# Indeksy, optymalizator
# Lab 2

**Imię i nazwisko: Wiktoria Zalińska, Magdalena Wilk**

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

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
    - ewentualnie inne narzędzie umożliwiające komunikację z MS SQL Server i analizę planów zapytań
- przykładowa baza danych AdventureWorks2017.

Oprogramowanie dostępne jest na przygotowanej maszynie wirtualnej

## Przygotowanie

Uruchom Microsoft SQL Managment Studio.

Stwórz swoją bazę danych o nazwie lab2.

```
create database lab2
go

use lab2
go
```

# Zadanie 1

Skopiuj tabelę `Person` do swojej bazy danych:

```
select businessentityid
      ,persontype
      ,namestyle
      ,title
      ,firstname
      ,middlename
      ,lastname
      ,suffix
      ,emailpromotion
      ,rowguid
      ,modifieddate
into person
from adventureworks2017.person.person
```

---

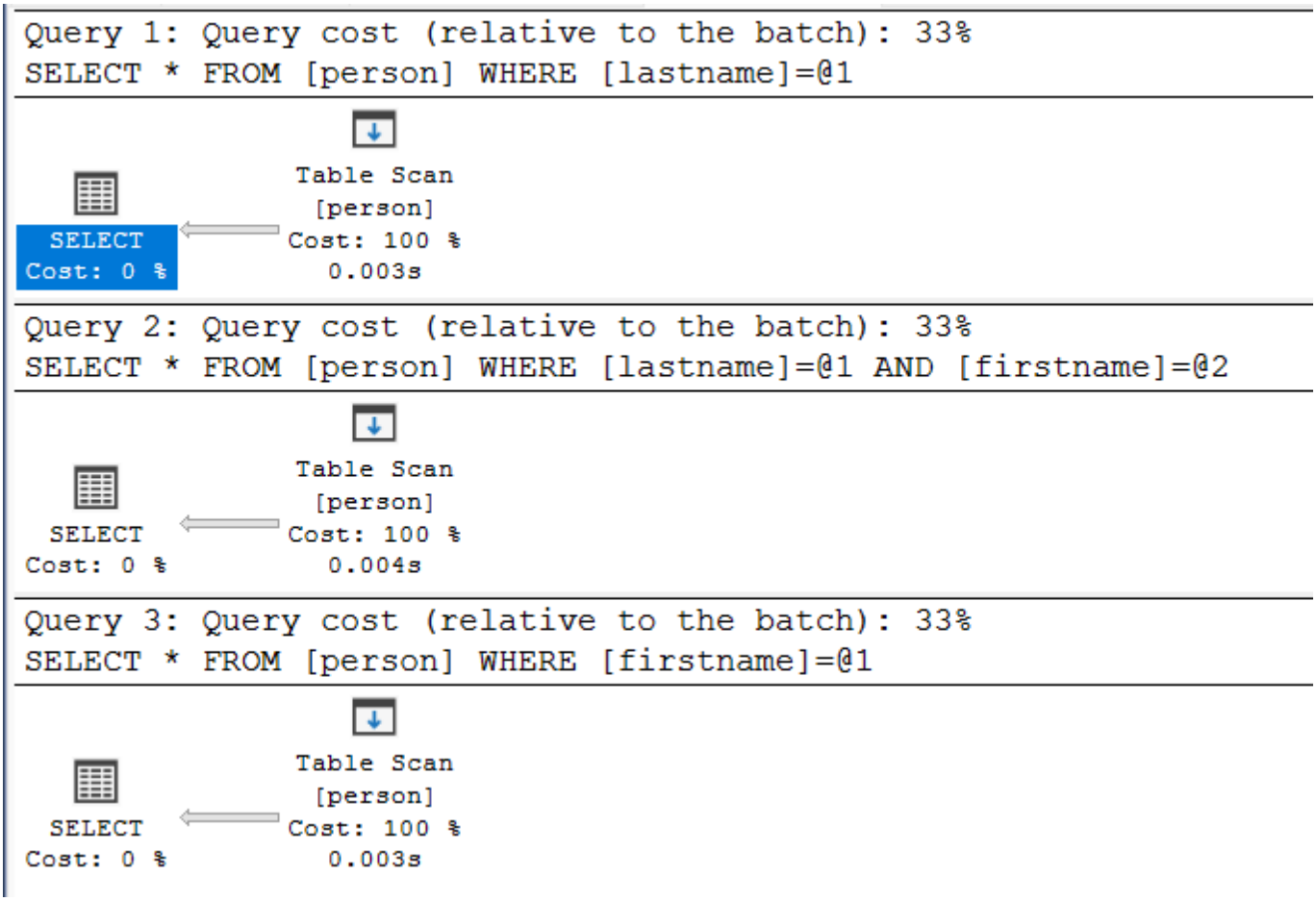
Wykonaj analizę planu dla trzech zapytań:

```
select * from [person] where lastname = 'Agbonile'

select * from [person] where lastname = 'Agbonile' and firstname = 'Osarumwense'

select * from [person] where firstname = 'Osarumwense'
```

Co można o nich powiedzieć?

---

> Wyniki:




Każde z zapytań dokonuje skanowania całej tabeli.

---

Przygotuj indeks obejmujący te zapytania:

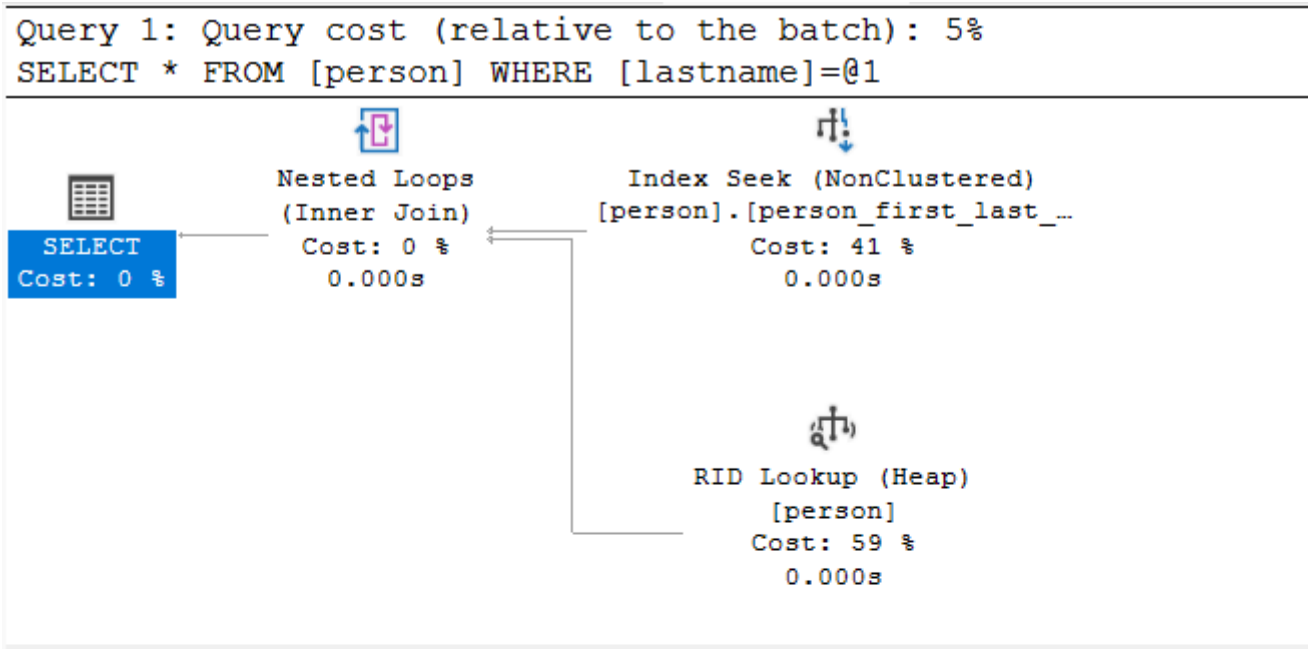
```
create index person_first_last_name_idx
on person(lastname, firstname);
```

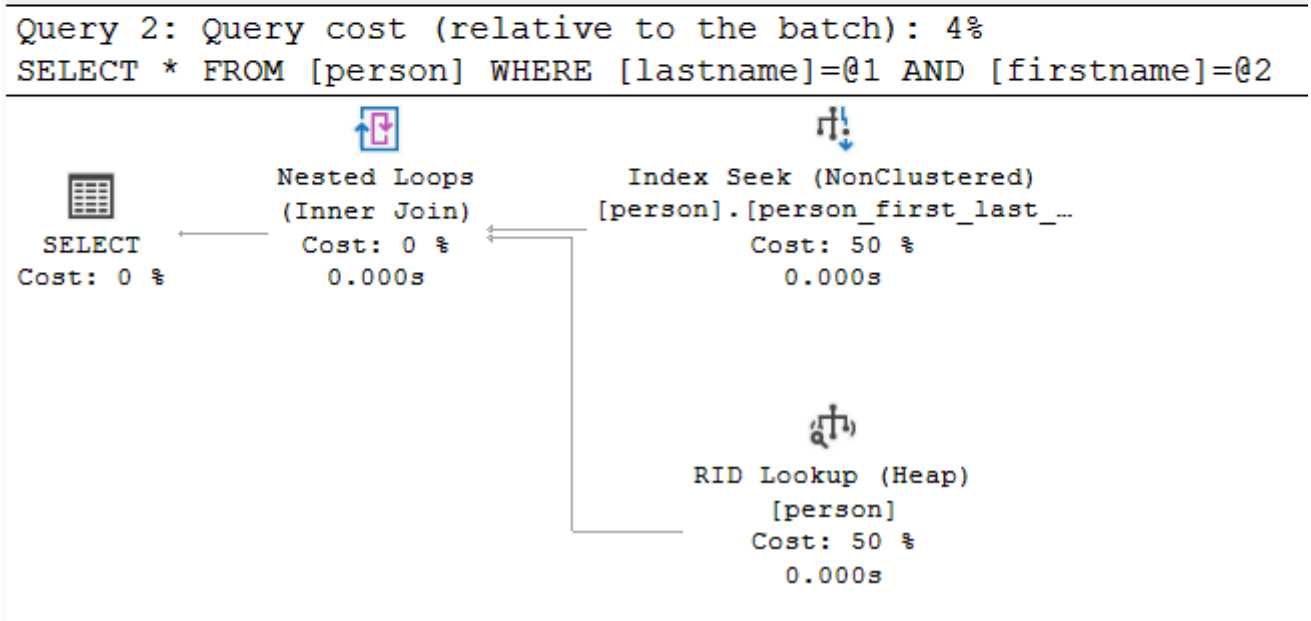
Sprawdź plan zapytania. Co się zmieniło?

