Indeksy, optymalizator
Lab 6-7

Imię i nazwisko: Dariusz Piwowarski, Wojciech Przybytek

Celem ćwiczenia jest zapoznanie się z planami wykonania zapytań (execution plans), oraz z budową i możliwością wykorzystaniem indeksów (cz. 2.)

Swoje odpowiedzi wpisuj w miejsca oznaczone jako:

> Wyniki:

```
-- ...
```

Ważne/wymagane są komentarze.

Zamieść kod rozwiązania oraz zrzuty ekranu pokazujące wyniki, (dołącz kod rozwiązania w formie tekstowej/źródłowej)

Zwróć uwagę na formatowanie kodu

Oprogramowanie - co jest potrzebne?

Do wykonania ćwiczenia potrzebne jest następujące oprogramowanie

- MS SQL Server,
- SSMS - SQL Server Management Studio
- przykładowa baza danych AdventureWorks2017.

Oprogramowanie dostępne jest na przygotowanej maszynie wirtualnej

Przygotowanie

Stwórz swoją bazę danych o nazwie lab6.

```
create database lab5
go

use lab5
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?

> Wyniki:

Pierwsze zapytanie

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		17		0.0033007	
Full Index Scan (Index Scan)	table: [dbo].[product]; index: [product_range_idx];	17	17	0.0033007	0.0

Drugie zapytanie

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		278		0.0127253	
Full Scan (Table Scan)	table: [dbo].[product];	278	278	0.0127253	0.0

```
Indeks został użyty w pierwszym zapytaniu (Index Scan), ponieważ klauzula where w zapytaniu mieści się w przedziale
klauzuli where w indeksie. Indeks nie został za to użyty w drugim zapytaniu (Table Scan), ponieważ przedziały w
klauzulach where zapytania i indeksu są rozłączne, a więc indeks nie zawiera rekordów, które są wyciągane z tabeli przy
tym zapytaniu. Indeksy z warunkiem zawierają tylko te rekordy, które spełniają owy warunek.
```

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?

> Wyniki:

Przed stworzeniem indeksu

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		1000		2.80124	
Transformation (TopN Sort - Sort)		1000	1000	2.80124	59.0
Full Scan (Table Scan)	table: [dbo].[salesorderheader2];	31465	31465	0.631968	22.0

Po stworzeniu indeksu

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		1000		0.0233442	
Transformation (Top)		1000	1000	0.0233442	4.0
Full Index Scan (Clu	table: [dbo].[salesorderheader2]; index: [order_date2_idx];	1000	1000	0.0232442	4.0

```
Bez indeksu wykonywane są operacje Table Scan oraz TopN Sort, a więc po pobraniu rekordów są one dodatkowo sortowane.
Po stworzeniu indeksu wykonywane są operacje Culstered Index Scan oraz Top, a więc po pobraniu rekordów nie są one
dodatkowo sortowane, ponieważ są już w poprawnej kolejności dzięki zastosowaniu indeksu klastującego.
```

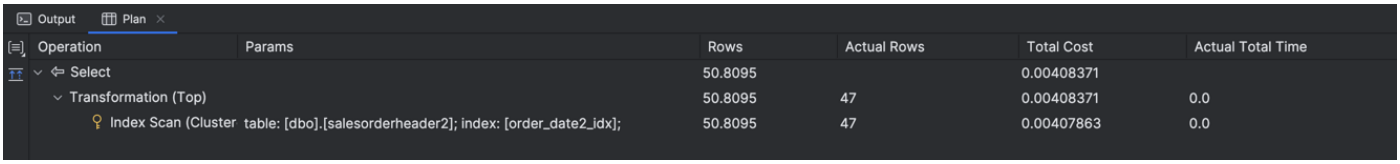
Sprawdź zapytanie:

```
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?

