Indeksy, optymalizator - lab5

Imię i Nazwisko: Daniel Kuc

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

Oprogramowanie - 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?

Pierwsze zapytanie wykorzystuje indeks "Product_Range_Idx", ponieważ warunek WHERE jest zgodny z warunkiem używanym w tworzeniu indeksu. W zapytaniu, wybierane są kolumny, gdzie "ProductSubcategoryID" jest pomiędzy 27 a 36. Ponieważ indeks obejmuje kolumny "ProductSubcategoryID" i "ListPrice", a warunek zapytania jest zgodny z zakresem indeksu, indeks zostanie użyty.

Drugie zapytanie nie wykorzystuje indeksu "Product_Range_Idx", ponieważ warunek WHERE nie jest zgodny z warunkiem używanym w tworzeniu indeksu. W zapytaniu, wybierane są kolumny, gdzie "ProductSubcategoryID" jest mniejsze niż 27 lub większe niż 36. Warunek ten nie jest zgodny z warunkiem zakresu indeksu, dlatego indeks nie będzie używany do optymalizacji zapytania.

Indeksy z warunkiem, takie jak "Product_Range_Idx" w tym przypadku, są przydatne, gdy chcemy optymalizować zapytania, które mają warunki na kolumnach, które są pokryte przez indeks, takie jak "ProductSubcategoryID" w tym przypadku. Indeksy te mogą przyspieszyć wyszukiwanie danych, które spełniają określone warunki, ponieważ są zoptymalizowane pod kątem tego konkretnego zakresu wartości. Jednakże, indeksy z warunkiem mogą nie być używane w przypadku, gdy warunki zapytań nie są zgodne z zakresem indeksu, tak jak w drugim zapytaniu w tym przypadku.

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:

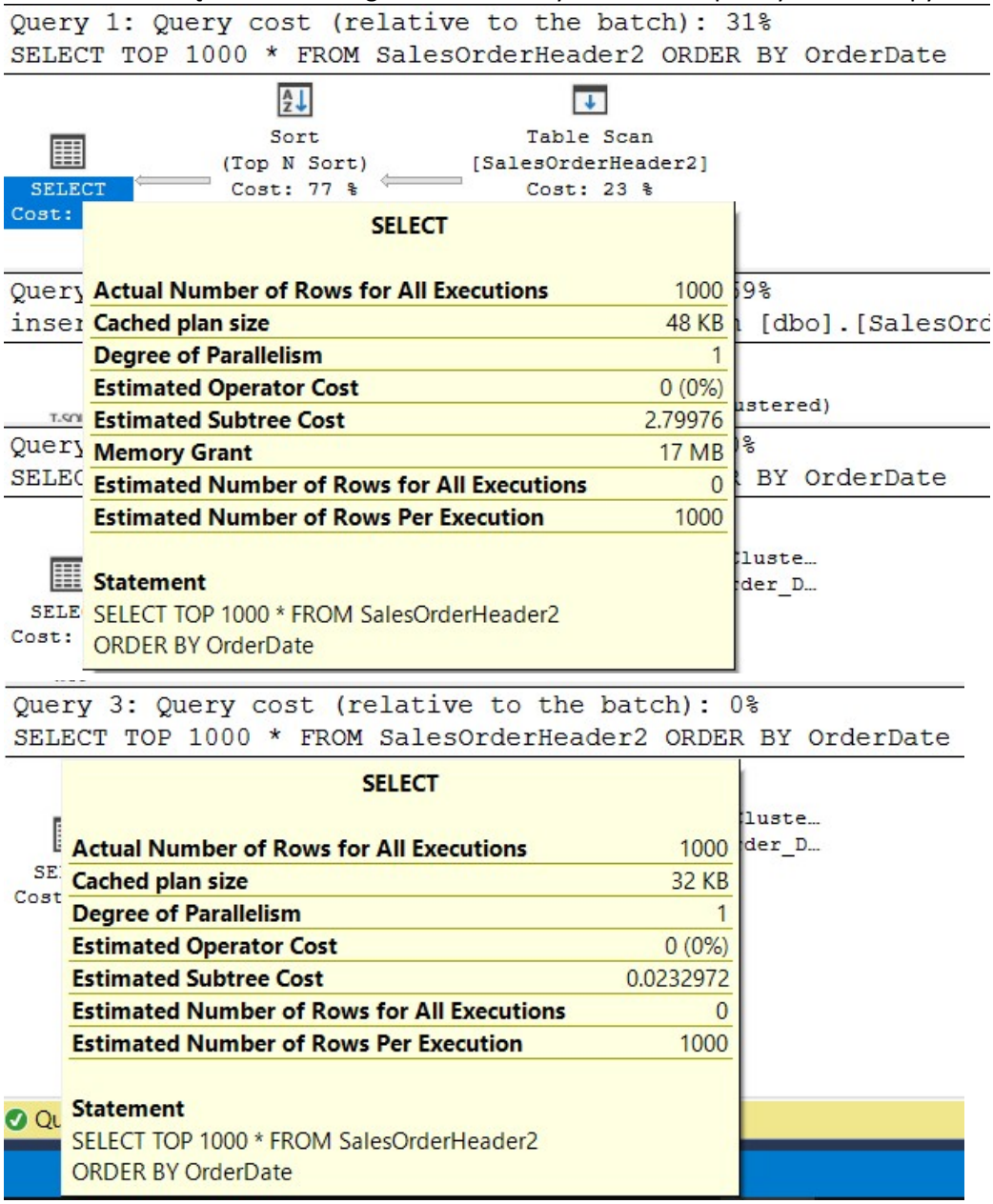
```
SELECT TOP 1000 * FROM SalesOrderHeader2
ORDER BY OrderDate
```