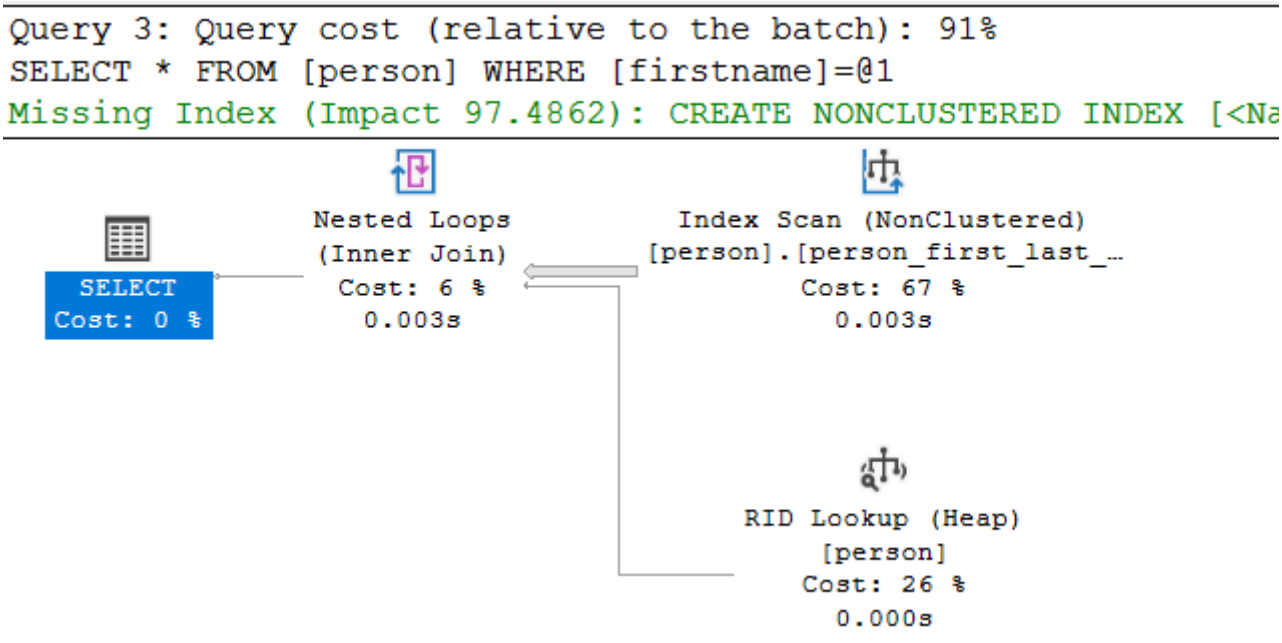
---

> Wyniki:

Plany zapytań po stworzeniu indeksu:




Po stworzeniu indeksu, dla pierwszego i drugiego zapytania tabela jest przeszukiwana po indeksie dla imienia i nazwiska (wykorzystywany jest Index Seek), ale w zapytania wywoływane są wszystkie kolumny, zatem po

wyszukaniu imienia, nazwiska po indeksie, przeglądana jest cała tabela - Rid Lookup w celu pobrania pozostałych kolumn.

W trzecim zapytaniu nie można użyć indeksu ponieważ indeks jest stworzony w kolejności lastname, firstname - a więc próbując wyszukiwać po samym firstname, nie ma dostępu do tego indeksu.

Ten indeks może zostać efektywnie użyty do zapytań z: lastname i firstname lub samym lastname.

---

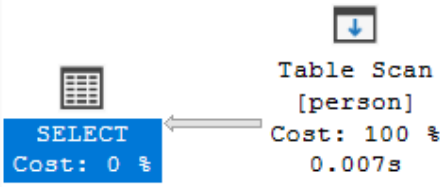
Przeprowadź ponownie analizę zapytań tym razem dla parametrów: `FirstName = 'Angela' LastName = 'Price'`. (Trzy zapytania, różna kombinacja parametrów).

Czym różni się ten plan od zapytania o `'Osarumwense Agbonile'` . Dlaczego tak jest?

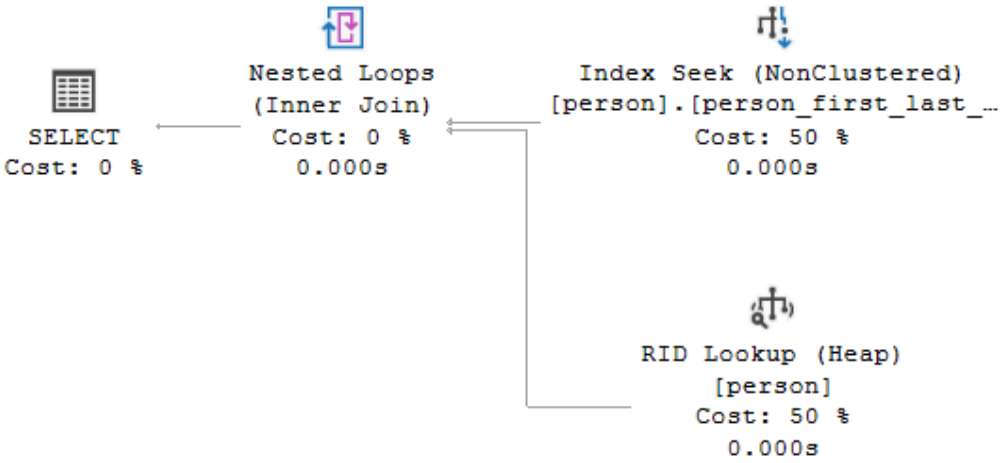
---

> Wyniki:

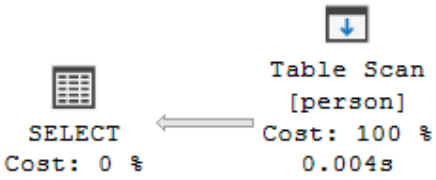
```
Query 1: Query cost (relative to the batch): 49%
SELECT * FROM [person] WHERE [lastname]=@1
```




```
Query 2: Query cost (relative to the batch): 2%
SELECT * FROM [person] WHERE [lastname]=@1 AND [firstname]=@2
```




```
Query 3: Query cost (relative to the batch): 49%
SELECT * FROM [person] WHERE [firstname]=@1
Missing Index (Impact 97.9617): CREATE NONCLUSTERED INDEX [<Name of Mi
```



Dla kombinacji parametrów: `FirstName = 'Angela' LastName = 'Price'`, indeks został użyty tylko w drugim przypdku (gdy w zapytaniu zostało ograniczone i imię i nazwisko do tych parametrów w `WHERE`). W przypadku ograniczania się tylko do imienia lub tylko do nazwiska, dokonywany jest zwyczajny skan tabeli.

Wynika to z tego, że dla `LastName = 'Price'` zostało znalezione bardzo dużo wyników - w tym przypadku skan tabeli okazał się efektywniejszy niż odczytywanie danych przez indeks i RID Lookup i optymalizator SQL Server wybrał bardziej opłacalną metodę dla tego zadania.

---

# Zadanie 2

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? Jak działają indeksy z warunkiem?

---

> Wyniki:

W pierwszym zapytaniu indeks został użyty, ale w drugim nie.

Wynika to z tego, że przy tworzeniu indeksu został zastosowany warunek, aby indeks został stworzony tylko dla productsubcategoryid z przedziału [27; 36] - a więc indeks może zostać użyty tylko wtedy gdy warunek

zapytania będzie zawierał się w zakresie tych wartości dla których stworzony jest indeks. Dla pozostałych danych indeks nie jest tworzony.

