

# Indeksy, optymalizator

## Lab 6-7

---

### Imię i nazwisko:

Przemysław Spyra, Piotr Urbańczyk

---

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
  - lub inne
- przykładowa baza danych AdventureWorks2017.

Oprogramowanie dostępne jest na przygotowanej maszynie wirtualnej

## Przygotowanie

Stwórz swoją bazę danych o nazwie lab6.

```
create database lab6
go

use lab6
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:

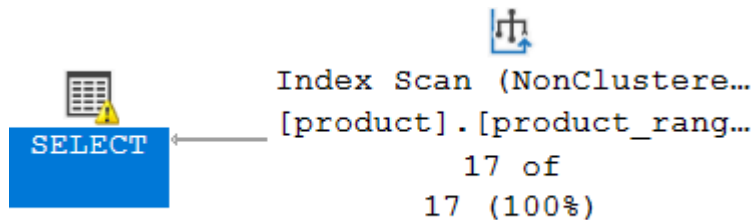
```
select name, productsubcategoryid, listprice  
from product  
where productsubcategoryid < 27 or productsubcategoryid > 36
```

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?

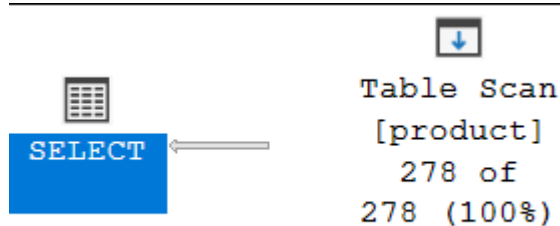
---

### Wyniki:

Indeks z warunkiem przedziałowym został użyty w zapytaniu z tym samym warunkiem:



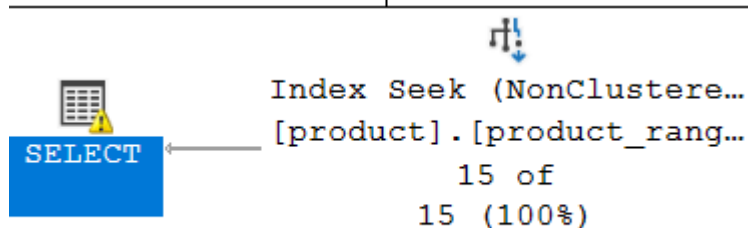
Natomiast nie został użyty w zapytaniu o wiersze niespełniające tego warunku. Zamiast tego SZBD wykonał pełne skanowanie tabeli (Table Scan).



Skoro indeks został ograniczony do wierszy z informacjami o produktach o productsubcategoryid  $\geq 27$  oraz  $\leq 36$ , to nie będzie używany przy przeszukiwaniu innych (niespełniających warunku zadanego przy tworzeniu indeksu) wierszy w tej tabli.

Ciekawostką jest fakt, że indeks nie został użyty w sposób optymalany (Index Scan a nie Index Seek). Z jakiegoś powodu stanie się tak dopiero wtedy, gdy zapytanie ograniczymy od góry nie do 36 a do 35

```
select name, productsubcategoryid, listprice
from product
where productsubcategoryid >= 27 and productsubcategoryid <= 35
```



## 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:

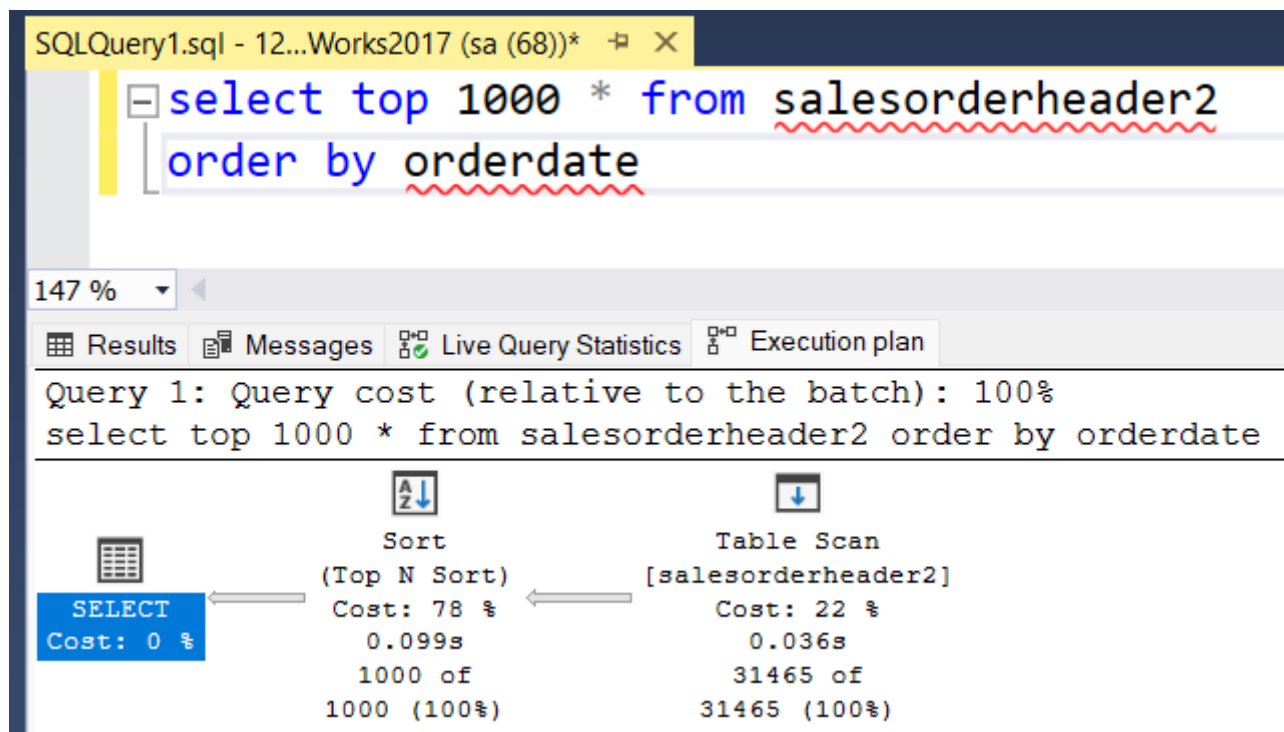
```
select top 1000 * from salesorderheader2  
order by orderdate
```

Stwórz indeks klastrowy według OrderDate:

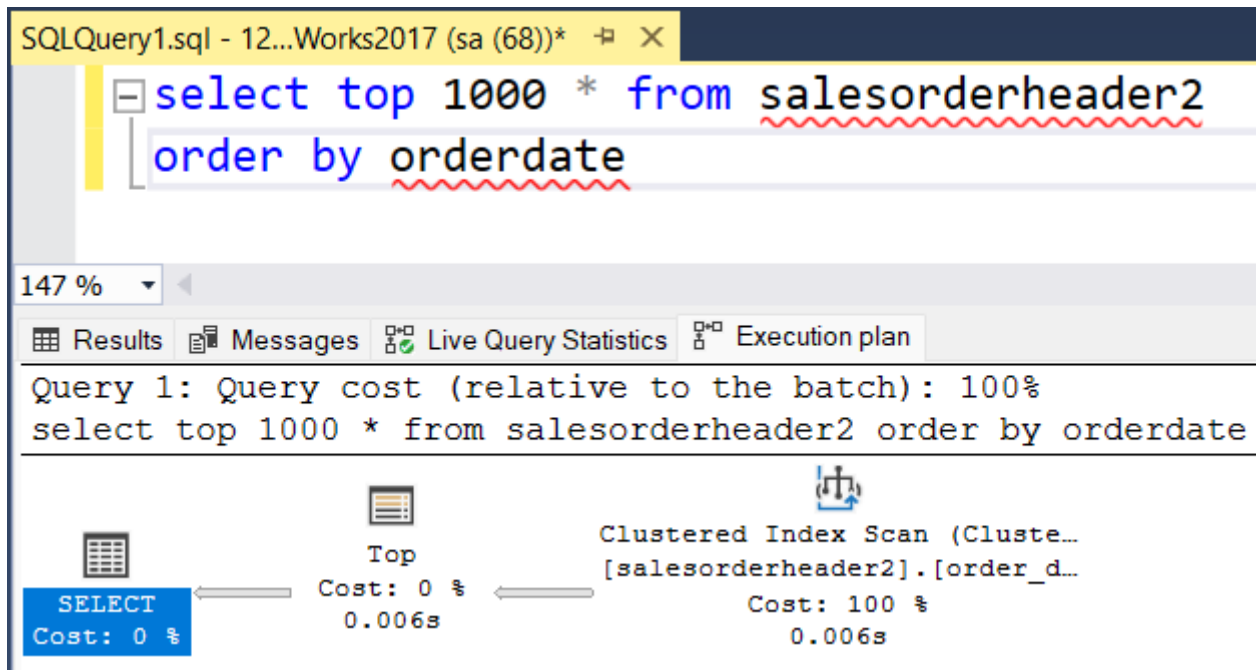
```
create clustered index order_date2_idx on salesorderheader2(orderdate)
```

Wypisz ponownie sto pierwszych zamówień. Co się zmieniło?

Przed:



Po:



SQLQuery1.sql - 12...Works2017 (sa (68))\*

