Transact-SQL. Czarna księga

Autor: Marcin Szeliga
ISBN: 83-7361-125-8
Format: B5, stron: 492
Przykłady na ftp: 65 kB

Transact-SQL to podstawowy język programowania baz danych w środowisku MS SQL Server. Transact-SQL jest zmodyfikowaną i uzupełnioną o elementy typowe dla proceduralnych języków programowania (jak zmienne i instrukcje sterujące wykonaniem programu) wersją standardu SQL-92. Dzięki temu rozszerzeniu, jego możliwości są znacznie większe niż możliwości standardowego SQL-a.

SQL Server został wyposażony w intuicyjne, a zarazem potężne narzędzie administracyjne – konsolę SQL Server Enterprise Menager. W rezultacie część administratorów SQL Servera nie zna albo nie korzysta z możliwości języka Transact-SQL. A okazuje się, że ta sama operacja może być przeprowadzona kilkukrotnie szybciej, jeżeli zamiast konsoli użyjemy Transact-SQLa.

Książka „Transact-SQL. Czarna Księga" to wyczerpujące i dogłębne kompendium omawiające nie tylko sam język Transact-SQL, ale również zasady projektowania baz danych, a także zarządzanie SQL Serverem za pomocą języka Transact-SQL.

Przedstawiono:

- Standardowe interfejsy języka SQL instalowane wraz z SQL Serverem
- Elementy języka Transact-SQL i składnię poszczególnych instrukcji języka wraz z praktycznymi przykładami ich wykorzystania
- Metody pobierania i modyfikowania danych
- Optymalizację zapytań
- Wyszukiwanie pełnotekstowe i usługa MS Search.
- Zasady projektowania relacyjnych baz danych
- Algorytmy przekształcania relacji poprzez kolejne postacie normalne
- Tworzenie, modyfikowanie i usuwanie wszystkich typów obiektów bazodanowych
- Bezpieczeństwo i kontrolowanie dostępu do danych
- Integrację SQL Servera z innymi serwerami firmy Microsoft
- Tworzenie i przywracanie kopii zapasowych
- Automatyzację czynności administracyjnych
- Monitorowanie pracy SQL Servera i optymalizację jego wydajności
- Łączenie serwerów bazodanowych
- Replikację danych pomiędzy wieloma SZBD

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 4.

Optymalizacja zapytań

Zapytania, tak jak pozostałe instrukcje języka Transact-SQL, przed skompilowaniem i wykonaniem są optymalizowane przez wewnętrzny proces SQL Servera o nazwie **Query Optimizer**. Jego zdaniem jest znalezienie **najtańszego sposobu wykonania instrukcji**. Query Optimizer bazuje na liczbie operacji wejścia-wyjścia oraz na liczbie obliczeń dokonanych przez procesor niezbędnych do wykonania instrukcji.

Query Optimizer szacuje koszt operacji wejścia-wyjścia na podstawie:

1. Struktury tabel przechowujących dane, do których odwołuje się instrukcja i istniejących, związanych z nimi indeksów.
2. Kosztu operacji złączenia różnego typu danych zapisanych w kilku tabelach źródłowych.
3. Istniejących lub tworzonych dynamicznie przez Query Optimizera statystyk opisujących dane źródłowe.
4. Metainformacji opisujących fizyczną strukturę plików, w których zapisane są żądane dane.

Ponieważ Query Optimizer oblicza koszt wykonania instrukcji na podstawie statystyk, nieaktualne lub nieadekwatne statystki spowodują utworzenie nieefektywnego planu jej wykonania. Mechanizm tworzenia i aktualizacji statystyk został opisany w rozdziale 10.

Wynikiem optymalizacji jest znalezienie takiego sposobu wykonania instrukcji, który zwraca najmniejszą z możliwych liczbę wierszy i w związku z tym wymaga odczytania najmniejszej liczby **stron** (8 KB spójnych bloków danych). Celem optymalizacji jest także znalezienie takiego wykonywania, którego całkowity czas przeprowadzania jest najkrótszy.

Fizyczna struktura baz danych została opisana w znajdującym się w. części poświęconej projektowaniu i tworzeniu baz danych rozdziale 8. W rozdziale 10. znajdują się szczegółowe wskazówki dotyczące tworzenia i wykorzystywania indeksów w celu poprawy wydajności zapytań.

Query Optimizer

Optymalizacja instrukcji języka Transact-SQL przebiega według następującego schematu:

1. Po sprawdzeniu poprawności syntaktycznej sprawdzana jest poprawność semantyczna (ang. *Parse*). Na tym etapie instrukcja zostanie „podzielona" na znaczniki interpretowane przez SQL Server.
2. Następuje **standaryzacja** — zapisanie znaczników instrukcji Transact-SQL w jednoznacznej postaci (np. ujednolicenie definiowania aliasów raz definiowanych za pomocą słowa kluczowego `AS`, raz bez niego). Na tym etapie wszystkie redundantne znaczniki instrukcji zostają usunięte.
3. Kolejnym etapem jest **optymalizacja** — wybór jednego z przygotowanych planów wykonania. Na tym etapie następuje analiza indeksów i statystyk oraz metod złączania danych. Etap można podzielić na trzy fazy:
   a. Analiza zapytania — wybór metod wyszukiwania i łączenia danych źródłowych mający na celu zminimalizowanie liczby danych, które muszą zostać odczytane do wykonania instrukcji.
   b. Wybór indeksów — na podstawie statystyk tabel i indeksów następuje wybór tych indeksów, do których odwołanie spowoduje zwrócenie najmniejszej liczby danych (minimalizacja liczby logicznych i fizycznych odczytów bloków danych).
   c. Wybór metody złączania tabel — na podstawie struktury i liczby danych oraz liczby pamięci operacyjnej niezbędnej do wykonania złączenia wybierany jest sposób złączania tabel. W tym momencie określone zostaje również, która tabela będzie traktowana jako wewnętrzna, a która jako zewnętrzna, w przypadku złączenia poprzez pętlę wyszukiwania.
4. Następnie instrukcja wykonana według opracowanego optymalnego planu zostaje skompilowana.
5. Query Optimizer wybiera optymalny sposób pobrania wybranych (wynikowych) danych. Na przykład: czy odwołać się do indeksu, czy odczytać tabelę — wbrew pozorom dla małych tabel odczytanie całej ich zawartości okazuje się tańszym sposobem na odczytanie danych niż odwoływanie się do nich poprzez indeksy.

Skompilowana według optymalnego planu instrukcja języka Transact-SQL zostaje zapisana w **buforze procedury** — przydzielonej przez SQL Server części pamięci operacyjnej wykorzystywanej wyłącznie do przechowywania skompilowanych procedur.

W buforze procedury mogą być przechowywane maksymalnie dwie wersje skompilowanej instrukcji — jedna wykonywana równolegle, druga szeregowo.

Ponieważ w pamięci procedury zapisana zostaje wyłącznie instrukcja języka Transact-SQL, bez informacji o tym, który użytkownik ją wykonał, odwołanie się do niej powoduje, że SQL Server musi wyznaczyć bieżący kontekst jej wykonania. Dlatego, jeżeli instrukcja zawiera wyłącznie identyfikatory obiektów, bez identyfikatora ich właściciela, SQL Server przyjmie, że właścicielem obiektów jest użytkownik wykonujący instrukcję, a dopiero potem sprawdzi, czy obiekty nie należą do użytkownika `dbo`. Wynika z tego, że jawne odwoływanie się do obiektów za pomocą nazwy użytkownika i nazwy obiektu przyspiesza wykonanie skompilowanych instrukcji.