Zakres productsubcategoryid w pierwszym zapytaniu pokrywa się z zakresem indeksu, dlatego został on użyty, a w drugim przypadku zakres productsubcategoryid w zapytaniu jest poza zakresem warunku indeksu, dlatego nie został on tutaj użyty.

---

# Zadanie 3

Skopiuj tabelę `PurchaseOrderDetail` do swojej bazy danych:

```
select * into purchaseorderdetail
from  adventureworks2017.purchasing.purchaseorderdetail
```

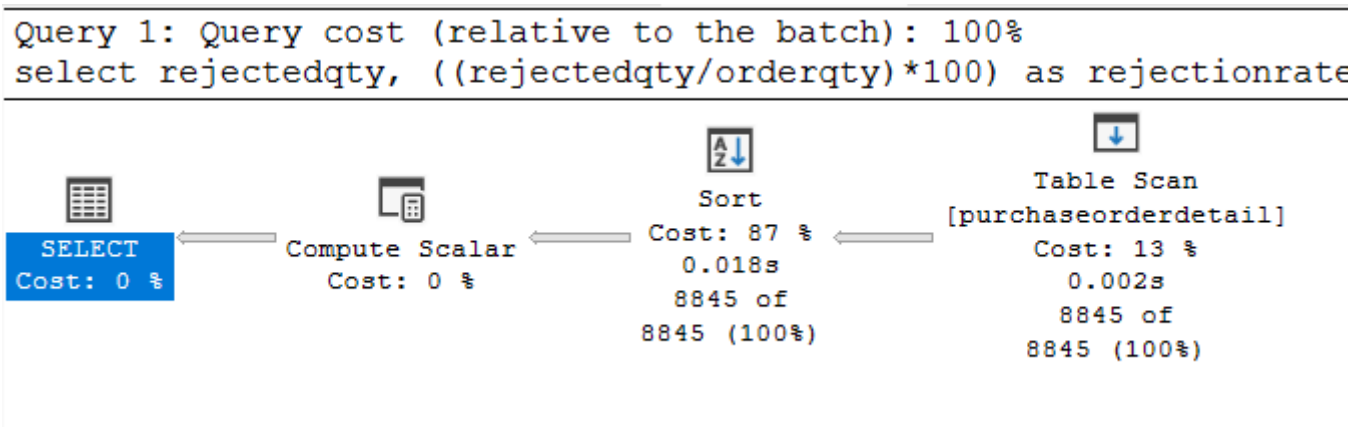
Wykonaj analizę zapytania:

```
select rejectedqty, ((rejectedqty/orderqty)*100) as rejectionrate,
  productid, duedate
from purchaseorderdetail
order by rejectedqty desc, productid asc
```

Która część zapytania ma największy koszt?

---

> Wyniki:




Największy koszt zapytania ma część odpowiadająca za sortowanie danych.

---

Jaki indeks można zastosować aby zoptymalizować koszt zapytania? Przygotuj polecenie tworzące index.

---

> Wyniki:

Aby zoptymalizować koszt tego zapytania, można zastosować indeks klastrowany na kolumnach po których sortujemy: rejectedqty oraz productid - w porządu takim jakim sortujemy (desc, asc):

```
create clustered index rejectedqty_productid_idx
on purchaseorderdetail(rejectedqty desc, productid asc);
```

---

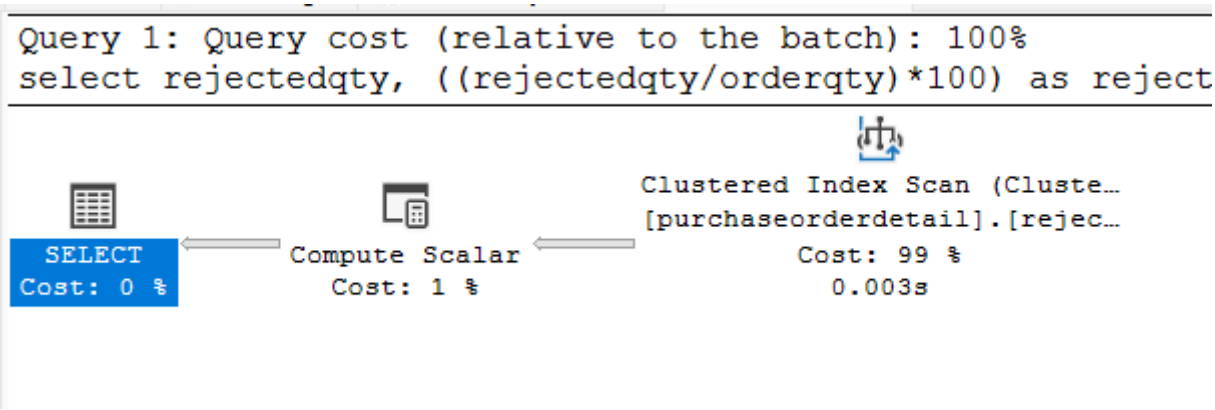
Ponownie wykonaj analizę zapytania:

---

> Wyniki:

W wyniku zastosowania indeksu koszt zapytania spadł z 0.534984 (gdzie 85% to sortowanie - 0.5341) do 0.0775997, gdzie 99% kosztu odpowiada kosztowi odczytu z indeksu.

W wyniku zastosowania ideksu, sortowanie zostało wyeliminowane, ponieważ dane są już w odpowiedniej kolejności.