```
select top 1000 * from salesorderheader2
order by orderdate
```

147 %

Results Messages Live Query Statistics Execution plan

Query 1: Query cost (relative to the batch): 100%

select top 1000 \* from salesorderheader2 order by orderdate

SELECT Cost: 0 %

Top Cost: 0 % 0.006s

Clustered Index Scan (Clustered Index Scan) [salesorderheader2].[order\_d... Cost: 100 % 0.006s

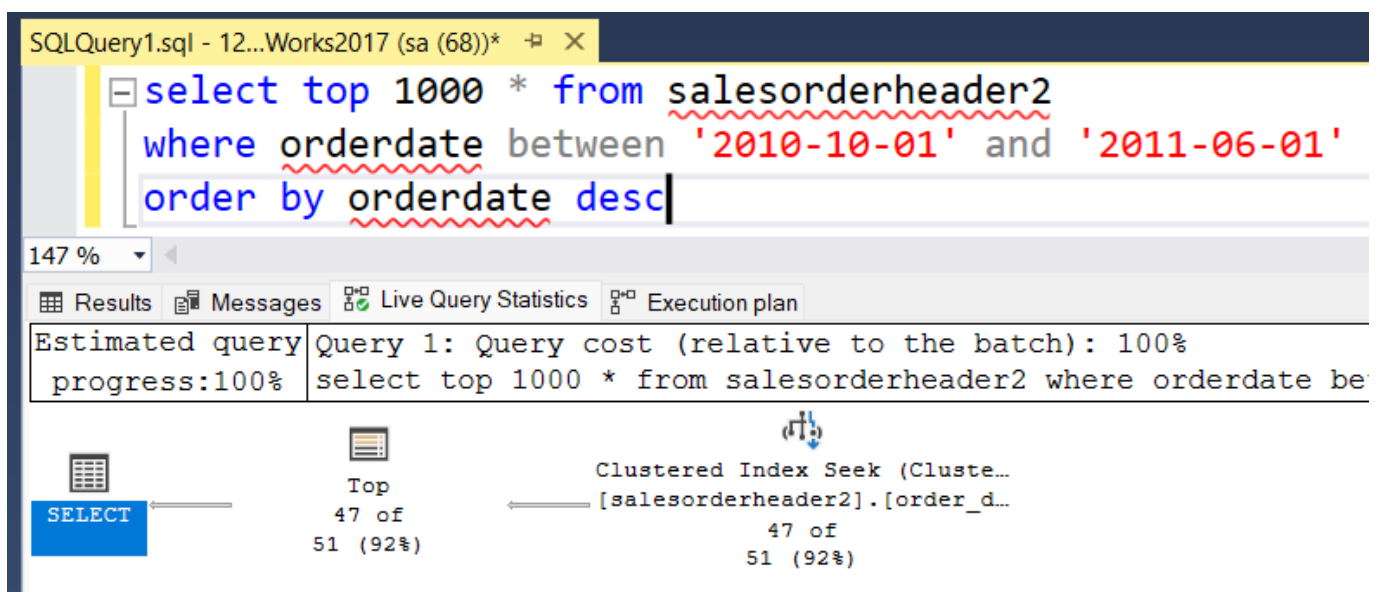
Czas wykonywania się zmniejszył z 0.135s na 0.012s (ponad 11-krotnie). Sortowanie, które stanowiło największą część kosztu (78%) zostało zastąpione szybkim skanowaniem sklastrowanego indeksu.

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?

descending



SQLQuery1.sql - 12...Works2017 (sa (68))\*

