Indeksy - Karta pracy nr 3

Imię i Nazwisko:	Oscar Teeninga

Swoje odpowiedzi wpisuj w **czerwone 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?

W przypadku zapytania 1 został wykorzystany indeks, natomiast w przypadku drugiego już nie. Dzieje się tak ponieważ został stworzony warunek na indeks, który dla drugiego zapytania się wyklucza i byłby bezużyteczny.

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 100 * 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?

Zmieniła się kolejność w obrębie krotek z tą samą datą

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?

Indeks działa w obu przypadkach i nie jest wykonywane dodatkowe sortowanie

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
```

Załóż 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
```

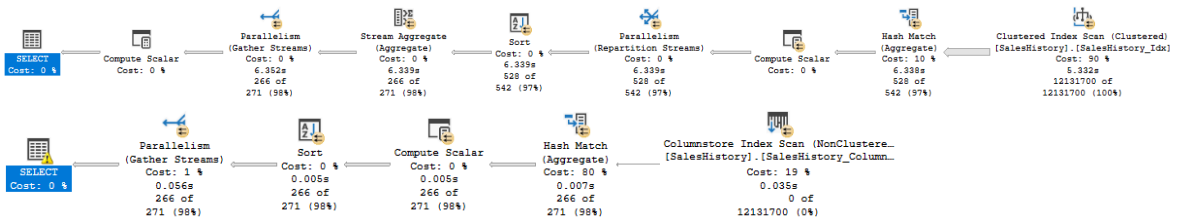
Sprawdź jak zachowa się zapytanie, które używa obecny indeks:

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

Załóż 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.



W przypadku wersji bez indeksu mamy dużo więcej operacji oraz czas jest znacznie dłuższy. Mamy dodatkową operację Stream Aggregate i Parallelism.

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 kubełków:

```
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?

Co ciekawe, czas wykonania SaleHistory_10000 trwało krócej niż SaleHistory_1000. W przypadku SaleHistory_1000 mamy bucket size równy 1000, a kopiowane jest 2000 elementów i z tego to wynika.

Sprawdź rozłożenie kubełków:

```
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	17900
2	SalesHistory_1000	Sales_Hash_1000	1024	931	90	2150	5900
3	SalesHistory_100000	Sales_Hash_100000	131072	130972	99	2000	4200

Zgodnie z oczekiwaniami, brakło nam miejsca w bucket'cie podczas kopiowania. W przypadku średniego bucketu zostało nad 931 wolnego miejsca, co odpowiada 90% jego rozmiaru (w przybliżeniu).

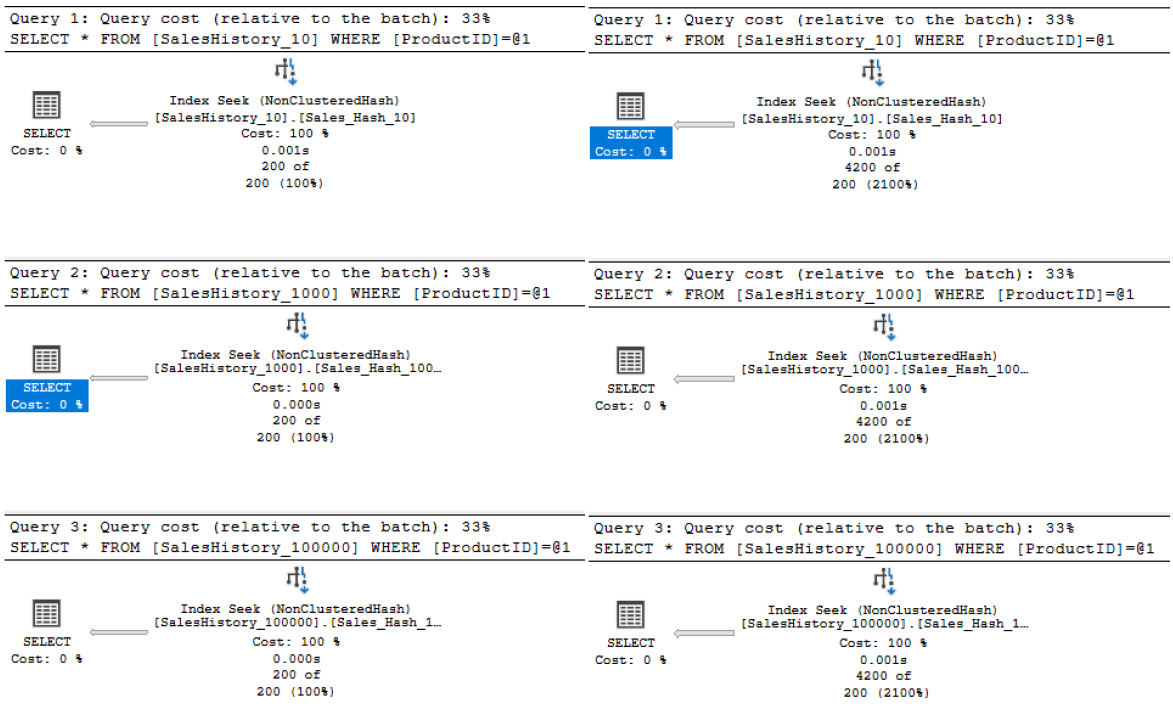
Znajdź ProductId z dużą liczbą wystąpień i małą:

```
SELECT ProductID, COUNT(*) FROM SalesHistory_10 GROUP BY ProductID
ORDER BY 2 (DESC)
```

Użyj te wartości w zapytaniach dla trzech tabel:

```
SELECT * FROM SalesHistory_10 WHERE ProductID = 7421/6999
SELECT * FROM SalesHistory_1000 WHERE ProductID = 7421/6999
SELECT * FROM SalesHistory_100000 WHERE ProductID = 7421/6999
```

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



Produkt z najmniejszą liczbą wystąpień to 7421, a najwięcej to 6999. Estymata liczby krotek jest równa wystąpień dla konkretnego produktu. Czasowo wygląda to bardzo podobnie, kosztowo tak samo. Ciekawe jest, że w przypadku najczęstszego produktu otrzymujemy informację w stylu 4200 of 200 (2100%).