---

# Zadanie 4 – 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 column store:

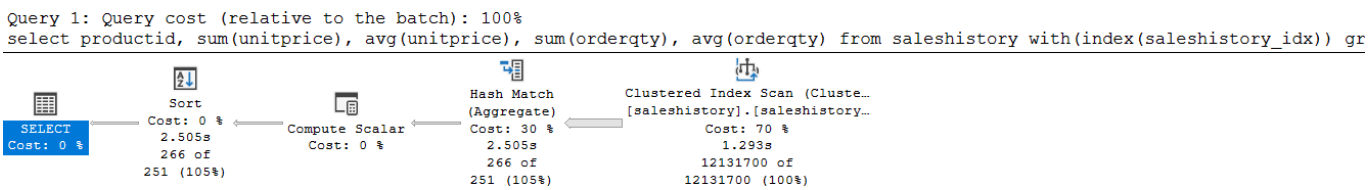
```
create nonclustered columnstore index saleshistory_columnstore
 on saleshistory(unitprice, orderqty, productid)
```

Sprawdź różnicę pomiędzy przetwarzaniem w zależności od indeksów. Porównaj plany i opisz różnicę. Co to są indeksy column store? Jak działają? (poszukaj materiałów w internecie/literaturze)

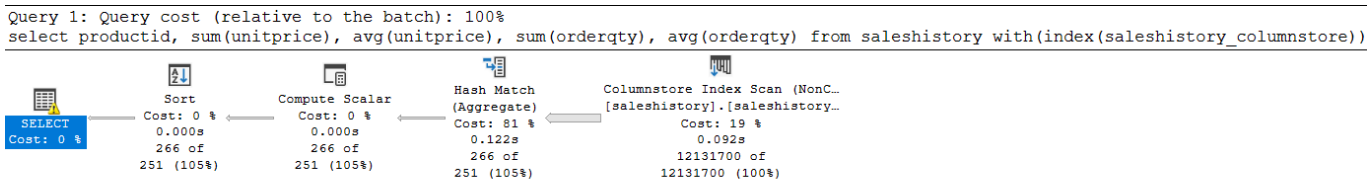
---

> Wyniki:

Plan zapytania przy użyciu pierwszego indeksu:



Plan zapytania przy użyciu indeksu kolumnowego:




Zapytanie z drugim indeksem jest o wiele szybsze i mniej kosztowne niż pierwsze. W pierwszym najbardziej kosztowną operacją było odczytywanie danych z tabeli, w drugim przypadku indeks uprościł odczyt, tak że najbardziej kosztowne są operacje agregujące.

Porównanie czasów i kosztów zapytań

| | czas [ms] | koszt |
|---|---|---|
| indeks 1 | 2706 | 187.89 |
| indeks 2 | 361 | 7.0192 |

**Indeksy column store**

Indeks kolumnowy column store przechowuje dane po kolumnach zamiast po wierszach, co umożlwiia efektywną kompresję i dostęp do danych podczas zapytań (tylko potrzebne kolumny są odczytywane z dysku). Są zoptymalizowane pod kątem zapytań analitycznych i dużych operacji agregujących.

---

# Zadanie 5 – 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 ekranow)
- Komentarze do zapytań, ich wyników
- Sprawdzenie, co proponuje Database Engine Tuning Advisor (porównanie czy udało się Państwu znaleźć odpowiednie indeksy do zapytania)

> Wyniki:

Tworzymy tabelę `EmployeeSalaryHistory` i wypełniamy ją przykładowymi danymi. Klucz główny nie może być indeksem klastrowym, jeśli chcemy stworzyć indeks klastrowy na innym atrybucie. Tabela ma 100000 wierszy.

```
CREATE TABLE EmployeeSalaryHistory (
    Id INT IDENTITY NOT NULL,
    EmployeeId INT,
    DepartmentId INT,
    JobTitle NVARCHAR(100),
    Salary DECIMAL(10,2),
    CurrencyCode CHAR(3),
    StartDate DATE,
    EndDate DATE,
    IsCurrent BIT,
    CONSTRAINT PK_Employee_Id PRIMARY KEY NONCLUSTERED (Id)
);


-- generowanie danych
DECLARE @i INT = 0;

WHILE @i < 100000
BEGIN
    INSERT INTO EmployeeSalaryHistory (
        EmployeeId, DepartmentId, JobTitle, Salary,
        CurrencyCode, StartDate, EndDate, IsCurrent
    )
    VALUES (
        ABS(CHECKSUM(NEWID())) % 1000 + 1,
        ABS(CHECKSUM(NEWID())) % 20 + 1,
        CONCAT('Job_', ABS(CHECKSUM(NEWID())) % 10 + 1),
        ROUND(RAND() * 5000 + 3000, 2),
        'USD',
        DATEADD(DAY, -1 * (ABS(CHECKSUM(NEWID())) % 3000), GETDATE()),
        NULL,
        CASE WHEN RAND() > 0.7 THEN 1 ELSE 0 END
    );
    SET @i += 1;
END
```

| | Id | EmployeeId | DepartmentId | JobTitle | Salary | CurrencyCode | StartDate | EndDate | IsCurrent |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 297 | 17 | Job_5 | 4019.14 | USD | 2022-08-22 | NULL | 1 |
| 2 | 2 | 183 | 3 | Job_3 | 6072.27 | USD | 2023-10-31 | NULL | 1 |
| 3 | 3 | 752 | 10 | Job_5 | 5639.73 | USD | 2018-10-30 | NULL | 1 |
| 4 | 4 | 72 | 18 | Job_1 | 4793.36 | USD | 2018-05-19 | NULL | 1 |
| 5 | 5 | 727 | 11 | Job_5 | 6100.50 | USD | 2019-01-23 | NULL | 0 |
| 6 | 6 | 367 | 11 | Job_9 | 4074.51 | USD | 2018-01-25 | NULL | 0 |
| 7 | 7 | 164 | 3 | Job_7 | 6947.36 | USD | 2017-08-08 | NULL | 0 |
| 8 | 8 | 162 | 4 | Job_5 | 4736.10 | USD | 2018-04-02 | NULL | 1 |
| 9 | 9 | 8 | 13 | Job_7 | 3527.59 | USD | 2021-03-11 | NULL | 0 |
| 10 | 10 | 101 | 12 | Job_8 | 4567.97 | USD | 2020-12-01 | NULL | 1 |
| 11 | 11 | 334 | 12 | Job_6 | 4816.14 | USD | 2020-07-28 | NULL | 0 |
| 12 | 12 | 94 | 5 | Job_8 | 6319.82 | USD | 2020-08-09 | NULL | 0 |
| 13 | 13 | 279 | 2 | Job_1 | 6283.85 | USD | 2021-05-16 | NULL | 0 |
| 14 | 14 | 101 | 7 | Job_10 | 4015.30 | USD | 2024-04-28 | NULL | 1 |
| 15 | 15 | 792 | 19 | Job_1 | 5313.71 | USD | 2021-06-17 | NULL | 0 |
| 16 | 16 | 783 | 14 | Job_9 | 5887.25 | USD | 2019-11-03 | NULL | 1 |
| 17 | 17 | 58 | 13 | Job_10 | 3038.73 | USD | 2023-10-19 | NULL | 0 |