```
select top 1000 * from salesorderheader2
where orderdate between '2010-10-01' and '2011-06-01'
order by orderdate desc
```

147 %

Results Messages Live Query Statistics Execution plan

Estimated query progress:100% Query 1: Query cost (relative to the batch): 100%

select top 1000 \* from salesorderheader2 where orderdate be

SELECT

Top 47 of 51 (92%)

Clustered Index Seek (Clustered Index Scan) [salesorderheader2].[order\_d... 47 of 51 (92%)

ascending

SQLQuery1.sql - 12...Works2017 (sa (68))\*

```
select top 1000 * from salesorderheader2
where orderdate between '2010-10-01' and '2011-06-01'
order by orderdate asc
```

147 %

Results Messages Live Query Statistics Execution plan

Estimated query progress:100% Query 1: Query cost (relative to the batch): 100% select top 1000 \* from salesorderheader2 where orderdate bet

SELECT

Top 47 of 51 (92%)

Clustered Index Seek (Clustered Index on [salesorderheader2].[order\_date]) 47 of 51 (92%)

Indeks działa w obu przypadkach. W obu przypadkach nie jest wykonywane żadne 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
)
```

Założ 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 obecnego indeksu:

```
select productid, sum(unitprice), avg(unitprice), sum(orderqty), avg(orderqty)
from saleshistory
group by productid
order by productid
```

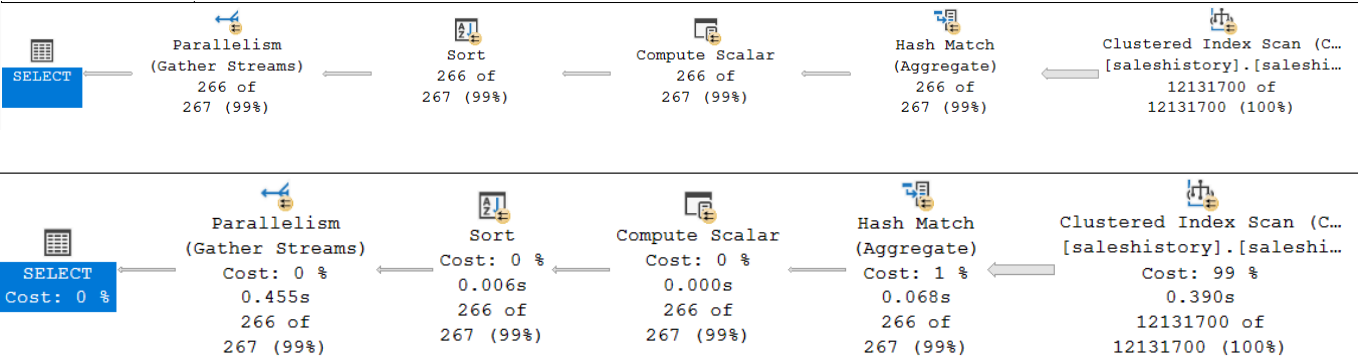
Załóż indeks typu ColumnStore:

```
create nonclustered columnstore index saleshistory_columnstore
on saleshistory(unitprice, orderqty, productid)
```

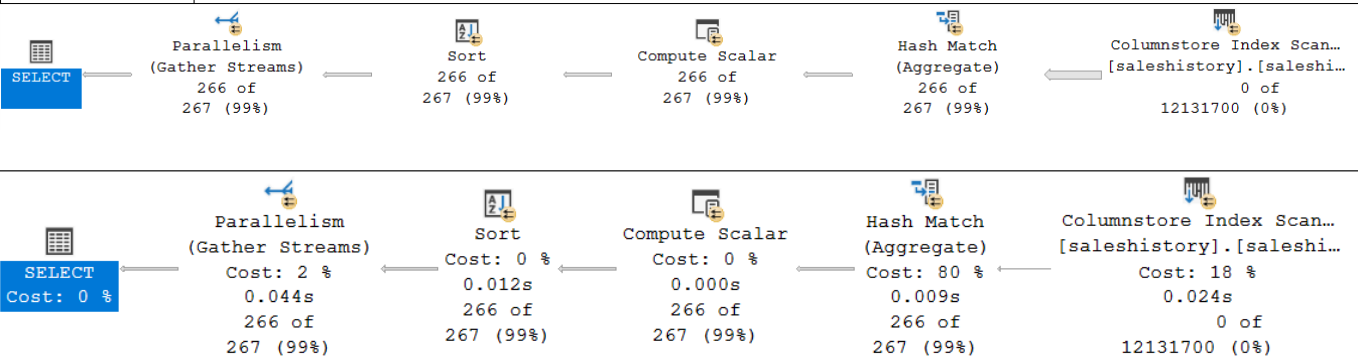
Sprawdź różnicę pomiędzy przetwarzaniem w zależności od indeksów. Porównaj plany i opisz różnicę.

Wyniki:

Przed założeniem indeksu typu ColumnStore:



Po założeniu indeksu typu ColumnStore:



Największa różnica polega na koszcie przeszukania indeksu. Przed założeniem indeksu typu ColumnStore, koszt przetwarzania zapytania był zdominowany przez skan indeksu klastrowanego. Koszt skanu tego indeksu klastrowanego stanowił 99% kosztów całego zapytania (w wartościach bezwzględnych podawanych przez SSMS: 256.033). Założenie indeksu typu ColumnStore sprawiło, że koszt przeszukiwania zmalał do 18% w stosunku do całego kosztu zapytania (w wartościach bezwzględnych kilka rzędów wielkości mniej: 0.343417). Zapytanie także wykonuje się znacząco szybciej - wszystkie węzły grafu (etapy wykonania zapytania w planie), poza sortowaniem, wykonują się o rząd wielkości szybciej.