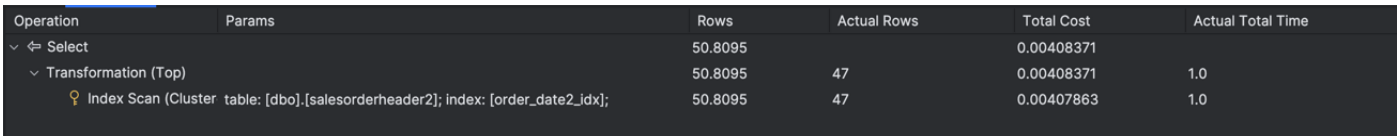
> Wyniki:

ASC



Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		50.8095		0.00408371	
Transformation (Top)		50.8095	47	0.00408371	0.0
Index Scan (Cluster	table: [dbo].[salesorderheader2]; index: [order_date2_idx];	50.8095	47	0.00407863	0.0

DESC



Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		50.8095		0.00408371	
Transformation (Top)		50.8095	47	0.00408371	1.0
Index Scan (Cluster	table: [dbo].[salesorderheader2]; index: [order_date2_idx];	50.8095	47	0.00407863	1.0

```
Indeks działa zarówno dla sortowania ASC (kolejność w jakiej rekordy są fizycznie posortowane) jak i DESC (kolejność
odwrotna) i w żadnym z przypadków nie ma dodatkowego sortowania. Wynika stąd, że nie ma potrzeby tworzenia dodatkowego
indeksu z odwrotną kolejnością sortowania, jeżeli planujemy wykonywać zapytania z sortowaniem.
```

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
 )
```

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.

> Wyniki:

Indeks klastrowany

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		290		1.8564	
Sort		290	266	1.8564	62.0
Value (Compute Scalar)		290		1.84133	
Aggregate (Aggregate - Hash Match)		290	266	1.84133	62.0
Full Index Scan (Clustered Index Scan)	table: [dbo].[saleshistory]; index: [saleshistory_idx];	121317	121317	1.25229	27.0

Indeks typu ColumnStore

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		290		0.0810956	
Sort		290	266	0.0810956	0.0
Value (Compute Scalar)		290	266	0.0768718	0.0
Aggregate (Aggregate - Hash Match)		290	266	0.0768718	0.0
Full Index Scan (Index Scan)	table: [dbo].[saleshistory]; index: [saleshistory_columnstore];	121317	0	0.0179671	3.0

```
Dla obu indeksów plany wykonania są bardzo podobne, najpierw wykonywany jest Index Scan (na odpowiednim indeksie)
następnie Hash Match aby pogrupować rekordy, Compute Scalar aby wyliczyć sumy i średnie, a na końcu rekordy są sortowane
według productid. Istotną różnicą między wykonaniami są natomiast czasy trwania zapytań oraz ich koszt, zapytanie z
indeksem typu ColumnStore wykonuje się dużo szybciej i efektowniej niż przy użyciu indeksu klastrowanego.
Indeksy ColumnStore przechowują fizycznie dane w kolumnach a nie w wierszach, co pozwala na dużo większą kompresję
i szybszy do nich dostęp ponieważ wszystkie dane są tego samego typu oraz mają podobne wartości.
```

Zadanie 4 – własne eksperymenty

Należy zaprojektować tabelę w bazie danych, lub wybrać dowolny schemat danych (poza używanymi na zajęciach), a następnie wypełnić ją danymi w taki sposób, aby zrealizować poszczególne punkty w analizie indeksów. Warto wygenerować sobie tabele o większym rozmiarze.

Do analizy, proszę uwzględnić następujące rodzaje indeksów:

- Klastrowane (np. dla atrybutu nie będącego kluczem głównym)
- Nieklastrowane
- Indeksy wykorzystujące kilka atrybutów, indeksy include
- Filtered Index (Indeks warunkowy)
- Kolumnowe

Analiza

Proszę przygotować zestaw zapytań do danych, które:

- wykorzystują poszczególne indeksy
- które przy wymuszeniu indeksu działają gorzej, niż bez niego (lub pomimo założonego indeksu, tabela jest w pełni skanowana) Odpowiedź powinna zawierać:
- Schemat tabeli
- Opis danych (ich rozmiar, zawartość, statystyki)
- Trzy indeksy:
- Opis indeksu
- Przygotowane zapytania, wraz z wynikami z planów (zrzuty ekranow)
- Komentarze do zapytań, ich wyników
- Sprawdzenie, co proponuje Database Engine Tuning Advisor (porównanie czy udało się Państwu znaleźć odpowiednie indeksy do zapytania)

> Wyniki:

Wygenerowano tabelę `dbo.PurchaseOrderHistory`