Zoptymalizowana na podstawie nieaktualnych statystyk instrukcja nie będzie dłużej wykonywana według najlepszego planu. Również, jeżeli zmieniła się struktura obiektów, do których odwoływała się instrukcja, próba wykonania tej instrukcji zakończy się błędem. Z tych powodów skompilowana instrukcja jest w zależności od potrzeb **rekompilowana**. Ponowna kompilacja instrukcji jest przeprowadzana, jeżeli:

1. Zmieniona została definicja obiektu, do którego instrukcja się odwołuje (wykonano polecenie `ALTER`).
2. Wymuszono aktualizacje statystyk, na podstawie których przygotowany został plan wykonania instrukcji (wykonano instrukcję `UPDATE STATISTIC`).
3. Usunięto indeks wykorzystywany przez instrukcję (wykonano instrukcję `DROP INDEX`).
4. Z tabeli źródłowej usunięto lub dodano do niej dużą (względem stanu w momencie optymalizacji instrukcji) liczbę wierszy.
5. Wymuszono rekompilację instrukcji (wywołano procedurę systemową `sp_recompile`).

Plan wykonania zapytania

Plan wykonania instrukcji można poznać odczytując zawartość tabeli systemowej `sysindexes`. Jednak bezpośrednie odwoływanie się do tabel systemowych nie jest zalecane, a w tym przypadku istnieją inne sposoby uzyskania żądanych danych.

Query Analyzer pozwala na wyświetlenie statystyk związanych z czasem (włączenie opcji `SET STATISTIC TIME`) i liczbą operacji wejścia-wyjścia (opcja `SET STATISTIC IO`). Ponadto możemy wyświetlić opracowany przez Query Optimizera plan wykonania instrukcji (opcja `SET SHOWPLAN TEXT` lub `SET SHOWPLAN_ALL`). Na przykład po włączeniu dwóch pierwszych opcji wykonanie poniższego zapytania spowoduje wyświetlenie poniższych informacji:

```
USE pubs
SELECT DISTINCT t1.type
FROM titles AS t1
INNER JOIN titles AS t2
ON t1.type = t2.type
WHERE t1.pub_id <> t2.pub_id
GO
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
type
------------
business
psychology
```

```
(2 row(s) affected)
Table 'titles'. Scan count 19, logical reads 38, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
```

Natomiast włączenie opcji `SET SHOWPLAN TEXT` spowoduje wyświetlenie pokazanego w rozdziale 3. planu wykonania instrukcji.

Informacje na temat czasu wykonania poszczególnych opisanych wcześniej faz przetwarzania przez SQL Server instrukcji języka Transact-SQL nie wymagają komentarza. Przyjrzyjmy się informacjom związanymi z liczbą operacji wejścia-wyjścia:

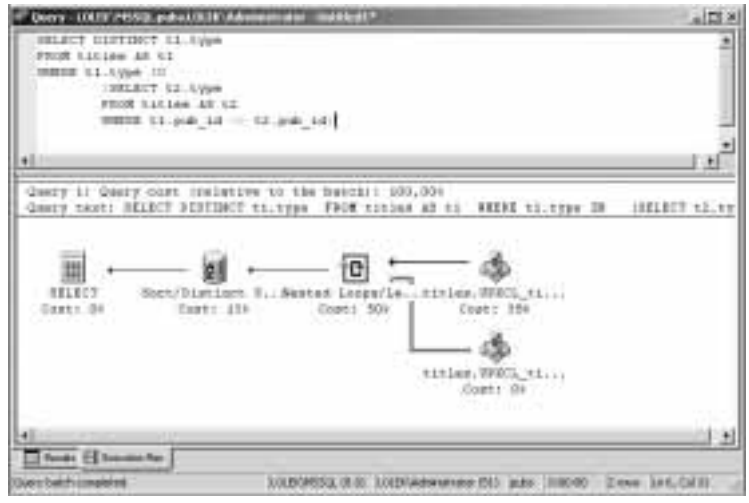
1. `scan count` — określa liczbę odwołań do tabeli źródłowej.
2. `logical reads` — określa liczbę stron danych odczytanych z pamięci podręcznej.
3. `physical reads` — określa liczbę stron danych odczytanych z dysku. Ta liczba nigdy nie jest większa od liczby stron odczytanych z pamięci podręcznej. Na podstawie tych dwóch wartości można obliczyć **współczynnik trafień**:

   współczynnik trafień = (*logical reads* – *physical reads*) / *logical reads*

4. `read-ahead reads` — określa liczbę stron umieszczoną w pamięci podręcznej.

Sam plan wykonania instrukcji może również zostać przedstawiony w postaci graficznej (rysunek 4.1). Zawarte są w nim informacje dotyczące następujących zagadnień:

1. Kroki wykonywania instrukcji i ich kolejność.
2. Logiczne operatory algebry zbiorów użyte podczas wykonywania instrukcji.
3. Fizyczna implementacja tych operatorów wykorzystana do wykonania zapytania (tabela 4.1).



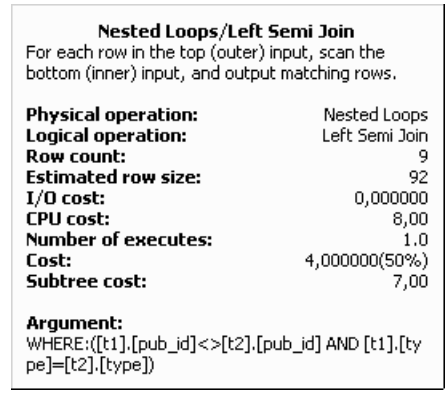
Rysunek 4.1. *Plan wykonania zapytania*

Tabela 4.1. *Lista operatorów fizycznych*

Symbol	Znaczenie	Opis
	Odczytanie indeksu	Wyszukanie (na podstawie podanych warunków) danych z indeksu niegrupującego
	Pętla wyszukiwania	Wyszukanie (z reguły w oparciu o indeks) w wewnętrznej tabeli złączenia wszystkich wierszy zgodnych z kolejnym wierszem zewnętrznej tabeli złączenia
	Porządkowanie danych	Uporządkowanie wszystkich danych źródłowych
	Przeszukanie indeksu	Wyszukanie w indeksie niegrupującym wierszy danych
	Przeszukanie tabeli	Wyszukanie w tabeli danych spełniających podane kryteria
	Przeszukanie wskaźników	Wyszukanie danych na podstawie identyfikatora wiersza lub klucza indeksu grupującego rekordy w tabeli lub w indeksie grupującym
	Wybieranie danych	Wyszukanie w zbiorze danych źródłowych rekordów spełniających podane kryteria
	Zgodność funkcji skrótu	Złączenie tabel na podstawie wyliczonej dla każdego wiersza wartości funkcji skrótu
	Złączenie	Dowolnego typu (z wyjątkiem złączenia tabeli z nią samą oraz złączenia krzyżowego) złączenie tabel lub widoków

Poszczególne kroki związane z wykonaniem instrukcji zapisywane są od strony prawej do lewej. W ramach każdego kroku możliwe jest wykonanie dowolnej liczby operacji — w tym przypadku odczytanie indeksów obu kolumn zostało wykonane jako jeden krok składający się z dwóch operacji. Związek pomiędzy poszczególnymi operacjami reprezentują strzałki. Ustawienie kursora na symbolu operatora fizycznego spowoduje wyświetlenie dodatkowych informacji o danej operacji (rysunek 4.2).

Rysunek 4.2. *Informacje o złączeniu tabel t1 i t2*



Wyświetlając dodatkowe informacje o każdej operacji poznamy:

- argumenty wywołania operacji (ang. *Argument*),
- koszt wykonania operacji i jego szacunkowy udział w koszcie wykonania instrukcji (ang. *Cost*),
- koszt wykonania operacji i operacji przez nią wywołanych (ang. *Subtree cost*),
- liczbę wykonania operacji w ramach instrukcji (ang. *Number of executes*),
- liczbę zwróconych przez operację wierszy (ang. *Row count*),
- szacunkową wielkość zwróconych przez operację wierszy (ang. *Estimated row size*),
- szacunkowy koszt operacji wejścia-wyjścia przeprowadzonych przez operację (ang. *I/O cost*),
- szacunkowy koszt wykorzystania zasobów procesora przez operację (ang. *CPU cost*).

