STUDIA INFORMATICA 2012
Volume 33 Number 2A (105)

Aleksandra WERNER, Małgorzata BACH
Politechnika Śląska, Instytut Informatyki

OCENA EFEKTYWNOŚCI STOSOWANIA INDEKSÓW KOLUMNOWYCH W BAZACH DANYCH[1]

Streszczenie. Jedną z odpowiedzi na wzrastający popyt na efektywne przetwarzanie danych potrzebnych do wieloaspektowych analiz jest koncepcja indeksów kolumnowych wprowadzona w Microsoft SQL Server 2012. W celu zweryfikowania tezy głoszącej przewagę techniki kolumnowego przetwarzania danych nad klasyczną – wierszową – metodą wykonano wiele badań, których wyniki zestawiono w niniejszym artykule. Porównano efektywność stosowania nowo wprowadzonych indeksów kolumnowych w stosunku do mechanizmów optymalizacji istniejących już we wcześniejszych wersjach MS SQL Server, takich jak: klasyczne indeksy zgrupowane i niezgrupowane, filtrowanie bitmapowe, indeksowane perspektywy czy indeksy filtrowane.

Słowa kluczowe: SQL Server 2012, indeksy kolumnowe, filtrowanie bitmapowe, indeksowane perspektywy, indeksy filtrowane

EVALUATION OF THE COLUMNSTORE INDEXES EFFICIENCY

Summary. The idea of column-based indexes, introduced in Microsoft SQL Server 2012, is one of the solutions, improving the efficiency of the analytical data processing. In order to verify the opinion, the column-oriented data processing gives more efficient results than classic, row-oriented, data processing method, a series of tests were executed and the results was summarized in this article. Besides, the efficiency of columnstore index was compared with other optimization mechanisms that exist in earlier versions of MS SQL Server – such as: classical clustered and unclustered indexes, bitmap filtering, indexed views and filtered indexes.

Keywords: SQL Server 2012, columnstore index, bitmap filtering, indexed views, filtered index

[1] Praca naukowa częściowo finansowana ze środków na naukę w latach 2010-2012 jako projekt rozwojowy O R00 0113 12.

1. Wprowadzenie

Na prawidłowe funkcjonowanie przedsiębiorstwa ma wpływ wiele czynników, z których na plan pierwszy z pewnością wysuwa się kwestia podejmowania szybkich, trafnych, a często i strategicznych decyzji. Brak albo nadmiar właściwych informacji, a także sposób ich selekcji w dużej mierze decydują o powodzeniu lub porażce realizowanych przedsięwzięć. Złożoność zapytań analitycznych i olbrzymie rozmiary danych przez nie adresowanych sprawiają, że ekstrakcja danych gromadzonych z codziennych procesów funkcjonowania firm staje się zadaniem czasochłonnym i wymagającym dużej ilości zasobów obliczeniowych i pamięciowych. W celu rozwiązania problemu małej efektywności przetwarzania analitycznego podczas wykonywania takich zapytań producenci systemów baz danych wyposażają swoje rozwiązania w coraz nowsze mechanizmy, umożliwiające optymalizację dostępu do przetwarzanych danych. Wśród wielu propozycji obiecujący wydaje się pomysł składowania danych w sposób kolumnowy, sięgający swoimi korzeniami początku lat 70. Choć nie spotkał się on wtedy z dużym zainteresowaniem, a Sysbase IQ przez długi czas był jedynym komercyjnym produktem tego typu, w ostatnich latach, wraz ze wzrostem znaczenia przetwarzania analitycznego, można obserwować prawdziwy renesans wspomnianej architektury. Jest to również widoczne w zwiększonych obecnie w systemach baz danych możliwościach definiowania struktur dostępowych – tj. indeksów dedykowanych specjalnie zaawansowanemu przetwarzaniu analitycznemu. Uwzględniając fakt, że elementarną jednostką przechowywania, ale i odczytywania danych jest strona, istnienie specjalnej struktury lokującej dane dla każdej indeksowanej kolumny na oddzielnym zestawie stron pamięci może w istotny sposób zwiększać efektywność przetwarzania[2]. Kluczowym spostrzeżeniem jest tu fakt, iż w zastosowaniach OLAP tabele mają wiele kolumn, są często mocno zdenormalizowane, a wiele zapytań operuje na niewielkiej liczbie kolumn (średnio ok. 15% kolumn [12]) i na bardzo dużej liczbie wierszy. Jest tak dlatego, że w zastosowaniach analitycznych rzadziej pobierane są pojedyncze wiersze, a częściej wykonywane różnego rodzaju agregaty właśnie na poszczególnych kolumnach. Tak więc redukcja czasu dostępu do danych, dzięki uniknięciu angażowania dla nich niepotrzebnie wielu operacji dyskowych, stanowi bardzo istotny element optymalizacji czasowej zapytań.

2. Opis analizowanych mechanizmów i środowiska testowego

W celu sprawdzenia wpływu nowego typu indeksów na wyszukiwanie danych do dalszych badań zakwalifikowano środowisko SQL Server 2012, którego pierwowzorem było

[2] Temu zagadnieniu poświęcony jest np. artykuł [11].

środowisko Denali, opracowane w ramach projektu Apollo i zaprezentowane po raz pierwszy w listopadzie 2010 r. Wybór tej wersji serwera baz danych został dokonany nie tylko ze względu na istniejący w nim nowy typ indeksów kolumnowych (tzw. indeksy columnstore), lecz także ze względu na jego mechanizm VertiPaq, który łączy w sobie idee podejścia kolumnowego i przetwarzania danych w pamięci, tym samym prowadząc do szybszego wyszukiwania przetwarzanych danych. W artykule SQL Server Technical Article [2] z końca 2010 r. twórcy systemu zademonstrowali przykład zapytania operującego na tabeli z prawie 1,5 mld wierszami danych, porównując szybkość jego wykonania przy użyciu tradycyjnych indeksów (501 s) do czasu uzyskanego przy wykorzystaniu indeksu kolumnowego (1,10 s).

W związku z tym, że na początku bieżącego roku firma Microsoft udostępniła możliwość pobierania wersji ewaluacyjnej silnika SQL Server 2012, pojawiła się możliwość praktycznego sprawdzenia wpływu stosowania tych obu nowatorskich mechanizmów, przeznaczonych do zastosowań analitycznych, na realny wzrost wydajności działania serwera.