```
use lab5;

create table dbo.PurchaseOrderHistory(
    PurchaseOrderID int not null,
    RevisionNumber tinyint,
    Status tinyint,
    EmployeeID int,
    VendorID int,
    ShipMethodID int,
    OrderDate datetime,
    ShipDate datetime,
    SubTotal money,
    TaxAmt money,
    Freight money,
    TotalDue money,
    ModifiedDate datetime
);

insert into dbo.PurchaseOrderHistory
 select sh.*
 from adventureworks2017.Purchasing.PurchaseOrderHeader sh
go 60;

select count(*) as count from PurchaseOrderHistory;
```

count
240720

Oraz tabelę `dbo.PurchaseOrderDetail`

```
create table dbo.PurchaseOrderDetail
(
    PurchaseOrderID       int           not null,
    PurchaseOrderDetailID int           not null,
    DueDate               datetime      not null,
    OrderQty              smallint      not null,
    ProductID             int           not null,
    UnitPrice             money         not null,
    LineTotal             money         not null,
    ReceivedQty           decimal(8, 2) not null,
    RejectedQty           decimal(8, 2) not null,
    StockedQty            decimal(9, 2) not null,
    ModifiedDate          datetime      not null
);

insert into dbo.PurchaseOrderDetail
select pod.*
from adventureworks2017.Purchasing.PurchaseOrderDetail pod;
```

Eksperyment 1

Stworzono nieklastrowany indeks z klauzulą include `date_employee_index` zawierający informacje o dacie zamówienia i wykonującego je pracownika.

```
create nonclustered index date_employee_index on PurchaseOrderHistory(OrderDate) include (EmployeeID);
```

Pozwala on efektywnie sprawdzać ile zamówień wykonali poszczególni pracownicy w danym dniu/przedziale czasowym, ponieważ wykonany na tabeli zostanie Index Seek lub Index Scan w zależności od tego jaki procent wszystkich rekordów objemuje wybrany przedział czasowy.

```
select EmployeeID, count(*) as OrdersCount from PurchaseOrderHistory where OrderDate = '2014-06-16' group by EmployeeID;
```

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		12		0.0302402	
Value (Compute Scalar)		12		0.0302402	
Aggregate (Aggregate - Hash Match)		12	10	0.0302402	0.0
Index Scan (Index Seek)	table: [dbo].[PurchaseOrderHistory]; index: [date_employee_index];	960	960	0.0058985	0.0

Jeżeli jednak postanowimy dodać nową kolumnę do zapytania, np. średnią wartość zamówienia w danym przedziale czasowym to zauważymy, że stworzony wcześniej indeks nie jest używany.

```
select EmployeeID, count(*) as OrdersCount, avg(TotalDue) as AverageCost
from PurchaseOrderHistory
where OrderDate = '2014-06-16'
group by EmployeeID;
```

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select ⚠		12		2.26506	
Value (Compute Scalar)		12		2.26506	
Aggregate (Aggregate - Hash Match)		12	10	2.26506	53.0
Full Scan (Table Scan)	table: [dbo].[PurchaseOrderHistory];	960	960	2.12511	52.0

Wymuszając użycie indeksu możemy zauważyć, że najbardziej kosztowną operacją jest RID Lookup, czyli dostęp do wartości kolumny TotalDue, przez co łączny koszt przewyższa ten kiedy przeszukiwana jest cała tabela.

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Ti...
Select ⚠		12		2.67696	
Value (Compute Scalar)		12		2.67696	
Aggregate (Aggregate - Hash Match)		12	10	2.67696	2.0
Nested Loops (Inner Join - Nested Loops)		960	960	2.65255	2.0
Index Scan (Index Seek)	table: [dbo].[PurchaseOrderHistory]; index: [date_employee_index];	960	960	0.0058985	0.0
Row Id Access (RID Lookup)	table: [dbo].[PurchaseOrderHistory];	1	960	2.64264	0.0

Można ten problem rozwiązać zamieniając indeks na taki, który będzie zawierał również kolumnę TotalDue.