Indeksy zawierające zapytania

Indeksy różnego typu i sposoby ich tworzenia zostały opisane w rozdziale 10.

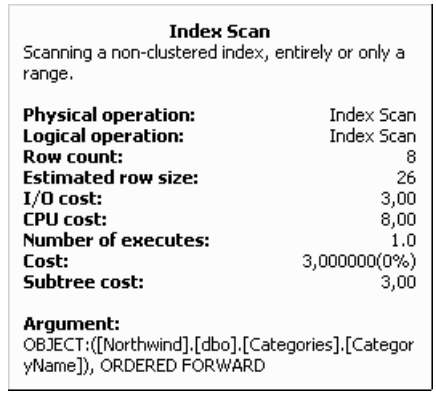
Optymalną pod względem szybkości odczytu danych jest sytuacja, w której wszystkie żądane dane (wyrażenia wymienione w instrukcji `SELECT`) mogą zostać odczytane z indeksu. Mówimy wtedy, że **indeks zawiera zapytanie**. Aby indeks zawierał zapytanie, wszystkie dane źródłowe muszą być poindeksowane. Dotyczy to kolumn wymienionych w poleceniu `SELECT`, w klauzuli `WHERE`, `GROUP BY` i `ORDER BY`. W takim przypadku pobranie danych sprowadza się do znalezienia i odczytania odpowiednich **liści** indeksu, bez konieczności odczytywania stron zawierających dane.

SQL Server pozwala na tworzenie indeksów dla danych będących wynikiem funkcji grupującej. Jeżeli zapytania dotyczą wyliczanych wartości utworzenie indeksów tego typu spowoduje wielokrotny wzrost wydajności zapytania, w ramach którego obliczane są te wartości.

Sprawdzić, czy istnieje indeks zawierający zapytanie, możemy wyświetlając graficzny plan jego wykonania, a następnie wyświetlając szczegóły operacji odczytania indeksu. Jeżeli znajduje się tam informacja: `Scanning a non-clustered index entirely or only a range` oznacza to, że do wykonania zapytania wykorzystano wyłącznie dane przechowywane w niegrupującym indeksie (rysunek 4.3).

Nie oznacza to, że najlepszym rozwiązaniem jest stworzenie indeksu zawierającego wszystkie kolumny wybranej tabeli — w takim przypadku utworzymy po prostu kopię tabeli i zamiast spodziewanego zysku wydajności uzyskamy jej spadek. Tworząc indeksy zawierające zapytania należy wybierać wyłącznie kolumny często występujące w zapytaniach i o podobnej wielkości (dodanie do indeksu zawierającego dane z trzech kolumn typu `smallint` danych z kolumny typu `varchar(255)` jest ekstremalnym przykładem źle zaprojektowanego indeksu).

Rysunek 4.3.
Odczytanie nazw poszczególnych kategorii z tabeli categories



Niegrupujący indeks zawierający zapytanie jest funkcjonalnym odpowiednikiem indeksu grupującego i korzystanie z niego wiąże się z tymi samymi korzyściami — po znalezieniu pierwszej spełniającej podany warunek wartości nie potrzebne jest tworzenie wskaźników do zewnętrznych danych (tabeli), a ponieważ dane indeksu zapisane są w określonym porządku, wystarczy odczytać określoną liczbę stron przechowujących żądane dane.

Analiza zapytań

W zależności od typu zapytania, wykorzystanych operatorów logicznych czy metod łączenia tabel, Query Optimizer posłuży się odmiennymi szablonami umożliwiającymi znalezienie optymalnego planu wykonania instrukcji. Znajomość tych szablonów jest niezbędna do tworzenia wydajnych zapytań.

Aby wyniki uzyskane przez Czytelników nie różniły się od przedstawianych w książce przed wykonaniem opisywanych programów należy przywrócić oryginalną postać baz `Northwind` i `pubs`. Można to osiągnąć zatrzymując SQL Server i nadpisując pliki *.mdf* i *.ldf* tych baz plikami znajdującymi się na płycie instalacyjnej serwera.

SARG

Akronim `SARG` (ang. *Search ARGuments*) określa pewien specjalny podzbiór argumentów wyszukiwania, czyli wyrażeń wymienionych w klauzuli `WHERE` instrukcji `SELECT`. Argumenty `SARG` charakteryzuje:

- Obecność stałej, której wartość jest porównywana z polami wybranej kolumny tabeli źródłowej.
- Wyszukiwanie wartości równych wzorcowi, należących do zakresu wyznaczonego przez wzorzec lub przez połączenie kilku argumentów `SARG` za pomocą operatora koniunkcji.

Wynika z tego, że dla argumentów `SARG` dopuszczalnymi operatorami są: =, <, <=, >, >=, `BETWEEN` oraz, po spełnieniu dodatkowych warunków, `LIKE`. To, czy argument zawierający operator `LIKE` może zostać uznany za argument `SARG`, zależy od pozycji symbolu wieloznacznego (`%`). Jeżeli występuje on jako ostatni znak wzorca, czyli możliwe będzie

ograniczenie liczby stron, które SQL Server będzie musiał odczytać, aby znaleźć żądane wartości, to taki argument będzie argumentem SARG.

W przypadku użycia operatora <>, NOT, !=, !>, !<, NOT EXISTS, NOT IN czy NOT LIKE konieczne okazuje się sprawdzenie wartości wszystkich wierszy tabeli źródłowej. Chociaż nie oznacza to, że SQL Server nie potrafi skorzystać z indeksów przy tworzeniu planu zapytania zawierającego wyżej wymienione operatory, to należy dążyć do zastąpienia takich operatorów argumentami SARG.

Wykonanie zapytania zawierającego argument SARG przebiega według następującego schematu:

1. Optymalizator sprawdza, czy istnieją przydatne do wykonania zapytania indeksy.
2. Jeżeli taki indeks zostanie znaleziony, rozpoczyna się wyszukiwanie (za pomocą operatora >=) stron indeksu przechowujących dane zgodne z żądanym wzorcem.

Domyślnie dane indeksów zapisane są na dysku w porządku rosnącym.

3. Wszystkie wartości spełniające zadane kryteria są odczytywane, a jeśli jest to konieczne, odczytywane są z tabeli wartości przechowywane w pozostałych polach danego wiersza.

Porównajmy koszt i plan wykonania zapytania wykorzystującego argument SARG z zapytaniem zwracającym ten sam wynik, ale niezawierającym argumentu SARG:

```
USE Northwind
SELECT *
FROM dbo.Orders
WHERE CustomerID = 'GROSR'
GO
StmtText
-----------------------------------------------------------------------------------------
  |--Bookmark Lookup(BOOKMARK:([Bmk1000]), OBJECT:([Northwind].[dbo].[Orders]))
       |--Index Seek(OBJECT:([Northwind].[dbo].[Orders].[CustomerID]),
       ↳SEEK:([Orders].[CustomerID]=Convert([@1])) ORDERED FORWARD)
(2 row(s) affected)
Table 'Orders'. Scan count 1, logical reads 6, physical reads 0, read-ahead reads 0.
```

Ponieważ pobieramy wszystkie dane z wszystkich kolumn tabeli, optymalizator musi użyć wskaźnika do tabeli — utworzenie indeksu kompozytowego dla wszystkich kolumn tabeli spowodowałby jedynie pogorszenie wydajności. Jednak wybór wierszy spełniających zadane kryteria odbywa się poprzez odczytanie kolejnych liści indeksu, dzięki czemu SQL Server musi odczytać jedynie wybrane strony tabeli.

