

Indeksy - Karta pracy nr 3

Imię i Nazwisko: Wojciech Koszyła

Swoje odpowiedzi wpisuj w **czzerwone pola**. Preferowane są zrzuty ekranu, **wymagane** komentarze.

Co jest potrzebne?

Do wykonania ćwiczenia potrzebne są:

MS SQL Server wersja co najmniej 2016,
przykładowa baza danych **AdventureWorks2017**.

Przygotowanie

Stwórz swoją bazę danych o nazwie **XYZ**. Jeśli jednak dzielisz z kimś serwer, to użyj swoich inicjałów:

```
CREATE DATABASE XYZ
GO

USE XYZ
GO
```

Dokumentacja

Obowiązkowo:

<https://docs.microsoft.com/en-us/sql/relational-databases/indexes/indexes>
<https://docs.microsoft.com/en-us/sql/relational-databases/indexes/create-filtered-indexes>

Zadanie 1

Skopiuj tabelę Product do swojej bazy danych:

```
SELECT * INTO Product FROM [AdventureWorks2017].[Production].Product
```

Stwórz indeks z warunkiem przedziałowym :

```
CREATE NONCLUSTERED INDEX Product_Range_Idx
ON Product (ProductSubcategoryID, ListPrice) Include (Name)
WHERE ProductSubcategoryID >= 27 AND ProductSubcategoryID <= 36
```

Sprawdź, czy indeks jest użyty w zapytaniu:

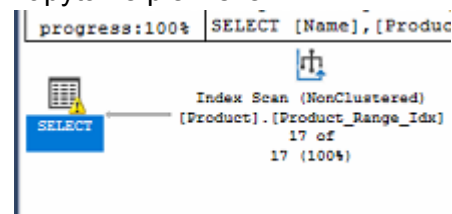
```
SELECT Name, ProductSubcategoryID, ListPrice
FROM Product
WHERE ProductSubcategoryID >= 27 AND ProductSubcategoryID <= 36
```

Sprawdź, czy indeks jest użyty w zapytaniu, który jest dopełnieniem zbioru:

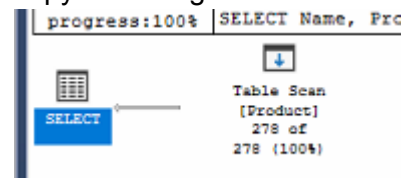
```
SELECT Name, ProductSubcategoryID, ListPrice
FROM Product
WHERE ProductSubcategoryID < 27 OR ProductSubcategoryID > 36
```

Skomentuj oba zapytania. Czy indeks został użyty w którymś zapytaniu, dlaczego? Czy indeks nie został użyty w którymś zapytaniu, dlaczego? Jak działają indeksy z warunkiem?

Zapytanie pierwsze:



Zapytanie drugie:



Stworzony przez nas indeks jest takzwanym “Filtered index”, który jest utworzony tylko dla wyspecyfikowanego przez nas subsetu. W pierwszym zapytaniu pytaliśmy o przedział, który jest objęty tym indeksem, więc został on użyty. W drugim przypadku pytaliśmy szczególnie o te pole, które nie są objęte - w tym wypadku indeks nie zawiera żadnych przydatnych informacji.

Zadanie 2 – indeksy klastrujące

Celem zadania jest poznanie indeksów klastrujących.

Skopiuj ponownie tabelę SalesOrderHeader do swojej bazy danych:

```
SELECT * INTO [SalesOrderHeader2] FROM
[AdventureWorks2017].[Sales].[SalesOrderHeader]
```

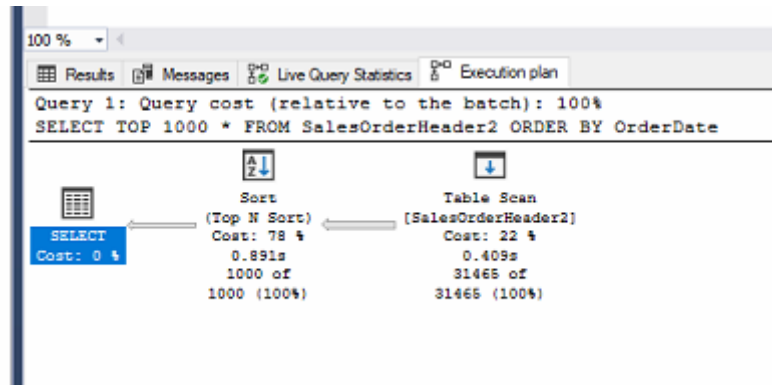
Wypisz sto pierwszych zamówień:

```
SELECT TOP 1000 * FROM SalesOrderHeader2
ORDER BY OrderDate
```