EmployeeId to liczby z przedziału [1, 1000]. DepartmentId to liczby z [1, 20]. JobTitle jest kodowany przy użyciu 10 kategorii. IsCurrent ustawione na 1 znaczy, że jest to obecna pensja (około 30% wierszy).

| | Num_Of_Employees |
|---|---|
| 1 | 1000 |

| | Num_Of_Departments |
|---|---|
| 1 | 20 |

| | AVG_Salary | MIN_Salary | MAX_Salary |
|---|---|---|---|
| 1 | 5503.064428 | 3000.01 | 7999.93 |

Tworzymy następujące indeksy

```
-- 1. Indeks klastrowy
CREATE CLUSTERED INDEX Salary_EmployeeId_Idx
ON EmployeeSalaryHistory(EmployeeId);

--2. Indeks nieklastrowy
CREATE NONCLUSTERED INDEX Salary_StartDate_Idx
ON EmployeeSalaryHistory(StartDate);

--3. Indeks typu include
CREATE NONCLUSTERED INDEX Salary_Dept_Include_Idx
ON EmployeeSalaryHistory(DepartmentId) INCLUDE(Salary, JobTitle);

--4. Indeks zawierający kilka atrybutów
CREATE NONCLUSTERED INDEX Employee_Salary_Idx
ON EmployeeSalaryHistory(EmployeeId, StartDate);

--5. Indeks warunkowy
CREATE NONCLUSTERED INDEX CurrentEmployees_Idx
```

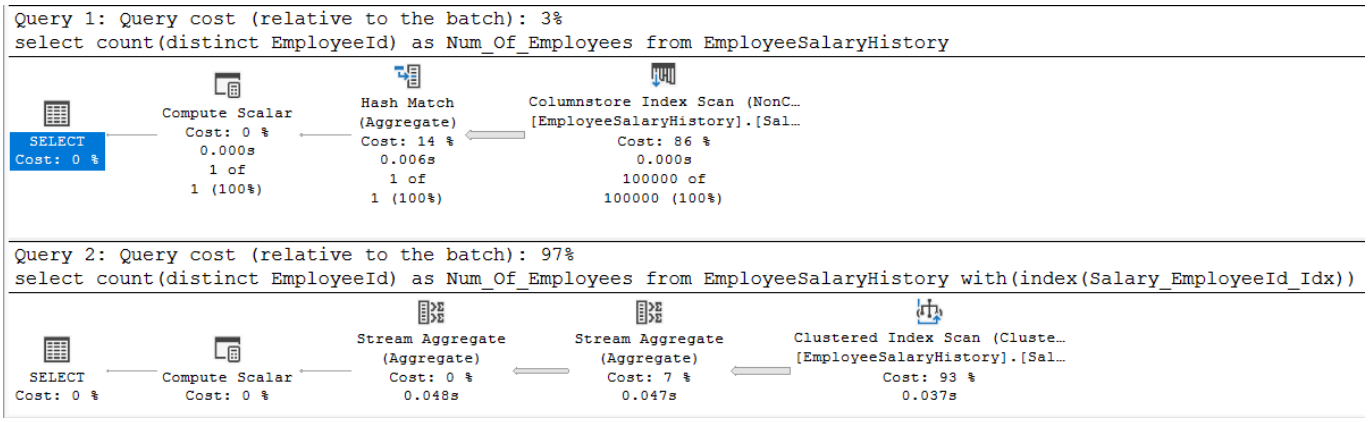
```
ON EmployeeSalaryHistory(IsCurrent) WHERE IsCurrent = 1;

--6. Indeks kolumnowy
CREATE NONCLUSTERED COLUMNSTORE INDEX Salary_Col_Idx
ON EmployeeSalaryHistory(Salary, EmployeeId, DepartmentId);
```

Rozmiary indeksów

| | indexname | size_kb |
|---|---|---|
| 1 | CurrentEmployees_Idx | 584 |
| 2 | Employee_Salary_Idx | 2256 |
| 3 | Salary_Col_Idx | 560 |
| 4 | Salary_Dept_Include_Idx | 4536 |
| 5 | Salary_EmployeeId_Idx | 6360 |
| 6 | Salary_StartDate_Idx | 2304 |

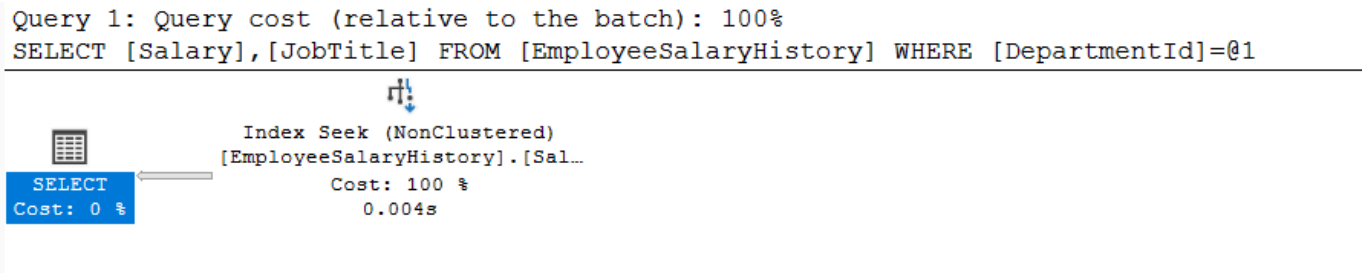
**Zapytanie 1** - SQL MS wykorzystuje `Salary_Col_Idx` (najbardziej optymalny). Po wymuszeniu `Salary_EmployeeId_Idx` koszt zapytania się zwiększa z 0.02 do ponad 0.7. Indeks kolumnowy jest w tym przypadku lepszy.

```
SELECT COUNT(DISTINCT EmployeeId) AS Num_Of_Employees
FROM EmployeeSalaryHistory;
```