Gdyby liczba odczytanych stron była stosunkowo duża, tak jak na przykład w przypadku klienta o identyfikatorze FRANK, optymalizator zdecydowałby się na wykorzystanie indeksu grupującego i sprawdzanie warunku bezpośrednio na danych tabeli, bez wykorzystywania indeksu powiązanego z kolumną CustomerID:

```
USE Northwind
SELECT *
FROM dbo.Orders
```

```
WHERE CustomerID = 'FRANK'
GO
StmtText
------------------------------------------------------------------------------------------
  |--Clustered Index Scan(OBJECT:([Northwind].[dbo].[Orders].[PK_Orders]),
  ↳WHERE:([Orders].[CustomerID]=Convert([@1])))
(1 row(s) affected)
Table 'Orders'. Scan count 1, logical reads 21, physical reads 0, read-ahead reads 0.
```

Wykonanie zapytania według wybranego przez optymalizator planu wymagało odczytania 21 stron danych. Przekonajmy się, czy próba wymuszenia użycia indeksu powiązanego z kolumną `CustomerID` poprawi wydajność:

```
USE Northwind
SELECT *
FROM dbo.Orders (INDEX = CustomerID)
WHERE CustomerID = 'FRANK'
GO
StmtText
------------------------------------------------------------------------------------------
  |--Bookmark Lookup(BOOKMARK:([Bmk1000]), OBJECT:([Northwind].[dbo].[Orders]))
       |--Index Seek(OBJECT:([Northwind].[dbo].[Orders].[CustomerID]),
       ↳SEEK:([Orders].[CustomerID]='FRANK ') ORDERED FORWARD)
(2 row(s) affected)
Table 'Orders'. Scan count 1, logical reads 40, physical reads 0, read-ahead reads 0.
```

Jak widać, optymalizator właściwie oszacował liczbę operacji wejścia-wyjścia i wybrał optymalny plan wykonania zapytania. Jeżeli któryś z Czytelników udokumentuje i prześle do firmy Microsoft informację o tym, że wystąpił przypadek, w którym optymalizator podjął błędną decyzję, będzie to podstawą do poprawienia kodu programu i (czasami) do nagrodzenia użytkownika.

Wróćmy do porównania planu wykonania zapytania z operatorem `SARG` z planem wykonania zapytania zwracającego te same dane, ale niewykorzystującego operatora `SARG`:

```
USE Northwind
SELECT *
FROM dbo.Orders
WHERE CustomerID LIKE '%ROSR'
GO
StmtText
------------------------------------------------------------------------------------------
  |--Clustered Index Scan(OBJECT:([Northwind].[dbo].[Orders].[PK_Orders]),
  ↳WHERE:(like([Orders].[CustomerID], '%ROSR')))

(1 row(s) affected)
Table 'Orders'. Scan count 1, logical reads 21, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 2 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
```

Jeżeli czas wykonania zapytania jest tak krótki, że Query Optimizer szacuje go na 0 milisekund, informacja ta nie jest zamieszczana w książce.

Ponieważ na podstawie warunku `CustomerID LIKE '%ROSR'` niemożliwe jest oszacowanie liczby spełniających go stron (argument nie jest argumentem `SARG`), SQL Server zdecydował się na „pewniejszą" opcję i wybrał odczytanie indeksu grupującego.

Zapytania z operatorem AND

Zapytania wykorzystujące koniunkcję kilku warunków logicznych przetwarzane są następująco:

1. W pierwszej kolejności Query Optimizer zwraca wszystkie wiersze spełniające poszczególne kryteria wymienione w klauzuli `WHERE`;
2. Z otrzymanego zbioru kolejno usuwane są wiersze niespełniające kolejnych warunków.

Podczas przetwarzania zapytań tego typu Query Optimizer:

1. Wykorzysta wszelkie dostępne indeksy zawierające dane wymienione w klauzuli `WHERE`. W przypadku ich braku przeszukana zostanie cała tabela zwierająca odpowiednie dane.
2. Może wykorzystać różne indeksy, o ile każdy z nich zawiera fragment danych wymienionych w klauzuli `WHERE`.

Największy wzrost wydajności dla zapytań tego typu uzyskamy, tworząc co najmniej jeden indeks zawierający dane o silnie zróżnicowanych (najlepiej unikalnych) wartościach (a więc np. wielkość zamówienia, a nie identyfikatory dostawców, których w przykładowej bazie jest kilkunastu, a każdy z nich składa wiele zamówień różnej wielkości), do których odwołują się warunki z klauzuli `WHERE`.

Przykład:

Utworzymy kopię tabeli `Order Details` i wybierzemy z niej dane spełniające koniunkcje dwóch warunków logicznych. Następnie utworzymy dla naszej tabeli indeks zawierający dane z obu kolumn wymienionych w klauzuli `WHERE` i ponownie wykonamy zapytanie.

```
USE Northwind
SELECT *
INTO dbo.od
FROM dbo.[Order Details]
GO
SET STATISTICS TIME ON
SET STATISTICS IO ON
GO
SELECT *
FROM dbo.od
WHERE UnitPrice >100 AND Quantity<5
GO
Table 'od'. Scan count 1, logical reads 10, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 1 ms.

CREATE INDEX i_od
ON dbo.od (UnitPrice, Quantity)
```

```
GO
SELECT *
FROM dbo.od
WHERE UnitPrice >100 AND Quantity<5
GO
Table 'od'. Scan count 1, logical reads 5, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
```

Pomimo tego, że nie utworzyliśmy indeksu zawierającego zapytanie, liczba odczytanych stron danych zmniejszyła się o połowę, chociaż tabela zawierająca najwięcej rekordów ze wszystkich tabel bazy `Northwind`, liczy jedynie nieco ponad 2000 rekordów.

Zapytania z operatorem OR

Zapytania wykorzystujące alternatywy kilku warunków logicznych przetwarzane są następująco:

1. W pierwszej kolejności Query Optimizer zwraca wszystkie wiersze spełniające poszczególne kryteria wymienione w klauzuli `WHERE`.
2. Do otrzymanego zbioru kolejno dodawane są wiersze niespełniające wcześniejszych warunków.

Podczas przetwarzania zapytań tego typu Query Optimizer:

1. Wykorzysta wszelkie dostępne indeksy zawierające wszystkie lub część danych wymienionych w klauzuli `WHERE`. W przypadku braku niegrupującego indeksu powiązanego z choćby jednym warunkiem logicznym przeszukana zostanie cała tabela zawierająca odpowiednie dane;
2. Może wykorzystać różne indeksy, o ile każdy z nich zawiera fragment danych wymienionych w klauzuli `WHERE`.

Operator `IN` na etapie standaryzacji jest przekształcany na odpowiadające mu wyrażenie z operatorami `OR` — zapytania z tym operatorem są przetwarzane w ten sam sposób co zapytania z operatorem `OR`.

W wypadku zapytań z operatorem `OR` istnienie indeksów zawierających wszystkie dane wymienione w klauzuli `WHERE` zapobiegnie (z wyjątkiem sytuacji, w której tabela przechowująca dane jest na tyle mała, że taniej jest odczytać wszystkie strony danych przechowujące dane z tabeli niż wybierać na podstawie odczytanych indeksów) przeszukaniu całej tabeli oraz zmniejszy liczbę operacji arytmetycznych związanych z sortowaniem i porównywaniem danych.