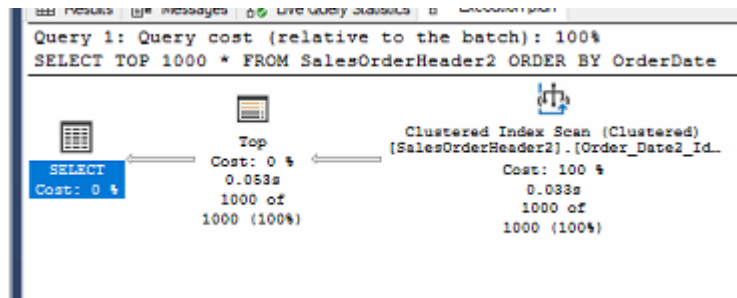
Stwórz indeks klastrujący według OrderDate:

```
CREATE CLUSTERED INDEX Order_Date2_Idx ON SalesOrderHeader2 (OrderDate)
```

Wypisz ponownie sto pierwszych zamówień. Co się zmieniło?



Przed stworzeniem indeksu:



Po stworzeniu:

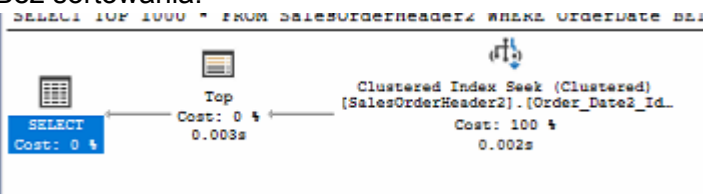
W trakcie tworzenia indeksa klastrowanego dane zostały posortowane. Teraz, gdy pobieramy te dane, wystarczy tylko pobrać 1000 sztuk z góry lub z dołu. Mimo, że ta kwerenda przed stworzeniem i po stworzeniu indeksu trwała tyle samo (0 sekund), to przewidywany czas w "Execution plan" znacząco spadł, tj. zmalał o cały rząd wielkości.

Sprawdź zapytanie:

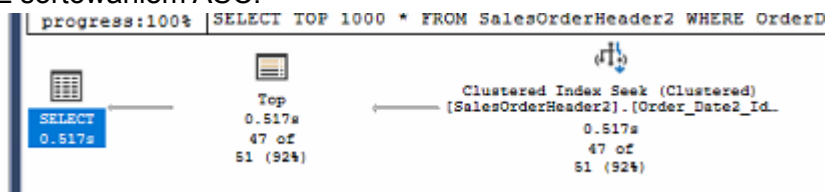
```
SELECT TOP 1000 * FROM SalesOrderHeader2
WHERE OrderDate BETWEEN '2010-10-01' AND '2011-06-01'
```

Dodaj sortowanie według OrderDate ASC i DESC. Czy indeks działa w obu przypadkach. Czy wykonywane jest dodatkowo sortowanie?

Bez sortowania:



Z sortowaniem ASC:



Z sortowaniem DESC:

Jak widzimy, nie zostało przeprowadzone dodatkowe sortowanie. Dane w indeksie klastrowanym już są posortowane. Jeśli chcemy mieć sortowanie ASC lub DESC, wystarczy dane odczytywać “w drugą stronę”, “od przodu do tyłu, lub od tyłu do przodu”.

Zadanie 3 – indeksy *column store*

Celem zadania jest poznanie indeksów typu column store.