Mimo, iż zastosowanie kolumnowej indeksacji wydaje się słusznym wyborem, prowadzącym do szybszego wyszukiwania danych, nie wolno pominąć faktu, iż struktura ta musi być przebudowywana w momencie dodawania lub usuwania danych, co oczywiście generuje dodatkowe koszty. Choć takie rozwiązanie jest akceptowalne dla hurtowni danych, gdzie ładowanie danych odbywa się w cyklach dziennych bądź tygodniowych, podjęto decyzję o oszacowaniu ilości pamięci potrzebnej do utworzenia indeksu kolumnowego oraz o porównaniu szybkości tworzenia niegrupujących indeksów kolumnowych w analizowanej bazie danych w stosunku do indeksów innego typu, dostępnych w systemie SQL Server. Ilość pamięci potrzebna do stworzenia indeksu kolumnowego zależy zarówno od liczby kolumn w tabeli, typów danych zapisanych w poszczególnych kolumnach, jak i od maksymalnej liczby procesów mogących jednocześnie wykonywać jedno polecenie[3] (steruje tym parametr serwera bazy danych *MAXDOP*). Można to wyrazić wzorem (1)[4]:

$$\begin{aligned} ilość_{pamięam} = {} & [(4{,}2 \cdot LiczbaIndeksowanychKolumn) + 68] \cdot DOP + \\ & (LiczbaKolumnTypuString \cdot 34)[\mathrm{MB}], \end{aligned} \tag{1}$$

gdzie *DOP* jest parametrem określającym stopień zrównoleglenia operacji.

Stosowanie indeksu *columnstore* zwiększa możliwość kompresji wybranych do niego kolumn danych, co w efekcie wpływa na obniżenie ilości zajmowanego miejsca przez indeks. Istnieje wiele efektywnych algorytmów kompresji danych, które mogą być kolumnowo wykorzystane przy mało zróżnicowanych danych składowanych [1], ale SQL Server korzysta ze wspomnianej już – i znanej z SQL Server Analysis Services i PowerPivot – technologii VertiPaq. Co istotne, indeksy kolumnowe serwera SQL Server wykorzystują tyle pamięci, ile jest jej dostępnej na serwerze, a dane zapisane w poszczególnych kolumnach są – w zależności

[3] Zakłada się istnienie wieloprocesorowej architektury sprzętowej.
[4] Źródło równania: pozycja [4] bibliografii.

od bieżących potrzeb użytkownika bazy – dynamicznie pobierane do pamięci RAM [2]. Tym samym wolumen danych ograniczony jest jedynie wielkością pamięci fizycznej. Mówiąc o korzyściach wynikających ze stosowania indeksów kolumnowych w przetwarzaniu klasycznych zapytań SQL, nie sposób pominąć faktu, iż możliwości silnika bazy źródłowej są w pełni wykorzystywane również w przypadku korzystania z tabelarycznego trybu[5] dostępu do danych w Analysis Services. Oznacza to, że również w przypadku późniejszego raportowania i analiz danych system działa znacznie wydajniej z tabelami wykorzystującymi indeksy columnstore.

Innym typem indeksu, zorientowanym na zwiększenie szybkości przetwarzania OLAP, jest indeks bitmapowy, będący zbiorem map bitowych dla każdego indeksowanego atrybutu [5], który cechuje się wysoką efektywnością wykonywania na nich operacji logicznych oraz łatwością zliczania wystąpień danej wartości. Tym samym indeksy te są zalecane do zapytań z duża liczbą warunków równościowych, co czyni je szczególnie przydatnymi dla celów selekcji danych analitycznych, choć trzeba zwrócić szczególną uwagę na tzw. krotność (ang. *cardinality*) – czyli unikalność indeksowanych kolumn[6]. Jeżeli krotność jest większa niż 30% sumarycznej liczby danych w kolumnie, optymalizator zapytań nie użyje indeksu w swoim planie wykonania. Warto przy tym podkreślić, że ze względu na występujące w klasycznych hurtowniach danych specyficzne problemy idea tych indeksów została w nich znacznie rozbudowana, prowadząc do możliwości indeksacji danych wielowymiarowych: grupowym, ograniczonym lub haszowym grupowym indeksem bitmapowym [6]. Jednak z uwagi na obszerność zagadnienia w ramach prezentowanego w artykule fragmentu prac badawczych kwestie te zostały pominięte, za to, mając na uwadze środowisko programistyczne, które zostało wzięte pod uwagę, skupiono się na technice filtrowania bitmapowego. Technika ta jest implementacją wspomnianej idei bitmapowej adresacji danych, dokonaną przez twórców SQL Servera i stanowi ich autorską odpowiedź na pilną potrzebę optymalizacji dużej liczby połączeń. W związku z tym, że dla powstania wartościowej informacji konieczne jest połączenie danych z tabeli faktów i kilku/kilkunastu tabel wymiarów, sprostanie tak wielu jednoczesnym operacjom łączenia, wykonywanym na tak dużej ilości danych, jest dla silnika bazy danych istotnie sporym wyzwaniem. Kluczem wydaje się więc odrzucenie nadmiarowej (niepotrzebnej) grupy wierszy tabel biorących udział w tej operacji – i to już na etapie odczytu wierszy z tabeli, a nie na podstawie warunku złączenia. Tym samym filtrowanie bitmapowe powoduje, że operacji łączenia od strony tabeli faktów podlegają wyłącznie wiersze, które do takiej operacji się kwalifikują [7]. Ponadto filtrowanie to opiera się na tworzeniu w pamięci dynamicznych struktur, które przechowują binarną reprezentację wierszy

[5] Jest to tzw. tryb DirectQuery, w którym analizowane są nie klasyczne kostki OLAP, ale zbiór powiązanych tabel, a wszelkie kalkulacje (wyliczenia) są wykonywane bezpośrednio na bazie źródłowej.

[6] Jest tak dlatego, że liczba generowanych map bitowych dla danej kolumny odpowiada właśnie liczbie różnych wartości, które się w niej znajdują.

uczestniczących w złączeniu (tzw. wektory bitowe), co dodatkowo jeszcze minimalizuje wpływ tworzenia filtru na całkowity koszt realizacji zapytania.