Przykład:

```
USE Northwind
DROP TABLE dbo.od --jeżeli przykładowa tabeli istnieje, usuwamy ją
SELECT *
INTO dbo.od
FROM dbo.[Order Details]
CREATE INDEX i_odup
```

```
ON dbo.od (UnitPrice)
GO
SET STATISTICS TIME ON
SET STATISTICS IO ON
GO
SELECT *
FROM dbo.od
WHERE UnitPrice >100 OR Quantity<5
GO
Table 'od'. Scan count 1, logical reads 10, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 2 ms,  elapsed time = 2 ms.
```

Wyświetlmy jeszcze plan wykonania instrukcji:

```
StmtText
-----------------------------------------------------------------------------------
  |--Table Scan(OBJECT:([Northwind].[dbo].[od]), WHERE:([od].[UnitPrice]>100.00 OR
  ↳[od].[Quantity]<5))
```

Jak widać, optymalizator przeszukał całą tabelę pomimo tego, że istniał indeks powiązany z danymi wykorzystywanymi w jednym z warunków logicznych. Dodajmy teraz indeks powiązany z drugą tabelą przechowującą dane. na podstawie których wybierany jest wynik zapytania i ponownie wyświetlmy statystyki i plan wykonania instrukcji:

```
CREATE INDEX i_odq
ON dbo.od (Quantity)
GO
SELECT *
FROM dbo.od
WHERE UnitPrice >100 OR Quantity<5
GO
Table 'od'. Scan count 1, logical reads 10, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 3 ms.
```

Liczba odczytanych stron danych się nie zmieniła, co sugeruje, że i tym razem przeszukana została cała tabela. Czas wykonania instrukcji wyraźnie się skrócił, w przeciwieństwie do czasu jej przetworzenia, co sugeruje, że optymalizator tym razem był w stanie opracować bardziej wydajny, choć nieco bardziej skomplikowany, plan jej wykonania:

```
StmtText
-----------------------------------------------------------------------------------
  |--Table Scan(OBJECT:([Northwind].[dbo].[od]), WHERE:([od].[UnitPrice]>100.00 OR
  ↳[od].[Quantity]<5))
```

Ponieważ tabela od jest niewielka (mieści się na 10 stronach pamięci) i tym razem optymalizator zdecydował się na przeszukanie wszystkich jej wierszy. Mimo to czas wykonania zapytania uległ skróceniu.

Zapytania pobierające dane z kilku źródeł danych

Jeżeli zapytanie odwołuje się do kilku tabel lub widoków, pierwszą wartością, którą oszacuje optymalizator jest liczba wierszy zwróconych w wyniku złączenia obiektów źródłowych. Liczba ta zależy odwrotnie proporcjonalnie od unikalności wartości danych

wykorzystanych do złączenia obiektów (łącząc tabele za pomocą kolumn przechowujących dane np. o województwie otrzymamy (statystycznie) większy zbiór wynikowy niż łącząc tabele za pomocą kolumn przechowujących dane o adresie konkretnego dostawcy).

Ponadto na podstawie szacunkowej **ziarnistości** danych optymalizator ocenia, ile danych zostanie zduplikowanych w wyniku złączenia i na tej podstawie wybiera typ indeksu wykorzystanego do złączenia:

- W przypadku danych o małej ziarnistości wykorzystany będzie w pierwszej kolejności indeks niegrupujący, w drugiej — grupujący.
- W przypadku danych o dużej ziarnistości wykorzystany będzie wyłącznie indeks grupujący.

Złączenia pośrednie

Wynik złączenia dowolnej liczby obiektów obliczany jest jako suma złączeń par obiektów. Każde **złączenie pośrednie** może zostać wykonane za pomocą operacji innego typu, wybranego przez optymalizator dla dwóch łączonych obiektów. Dla każdego złączenia pośredniego niezależnie wyznaczana jest tabela zewnętrzna i wewnętrzna złączenia.

Kolejność wykonywania złączeń pośrednich wyznaczana jest przez Query Optimizera i nie musi odpowiadać kolejności wymienionej w instrukcji SELECT.

Klauzula WHERE

Jeżeli zapytanie zawiera klauzulę WHERE, optymalizator może zadecydować o wybraniu wierszy spełniających podane kryteria, zanim wykona operację złączenia. W ten sposób wielokrotnie zmniejsza się liczba wierszy, które będą łączone.

Przykład:

Pierwsze zapytanie zwraca wszystkie rekordy łączonych tabel, drugie — jedynie wybrane rekordy z obu tabel. Pokazano statystyki związane z wykonaniem zapytań i plan ich wykonania:

```
USE Northwind
SELECT LastName, OrderID
FROM dbo.Orders o JOIN  dbo.Employees e
  ON e.EmployeeID = o.EmployeeID
GO
Table 'Orders'. Scan count 9, logical reads 19, physical reads 0, read-ahead reads 0.
Table 'Employees'. Scan count 1, logical reads 1, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 34 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
StmtText
------------------------------------------------------------------------------------
  |--Nested Loops(Inner Join, OUTER REFERENCES:([e].[EmployeeID]))
       |--Index Scan(OBJECT:([Northwind].[dbo].[Employees].[LastName] AS [e]))
       |--Index Seek(OBJECT:([Northwind].[dbo].[Orders].[EmployeeID] AS [o]),
       ↳SEEK:([o].[EmployeeID]=[e].[EmployeeID]) ORDERED FORWARD)
```

```
(3 row(s) affected)

USE Northwind
SELECT LastName, OrderID
FROM dbo.Orders o JOIN  dbo.Employees e
  ON e.EmployeeID = o.EmployeeID
WHERE e.FirstName LIKE 'A%' AND Freight<10
GO
Table 'Employees'. Scan count 176, logical reads 352, physical reads 0, read-ahead reads 0.
Table 'Orders'. Scan count 1, logical reads 21, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 2 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
StmtText
------------------------------------------------------------------------------------
  |--Nested Loops(Inner Join, OUTER REFERENCES:([o].[EmployeeID]))
       |--Clustered Index Scan(OBJECT:([Northwind].[dbo].[Orders].[PK_Orders] AS [o]),
       ↳WHERE:([o].[Freight]<10.00))
       |--Clustered Index Seek(OBJECT:([Northwind].[dbo].[Employees].[PK_Employees] AS
       ↳[e]), SEEK:([e].[EmployeeID]=[o].[EmployeeID]),  WHERE:(like([e].[FirstName],
       ↳'A%')) ORDERED FORWARD)
(3 row(s) affected)
```

W drugim przypadku obie tabele najpierw zostały przeszukane pod kątem zgodności z kryteriami podanymi w klauzuli `WHERE` (stąd duża liczba przeszukań tabel), ale w rezultacie uzyskano wielokrotnie mniejszy zbiór wierszy, które zostały złączone.

Pętla wyszukiwania

Złączenie przeprowadzone za pomocą operatora pętli wyszukiwania (ang. *Nested Loop*) jest tym sposobem łączenia tabel, o którym myślimy w pierwszej kolejności, szczególnie jeżeli piszemy programy w językach takich jak C czy Basic. Złączenie tego typu polega na porównaniu przez SQL Server każdego wiersza w wewnętrznej tabeli złączenia (decyzja o tym, która tabela zostanie uznana za wewnętrzną jest podejmowana przez usługę Query Optimizer na podstawie liczby wierszy, ich unikalności i ziarnistości) z kolejnymi wierszami zewnętrznej tabeli złączenia. Najgorszym z możliwych scenariuszów jest ten, w którym nie istnieją ani indeksy powiązane z kolumnami, według których następuje złączenie, ani indeksy związane z kolumnami wymienionymi w klauzuli `WHERE` — w takim przypadku zbiór danych źródłowych jest równy całym tabelom. Nie wnikając w opisaną w dalszej części książki strukturę fizycznych plików bazy danych, spróbujemy obliczyć liczbę stron danych, które muszą zostać odczytane w przypadku, gdy tabela *t1*, składająca się z *w1* wierszy zapisanych na *p1* stron, zostanie złączona z tabelą *t2* zawierającą *w2* wierszy zapisanych na *p2* stronach — SQL Server musi odczytać $p1 + w1 * s2$ stron danych. Oznacza to, że łącząc niewielkie tabele zawierające odpowiednio: tabela *t2* (wewnętrzna) — 100 stron, tabela *t1* (zewnętrzna) — 5 000 wierszy zapisanych na 200 stronach, SQL Server musi odczytać ponad 500 000 stron, czyli ponad 4 MB danych.