Stwórz indeks klastrujący według OrderDate:

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

Wypisz ponownie sto pierwszych zamówień. Co się zmieniło?

Dodanie indeksu pozwoliło na efektywne posortowanie wyników zapytania według tej kolumny (OrderDate). Dzięki temu, czas wykonywania polecenia po dodaniu indeksu ulega znacznemu skróceniu, ponieważ dane są uporządkowane w sposób optymalny w ramach indeksu, co umożliwa uniknięcie kosztownego sortowania wyników na etapie wykonania zapytania.



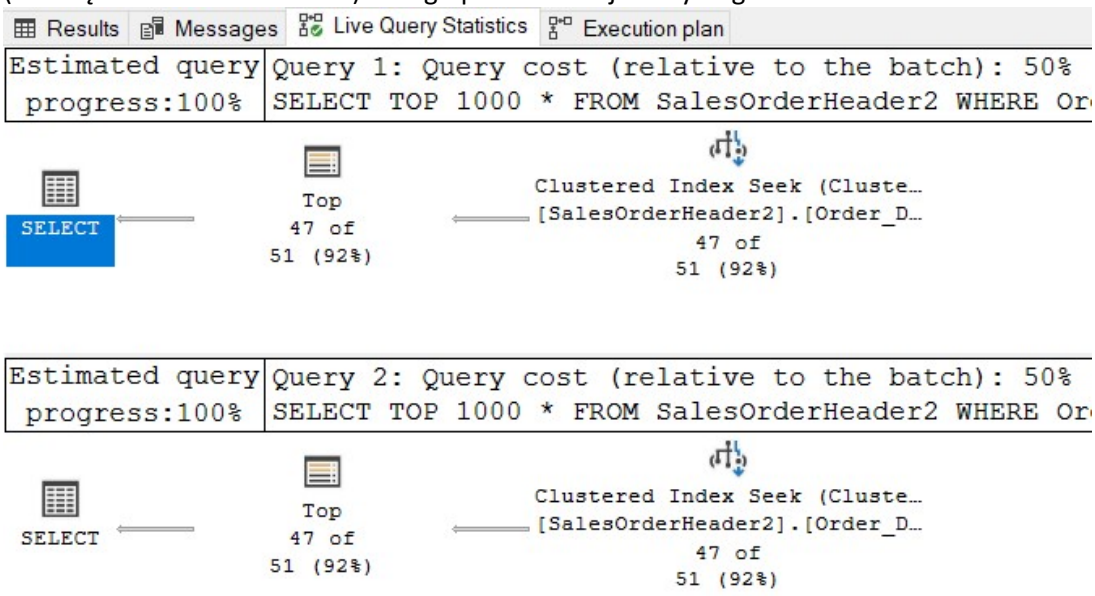
Sprawdź zapytanie:

```
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?

Indeksy zadziałają w obu przypadkach oraz nie jest wykonywane dodatkowe sortowanie.

Wynika z tego ciekawy wniosek, że silnik bazo danowy potrafi czytać strony pamięci od tyłu (mówiąc bardzo kolokwialnie) i z tego powodu nie jest wymagane 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.

Polecenie wykonuje się o wiele wolniej z indeksem SalesHistory_Idx w porównaniu do indeksu SalesHistory_ColumnStore z uwagi na sposób, w jaki indeksy są zdefiniowane i jakie są ich właściwości.

Indeks SalesHistory_Idx jest typem indeksu gromadzącego dane w sposób sekwencyjny, tzw. indeksem klastorwym, co oznacza, że dane w indeksie są przechowywane na dysku w takiej samej kolejności, jak w tabeli podstawowej. W związku z tym, kiedy zapytanie grupuje i sortuje dane według kolumny ProductID, może występować konieczność przeszukania całego indeksu w celu zebrania danych o produktach o tym samym identyfikatorze.

Z drugiej strony, indeks SalesHistory_ColumnStore jest typem indeksu kolumnowego, co oznacza, że dane są przechowywane kolumnowo, a nie wierszowo, co może być bardziej efektywne w przypadku zapytań agregujących takich jak to zapytanie. Indeks ten jest zoptymalizowany pod kątem analizy danych i może być bardziej efektywny w przypadku operacji agregacyjnych, takich jak SUM i AVG, ponieważ umożliwia szybkie dostęp do danych tylko związanych z wymaganymi kolumnami (UnitPrice, OrderQty, ProductID), bez konieczności przeszukiwania całego indeksu czy wczytywania niepotrzebnych danych z innych kolumn.