W celu skrócenia czasu wykonywania skomplikowanych, czasochłonnych zapytań, zawierających wiele operacji łączeń i grupowania, można również stosować – oferowane przez niektórych dostawców rozwiązań bazodanowych – perspektywy zmaterializowane (ang. *materialized view*). Perspektywy (widoki) zmaterializowane – w odróżnieniu od perspektyw wirtualnych istniejących wyłącznie w postaci definicji (wyznaczenie zbioru krotek perspektywy wirtualnej następuje dopiero w momencie odwołania się do niej) – są wyliczone „na zapas" i ich wyniki są przechowywane w bazie danych. Materializowanie danych ma sens, jeżeli często wykonywane są zapytania identyczne lub podobne do tego, które występuje w definicji perspektywy. Istotne jest, aby dane w tabelach bazowych takiej perspektywy nie ulegały zbyt częstym zmianom, pociąga to bowiem za sobą konieczność aktualizacji zmaterializowanego widoku, co oznacza spadek wydajności operacji modyfikacji danych. Jeżeli w systemie pojawia się zapytanie, które może zostać wykonane z wykorzystaniem zmaterializowanych perspektyw, wówczas następuje tzw. przepisanie zapytania (ang. *query rewriting*), co oznacza, że oryginalne zapytanie użytkownika zostanie przez optymalizator zastąpione (przepisane) zapytaniem wykorzystującym widok zmaterializowany. Proces „zamiany" jest niewidoczny dla użytkownika [9].

W systemie SQL Server odpowiednikiem zmaterializowanych perspektyw są perspektywy indeksowane (ang. *indexed views).* Gdy następuje odwołanie do takiej perspektywy, serwer nie wykonuje zapytania w niej zawartego, lecz odczytuje przeliczone dane z indeksu.

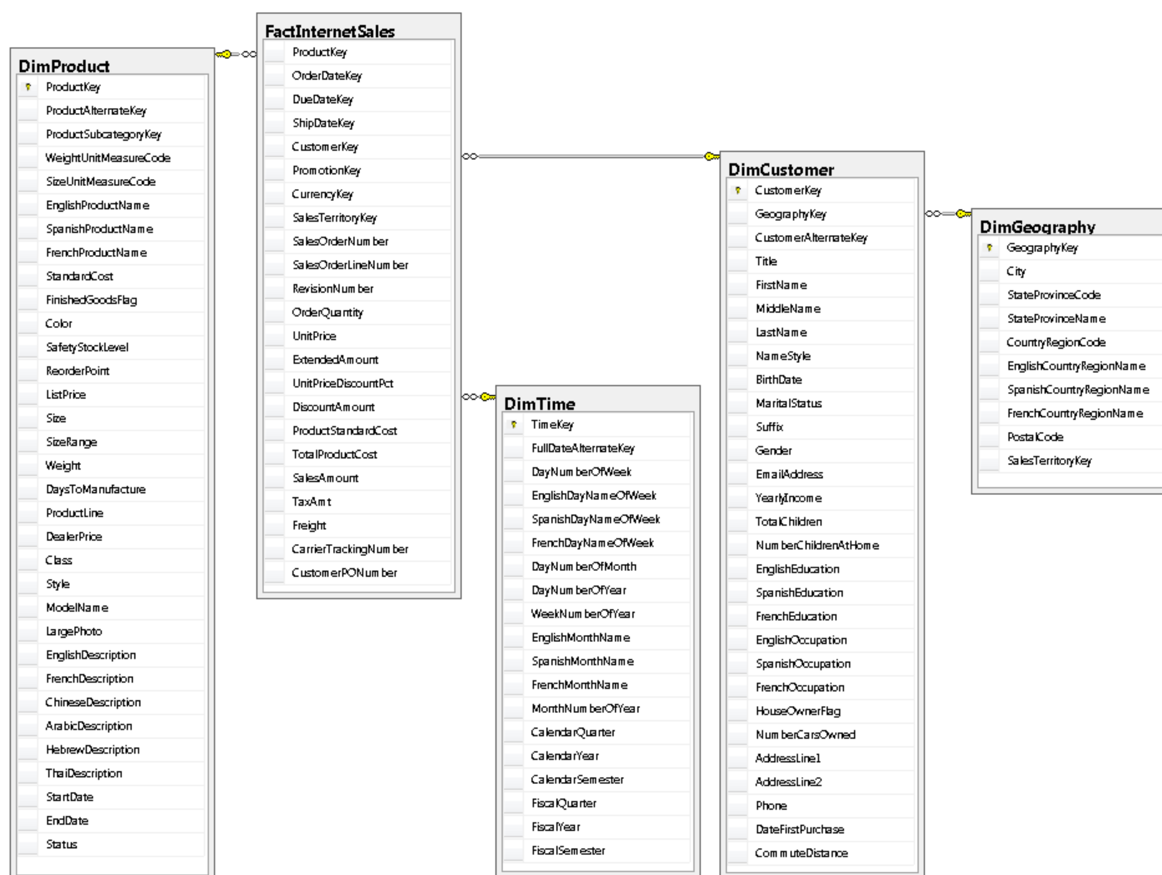
Kolejnym rozwiązaniem stworzonym z myślą o poprawie wydajności zapytań są indeksy filtrowane (ang. *filtered index*), wprowadzone przez firmę Microsoft od wersji SQL Server 2008. Jak wskazuje nazwa, w indeksie tym można użyć filtru w celu wybrania dokładnie tych wierszy, które trzeba indeksować. Indeks filtrowany zawiera tylko wybrane (wyselekcjonowane) wartości, dzięki czemu zajmuje mniej miejsca, poza tym korzysta z filtrowanych statystyk, które są dokładniejsze niż statystyki oparte na danych z całej tabeli. Wszystko to powoduje, iż dobrze skonstruowany indeks filtrowany może znacząco wpłynąć na efektywność realizowanych zapytań [10].

2.1. Przebieg testów

Badania przeprowadzano przy użyciu 4-rdzeniowego komputera z procesorem Intel® Core™ i5-2520M, o częstotliwości 2 GHz i pamięci RAM 8 GB oraz 2-rdzeniowego komputera o częstotliwości 2 GHz i pamięci RAM 8 GB, a środowiskiem testowym było oprogramowanie SQL Server 2012.

W celu weryfikacji tezy głoszącej przewagę techniki kolumnowego przetwarzania danych nad metodą klasyczną, tj. wierszową, wykonano wiele testów przy wykorzystaniu przykła-

dowej bazy danych AdventureWorksDW o zwiększonej do 4,5 mln liczbie wierszy w tabeli *FactInternetSales* do 350 tys. w tabeli *DimCustomer*.