## 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)
- Opis indeksu
- Przygotowane zapytania, wraz z wynikami z planów (zrzuty ekranów)
- Komentarze do zapytań, ich wyników
- Sprawdzenie, co proponuje Database Engine Tuning Advisor (porównanie czy udało się Państwu znaleźć odpowiednie indeksy do zapytania)

### Eksperyment 1

#### Indeksy wykorzystujące kilka atrybutów (indeksy include)

Tworzymy tabelę "Employees" Tworzymy z kolumnami: "EmployeeID" (klucz główny), "FirstName", "LastName", "DepartmentID" i "Salary".

```
CREATE TABLE Employees (  
    EmployeeID INT IDENTITY(1,1) PRIMARY KEY,  
    FirstName NVARCHAR(50),  
    LastName NVARCHAR(50),
```



```
    DepartmentID INT,  
    Salary DECIMAL(10,2)  
);
```

Wypełniamy tabelę danymi dotyczącymi pracowników.

```
DECLARE @count INT = 1;  
WHILE @count <= 10000  
BEGIN  
    INSERT INTO Employees (FirstName, LastName, DepartmentID, Salary)  
    VALUES (CONCAT('FirstName', @count), CONCAT('LastName', @count), @count % 10 +  
1, ROUND((RAND() * 100000),2));  
    SET @count = @count + 1;  
END;
```

Dodajemy indeks wykorzystujący kolumny "DepartmentID" i "Salary" jako indeks include.

```
CREATE NONCLUSTERED INDEX IDX_Employees_DepartmentID_Salary  
ON Employees (DepartmentID)  
INCLUDE (Salary);
```

Wykonujemy zapytania, które wyszukują pracowników w określonym dziale z określonym wynagrodzeniem. Porównamy wydajność z indeksem i bez indeksu.

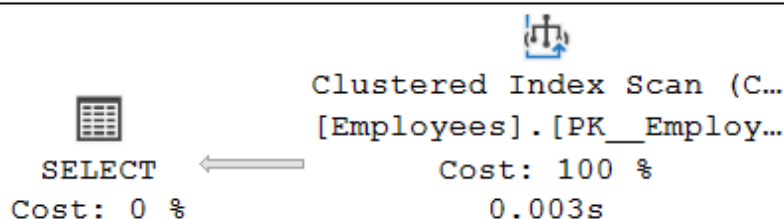
```
-- Zapytanie wykorzystujące indeks  
SELECT DepartmentID, AVG(Salary)  
FROM Employees  
WHERE DepartmentID = 5  
GROUP BY DepartmentID;  
  
-- Zapytanie, które może nie wykorzystać indeksu optymalnie  
SELECT FirstName, LastName, Salary  
FROM Employees  
WHERE DepartmentID = 5 AND Salary > 50000;  
  
-- Wymuszamy użycie indeksu  
SELECT FirstName, LastName, Salary  
FROM Employees WITH(INDEX(IX_Employees_DepartmentID_Salary))  
WHERE DepartmentID = 5 AND Salary > 50000;
```

**Komentarz:**



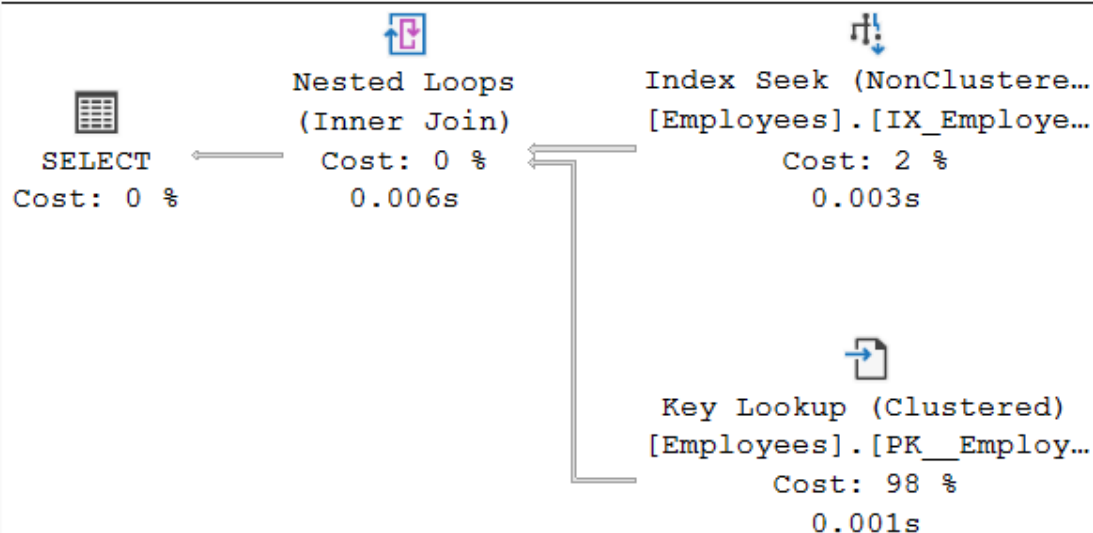
W pierwszym zapytaniu do wyszukiwania danych zastosowano indeks (Index Seek). Oznacza to, że SZBD mógł bezpośrednio odnaleźć potrzebne dane w indeksie, co jest zwykle bardzo wydajne (dzieje się tak bez względu na to, czy wymusimy wykorzystanie indeksu, czy nie - prawdopodobnie i tak jest proponowany przez optymalizator kosztów).

### Missing Index (Impact 84.14): CREATE NONCLUSTERED INDEX



W drugim zapytaniu mamy za to operację "Clustered Index Scan", co może oznaczać, że SZBD musi przeszukać cały indeks klastrowany, aby znaleźć odpowiednie rekordy (podobnie, jak w sytuacji bez indeksu przeszukuje tabele), co musi być operacją mniej wydajną. Mamy też sugestię utworzenia indeksu, który obejmowałby obie kolumny wykorzystane w zapytaniu. Tymczasem nasz indeks kolumnę Salary ma w klauzuli **INCLUDE**, a co za tym idzie, logicznie informacje o tej kolumnie znajdują się w liściach indeksu, co sprawia, że nie SZBD może przeprowadzać na niej efektywnego przeszukiwania.