Rozwiązaniem jest utworzenie indeksu grupującego na kolumnie wykorzystanej do łączenia tabel. Ponieważ dane w tabeli są uporządkowane według wartości indeksu grupującego, nie ma potrzeby przeszukiwania całej tabeli. W przypadku wewnętrznej tabeli złączenia SQL Server może odczytać jedynie stronę zawierającą dokładnie ten wiersz,

który odpowiada kolejnemu wierszowi tabeli zewnętrznej. W ten sposób zastępujemy w naszym wzorze wartość *s2* stałą wyliczoną na podstawie ilości poziomów indeksu (będzie to liczba z zakresu od 1 do 3). Oznacza to, że złączenie naszych przykładowych tabel wymaga już jedynie (w najgorszym razie) odczytania 15 200 stron, czyli około 120 KB danych — ponad 30-krotnie mniej.

Dodatkowo, jeżeli na zewnętrzną tabelę złączenia nałożono warunek `WHERE`, możemy ograniczyć liczbę wierszy odczytanych z tej tabeli, tworząc niegrupujący indeks dla kolumny, według której wybierane są dane.

Query Optimizer tworzy co najmniej cztery plany wykonania złączenia uwzględniające różną rolę tabel w złączeniu oraz kolejność wybierania danych i realizuje plan najtańszy.

Złączenie za pomocą operatora pętli wyszukiwania wybierane jest, jeżeli zewnętrzna tabela złączenia zawiera dużą ilość wierszy, a wewnętrzna jest mała lub zawiera użyteczne indeksy.

Łączenie

Ograniczeniem złączenia przez pętle wyszukiwania jest niewykorzystywanie ewentualnie istniejących indeksów dla kolumny łączącej tabeli zewnętrznej. Niestety podczas projektowania i strojenia bazy danych nie mamy pewności, która z tabel zostanie wybrana przez optymalizator jako tabela zewnętrzna złączenia.

Jeżeli utworzymy indeks grupujący dla kolumny łączącej w obu złączonych tabelach (czyli obie kolumny będą posortowane według tej samej kolumny) SQL Server będzie mógł złączyć tabele poprzez łączenie (ang. *Merge*). Złączenie tego typu polega na odczytaniu i porównaniu kolejnych wierszy obu tabel.

Złączenie poprzez łączenie wybierane jest wtedy, gdy obie tabele są posortowane według kolumny złączenia.

Łączenie tabel powiązanych związkiem typu jeden do wielu i jeden do jednego przebiega dokładnie według opisanego schematu. Natomiast łączenie tabel powiązanych związkiem typu wiele do wielu wymaga utworzenia tymczasowej tabeli do przechowywania kolejnych wierszy, zanim zostaną sprawdzone pozostałe wiersze drugiej tabeli i zapadnie decyzja o włączeniu lub odrzuceniu danego wiersza z wyniku złączenia. Ponieważ wielokrotnie zwiększa to liczbę operacji wejścia-wyjścia, optymalizator z reguły nie decyduje się na złączenie poprzez łączenie, jeżeli przynajmniej jedna tabela wykorzystywana do złączenia nie zawiera wartości niepowtarzalnych.

Przykład:

Tworzymy kopię tabel `Orders` i `Orders Detail`, następnie indeks grupujący dla kolumn `OrderID` i wykonujemy proste zapytanie łączące obie utworzone tabele:

```
USE Northwind
SELECT *
INTO dbo.o
FROM dbo.Orders
SELECT *
INTO dbo.od
FROM dbo.[Order Details]
```

```
CREATE CLUSTERED INDEX i_o
ON o(OrderID)
CREATE CLUSTERED INDEX i_od
ON od(OrderID)
GO
SET STATISTICS TIME ON
SET STATISTICS IO ON
GO
SELECT *
FROM o JOIN od
ON o.OrderID = od.OrderID
GO
Table 'od'. Scan count 830, logical reads 1672, physical reads 0, read-ahead reads 0.
Table 'o'. Scan count 1, logical reads 22, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 10 ms,  elapsed time = 98 ms.
```

Liczba odczytanych stron pozwala przypuszczać, że tabele zostały złączone za pomocą pętli wyszukiwania. Wyświetlając plan wykonania przekonamy się, że nasze obawy są słuszne — ponieważ żaden z indeksów grupujących nie został zdefiniowany jako unikalny, optymalizator, który nie analizuje danych zapisanych w obu tabelach pod kątem ich niepowtarzalności, zdecydował się na takie złączenie tabel.

```
StmtText
-----------------------------------------------------------------------------------------
  |--Nested Loops(Inner Join, OUTER REFERENCES:([o].[OrderID]))
       |--Clustered Index Scan(OBJECT:([Northwind].[dbo].[o].[i_o]))
       |--Clustered Index Seek(OBJECT:([Northwind].[dbo].[od].[i_od]),
       ↳SEEK:([od].[OrderID]=[o].[OrderID]) ORDERED FORWARD)
(3 row(s) affected)
```

Zmieńmy definicję jednego indeksu i ponownie wykonajmy zapytanie, porównując koszty jego wykonania:

```
CREATE UNIQUE CLUSTERED INDEX i_o
ON o(OrderID)
WITH DROP_EXISTING
GO
SELECT *
FROM o JOIN od
ON o.OrderID = od.OrderID
GO
Table 'od'. Scan count 1, logical reads 11, physical reads 0, read-ahead reads 0.
Table 'o'. Scan count 1, logical reads 21, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 10 ms,  elapsed time = 236 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
StmtText
-----------------------------------------------------------------------------------------
  |--Merge Join(Inner Join, MERGE:([o].[OrderID])=([od].[OrderID]),
  ↳RESIDUAL:([od].[OrderID]=[o].[OrderID]))
       |--Clustered Index Scan(OBJECT:([Northwind].[dbo].[o].[i_o]), ORDERED FORWARD)
       |--Clustered Index Scan(OBJECT:([Northwind].[dbo].[od].[i_od]), ORDERED FORWARD)
(3 row(s) affected)
```

Liczba odczytanych stron danych zmniejszyła się z 1694 do 32 (prawie 53 razy), a liczba wykonanych operacji przeszukiwania indeksu z 830 do 1! Pomimo tak spektakularnego zmniejszenia liczby operacji wejścia-wyjścia czas wykonania instrukcji uległ wydłużeniu, co związane jest z tym, że wszystkie potrzebne dane były zapisane w pamięci podręcznej, a same tabele były raczej niewielkie.

Złączenie przez łączenie jest na tyle wydajnym sposobem złączania tabel, że optymalizator może zdecydować się na to rozwiązanie, nawet jeżeli jedna z łączonych tabel nie jest posortowana według odpowiedniej kolumny. W takim przypadku najpierw następuje posortowanie tabeli, a następnie jej złączenie.

Zgodność funkcji skrótu

Złączenie na podstawie zgodności funkcji skrótu (ang. *Hash*) przeprowadzane jest, jeżeli optymalizator nie może znaleźć użytecznych dla złączenia indeksów. Brak indeksów oznacza między innymi, że dane zawarte w tabelach źródłowych nie są posortowane. O ile w przypadku niewielkich danych ich posortowanie i złączenie przez łączenie może okazać się najtańszym rozwiązaniem, o tyle dla tabel zawierających miliony wierszy próba ich posortowania byłaby operacją wyjątkowo kosztowną. Również sekwencyjne porównywanie wszystkich wierszy jednej tabeli z kolejnymi wierszami drugiej tabeli okazuje się zbyt kosztownym rozwiązaniem. Pozostaje podzielenie danych na grupy, co jest operacją szybszą niż pełne ich posortowanie, a jednocześnie pozwalającą ograniczyć liczbę operacji wejścia-wyjścia.