Rys. 1. Fragment bazy AdventureWorksDW
Fig. 1. The part of the AdventureWorksDW database

Do wybranych tabel (rys. 1) bazy, zawierającej dane dotyczące internetowej sprzedaży różnych produktów, kierowano zapytania, należące do różnych klas zapytań, takie jak np.:

1. Znaleźć 2 miasta, w których uzyskano największą łączną wartość transakcji.
2. Sporządzić ranking miast pod względem łącznej wartości zawartych transakcji.
3. Znaleźć łączne kwoty transakcji na kolejnych poziomach wymiaru geograficznego.
4. Znaleźć łączne kwoty transakcji na kolejnych poziomach wymiaru czasu (rok, kwartał) dla produktów zawierających w nazwach literę ‘M’.
5. Znaleźć wartości transakcji dla produktów zawierających w nazwie literę ‘L’ na poziomach agregacji obejmujących lata i prowincje.
6. Znaleźć liczbę transakcji kupna produktów zawierających w nazwach literę ‘L’, zamówionych w Alabamie, dla poszczególnych kwartałów lat 2003 i 2004.
7. Znaleźć średnią i sumaryczną wartości transakcji kupna produktów, zawartych przez klientów w I półroczu 2003 roku, dla poszczególnych krajów, prowincji i miast.

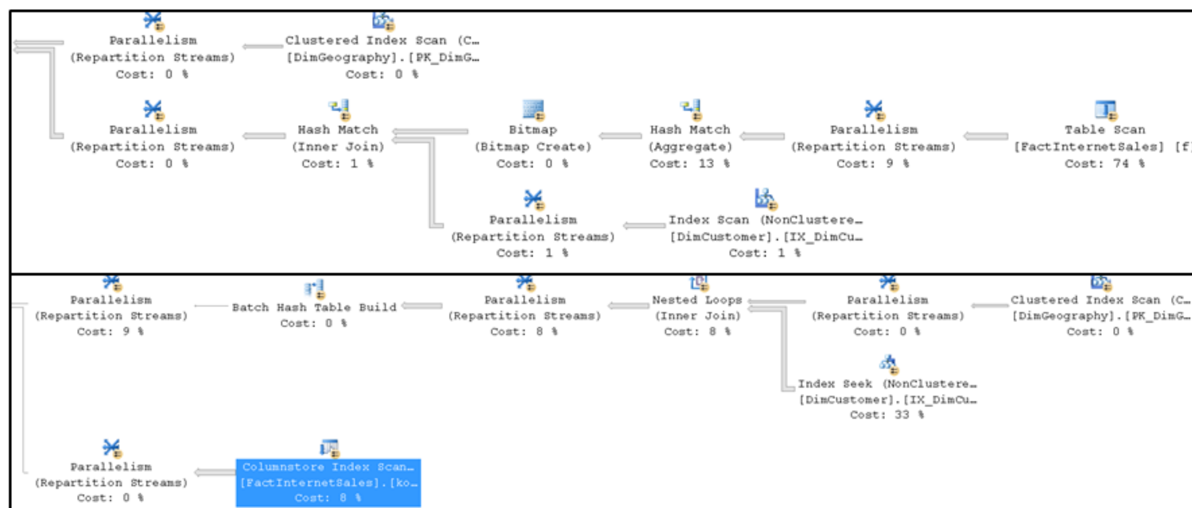
Następnie, w kolejnym kroku analizowano plany wykonania tych zapytań.

Aby przyspieszyć dostęp do danych, do tabeli faktów dodano niezbędne tradycyjne indeksy (tj. indeksy na kolumnach klucza obcego: *CustomerKey*, *ProductKey*, *CurrencyKey*, *DueDateKey*, *OrderDateKey*, *PromotionKey*, *SalesOrderNumber* oraz *ShipDateKey*), a w kolejnym kroku również indeks kolumnowy, oparty na wszystkich kolumnach tabeli z rys. 1. Taka definicja indeksu kolumnowego wymusiła na systemie zapisywanie danych poszczególnych kolumn tabeli na osobnych stronach, co – z założenia – powinno było skutkować zmniejszeniem liczby operacji wejścia/wyjścia w przypadku korzystania z tego indeksu. Testy wykonywano wykorzystując wbudowany optymalizator zapytań (tj. nie wymuszano stosowania przez optymalizator wybranych indeksów, np. przez dodanie do zapytania wskazówki `WITH (INDEX (`*`nazwa_indeksu`*`)` po nazwie tabeli). Jednocześnie, dla celów porównawczych, każdorazowo dane zapytanie było wykonywane dwukrotnie – raz przy aktywowanym indeksie kolumnowym i raz przy wyłączonym.

Fragmenty przykładowego planu wykonania zapytania 2 wyświetlającego ranking miast pod względem łącznej wartości zawartych transakcji, postaci:

```
SELECT RANK () OVER (ORDER BY SUM(SalesAmount) desc), City, SUM(SalesAmount)
FROM dbo.DimGeography g, dbo.DimCustomer c, dbo.FactInternetSales f
WHERE g. GeographyKey=c. GeographyKey AND c.CustomerKey=f.CustomerKey
GROUP BY City
[OPTION (IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX)];
```

zaprezentowano na rys. 2.



Rys. 2. Plan wykonania zapytania 2 z wyłączonym (góra)/aktywowanym (dół) indeksem kolumnowym
Fig. 2. Execution plan of query 2 with (top)/without (down) enabling the columnstore index

Efekt wyłączenia wspomnianego indeksu uzyskiwano, dodając na końcu realizowanego zapytania frazę `OPTION (IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX)`, co pozwalało optymalizatorowi ignorować istnienie tego indeksu. Początkowo zaniedbywano fakt, że dodanie dodatkowego indeksu powoduje spowolnienie dodawania i aktualizacji wierszy tabel, gdyż operacjami najczęściej wykonywanymi na danych analitycznych są ich wyszukiwanie i analiza. Jednak aby dokładniej określić całkowity bilans kosztów utworzenia indeksu kolumnowego,

wykonano zapytania zawarte w [4], które pozwoliły oszacować jego rozmiar na dysku, i zmierzono czas tworzenia takiego indeksu dla tabeli faktów. Aby zapewnić możliwie maksymalną precyzyjność dokonywanych obserwacji i wysnutych wniosków, dla kolumn, które wchodziły w skład indeksu kolumnowego, liczono tzw. krotność według wzoru:

$$krotność = \frac{LiczbaRóżnychWartościWKolumnie}{ŁącznaLiczbaWierszy} \cdot 100[\%]. \tag{2}$$