```
drop index date_employee_index on PurchaseOrderHistory;
create nonclustered index date_employee_total_index on PurchaseOrderHistory (OrderDate) include (EmployeeID, TotalDue);
```

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		12		0.0310048	
Value (Compute Scalar)		12		0.0310048	
Aggregate (Aggregate - Hash Match)		12	10	0.0310048	0.0
Index Scan (Index Seek)	table: [dbo].[PurchaseOrderHistory]; index: [date_employee_total_index];	960	960	0.00659863	0.0

Jeżeli jednak wykonamy teraz pierwotne zapytanie to zauważymy, że koszt wykonania zapytania zwiększył się używając rozszerzonego indeksu.

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		12		0.0309403	
Value (Compute Scalar)		12		0.0309403	
Aggregate (Aggregate - Hash Match)		12	10	0.0309403	0.0
Index Scan (Index Seek)	table: [dbo].[PurchaseOrderHistory]; index: [date_employee_total_index];	960	960	0.00659863	0.0

Wynika stąd, że podczas tworzenia indeksów z klauzulą include należy uważać na to, których kolumn używamy, zbyt mała ilość sprawi, że zapytania nieuwzględnione w indeksie będą bardziej kosztowne, nastomiast zbyt duża ilość spowolni wszystkie pozostałe zapytania.

Eksperyment 2

Przenalizujmy podobny przykład co w zadaniu 3, stworzono 2 indeksy na kolumnach OrderDate, PurchaseOrderID, TotalDue - jeden klastrowany a drugi typu columnstore.

```
create columnstore index columnstore_index on PurchaseOrderHistory(OrderDate, PurchaseOrderID, TotalDue);
create clustered index clustered_index on PurchaseOrderHistory(OrderDate, PurchaseOrderID, TotalDue);
```

Następnie za pomocą tych indeksów pobrano ID i wartość 3 najdroższych zamówień w roku 2014.

```
select top 3 PurchaseOrderID, TotalDue
from PurchaseOrderHistory with (index (clustered_index))
where year(OrderDate) = 2014
order by TotalDue;

select top 3 PurchaseOrderID, TotalDue
from PurchaseOrderHistory with (index (columnstore_index))
where year(OrderDate) = 2014
order by TotalDue;
```

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		3		4.04035	
Transformation (Top)		3	3	4.04035	19.0
Transformation (Gather Streams - Parallelism)		3	3	4.04035	19.0
Transformation (TopN Sort - Sort)		3	36	4.01181	16.0
Full Index Scan (Clustered Index Scan)	table: [dbo].[PurchaseOrderHistory]; index: [clustered_index];	145500	145500	2.07765	11.0
Select		3		1.21209	
Transformation (TopN Sort - Sort)		3	3	1.21209	1.0
Filter		145500	145500	0.0628409	1.0
Value (Compute Scalar)		240720	240720	0.0512864	25.0
Full Index Scan (Index Scan)	table: [dbo].[PurchaseOrderHistory]; index: [columnstore_index];	240720	240720	0.0488792	1.0

Jak widać użycie indeksu typu columnstore powoduje prawie 4-krotnie mniejszy koszt wykonania zapytania. Można więc zastanowić się, czy indeksy columnstore nie są lepsze w każdym przypadku lepsze niż klasyczne indeksy. Jeżeli jednak dodamy do zapytania jedną dodatkową kolumnę, np. ID sprzedawcy, to wyniki będą już nieco inne.

```
select top 3 PurchaseOrderID, TotalDue, VendorID
from PurchaseOrderHistory with (index (clustered_index))
where year(OrderDate) = 2014
order by TotalDue;

select top 3 PurchaseOrderID, TotalDue, VendorID
from PurchaseOrderHistory with (index (columnstore_index))
where year(OrderDate) = 2014
order by TotalDue;
```

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		3		4.04035	
Transformation (Top)		3	3	4.04035	19.0
Transformation (Gather Streams - Parallelism)		3	3	4.04035	19.0
Transformation (TopN Sort - Sort)		3	36	4.01181	17.0
Full Index Scan (Clustered Index Scan)	table: [dbo].[PurchaseOrderHistory]; index: [clustered_index];	145500	145500	2.07765	12.0
Select		3		28.5559	
Transformation (Top)		3	3	28.5559	194.0
Transformation (Gather Streams - Parallelism)		3	3	28.5559	194.0
Transformation (TopN Sort - Sort)		3	3	28.5273	18.0
Nested Loops (Inner Join - Nested Loops)		145500	145500	28.321	174.0
Full Index Scan (Index Scan)	table: [dbo].[PurchaseOrderHistory]; index: [columnstore_index];	240720	145500	0.032726	2.0
Index Scan (Clustered Index Seek)	table: [dbo].[PurchaseOrderHistory]; index: [clustered_index];	1	145500	28.184	122.0

Dla indeksu klastrowanego koszt wykonania zapytania pozostał taki sam, natomiast dla indeksu typu columnstore wzrósł ponad 20-krotnie! Oznacza to, że klastrowane indeksy są dużo bardziej uniwersalne, natomiast indeksy typu columnstore, choć bardzo szybkie, muszą być dostosowane pod konkretne zapytanie.

Eksperyment 3

Spróbujemy teraz przeanalizować indeksy warunkowe oraz co zmieni zastąpienie indeksu warunkowego indeksem bez warunku.

W tym celu rozważmy scenariusz, w którym chcemy znaleźć dla zamówień o statusie 1 sumę OrderQty dla każdego produktu, możemy sobie wyobrazić że status ten odpowiada zamówieniom, które dopiero co wpłynęły do systemu i ważne jest dla nas, aby często sprawdzać jaką ilość danego produktu potrzebujemy, aby zrealizować te zamówienia.

Powyższy scenariusz jest realizowany przez następujące zapytanie:

```
select ProductID, sum(OrderQty) as SumQty
from PurchaseOrderHistory as o
         join dbo.purchaseorderdetail p on o.PurchaseOrderID = p.PurchaseOrderID
where Status = 1
group by ProductID
order by SumQty desc
```

I daje następujące wyniki:

	ProductID	SumQty
54	484	35750
55	481	35750
56	487	35750
57	488	35750
58	482	35750
59	483	35750
60	486	35750
61	489	35750
62	357	23400
63	356	23400
64	322	19500
65	493	15600
66	496	15600
67	494	15600
68	495	15600
69	492	15600
70	506	11700
71	358	11700

W pierwszej kolejności tworzymy prosty indeks klastrowany na PurchaseOrderID w tabeli PurchaseOrderDetail, natomiast w analizie skupimy się na wyborze indeksu w tabeli PurchaseOrderHistory.

```
create clustered index purchaseorderdetail_clustered_index on PurchaseOrderDetail (PurchaseOrderID)
```

Koszt zapytania bez żadnego indeksu (w tabeli PurchaseOrderHistory) to: 3.05357

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		262.691		3.05357	
Sort		262.691	195	3.05357	77.0
Aggregate (Aggrega		262.691	195	3.03891	77.0
Hash Join (Inn		28101.2	31005	2.88703	69.0
Full Index S	table: [dbo].[PurchaseOrderDetail]; index: [purchaseorderdetail_clustered_index];	8845	8845	0.0737522	6.0
Full Scan (1	table: [dbo].[PurchaseOrderHistory];	12746.1	14625	2.48495	46.0

Patrząc na zapytanie, widzimy tu możliwość zastosowania indeksu warunkowego z warunkiem `where Status = 1`, aby zoptymalizować nasze zpytanie wystarczy, że indeks będzie obejmował kolumnę kluczową `PurchaseOrderID`. Zakładając (zgodnie ze scenraiuszem eksperymentu), że status 1 dotyczy tylko nowych zamówień, to indeks ten obejmuje jedynie niewielką część wierszy z tabeli, zatem przewaga takiego indeksu (nad indeksem bez warunku obejmującym wszystkie wiersze), jest taka, że wymaga on dużo rzadszych aktualizacji (niższy koszt utrzymania indeksu) oraz będzie zajmował fizycznie mniej miejsca.

```
create nonclustered index purchaseorderhistory_range_index
    on PurchaseOrderHistory (PurchaseOrderID)
    where Status = 1;
```

Koszt zapytania z takim indeksem wynosi 0.469743

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		174.958		0.469743	
Sort		174.958	195	0.469743	38.0
Aggregate (Aggregate - I		174.958	195	0.456348	38.0
Merge Join (Inner ,		32146.4	31005	0.286473	27.0
Full Index Scan	table: [dbo].[PurchaseOrderHistory]; index: [purchaseorderhistory_range_index];	14625	14625	0.0201102	3.0
Full Index Scan	table: [dbo].[PurchaseOrderDetail]; index: [purchaseorderdetail_clustered_index];	8845	8789	0.0737522	3.0

Jeśli przy tworzeniu pominiemy warunek, to zauważmy że w przypadku naszego zapytania indeks ten stanie się nieużyteczny.