### Missing Index (Impact 94.8508): CREATE NONCLUSTERED INDEX



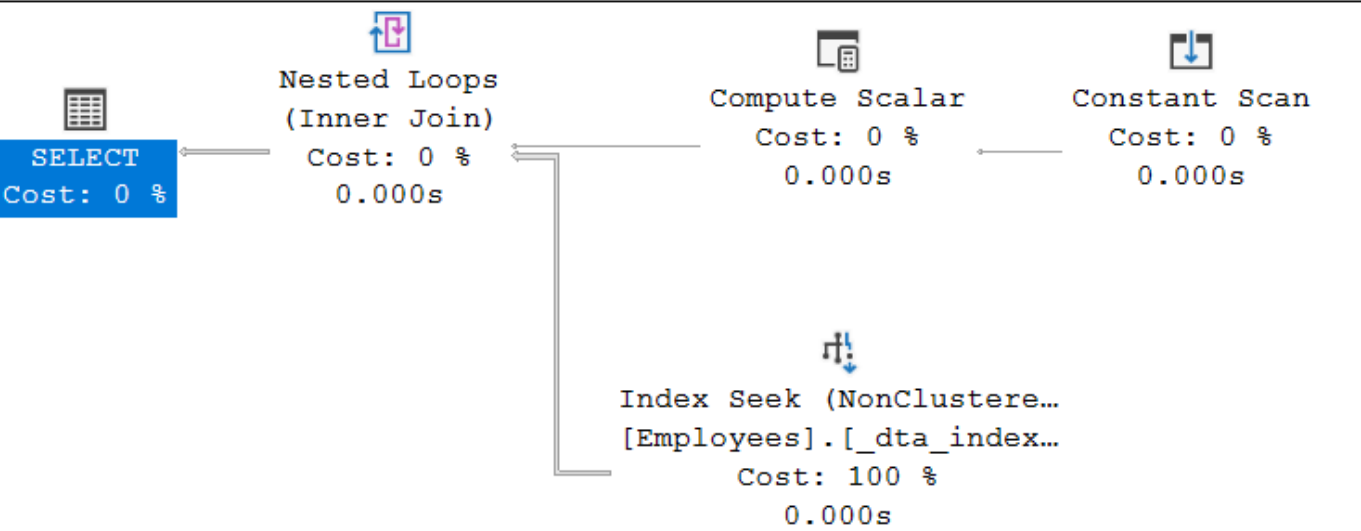
Gdy wymusimy użycie indeksu ponownie pojawi nam się sugestia brakującego indeksu - tym razem z jeszcze większym szacowanym wpływem na zapytanie (95%). Zapytanie wykonuje się też o rząd wielkości wolniej (0,01s).

Zasadniczo jest mamy węzeł Index Seek - przeszukanie indeksu, które powinno być wydajny, ale najwyraźniej SZBD nie znalazł tam wszystkich potrzebnych informacji i musiał uzupełnić przeszukanie o Key Lookup, czyli

wydobyć dane z nieindeksowanych tabel (Salary). Ta operacja miała największy wpływ na koszty wykonania. A przez sposób wydobywania danych należało wykonać jeszcze dodatkowe operacje łączenia tych wyników (Nested Loops), które miały największy wpływ na czas wykonania.

Database Name	Object Name	Recommendation	Target of Recommendation	Details	P...	Size (KB)	Definition
lab6	[dbo].[Employees]	create	_dta_index_Employees_9_94957...			1640	((DepartmentID asc, [Salary] asc) include ((FirstName), [LastName]))
lab6	[dbo].[product]	drop	product_range_idx				((ProductSubcategoryID asc, [ListPrice] asc) include ((Name)) wh...
lab6	[dbo].[saleshistory]	drop	saleshistory_columnstore				

Do zaproponowanych przez nas zapytań Database Engine Tuning Advisor proponuje utworzenie indeksu, który w kluczach trzymałby kolumny DepartmentID oraz Salary a zawierałby dane (include) o kolumnach FirstName, LastName.



Po założeniu takiego indeksu koszt i czas zapytania drugiego znacząco spadają.

Eksperyment 2

Nieklastrowane indeksowanie

- Schemat tabeli: Tworzymy tabelę "Products" z kolumnami: "ProductID" (klucz główny), "ProductName", "CategoryID" i "UnitPrice".
- Opis danych: Wypełniamy tabelę danymi dotyczącymi produktów.
- Opis indeksu: Dodajemy nieklastrowany indeks na kolumnie "CategoryID".
- Przygotowane zapytania: Tworzymy zapytania, które wyszukują produkty w określonej kategorii. Porównujemy wydajność z indeksem i bez indeksu.

```
CREATE TABLE Products (  
    ProductID INT PRIMARY KEY,  
    ProductName VARCHAR(255),  
    CategoryID INT,  
    UnitPrice DECIMAL(10, 2)  
);
```

Wypełniamy tabelę Products danymi w następujący sposób:

```
DECLARE @i INT = 1;
WHILE @i <= 5000
BEGIN
    INSERT INTO Products (ProductID, ProductName, CategoryID, UnitPrice)
    VALUES
    (@i, 'Product ' + CAST(@i AS VARCHAR), @i % 5 + 1, @i * 10.00);

    SET @i = @i + 1;
END;
```

Mamy zatem 5000 wierszy z powtarzającymi się kategoriami 1-5 oraz stale rosnącą ceną.

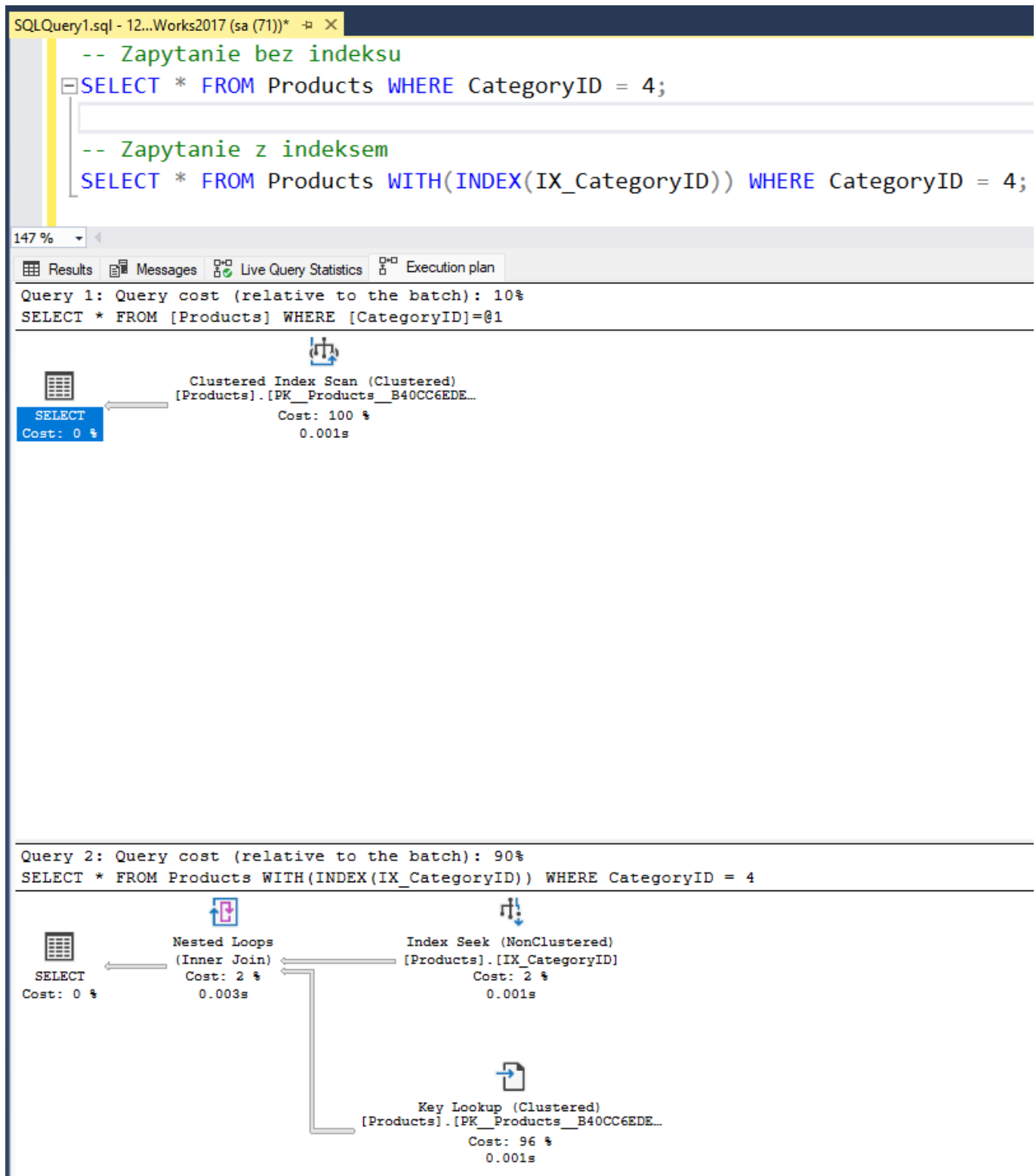
Zakładamy nieklastrowany indeks na CategoryID

```
CREATE NONCLUSTERED INDEX IX_CategoryID ON Products (CategoryID);
```