Dokonane pomiary czasowe wykazały, że dla tabeli liczącej ok. 4,5 mln wierszy, indeks kolumnowy względem pola o krotności 1,8% był tworzony prawie 3 razy wolniej (13554 ms vs 4721 ms) niż klasyczny indeks niezgrupowany na tej kolumnie. Jeżeli krotność indeksowanej kolumny była bardzo mała (np. rzędu 0,02%), wtedy czas tworzenia indeksu columnstore był nawet kilkadziesiąt (miejscami blisko 30) razy dłuższy niż w przypadku klasycznego indeksu nieklastrowanego. Tak było np. w przypadku kolumny *CustomerKey* tabeli *FactInternetSales*, dla której czas tworzenia indeksu kolumnowego przekroczył o 2862% czas tworzenia indeksu nieklastrowanego. Takie wyniki należały jednak do rzadkości, bo – porównując czasy tworzenia nieklastrowanych indeksów kolumnowych i klasycznych indeksów klastrowanych – najczęściej obserwowano znaczący spadek czasu (od 30% dla jednej z kolumn tabeli *FactInternetSales* o krotności 1,5% do blisko 94% dla kolumny *CustomerKey* tabeli *DimCustomer*, której krotność wynosiła 100%) na korzyść indeksu kolumnowego. Uwzględniając specyfikę przetwarzania analitycznego, analizowano również koszt tworzenia indeksów opartych na więcej niż 1 kolumnie. W tym jednak wypadku odnotowywane wcześniej spore różnice czasowe stawały się nieistotne i pomijalne. Przykładowo tworzenie indeksu kolumnowego opartego na 23 kolumnach tabeli było jedynie 1,05 raza dłuższe niż tworzenie analogicznego indeksu niekolumnowego, a dla tabeli o 16 kolumnach – nawet o 10% krótsze.

Jeżeli chodzi o ilość pamięci potrzebnej do utworzenia indeksu kolumnowego [4], to ze wzoru zamieszczonego w poprzednim punkcie wyliczono potrzebę alokacji blisko 432 MB pamięci, ale po wykonaniu stosownych zapytań, operujących na tabelach systemowych *sys.column_store_segments* i *sys.column_store_dictionary*, okazało się, że faktycznie zaalokowano tylko 79 MB.

Generalnie jako krytyczny parametr dokonywanych testów przyjęto czas dostępu do danych, rozumiany jako czas odpowiedzi na zadane zapytanie, oraz liczbę logicznych odczytów z pamięci.

W przeprowadzonych badaniach zmierzono czasy wykonania zapytań z wykorzystaniem indeksów tradycyjnych i kolumnowych, dla wybranych zapytań zdefiniowano również odpowiednie perspektywy zmaterializowane oraz indeksy filtrowane.

Aby uzyskiwane wyniki czasowe były w pełni porównywalne, każdorazowo przed wykonaniem danego zapytania czyszczono pamięć cache serwera, wykonując następującą sekwencję poleceń:

```
CHECKPOINT;
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;
```

3. Wyniki testów i wnioski

W wyniku przeprowadzonych eksperymentów stwierdzono, że w większości z nich liczba logicznych odczytów stron, potrzebnych do wykonania zapytania, znacząco się zmniejszała po dodaniu do tabeli indeksu kolumnowego. Pozytywne rezultaty utworzenia indeksu były szczególnie widoczne dla najliczniejszej spośród wszystkich tabel bazy danych – tj. tabeli *FactInternetSales* – dla której liczba logicznych odczytów zmniejszała się od kilkudziesięciu (w przypadku zapytania 3) do nawet kilkuset (w przypadku zapytania 2) razy. Przyjmując liczbę odczytów dokonywanych bez użycia indeksu kolumnowego za wartość bazową, liczba odczytów dokonywana z wykorzystaniem tych indeksów spadała średnio do 1,65% wyjściowej liczby odczytów. Czyli maksymalny odnotowany zysk, rozumiany jako procent różnicy liczby odczytów logicznych dokonywanych z/bez indeksu kolumnowego w stosunku do pomiaru bazowego, był równy nawet 100%. Tabela 1 zawiera szczegółowe wyniki analiz dla wybranej grupy zapytań. W przypadku tabel zawierających mniejszą liczbę wierszy (np. tabeli *DimCustomer*) zarejestrowany spadek liczby odczytów nie był tak duży, jak obserwowano to dla największej z tabel, i średnio był rzędu 95%.

Tabela 1

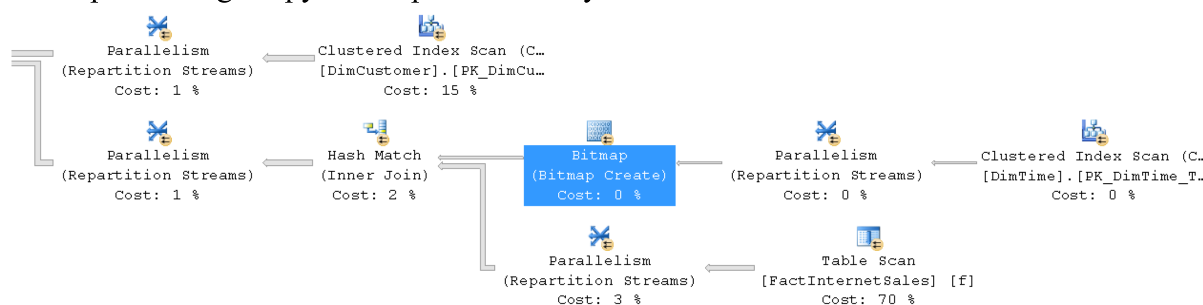
Zestawienie liczby logicznych odczytów wybranej tabeli dla poszczególnych zapytań

Numer zapytania	Liczba zwracanych wierszy	Liczba logicznych odczytów tabeli *FactInternetSales*		Zysk $(L_{BK}-L_{ZK})/L_{BK}*100$
		Bez indeksu kolumnowego L_{BK}	Z indeksem kolumnowym L_{ZK}	
1	2	82083	3052	96,28
2	269	82083	3052	96,28
3	345	82083	3052	96,28
4	18	82083	0	100
5	234	82083	0	100
6	6	18928	4790	74,69
7	317	82083	4745	94,22

W przypadku wielu zapytań – m.in. zapytania 6 czy 7, pozwalającego znaleźć średnią i sumaryczną wartość transakcji kupna produktów, zawartych przez klientów w I półroczu 2003 roku, dla poszczególnych krajów, prowincji i miast, postaci:

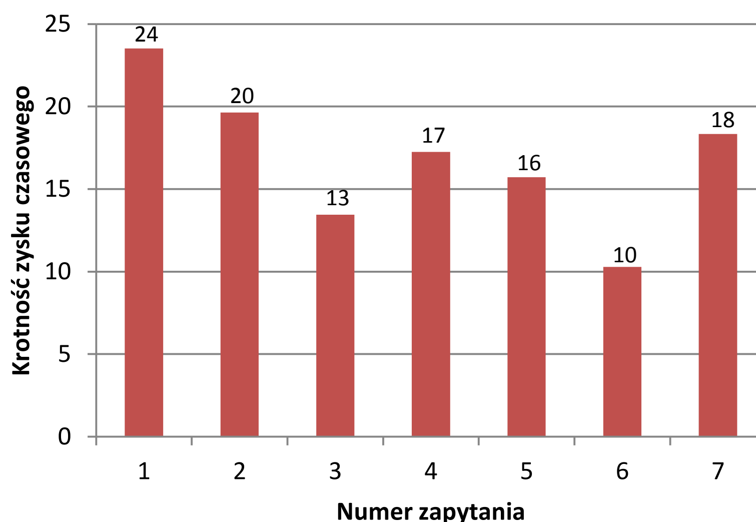
```
SELECT EnglishCountryRegionName, StateProvinceName, City, AVG(SalesAmount),
       SUM(SalesAmount)
FROM   dbo.FactInternetSales f, dbo.DimTime t, dbo.DimGeography g,
       dbo.DimCustomer c
WHERE  t.TimeKey=f.OrderDateKey AND c.CustomerKey=f.CustomerKey AND g.
       GeographyKey=c. GeographyKey and CalendarYear =2003 AND
       (CalendarQuarter=1 OR CalendarQuarter=2)
GROUP BY EnglishCountryRegionName, StateProvinceName, City WITH ROLLUP
[OPTION (IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX)];
```

w ich planie wykonania zaobserwowano użycie filtru bitmapowego. Przykład takiego planu dla wspomnianego zapytania 7 przedstawia rys. 3.



Rys. 3. Przykładowy plan wykonania zapytania 7 z użyciem filtru bitmapowego
Fig. 3. Sample query 7 execution plan with bitmap filtering optimization

Filtr bitmapowy nakładany jest na wiersze danej tabeli jeszcze przed ich przesłaniem do kolejnego operatora – np. widocznego w planie na rys. 3 operatora Parallelism, przy eliminacji tych spośród nich, których klucze (wartości klucza) nie biorą udziału w operacji łączenia z wierszami drugiej tabeli.

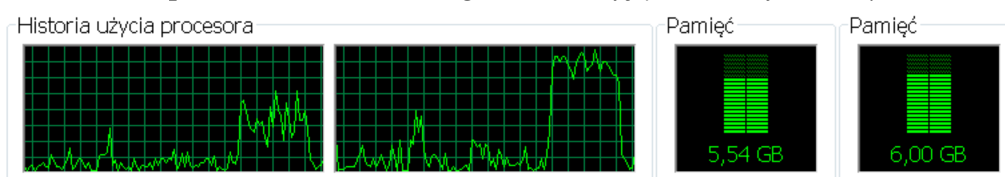


Rys. 4. Zysk czasowy osiągnięty po wykorzystaniu indeksu kolumnowego przez optymalizator zapytań
Fig. 4. The query execution time savings gained after columnstore index usage

Jeżeli chodzi o zysk czasowy, wynikający ze stosowania indeksów kolumnowych, to był on mocno zauważalny szczególnie podczas realizacji zapytań 1, 2 oraz 7. Pomiar czasu realizacji zapytań przy wykorzystanym przez optymalizator indeksie kolumnowym wykazał, że odpowiedź na zadane zapytanie pojawiała się od 10 do ponad 23 razy szybciej niż w sytuacji,

gdy optymalizator generował plan wykonania zapytania bez wykorzystania tego indeksu (rys. 4).

Ta oszczędność czasu w praktyce oznaczała skrócenie czasu wykonania zapytania z (przykładowo) 22 (pytanie 7) czy 19 sekund (pytanie 3) do – odpowiednio – 1,1 i 1,3 sekundy. Spostrzeżono przy tym, że nie zawsze koszt skanowania indeksu kolumnowego jest równy 0%. Często wynosił on 3, 12, a nawet 65%. Aby pogłębić wiedzę na temat indeksów kolumnowych i ich wykorzystania przez wbudowany optymalizator zapytań, wykonano sekwencję prostych zapytań na tabeli faktów – *FactInternetSales*. Równolegle monitorowano zużycie pamięci RAM oraz stopień wykorzystania procesora serwera bazy danych. Podczas testów zauważono znaczny wzrost wykorzystania wymienionych zasobów– szczególnie podczas tworzenia indeksu kolumnowego, ale i podczas wykonywania długotrwałych zapytań (rys. 5), co czasami prowadziło nawet do zgłoszenia wyjątku *OutOfMemory*.



Rys. 5. Ilustracja wzrostu zużycia podstawowych zasobów serwera
Fig. 5. The increase of the basic server resource consumption

Bazując na tych doświadczeniach, można stwierdzić, że aby w pełni wykorzystać możliwości indeksowania kolumnowego, należy stosować serwery o sporej mocy obliczeniowej i odpowiednio pojemnej pamięci RAM.

Analiza planów wykonania zapytań, w których szczególnie uważnie śledzono szacunkowy koszt przeszukiwania indeksu kolumnowego przez optymalizator, potwierdziła obserwacje odnośnie opłacalności wykorzystania indeksów kolumnowych. Okazało się, że miały na nią wpływ takie czynniki, jak np.:

- liczba zwracanych kolumn (im więcej – tym mniej opłacalne było stosowanie tego typu indeksu); przykładowo jeżeli w zapytaniu operującym na tabeli *FactInternetSales* we frazie `SELECT` specyfikowano tylko 2-3 kolumny, szacunkowy koszt przeszukiwania indeksu kolumnowego wynosił tylko 3%, podczas gdy po zwiększeniu liczby zwracanych kolumn do 9 wzrastał on już do 35%;
- użycie funkcji agregujących (po użyciu funkcji agregującej koszt wzrastał); przykładowo dla zapytania zwracającego tylko wartości kolumny *DiscountAmount* tabeli *FactInternetSales* koszt wynosił 5%, podczas gdy dla zapytania zwracającego sumaryczny rabat (`sum(DiscountAmount)`) koszt ten wzrastał już do 50%;
- liczba wartości w kolumnie (im większa – tym większy koszt); przykładowo jeżeli zapytanie zwracało wszystkie (4,5 mln) wartości wpisane w kolumnie, wtedy szacunkowy koszt wykonania operacji przeszukania indeksu columnstore osiągał wartość 100%, ale

po ograniczeniu liczby zwracanych wierszy do 1 mln (`SELECT TOP 1000000 …`) koszt spadał do poziomu 92%.