```
create nonclustered index purchaseorderhistory_nonrange1_index
    on PurchaseOrderHistory (PurchaseOrderID)
```

Wymuszenie takiego indeksu, powowduje że koszt zapytania jest ponad 10 krotnie wyższy niż zapytania bez żadnego indeksu.

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		174.958		52.1411	
Sort		174.958	195	52.1411	113.0
Transformation (Gather Streams - Parallelism)		174.958	195	52.1277	145.0
Aggregate (Aggregate - Stream Aggregate)		174.958	195	52.0989	78.0
Sort		349.915	371	52.0988	78.0
Transformation (Repartition Streams - Parallelism)		349.915	371	52.0908	78.0
Aggregate (Partial Aggregate - Hash Match)		349.915	371	52.0617	90.0
Hash Join (Inner Join - Hash Match)		32146.4	31005	51.9653	89.0
Transformation (Repartition Streams - Parallelism)		8845	8845	0.114009	10.0
Full Index Scan (Clustered Index Scan)	table: [dbo].[PurchaseOrder...	8845	8845	0.068809	8.0
Transformation (Repartition Streams - Parallelism)		14625	14625	51.7225	76.0
Nested Loops (Inner Join - Nested Loops)		14625	14625	51.6139	90.0
Full Index Scan (Index Scan)	table: [dbo].[PurchaseOrder...	260780	260780	0.577003	13.0
Row Id Access (RID Lookup)	table: [dbo].[PurchaseOrder...	0.0560818	14625	50.4918	61.0

Aby stworzyć użyteczny indeks musimy dodać `Status` jako kolumnę kluczową, natomiast `PurchaseOrderID` może pozostać kolumną kluczową lub dołączoną.

```
create nonclustered index purchaseorderhistory_nonrange2_index
    on PurchaseOrderHistory (Status, PurchaseOrderID)

create nonclustered index purchaseorderhistory_nonrange3_index
    on PurchaseOrderHistory (Status) include (PurchaseOrderID)
```

Różnica pomiędzy dwoma powyższymi indeksami jest taka, że w przypadku gdy `PurchaseOrderID` pozostanie kolumną kluczową to powoduje to zastąpienie Hash Join przez Merge Join, który ma wtedy minimalnie mniejszy koszt.

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		174.958		0.493727	
Sort		174.958	195	0.493727	33.0
Aggregate (Aggregate - Hash Match)		174.958	195	0.480332	32.0
Merge Join (Inner Join - Merge Join)		32146.4	31005	0.310457	23.0
Index Scan (Index Seek)	table: [dbo].[PurchaseOrderHi...	14625	14625	0.044094	3.0
Full Index Scan (Clustered Index Scan)	table: [dbo].[PurchaseOrderD...	8845	8789	0.0737522	3.0

Operation	Params	Rows	Actual Rows	Total Cost	Actual Total Time
Select		174.958		0.516336	
Sort		174.958	195	0.516336	17.0
Aggregate (Aggregate -		174.958	195	0.502941	17.0
Hash Join (Inner J		32146.4	31005	0.333066	12.0
Full Index Scan	table: [dbo].[PurchaseOrderDetail]; index: [purchaseorderdetail_clustered_index];	8845	8845	0.0737522	2.0
Index Scan (Inc	table: [dbo].[PurchaseOrderHistory]; index: [purchaseorderhistory_nonrange3_index];	14625	14625	0.044094	2.0

Ostatecznie najniższy koszt został uzyskany przy pomocy indeksu warunkowego, chociaż różnica jest nieznaczna (0.469 z warunkowym, 0.493 bez warunku z kolumnami kluczowymi Status i PurchaseOrderID). Należy jednak pamiętać o innych zaletach indeksów warunkówych, czyli niższym koszcie utrzymywania oraz przechowywania indeksu.

Filtered indexes can provide the following advantages over full-table indexes:

1. Improved query performance and plan quality.

   A well-designed filtered index improves query performance and execution plan quality because it is smaller than a full-table nonclustered index and has filtered statistics. The filtered statistics are more accurate than full-table statistics because they cover only the rows in the filtered index.

2. Reduced index maintenance costs.

   An index is maintained only when data manipulation language (DML) statements affect the data in the index. A filtered index reduces index maintenance costs compared with a full-table nonclustered index because it is smaller and is only maintained when the data in the index is changed. It is possible to have a large number of filtered indexes, especially when they contain data that is changed infrequently. Similarly, if a filtered index contains only the frequently modified data, the smaller size of the index reduces the cost of updating the statistics.

3. Reduced index storage costs.

   Creating a filtered index can reduce disk storage for nonclustered indexes when a full-table index isn't necessary. You can replace a full-table nonclustered index with multiple filtered indexes without significantly increasing the storage requirements.

zadanie	pkt
1	2
2	2
3	2
4	10
razem	16