Utwórz tabelę testową:

```
CREATE TABLE [dbo].[SalesHistory] (
  [SalesOrderID] [int] NOT NULL,
  [SalesOrderDetailID] [int] NOT NULL,
  [CarrierTrackingNumber] [nvarchar] (25) NULL,
  [OrderQty] [smallint] NOT NULL,
  [ProductID] [int] NOT NULL,
  [SpecialOfferID] [int] NOT NULL,
  [UnitPrice] [money] NOT NULL,
  [UnitPriceDiscount] [money] NOT NULL,
  [LineTotal] [numeric] (38, 6) NOT NULL,
  [rowguid] [uniqueidentifier] NOT NULL,
  [ModifiedDate] [datetime] NOT NULL
) ON [PRIMARY]
GO
```

Załącz indeks:

```
CREATE CLUSTERED INDEX [SalesHistory_Idx]
ON [SalesHistory] ([SalesOrderDetailID])
```

Wypełnij tablicę danymi:

(UWAGA! 'GO 100' oznacza 100 krotne wykonanie polecenia. Jeżeli podejrzewasz, że Twój serwer może to zbyt przeciążyć, zacznij od GO 10, GO 20, GO 50 (w sumie już będzie 80))

```
INSERT INTO SalesHistory
SELECT SH.*
FROM [AdventureWorks2017].[Sales].SalesOrderDetail SH
GO 100
```

Sprawdź jak zachowa się zapytanie, które używa obecny indeks:

```
SELECT ProductID, SUM(UnitPrice), AVG(UnitPrice), SUM(OrderQty),
AVG(OrderQty)
FROM SalesHistory
GROUP BY ProductID
ORDER BY ProductID
```

Załącz indeks typu ColumnStore:

```
CREATE NONCLUSTERED COLUMNSTORE INDEX SalesHistory_ColumnStore
ON SalesHistory(UnitPrice, OrderQty, ProductID)
```

Sprawdź różnicę pomiędzy przetwarzaniem w zależności od indeksów. Porównaj plany i opisz różnicę.

Wypełnianie tablicy zajęło bardzo dużo czasu :(00:26:52
Nie wiem, czy tak miało być, ale w nowej tablicy mam teraz ponad 12 milionów wierszy

Zadanie 4 – indeksy w pamięci

Celem zadania jest poznanie indeksów w pamięci.

Najpierw przygotujmy możliwość tworzenia optymalizacji w pamięci:

(UWAGA! Musi istnieć katalog c:\tmp)

```
ALTER DATABASE XYZ ADD FILEGROUP a_mod CONTAINS MEMORY_OPTIMIZED_DATA

ALTER DATABASE XYZ ADD FILE (name='a_mod1', filename='c:\tmp\a_mod1') TO
FILEGROUP a_mod
```

W tym zadaniu wykorzystamy ponownie schemat SalesHistory. Stwórz 3 tabele, dla 10, 1000 i 100000 kulek:

```
CREATE TABLE [dbo].[SalesHistory_10](
[SalesOrderID] [int] NOT NULL PRIMARY KEY NONCLUSTERED IDENTITY(1,1),
[SalesOrderDetailID] [int] NOT NULL,
[CarrierTrackingNumber] [nvarchar](25) NULL,
[OrderQty] [smallint] NOT NULL,
[ProductID] [int] NOT NULL,
[SpecialOfferID] [int] NOT NULL,
[UnitPrice] [money] NOT NULL,
[UnitPriceDiscount] [money] NOT NULL,
[LineTotal] [numeric](38, 6) NOT NULL,
[rowguid] [uniqueidentifier] NOT NULL,
[ModifiedDate] [datetime] NOT NULL,
INDEX Sales_Hash_10 HASH ([ProductID]) WITH (BUCKET_COUNT = 10)
) WITH (
MEMORY_OPTIMIZED = ON,
DURABILITY = SCHEMA_AND_DATA);
GO

CREATE TABLE [dbo].[SalesHistory_1000](
[SalesOrderID] [int] NOT NULL PRIMARY KEY NONCLUSTERED IDENTITY(1,1),
[SalesOrderDetailID] [int] NOT NULL,
[CarrierTrackingNumber] [nvarchar](25) NULL,
[OrderQty] [smallint] NOT NULL,
[ProductID] [int] NOT NULL,
[SpecialOfferID] [int] NOT NULL,
[UnitPrice] [money] NOT NULL,
[UnitPriceDiscount] [money] NOT NULL,
[LineTotal] [numeric](38, 6) NOT NULL,
```

```

[rowguid] [uniqueidentifier] NOT NULL,
[ModifiedDate] [datetime] NOT NULL,
INDEX Sales_Hash_1000 HASH ([ProductID]) WITH (BUCKET_COUNT = 1000)
) WITH (
    MEMORY_OPTIMIZED = ON,
    DURABILITY = SCHEMA_AND_DATA);
GO

CREATE TABLE [dbo].[SalesHistory_100000](
    [SalesOrderID] [int] NOT NULL PRIMARY KEY NONCLUSTERED IDENTITY(1,1),
    [SalesOrderDetailID] [int] NOT NULL,
    [CarrierTrackingNumber] [nvarchar](25) NULL,
    [OrderQty] [smallint] NOT NULL,
    [ProductID] [int] NOT NULL,
    [SpecialOfferID] [int] NOT NULL,
    [UnitPrice] [money] NOT NULL,
    [UnitPriceDiscount] [money] NOT NULL,
    [LineTotal] [numeric](38, 6) NOT NULL,
    [rowguid] [uniqueidentifier] NOT NULL,
    [ModifiedDate] [datetime] NOT NULL,
    INDEX Sales_Hash_100000 HASH ([ProductID]) WITH (BUCKET_COUNT = 100000)
) WITH (
    MEMORY_OPTIMIZED = ON,
    DURABILITY = SCHEMA_AND_DATA);
GO