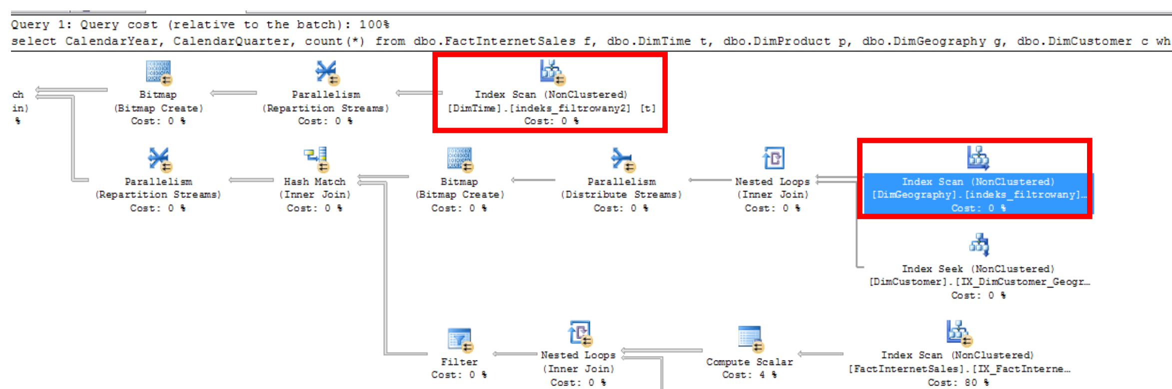
Nie zauważono natomiast znaczącego wpływu stopnia kompresji wartości wpisanych w danej kolumnie na koszt przeszukiwania indeksu kolumnowego (bez względu na to jak liczne grupy wierszy zawierały tę samą wartość w danej kolumnie, koszt kształtował się na tym samym poziomie).

W celu dalszej weryfikacji wydajności (przydatności) indeksów kolumnowych porównano je z indeksami filtrowanymi. Przykładowo dla potrzeb zapytania 6, mającego na celu określenie liczby transakcji kupna produktów zawierających w nazwach literę ‘L’, zamówionych w Alabamie dla poszczególnych kwartałów lat 2003 i 2004, zaproponowano dwa indeksy filtrowane, zdefiniowane następująco:

```
CREATE nonclustered index indeks_filtrowany on DimGeography(GeographyKey ASC)
INCLUDE (StateProvinceName )
WHERE (StateProvinceName = 'Alabama') ON [PRIMARY];
```

i

```
CREATE nonclustered index indeks_filtrowany2 on DimTime(TimeKey ASC)
INCLUDE (CalendarYear, CalendarQuarter)
WHERE (CalendarYear>='2003' AND CalendarYear<='2004') ON [PRIMARY];
```
[7]



Rys. 6. Plan wykonania zapytania 6 z aktywnym indeksem filtrowanym
Fig. 6. Execution plan of query 6 with enabling the filtered index

Optymalizator zapytań skorzystał z tych indeksów (fragment planu wykonania zamieszczono na Rys. 6.), co spowodowało prawie 2,5-krotne skrócenie czasu wykonania zapytania (z 22055 ms do 8922 ms) w stosunku do czasu wykonania tego zapytania z wykorzystaniem klasycznych indeksów pełnotablicowych. Porównując natomiast szybkość wykonania tego zapytania z aktywnymi indeksami filtrowanymi oraz z użyciem indeksu kolumnowego, odnotowano, iż czas wykonania był niemal 6 razy krótszy w przypadku indeksu kolumnowego (8922 ms dla indeksów filtrowanych vs 1548 ms dla indeksu kolumnowego). W przypadku

[7] Nie zastosowano indeksu filtrującego przyspieszającego wyszukiwanie produktów zawierających w nazwach literę ‘L’ ze względu na ograniczenie tego typu indeksów – nie ma możliwości zastosowania operatora `LIKE '%L%'`.

pozostałych zapytań również zauważono przewagę indeksu kolumnowego nad filtrowanym, przy czym obserwowane przyspieszenie, oscylujące w granicach 2-6, zależało od selektywności warunków filtrujących stosowanych w poszczególnych zapytaniach. Im mniejszej liczby wierszy dotyczył filtr, tym zysk był mniejszy.

W celu dopełnienia analizy nowo wprowadzonego indeksu kolumnowego porównano go jeszcze z perspektywami indeksowanymi. Przykładowo dla omawianego wcześniej zapytania 6 zdefiniowano perspektywę i stosowny indeks:

```
CREATE VIEW perspektywa_z6 (CalendarYear, CalendarQuarter, liczba)
WITH SCHEMABINDING
AS
(SELECT CalendarYear, CalendarQuarter, count_big(*)
FROM   dbo.FactInternetSales f, dbo.DimTime t, dbo.DimProduct p,
       dbo.DimGeography g, dbo.DimCustomer c
WHERE  t.TimeKey=f.OrderDateKey and p.ProductKey=f.ProductKey and
       c.CustomerKey=f.CustomerKey and g. GeographyKey=c. GeographyKey and
       StateProvinceName = 'Alabama' and EnglishProductName like '%L%' and
       CalendarYear>='2003' and CalendarYear<='2004'
GROUP BY CalendarYear, CalendarQuarter);
CREATE UNIQUE CLUSTERED INDEX indeks_persp_z6 on perspektywa_z6(CalendarYear,
       CalendarQuarter);
```

Po zdefiniowaniu widoku przy próbie uruchomienia zapytania 6 następowało jego przepisanie, co oznacza, że optymalizator zastępował oryginalne zapytanie zapytaniem wykorzystującym stworzony widok. Czas wykonania przepisanego zapytania wyniósł 235 ms – czyli był ok. 6,5 raza krótszy niż w przypadku realizacji tego zapytania z wykorzystaniem indeksów kolumnowych. Należy jednak zwrócić uwagę na to, że choć prędkość wykonania zapytań z wykorzystaniem indeksowanych perspektyw jest znacznie wyższa, to mechanizm ten posiada wiele ograniczeń, dotyczących m.in. dopuszczalnych typów kolumn (widok nie może zawierać kolumn typów: TEXT, NTEXT oraz IMAGE), złożonych funkcji agregujących (nie można używać funkcji: AVG, MIN, MAX, STDEV,STDEVP, VAR, VARP) czy stosowanych klauzul (niedopuszczalne są frazy: TOP, ORDER BY, DISTINCT, COMPUTE, COMPUTE BY, HAVING, CUBE oraz ROLLUP). Poza tym indeksowana perspektywa jest wykorzystywana jedynie w przypadku uruchamiania zapytań identycznych lub bardzo podobnych do zapytania zawartego w jej definicji. Oznacza to, że w przypadku wykonywania różnorodnych zapytań konieczne jest definiowanie wielu indeksowanych perspektyw, co w oczywisty sposób przekłada się na zwiększenie ilości miejsca niezbędnego do przechowywania tych struktur. W tym względzie indeksy kolumnowe wydają się bardziej elastyczne, mogą bowiem być stosowane do znacznie szerszej klasy zapytań.