Dane zawarte w tabelach źródłowych dzielone są na grupy według wartości obliczonych dla kolejnych wierszy funkcji skrótu. Każda grupa zawiera dane, dla których wyliczona wartość była taka sama, więc jeżeli nastąpi porównanie na podstawie wartości funkcji skrótu, wystarczy, że SQL Server sprawdzi zgodność z wierszami z wybranej grupy.

Na przykład, jeżeli funkcją skrótu byłaby funkcja `modulo 13`, dane liczbowe zostałyby podzielone na 13 grup (możliwe wartości funkcji `modulo 13` należą do zbioru <0, 12>). W rzeczywistości funkcje skrótu stosowane przez SQL Server są bardziej skomplikowane, a liczba grup może sięgać kilku tysięcy. Dzieląc dane źródłowe pomiędzy grupy i porównując, wiersz po wierszu, dane z jednego zbioru wejściowego z odpowiadającymi im danymi ze zbioru danych drugiej tabeli, SQL Server może złączyć obie tabele, wykonując tylko jedną operację przeszukania tabeli (jak w przypadku złączenia przez łączenie).

Przykład:

Wykonamy kopie tabel `Orders` i `Orders Detail` i połączymy nowo utworzone, pozbawione jakichkolwiek indeksów tabele:

```
USE Northwind
DROP TABLE o --jeżeli tabele istniały
DROP TABLE od --należy je usunąć
SELECT *
INTO dbo.o
FROM dbo.Orders
SELECT *
INTO dbo.od
FROM dbo.[Order Details]
GO
```

```
SET STATISTICS TIME ON
SET STATISTICS IO ON
GO
SELECT *
FROM o JOIN od
ON o.OrderID = od.OrderID
GO
Table 'od'. Scan count 1, logical reads 10, physical reads 0, read-ahead reads 0.
Table 'o'. Scan count 1, logical reads 21, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 20 ms,  elapsed time = 244 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
StmtText
------------------------------------------------------------------
  |--Hash Match(Inner Join, HASH:([o].[OrderID])=([od].[OrderID]))
       |--Table Scan(OBJECT:([Northwind].[dbo].[o]))
       |--Table Scan(OBJECT:([Northwind].[dbo].[od]))
(3 row(s) affected)
```

Liczba odczytanych stron jest równie mała jak dla złączenia przez łączenie, ale czas wykonania operacji uległ wydłużeniu.

W dokumentacji technicznej SQL Servera możemy przeczytać, że złączenie na podstawie zgodności funkcji skrótu powinno być stosowane jedynie dla zapytań ad hoc i świadczy o konieczności optymalizacji (strojenia) bazy danych.

Wybór najlepszego indeksu

Optymalizacja zapytań związana jest bezpośrednio z optymalizacją bazy danych i nie ma ogólnego schematu pozwalającego na osiągnięcie maksymalnej wydajności konkretnego zapytania. Jedynym rozwiązaniem jest testowanie różnych rozwiązań i porównywanie wyników. Właściwie wyglądający proces testowania i optymalizacji prowadzący do wyboru najlepszego indeksu dla zapytania wybierającego dane z dwóch tabel opisano poniżej.

W poniższych przykładach dodatkowo została włączona opcja wyświetlania graficznego planu wykonania poszczególnych zapytań, co wpłynęło na wydłużony czas ich wykonania i zwiększenie liczby operacji wejścia-wyjścia.

Należy utworzyć linię bazową zawierającą dane, z którymi będziemy porównywali wydajność zapytania korzystającego z różnych indeksów. Kolejno przedstawiono instrukcje (częściowo wypełnione losowymi danymi) tworzące kopię dwóch tabel bazy `Northwind` oraz zapytanie bazowe.

Tworzymy kopię tabel pozbawionych jakichkolwiek indeksów:

```
USE Northwind
DROP TABLE o -- jeżeli tabele istniały
DROP TABLE od -- należy je usunąć
SELECT *
INTO dbo.o
FROM dbo.Orders
SELECT *
INTO dbo.od
FROM dbo.[Order Details]
GO
```

Tworzymy zapytanie bazowe pobierające dane z tabel o i od:

```
SELECT o.OrderID, CustomerID, OrderDate, ProductID, UnitPrice, Discount
FROM o JOIN od
  ON o.OrderID = od.OrderID
WHERE CustomerID IN ('SUPRD', 'HANAR', 'RICSU')
AND UnitPrice >30
GO
Table 'od'. Scan count 1, logical reads 10, physical reads 0, read-ahead reads 0.
Table 'o'. Scan count 1, logical reads 21, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 10 ms,  elapsed time = 22 ms.
```

Statystyki wykonania zapytania bazowego, przede wszystkim liczba milisekund, którą SQL Server potrzebował na jego wykonanie (10 ms), są wartościami bazowymi, które należy poprawić. Niewielka liczba odczytanych stron danych wynika z tego, że SQL Server połączył obie tabele na podstawie zgodności funkcji skrótu.

W opisywanym przykładzie wszystkie argumenty wymienione w klauzuli WHERE to argumenty SARG. Są minimalnym zbiorem argumentów niezbędnym do przeprowadzenia zaplanowanej selekcji, więc pozostaje nam utworzyć brakujące indeksy.

W pierwszej kolejności dodamy indeksy do kolumn łączących tabelę. W przypadku tabeli o będzie to indeks unikalny, ponieważ tworzymy go dla kolumny będącej kluczem głównym tabeli:

```
CREATE UNIQUE INDEX o_zl ON o(OrderID)
CREATE  INDEX od_zl ON od(OrderID)
GO
The command(s) completed successfully.
```

Ponowne wykonanie zapytania bazowego zwróci następujące informacje:

```
Table 'od'. Scan count 50, logical reads 235, physical reads 0, read-ahead reads 0.
Table 'o'. Scan count 1, logical reads 21, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 21 ms.
```

Czas wykonania zapytania uległ wyraźnemu skróceniu, ale liczba odczytanych stron danych jest wielokrotnie większa. Sprawdźmy, jaki wpływ na wydajność zapytania będzie miało zastąpienie utworzonych indeksów indeksami utworzonymi dla kolumn wymienionych w klauzuli WHERE:

```
DROP INDEX o.o_zl
DROP INDEX od.od_zl
CREATE INDEX o_sz ON o(CustomerID)
CREATE INDEX od_sz ON od(UnitPrice)
GO
```

Po wykonaniu zapytania okazuje się, że poindeksowanie kolumn, według których wyszukiwane są dane, zamiast kolumn, według których łączone są tabele, nie przyniosło żadnego wzrostu wydajności:

```
Table 'od'. Scan count 1, logical reads 10, physical reads 0, read-ahead reads 0.
Table 'o'. Scan count 1, logical reads 21, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 10 ms,  elapsed time = 20 ms.
```

Kolejna próba będzie polegała na utworzeniu indeksów wspólnych dla obu kolumn (tej wykorzystywanej do złączenia i tej wykorzystywanej do wyszukiwania danych) i na ponownym wykonaniu zapytania bazowego:

```
DROP INDEX o.o_sz
DROP INDEX od.od_sz
CREATE UNIQUE INDEX o_ws ON o(OrderID,CustomerID)
CREATE INDEX od_ws ON od(OrderID,UnitPrice)
GO
Table 'od'. Scan count 50, logical reads 209, physical reads 0, read-ahead reads 0.
Table 'o'. Scan count 1, logical reads 21, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 10 ms,  elapsed time = 17 ms.
```