```

Wypełnij tabele danymi, pierwszą wygeneruj (HASH_10), kolejne skopiuj (HASH_1000 i HASH_100000).

Generowanie:

```

INSERT INTO SalesHistory_10
(SalesOrderDetailID, CarrierTrackingNumber, OrderQty, ProductID,
SpecialOfferID, UnitPrice, UnitPriceDiscount, LineTotal, rowguid,
ModifiedDate)
SELECT TOP(100000) SH.SalesOrderDetailID, SH.CarrierTrackingNumber,
SH.OrderQty, SH.ProductID+ROUND(RAND()*9000, 0),
SH.SpecialOfferID, SH.UnitPrice, SH.UnitPriceDiscount, SH.LineTotal,
SH.rowguid, SH.ModifiedDate FROM SalesHistory SH
GO 2

```

Kopie:

```

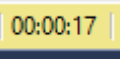
INSERT INTO SalesHistory_1000
(SalesOrderDetailID, CarrierTrackingNumber, OrderQty, ProductID,
SpecialOfferID, UnitPrice, UnitPriceDiscount, LineTotal, rowguid,
ModifiedDate)
SELECT SH.SalesOrderDetailID, SH.CarrierTrackingNumber, SH.OrderQty,
SH.ProductID,
SH.SpecialOfferID, SH.UnitPrice, SH.UnitPriceDiscount, SH.LineTotal,
SH.rowguid, SH.ModifiedDate FROM SalesHistory_10 SH

INSERT INTO SalesHistory_100000
(SalesOrderDetailID, CarrierTrackingNumber, OrderQty, ProductID,
SpecialOfferID, UnitPrice, UnitPriceDiscount, LineTotal, rowguid,
ModifiedDate)
SELECT SH.SalesOrderDetailID, SH.CarrierTrackingNumber, SH.OrderQty,
SH.ProductID,
SH.SpecialOfferID, SH.UnitPrice, SH.UnitPriceDiscount, SH.LineTotal,

```

```
SH.rowguid, SH.ModifiedDate FROM SalesHistory_10 SH
```

Co powiesz o czasie działania operacji? Dlaczego tak było?

Hash10: 

Hash1000 wraz z Hash100000: 

Przy tworzeniu indeksów podawaliśmy ilość “bucket’ów”, 10, 1000 i 100’000. Są one wykorzystywane przy funkcji hashowania.

Z racji dużej ilości dodawanych wierszy, naturalnie przy zbyt małej ilości bucket’ów występowałyby dużo kolizji, tj. dużo wierszy przypisane byłoby do tego samego wiersza, co przedłuża czas operacji do $O(n)$.

W drugim przypadku tych wiaderek jest dużo więcej, a w trzecim jeszcze więcej. Mniej kolizji = mniejsza złożoność obliczeniowa.