4. Podsumowanie

Dynamiczny przyrost analizowanych danych powoduje konieczność poszukiwania coraz nowszych mechanizmów poprawiających wydajność takich analiz. Wprowadzone w najnowszej wersji SQL Serwera indeksy kolumnowe są właśnie próbą poszerzenia możliwości w tym zakresie.

W rozdziale zaprezentowano wyniki wieloaspektowej analizy nowego typu indeksu. Zbadano zarówno czas tworzenia, jak i ilość pamięci potrzebnej do utworzenia struktur kolumnowych. Testy dotyczące czasu realizacji zapytań wykazały, że w przypadku systemów analitycznych indeksy kolumnowe sprawdzają się znacznie lepiej niż klasyczne indeksy pełnotablicowe (zarówno zgrupowane, jak i niezgrupowane) – w wielu przypadkach również lepiej niż indeksy filtrowane, skracając czas wykonania od kilku do kilkudziesięciu razy. Porównanie czasowe z indeksowanymi perspektywami nie wypadło już tak dobrze, trzeba jednak podkreślić fakt, iż indeksy kolumnowe są mechanizmem znacznie elastyczniejszym, mają dużo mniej ograniczeń niż indeksowane widoki, a co za tym idzie mogą mieć znacznie szersze zastosowanie.

Podsumowując, w niniejszym artykule podjęto próbę nie tylko sprawdzenia efektywności indeksów kolumnowych dostępnych w serwerze SQL Server 2012, ale również przedstawienia możliwie szerokiego spektrum mechanizmów i rozwiązań, które są zwykle stosowane w celu optymalizacji zapytań o charakterze analitycznym. Stąd właśnie konfrontacja najnowszego rozwiązania firmy Microsoft z istniejącymi od dawna i dobrze poznanymi indeksami (indeksy pełnotablicowe, filtrowane i bitmapowe) oraz indeksowanymi perspektywami.

BIBLIOGRAFIA

1. Abadi D. J., Madden S. R., Ferreira M. C.: Integrating Compression and Execution in Column-Oriented Database Systems, http://db.csail.mit.edu/projects/cstore/abadisigmod06.pdf (stan na rok 2012).
2. Hanson E. N.: Columnstore Indexes for Fast Data Warehouse Query Processing in SQL Server 11.0, http://download.microsoft.com/ (stan na rok 2012).
3. Przykładowa baza danych AdventureWorksDW, http://szybkidownload.pl/download/4138/pobierz,adventureworks---przykladowa-baza.aspx (stan na rok 2012).
4. SQL Server Columnstore Index FAQ, http://social.technet.microsoft.com/wiki/contents/articles/sql-server-columnstore-index-faq.aspx (stan na rok 2012).
5. Wojciechowski M.: Systemy baz danych – indeksy bitmapowe, http://www.cs.put.poznan.pl/mmorzy/sbd-gniezno/wyklad/09a-Indeksy-bitmapowe.pdf (stan na rok 2012).

6. Idkowiak Ł., Kamiński T.: Indeksy w hurtowniach danych, https://ophelia.cs.put.poznan.pl/webdav/dbdw/students/dbdw-summer_2011/lectures/seminars/IndeksyHD_Idkowiak_Kaminski.pdf (stan na rok 2012).
7. Guzowski M.: Co nowego w silniku bazodanowym SQL Server 2008 June CTP, http://technet.microsoft.com/pl-pl/library/co-nowego-w-silniku-bazodanowym-sql-server-2008-june-ctp.aspx (stan na rok 2012).
8. How does database indexing work? http://stackoverflow.com/questions/1108/how-does-database-indexing-work (stan na rok 2012).
9. Roussopoulos N.: Materialized Views and Data Warehouses. SIGMOD Record, Vol. 27, No. 1, 1998, s. 21÷26.
10. Mendrala D., Potasiński P., Szeliga M., Widera D.: Serwer SQL 2008 Administracja i Programowanie, Helion, Gliwice 2009.
11. Sack J.: Row and batch execution modes and columnstore indexes, http://www.sqlskills.com/blogs/joe/category/columnstore-indexes.aspx (stan na rok 2012).
12. Nevarez B.: Columnstore Indexes and other new Optimizations in Denali, http://www.benjaminnevarez.com/2011/04/columnstore-indexes-and-other-new-optimizations-in-denali/ (stan na rok 2012).

Wpłynęło do Redakcji 31 stycznia 2012 r.

Abstract

With the increasing need to analyze the large amounts of data, the requirements for the time reaction of analytical processing systems or time of access to transactional data, also increasing. For this reason, a lot of solutions and mechanisms were developed, to accelerate data processing – especially in field of query execution time. One of the most interesting query acceleration feature is a new type of index – so called columnstore index, implemented in the SQL Server 2012 release, that employs Vertipaq™ technology (the same, as exists in SQL Server Analysis Services and PowerPivot).

In order to revise the opinion that column-oriented data processing improves query performance, the sample database *AdventureWorksDW* with over 4.5 millions of rows was analyzed and different types of queries were executed (point 3). It was found that in order to observe the most time reduction, the queries on the fact table (here: *FactInternetSales*), which has a star configuration (i.e. depends to other dimensions, such as *DimCustomer* or *DimProduct*), should be optimized.

As it was expected, the new columnstore index significantly improves data warehouse query performance in some cases, as it is seen in fig. 4. The small number of logical reads in the case of columnstore index enabling in comparision with the number of reads without column index, confirms this observation (Table 2).

Another interesting – from the analytical processing point of view – mechanism applied to SQL Server and examined in the research, was bitmap filtering. The fig. 3 presents the sample query execution plan with bitmap filtering optimization.

Alternative optimization mechanisms that exist in earlier versions of MS SQL Server – such as: classical clustered and unclustered indexes, indexed views and filtered indexes are also described in the article (mainly in point 2).

Adresy

Aleksandra WERNER: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-100 Gliwice, Polska, aleksandra.werner@polsl.pl.

Małgorzata BACH: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-100 Gliwice, Polska, malgorzata.bach@polsl.pl.