Jak widać, utworzenie wspólnego indeksu dla obu kolumn nie wpłynęło w znaczący sposób na wydajność zapytania w porównaniu z indeksem utworzonym dla kolumny, według której łączone są tabele. Czytelnicy mogą samodzielnie przekonać się, że zmiana kolejności kolumn w definicji indeksu wspólnego również nie przynosi żadnych korzyści. Ponieważ to zapytanie wybiera wszystkie dane z łączonych tabel, nie możemy utworzyć indeksu zawierającego zapytanie, który z pewnością gwarantowałby jego najszybsze wykonanie. Zamiast tego spróbujemy utworzyć indeksy grupujące dla kolumn złączenia i niegrupujące dla kolumn przechowujących dane, według których wyszukiwane są wiersze:

```
DROP INDEX o.o_ws
DROP INDEX od.od_ws
CREATE UNIQUE CLUSTERED INDEX o_zl ON o(OrderID)
CREATE CLUSTERED INDEX od_zl ON od(OrderID)
CREATE INDEX o_sz ON o(CustomerID)
CREATE INDEX od_sz ON od(UnitPrice)
GO
Table 'od'. Scan count 50, logical reads 101, physical reads 0, read-ahead reads 0.
Table 'o'. Scan count 1, logical reads 21, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 16 ms.
```

Ostatnie rozwiązanie okazuje się najlepsze. Chociaż liczba odczytanych stron świadczy wyraźnie o tym, że złączenie zostało dokonane poprzez pętle wyszukiwania, to początkowy wybór łączonych danych i porównywanie posortowanych danych zaowocowało krótkim czasem wykonania zapytania przy stosunkowo niewielkiej liczbie odczytanych stron danych.

Zapytania grupujące dane

Zapytania zawierające klauzulę `GROUP BY` lub `DISTINCT` wykonywane są według tego samego planu — w obu przypadkach SQL Server zwraca jeden rekord reprezentujący wiersze przechowujące te same wartości. Różnica pomiędzy klauzulami sprowadza się do tego, że w przypadku klauzuli `GROUP BY` możliwe jest użycie funkcji grupującej, zwracającej jedną wartość na podstawie dowolnej liczby przekazanych argumentów. Wspólne cechy obu klauzul obrazują poniższe zapytania, które nie tylko zwracają te same dane, ale także są wykonane według tego samego planu:

```
USE Northwind
SELECT CustomerID
FROM dbo.Orders
GROUP BY CustomerID
```

```
GO
StmtText
------------------------------------------------------------------------------------------
  |--Stream Aggregate(GROUP BY:([Orders].[CustomerID]))
       |--Index Scan(OBJECT:([Northwind].[dbo].[Orders].[CustomerID]), ORDERED FORWARD)
(2 row(s) affected)
SELECT DISTINCT CustomerID
FROM dbo.Orders
GO
StmtText
------------------------------------------------------------------------------------------
  |--Stream Aggregate(GROUP BY:([Orders].[CustomerID]))
       |--Index Scan(OBJECT:([Northwind].[dbo].[Orders].[CustomerID]), ORDERED FORWARD)
(2 row(s) affected)
```

Aby uwidocznić różnice pomiędzy klauzulami, wykonajmy zapytanie, które oprócz identyfikatorów klientów zwraca datę ostatniego zamówienia:

```
USE Northwind
SELECT CustomerID, MAX (OrderDate)
FROM dbo.Orders
GROUP BY CustomerID
GO
StmtText
------------------------------------------------------------------------------------------
  |--Hash Match(Aggregate, HASH:([Orders].[CustomerID]),
  ↳RESIDUAL:([Orders].[CustomerID]=[Orders].[CustomerID])
  ↳DEFINE:([Expr1002]=MAX([Orders].[OrderDate])))
       |--Clustered Index Scan(OBJECT:([Northwind].[dbo].[Orders].[PK_Orders]))
(2 row(s) affected)
```

Plan wykonania tego zapytania różni się od poprzednich — w tym wypadku SQL Server przeprowadzi **grupowanie dla funkcji skrótu** (ang. *Hash Aggregation*), które (w przeciwieństwie do odczytania indeksu) nie spowoduje uporządkowania zwracanych danych.

Inną techniką wykorzystywana do wykonania zapytania zawierającego funkcję grupującą jest **grupowanie dla danych** (ang. *Stream Aggregation*) — w tym przypadku najpierw nastąpi posortowanie danych, następnie usunięcie duplikatów i obliczenie wartości funkcji grupującej.

Do wersji 7.0 SQL Server grupował dane i obliczał wartości funkcji grupujących wyłącznie poprzez grupowanie dla danych. W rezultacie wynik zawsze zawierał dane uporządkowane. W wersji 7.0 i następnych dla osiągnięcia takiego samego rezultatu konieczne jest dodanie klauzuli `ORDER BY`.

Analizując plan wykonania dwóch pierwszych zapytań zauważymy, że zostały one wykonane poprzez odczytanie indeksu zawierającego żądane dane. Dodatkowo wpis `ORDERED FORWARD` świadczy o tym, że zostały sekwencyjnie odczytane uporządkowane liście indeksu. W rezultacie SQL Server nie potrzebował ponownie sortować danych.

Wykonajmy zapytanie zwracające liczbę pracowników związanych z realizowaniem zamówień poszczególnych klientów. W tym przypadku w bazie `Northwind` nie ma indeksu zawierającego wynik zapytania:

```
USE Northwind
SELECT CustomerID, count (EmployeeID)
FROM dbo.Orders
GROUP BY CustomerID
GO
StmtText
-----------------------------------------------------------------------------------------
  |--Compute Scalar(DEFINE:([Expr1002]=Convert([Expr1006])))
       |--Hash Match(Aggregate, HASH:([Orders].[CustomerID]),
       ↳RESIDUAL:([Orders].[CustomerID]=[Orders].[CustomerID])
       ↳DEFINE:([Expr1006]=COUNT_BIG([Orders].[EmployeeID])))
            |--Clustered Index Scan(OBJECT:([Northwind].[dbo].[Orders].[PK_Orders]))
(3 row(s) affected)
Table 'Orders'. Scan count 1, logical reads 21, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 7 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
```

Porównajmy koszt i plan wykonania tego zapytania po dodaniu brakującego indeksu:

```
CREATE INDEX orders_c_e
ON Orders (CustomerID, EmployeeID)
GO
SELECT CustomerID, count (EmployeeID)
FROM dbo.Orders
GROUP BY CustomerID
GO
StmtText
-----------------------------------------------------------------------------------------
  |--Compute Scalar(DEFINE:([Expr1002]=Convert([Expr1006])))
       |--Stream Aggregate(GROUP BY:([Orders].[CustomerID])
       ↳DEFINE:([Expr1006]=COUNT_BIG([Orders].[EmployeeID])))
            |--Index Scan(OBJECT:([Northwind].[dbo].[Orders].[orders_c_e]), ORDERED
            ↳FORWARD)
(3 row(s) affected)
Table 'Orders'. Scan count 1, logical reads 4, physical reads 0, read-ahead reads 0.
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
```

Po dodaniu niegrupującego indeksu zawierającego zapytanie koszt jego wykonania zmniejszył się wielokrotnie. Ponieważ zapytania zawierające funkcje grupujące rzadko pobierają dane z wielu kolumn, najlepszym rozwiązaniem jest utworzenie niegrupującego indeksu zawierającego takie zapytanie albo indeksu grupującego, który sortuje dane według wartości kolumny wymienionej w klauzuli `GROUP BY` lub po słowie kluczowym `DISTINCT`.