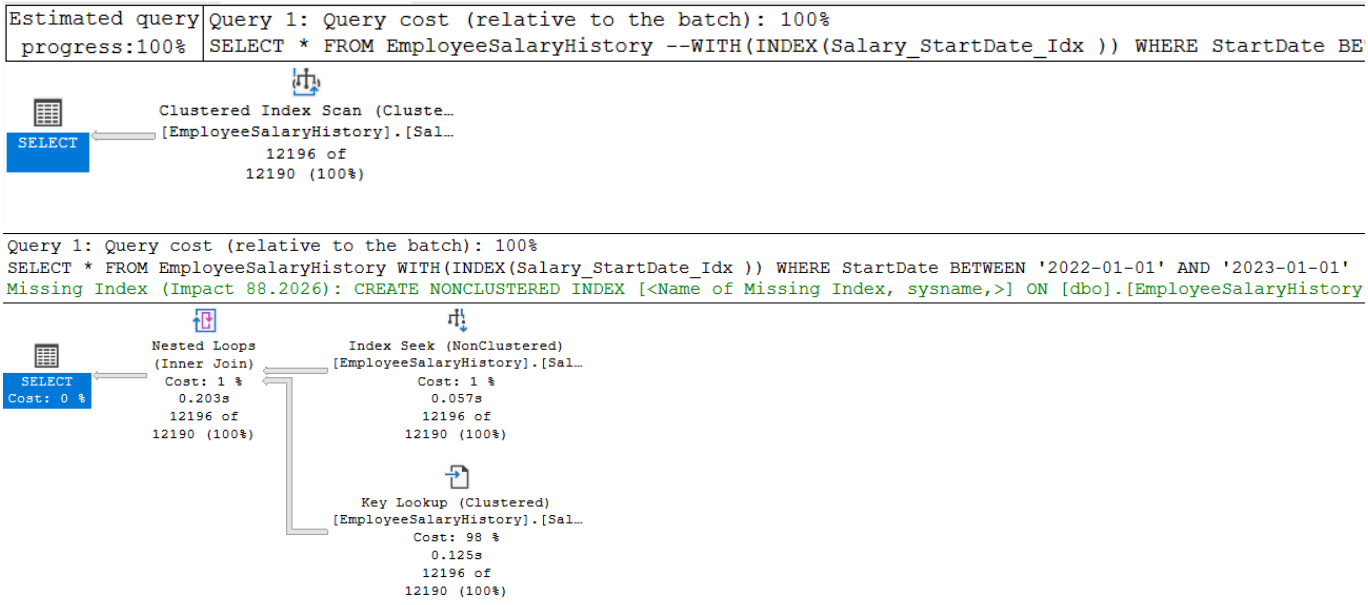

**Zapytanie 2** - wykorzystuje `Salary_Dept_Include_Idx`. Zapytanie wykonuje tylko Index Seek. Jego koszt to 0.027.

```
SELECT Salary, JobTitle
FROM EmployeeSalaryHistory
WHERE DepartmentId = 5;
```




**Zapytanie 3** - automatycznie wykorzystuje indeks klastrowy `Salary_EmployeeId_Idx`. Jeśli wymusimy `Salary_StartDate_Idx` zapytanie i taak będzie korzystać z indeksu klastrowego, aby pobrać wszystkie dane dla danych wierszy.

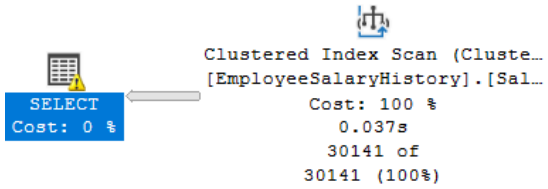
```
SELECT *
FROM EmployeeSalaryHistory --WITH(INDEX(Salary_StartDate_Idx ))
WHERE StartDate BETWEEN '2022-01-01' AND '2023-01-01';
```



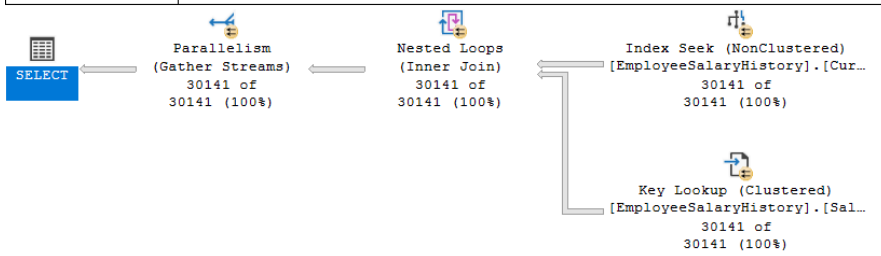

**Zapytanie 4** - mimo, że istnieje indeks `CurrentEmployees_Idx` korzysta automatycznie z indeksu klastrowego, ponieważ musi pobierać dane z innych kolumn niż IsCurrent. Po wymuszeniu efektywność drastycznie spada.

```
SELECT EmployeeId, Salary
FROM EmployeeSalaryHistory
WHERE IsCurrent = 1;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT [EmployeeId],[Salary] FROM [EmployeeSalaryHistory] WHERE [IsCurrent]=@1
Missing Index (Impact 82.8956): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
```



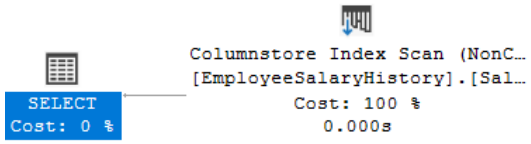

```
Estimated query progress:100%
Query 1: Query cost (relative to the batch): 100%
SELECT EmployeeId, Salary FROM EmployeeSalaryHistory with(index(CurrentEmployees_Idx)) WHERE IsCurrent = 1
Missing Index (Impact 97.8326): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Emp
```




**Zapytanie 5** - wykorzystuje indeks kolumnowy, który idealnie nadaje się do tego zapytania, ponieważ zawiera wszystkie kolumny, które są użyte w zapytaniu.