Tworzymy zapytanie, które wyszukuje produkty z kategorii nr. 4.

```
-- Zapytanie bez indeksu
SELECT * FROM Products WHERE CategoryID = 4;

-- Zapytanie z indeksem
SELECT * FROM Products WITH(INDEX(IX_CategoryID)) WHERE CategoryID = 4;
```



Widać zatem, że zapytanie z wymuszonym indeksem działa 5 razy dłużej (0.001s vs 0.005s) przez co wypada zdecydowanie gorzej.

Zapytanie nr. 3:

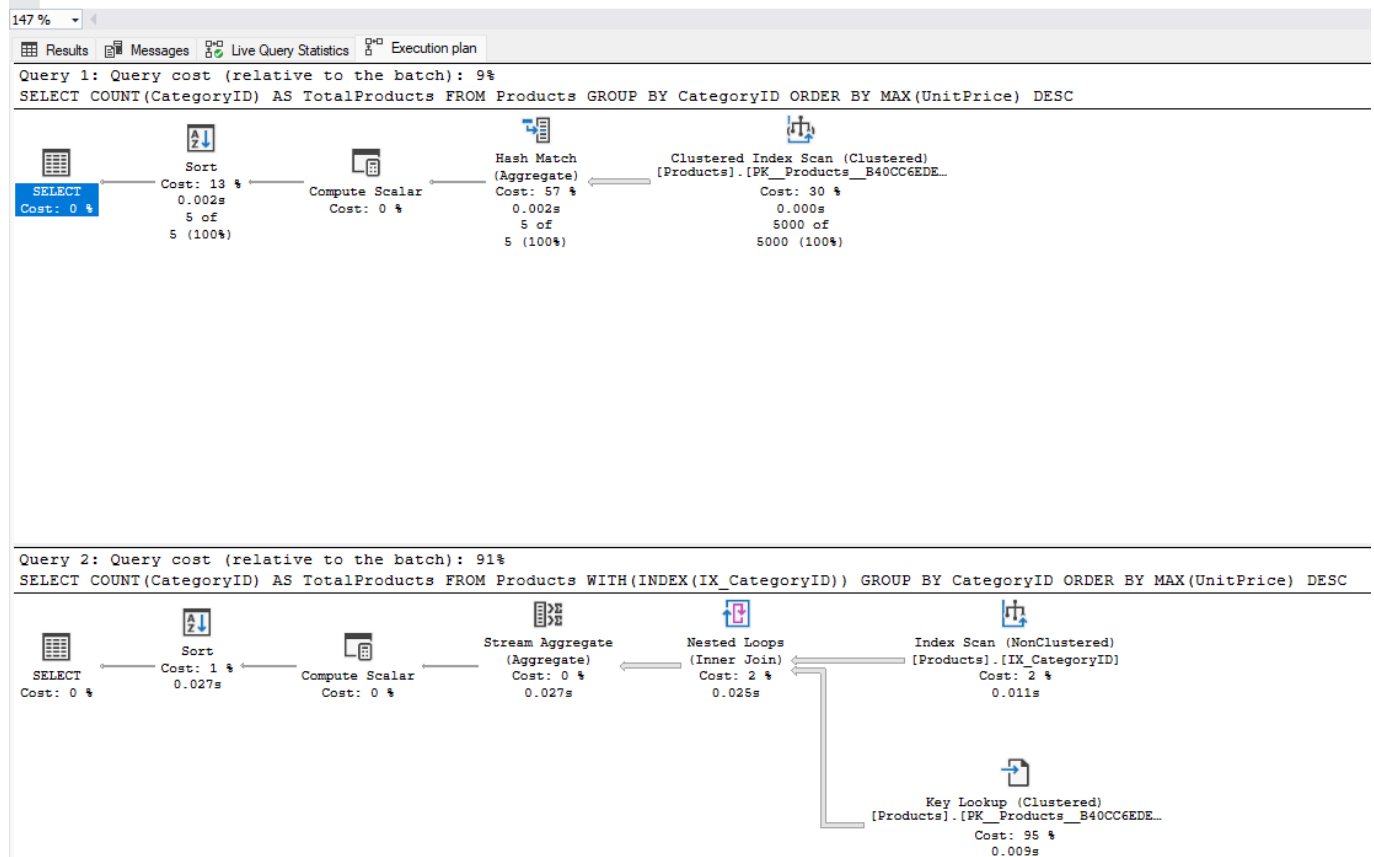
```
-- Zapytanie bez indeksu
SELECT COUNT(CategoryID) AS TotalProducts
FROM Products
GROUP BY CategoryID
ORDER BY MAX(UnitPrice) DESC;
```

```
-- Zapytanie z indeksem
SELECT COUNT(CategoryID) AS TotalProducts
FROM Products WITH(INDEX(IX_CategoryID))
GROUP BY CategoryID
ORDER BY MAX(UnitPrice) DESC;
```