Sprawdź rozłożenie kubelków:

```
SELECT
    object_name(hs.object_id) AS 'object name',
    i.name as 'index name',
    hs.total_bucket_count,
    hs.empty_bucket_count,
    floor((cast(empty_bucket_count as float)/total_bucket_count) * 100) AS
'empty_bucket_percent',
    hs.avg_chain_length,
    hs.max_chain_length
FROM sys.dm_db_xtp_hash_index_stats AS hs
JOIN sys.indexes AS i
    ON hs.object_id=i.object_id AND hs.index_id=i.index_id
```

Skomentuj rozłożenie:

	object name	index name	total_bucket_count	empty_bucket_count	empty_bucket_percent	avg_chain_length	max_chain_length
1	SalesHistory_10	Sales_Hash_10	16	0	0	12500	20500
2	SalesHistory_1000	Sales_Hash_1000	1024	925	90	2020	5200
3	SalesHistory_100000	Sales_Hash_100000	131072	130972	99	2000	4200

Pomimo podawania ilości bucketów jako 10, 1000 i 100000 SQL zaokrąglił je w górę do najbliższej potęgi dwójki.

Oprócz tego, potwierdza się moje przypuszczenie, że przy zwiększeniu ilości bucketów zmniejszy się złożoność obliczeniowa (patrzemy tutaj na avg_chain_length).

Nie wiem, jaka jest użyta funkcja hashująca, lecz jest ona dość kiepska - w tym przypadku zwiększenie ilości bucketów z 1024 na 131’072 poprawiło “złożoność” o jeden procent, co tak naprawdę jest tylko jednym więcej używanym bucketem (99 vs 100 wiaderek).

Znajdź ProductId z dużą liczbą wystąpień i małą:

```
SELECT ProductID, COUNT(*) FROM SalesHistory_10 GROUP BY ProductID
```

Użyj te wartości w zapytaniach dla trzech tabel:

```
SELECT * FROM SalesHistory_10 WHERE ProductID = ID
SELECT * FROM SalesHistory_1000 WHERE ProductID = ID
```

```
SELECT * FROM SalesHistory_100000 WHERE ProductID = ID
```

Skomentuj uzyskane wyniki kosztowe, czasowe oraz estymacji liczby krotek w planie:

	ProductID	(No column name)
47	4973	3100
48	5028	200
49	9605	2000

Używane ID:

SalesHistory_10:

```
SELECT * FROM SalesHistory_10 WHERE ProductID = 4973 00:00:00
SELECT * FROM SalesHistory_10 WHERE ProductID = 5028 00:00:00
SELECT * FROM SalesHistory_10 WHERE ProductID = 9605 00:00:00
```

SalesHistory_1000:

```
SELECT * FROM SalesHistory_1000 WHERE ProductID = 4973 00:00:00
SELECT * FROM SalesHistory_1000 WHERE ProductID = 5028 00:00:00
SELECT * FROM SalesHistory_1000 WHERE ProductID = 9605 00:00:00
```

SalesHistory_100000:

```
SELECT * FROM SalesHistory_100000 WHERE ProductID = 4973 00:00:00
SELECT * FROM SalesHistory_100000 WHERE ProductID = 5028 00:00:00
SELECT * FROM SalesHistory_100000 WHERE ProductID = 9605 00:00:00
```

Przy mniejszej ilości elementów o tym samym ProductID wszystkie trzy zachowywały się “tak samo szybko”, tj. czas wykonania był pomijalny na tyle, że zmiany środowiskowe komputera byłyby dużym “szumem”.

W przypadku największego zapytania, tj. zapytanie o indeks pojawiający się 3100 razy również nie zauważyliśmy różnicy - aby upewnić się, że nie jest to “przypadek”, powtórzyłem te zapytanie wiele razy i za każdym razem czas wykonania “był ten sam”. Włączyłem opcję “Execution plan” (Ctrl+M) i zobaczyłem, że wszystkie trzy zapytania korzystają z “Index Seek (NonClustered Hash)”. Wszystkie trzy odnoszą się do tego samego wiaderka, więc ich zapytania są wręcz jednakowe.