00.85
```

Trzydzieści trzy duplikaty ze zbioru trzech tysięcy to trochę powyżej 1%, być może zatem warto byłoby obsłużyć te 99%, a elementy problemowe obsłużyć z użyciem wyjątków? W końcu 1% danych nie powinien powodować znacznego obciążenia bazy w wyniku procedury obsługi wyjątków. Poniżej kod realizujący ten optymistyczny scenariusz:

```
SQL> declare
  2     v_counter  varchar2(12);
  3     b_ok       boolean;
  4     n_counter  number;
  5     cursor c is select emp_num,
  6                        emp_name,
  7                        emp_firstname
  8                 from employees_old;
  9 begin
 10  for rec in c
 11  loop
 12    begin
 13      insert into employees(emp_num, emp_name,
 14                            emp_firstname, emp_email)
```

```
15      values (rec.emp_num,
16              rec.emp_name,
17              rec.emp_firstname,
18              substr(substr(rec.emp_firstname, 1, 1)
19              ||translate(rec.emp_name, ' ''', ' '), 1, 12));
20      exception
21       when dup_val_on_index then
22         b_ok := FALSE;
23         n_counter := 1;
24         begin
25           v counter := ltrim(to_char(n_counter));
26           insert into employees(emp_num, emp_name,
27                                 emp_firstname, emp_email)
28           values (rec.emp_num,
29                   rec.emp_name,
30                   rec.emp_firstname,
31                   substr(substr(rec.emp_firstname, 1, 1)
32                     ||translate(rec.emp_name, ' ''', '__'), 1,
33                     12 - length(v_counter)) || v_counter);
34           b_ok : = TRUE;
35         exception
36          when dup_val_on_index then
37            n_counter := n_counter + 1;
38         end;
39       end;
40    end loop;
41  end;
40 /

PL/SOL procedure successfully completed.
Elapsed: 00:00:18.41
```

Jaki jest jednak rzeczywisty koszt obsługi wyjątków? Gdyby ten sam test przeprowadzić na danych pozbawionych duplikatów, okaże się, że koszt rzeczywistej obsługi wyjątków (ich wystąpień) jest pomijalny. Procedura wywołana na danych z duplikatami działa około osiemnaście sekund, podobnie jak na danych bez duplikatów. Jednak gdy wykonamy ten test (dane bez duplikatów) na naszej oryginalnej procedurze nieobsługującej wyjątków (`insert...select`), zauważymy, że wykona się znacznie szybciej od pętli. Przełączenie się w tryb „wiersz po wierszu" powoduje około 50-procentowy narzut czasu przetwarzania. Czy w takim razie możliwe jest uniknięcie tego trybu? To kwestia tego, czy zdecydujemy się na rezygnację z mechanizmu obsługi wyjątków, który to właśnie zmusił nas do obsługi danych w trybie wierszowym.

Innym sposobem mogłoby być zidentyfikowanie wierszy powodujących powstanie duplikatów i uzupełnienie w nich adresów e-mail kolejnymi liczbami.

Łatwo określić liczbę problematycznych wierszy, wystarczy odpowiednio je zgrupować w zapytaniu SQL. Jednakże uzupełnienie o unikalne liczby może być trudne bez zastosowania funkcji analitycznych dostępnych w niektórych zaawansowanych systemach baz danych. Określenie „funkcje **analityczne**" pochodzi z nomenklatury Oracle. W DB2 funkcje te znane są jako funkcje **OLAP** (ang. *online analytical processing*), w Microsoft SQL Server jako **funkcje rankingu** (ang. **ranking functions**). Warto przyjrzeć się bliżej rozwiązaniom tego typu z punktu widzenia czystego SQL-a.

Każdy adres e-mail może mieć dopisany unikalny numer przy wykorzystaniu do tego rankingu według wieku pracownika. Numer 1 otrzyma najstarszy pracownik w danej grupie duplikatów, numer 2 kolejny pod względem wieku z tej grupy itd. Umieszczając tę liczbę w podzapytaniu, mamy możliwość uniknięcia dopisania czegokolwiek do pierwszego znalezionego adresu e-mail w każdej grupie, natomiast pozostałym przypisywane są kolejne liczby sekwencji. Sposób realizacji tego zadania demonstruje poniższy kod:

```
SQL> insert into employees(emp_num, emp_firstname,
  2                        emp_name, emp_email)
  3  select emp_num,
  4         emp_firstname,
  5         emp_name,
  6         decode(rn, 1, emp_email,
  7                       substr(emp_email,
  8                       1, 12 - length(ltrim(to_char(rn))))
  9                        || ltrim(to_char(rn)))
 10 from (select emp_num,
 11              emp_firstname,
 12              emp_name,
 13              substr(substr(emp_firstname, 1, 1)
 14                ||translate(emp_name, ' ''', '_'), 1, 12)
 15                         emp_email,
 16              row_number()
 17                 over (partition by
 18                        substr(substr(emp_firstname, 1, 1)
```

```
  19                         ||translate(emp_name,' ''','_'), 1, 12)
  20                        order by emp_num) rn
  21       from employees_old)
  22 /

3000 rows created.

Elapsed: 00:00:11.68
```

Unikamy kosztu przetwarzania w trybie wierszowym, dzięki czemu to rozwiązanie zajmuje około 60% czasu w porównaniu z pętlą.

Obsługa wyjątków zmusza do zastosowania logiki proceduralnej. Zawsze warto brać pod uwagę obsługę wyjątków, jednak w zakresie niezmuszającym do rezygnacji z deklaratywnej specyfiki SQL-a.