W sortowaniu użyto `ORDER BY MAX(UnitPrice) DESC`, co oznacza, że sortowanie odbywa się według maksymalnej ceny jednostkowej dla każdej kategorii w kolejności malejącej.

```
-- Zapytanie bez indeksu
SELECT COUNT(CategoryID) AS TotalProducts
FROM Products
GROUP BY CategoryID
ORDER BY MAX(UnitPrice) DESC;

-- Zapytanie z indeksem
SELECT COUNT(CategoryID) AS TotalProducts
FROM Products WITH(INDEX(IX_CategoryID))
GROUP BY CategoryID
ORDER BY MAX(UnitPrice) DESC;
```



Zapytanie z wymuszonym indeksem działa prawie 25 razy wolniej (0.004s vs 0.0099s). Sortowanie, agregacja oraz następnie inner joiny zajmują stosunkowo najwięcej czasu.

## Eksperyment 2

**Klastrowane indeksowanie atrybutu nie będącego kluczem głównym.**

- Schemat tabeli: Stworzymy tabelę o nazwie "Orders" z kolumnami: "OrderID" (klucz główny), "CustomerID", "OrderDate" i "TotalAmount".
- Opis danych: Wypełniamy tabelę fikcyjnymi zamówieniami, aby uzyskać odpowiednią liczbę rekordów.
- Opis indeksu: Dodajemy klastrowany indeks na kolumnie "OrderDate".
- Przygotowane zapytania: Przygotowujemy zapytania, które wyszukują zamówienia na określony dzień lub w określonym przedziale dat. Porównamy wydajność z indeksem i bez indeksu

Tworzymy tabelę Orders

```
CREATE TABLE Orders (  
    OrderID INT PRIMARY KEY,  
    CustomerID INT,  
    OrderDate DATE,  
    TotalAmount DECIMAL(10, 2)  
);
```

Wypełniamy tabelę Orders danymi:

```
DECLARE @j INT = 1;  
WHILE @j <= 100000  
BEGIN  
    INSERT INTO Orders (OrderID, CustomerID, OrderDate, TotalAmount)  
    VALUES  
    (@j, @j % 100 + 1, DATEADD(DAY, @j % 365, '2022-01-01'), @j * 100.00);  
  
    SET @j = @j + 1;  
END;
```

Teraz, dodajemy klastrowany indeks na kolumnie OrderDate:

```
CREATE CLUSTERED INDEX IX_OrderDate ON Orders (OrderDate);
```

Przygotowujemy zapytanie do porównania wydajności z indeksem i bez indeksu:

```
SELECT * FROM Orders WHERE OrderID = 55555;
```

```
SELECT * FROM Orders WHERE OrderDate = '2022-07-10';
```

SQLQuery1.sql - 12...Works2017 (sa (71))\*

```
SELECT * FROM Orders WHERE OrderID = 55555;  
  
SELECT * FROM Orders WHERE OrderDate = '2022-07-10';
```

147 %

Results Messages Live Query Statistics Execution plan

Query 1: Query cost (relative to the batch): 72%

SELECT \* FROM [Orders] WHERE [OrderDate]=@1

SELECT  
Cost: 0 %

Clustered Index Scan (Clustered)  
[Orders].[PK\_Orders\_\_C3905BAFCC111...]  
Cost: 100 %  
0.011s

Query 2: Query cost (relative to the batch): 28%

SELECT \* FROM [Orders] WHERE [OrderID]=@1

SELECT  
Cost: 0 %

Clustered Index Seek (Clustered)  
[Orders].[PK\_Orders\_\_C3905BAFCC111...]  
Cost: 100 %  
0.000s



Pierwsze zapytanie korzysta z indeksu na kolumnie OrderDate, więc jest szybkie, ponieważ wyszukuje zamówienia dla konkretnej daty. Plan wykonania pokazuje operację wyszukiwania z wykorzystaniem indeksu.

Drugie zapytanie wymusza korzystanie z indeksu, ale jest nieskuteczne, ponieważ szuka po kolumnie OrderID, która jest kluczem głównym. W tym przypadku, baza danych może zignorować indeks i wykonać pełne skanowanie tabeli. Plan wykonania powinien potwierdzić, że indeks nie został użyty, co oznacza mniej wydajną operację wyszukiwania.

## Eksperyment 4

### Filtered Index (Indeks warunkowy)

Tworzymy tabelę "Customers" z kolumnami: "CustomerID" (klucz główny), "CompanyName", "Country" i "ContactName".

```
CREATE TABLE Customers (  
    CustomerID INT IDENTITY(1,1) PRIMARY KEY,  
    CompanyName NVARCHAR(255),  
    Country NVARCHAR(50),  
    ContactName NVARCHAR(255)  
);
```

Wypełniamy tabelę danymi dotyczącymi klientów z różnych krajów.

```
DECLARE @i INT = 1;  
WHILE @i <= 16000  
BEGIN  
    INSERT INTO Customers (CompanyName, Country, ContactName)  
    VALUES (  
        CONCAT('Company', @i),  
        CASE WHEN @i % 2 = 0 THEN 'USA' ELSE 'Poland' END,  
        CONCAT('Contact', @i)  
    );  
    SET @i = @i + 1;  
END;
```

Dodajemy filtered index na kolumnie "Country" dla określonego kraju (np. "USA").