Wniosek:
Indeks kolumnowy może być bardziej efektywny w przypadku zapytań agregujących takich jak to zapytanie, które korzysta z funkcji SUM i AVG.

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_100000, SaleHistory_1000 trwało krócej niż SalesHistory_10. Wynika to z wielkości "bucket size" ponieważ 10 nie jest wystarczającą ilość do tej ilości danych i kubełki szybko się "przepełniają" przez co insert staję się nie efektywny w taki sposób, że operacje insert trwała na nim około 1,5s a na tabelach z 100000/1000 kubełkami 0,2s.

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	927	90	2061	4400
3	SalesHistory_100000	Sales_Hash_100000	131072	130972	99	2000	4200

W przypadku 10 kubełków wyekspletowaliśmy je do cna, co sugeruje, że powinniśmy zwiększyć ich ilość. W przypadku średniej ilości bucketów zostało nam 927 wolnego miejsca, co odpowiada 90% jego rozmiaru a w przypadku największej ilości zostało nam aż 130972 wolnych kubełków, co odpowiada 99%. Mamy sygnał, że taka ilość kubełków jest nadmiarowa i nie zarządzamy pamięcią w efektywny sposób.

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:

Produkt o najmniejszej liczbie wystąpień to 7421, zaś o największej to 6999. Estymowana liczba krotek jest równa liczbie wystąpień danego produktu. Czasowo i kosztowo wygląda to bardzo podobnie, ponieważ tutaj nie ma już operacji "insert", lecz "select" i przeszukiwanie kubełków dzieje się w stosunkowo podobnym czasie, nawet gdy liczba kubełków jest mniejsza, ale zawierają one większą ilość danych, lub gdy jest ich więcej, ale z mniejszą liczbą danych.

Punktacja

zadanie	pkt
1	2.5
2	2.5
3	2.5
4	2.5
razem	10