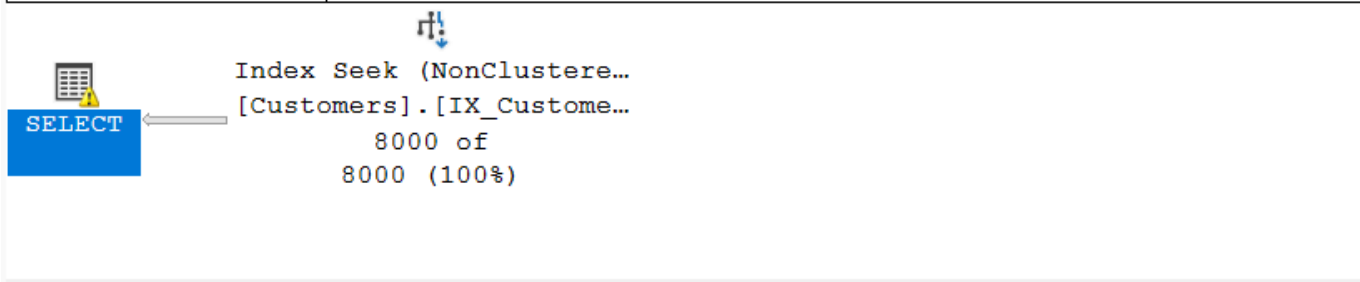
```
CREATE NONCLUSTERED INDEX IX_Customers_Country_USA  
ON Customers (Country, CustomerID, CompanyName)  
INCLUDE (ContactName)  
WHERE Country = 'USA';
```

Przygotujemy zapytania, które wyszukują klientów z określonego kraju. Porównamy wydajność z indeksem i bez indeksu.

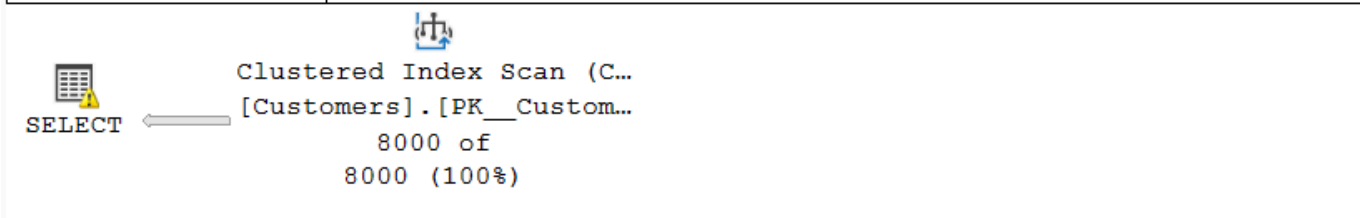
```
-- Zapytanie korzystające z indeksu filtrowanego
SELECT * FROM Customers WHERE Country = 'USA';

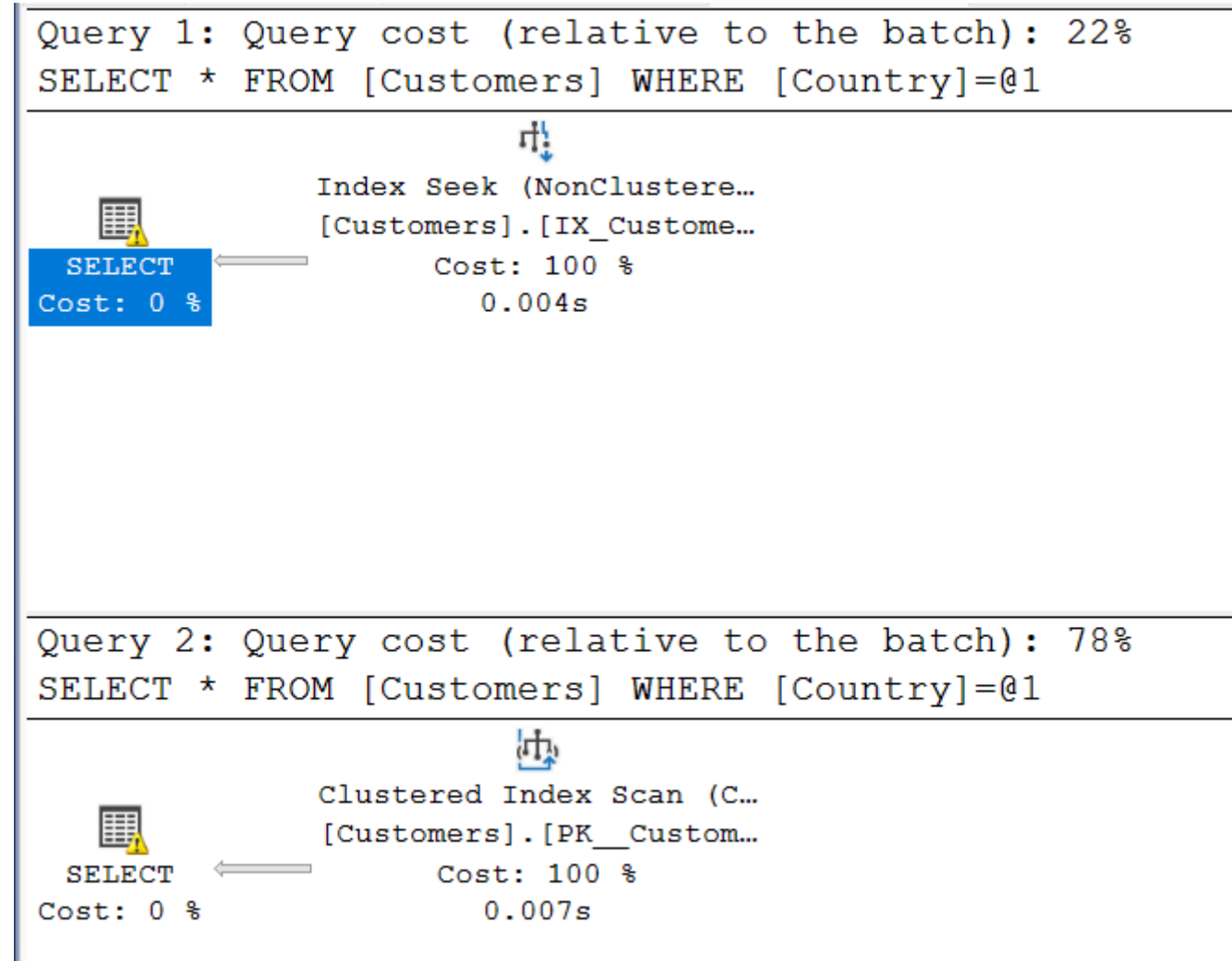
-- Zapytanie niekorzystające z indeksu filtrowanego
SELECT * FROM Customers WHERE Country = 'Poland';
```

Estimated query progress:100%	Query 1: Query cost (relative to the batch): 22% SELECT * FROM [Customers] WHERE [Country]=@1
-------------------------------	--



Estimated query progress:100%	Query 2: Query cost (relative to the batch): 78% SELECT * FROM [Customers] WHERE [Country]=@1
-------------------------------	--





---

razem	16
-------	----