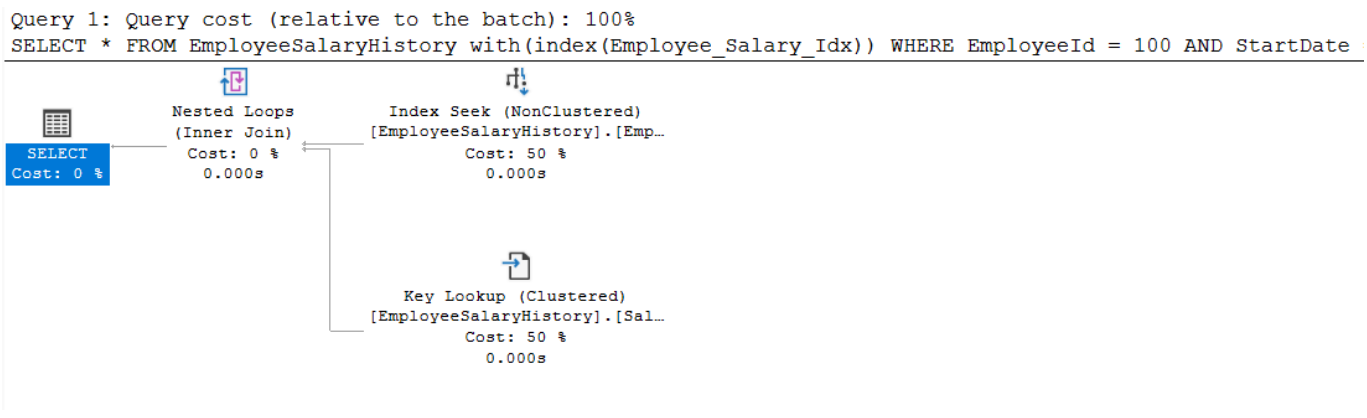
```sql
SELECT EmployeeId, DepartmentId, Salary
FROM EmployeeSalaryHistory
WHERE Salary > 12000;
```

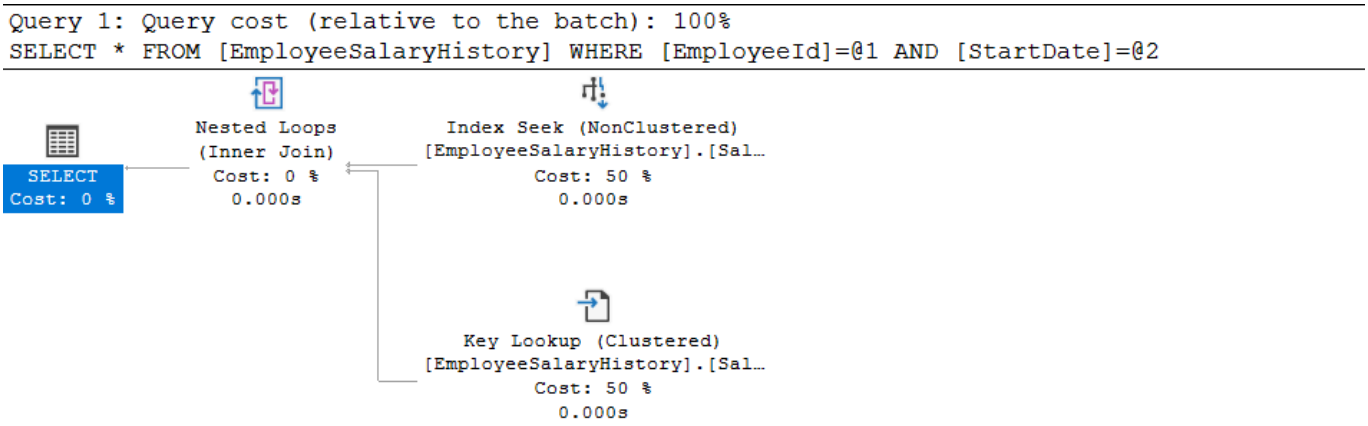
```
Query 1: Query cost (relative to the batch): 100%
SELECT [EmployeeId],[DepartmentId],[Salary] FROM [EmployeeSalaryHistory] WHERE [Salary]>@1
```




**Zapytanie 6** - wykorzystuje automatycznie `Salary_StartDate_Idx`. Po wymuszeniu `Employee_Salary_Idx` plan zapytania się nie zmienia, koszt pozostaje taki sam i wynosi 0.0066, a czas maleje z 2ms do 1ms.

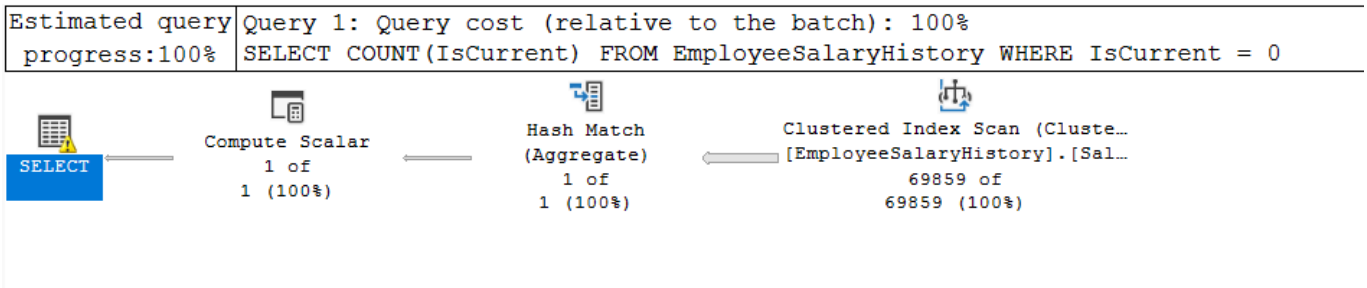
```sql
SELECT *
FROM EmployeeSalaryHistory with(index(Employee_Salary_Idx))
WHERE EmployeeId = 100 AND StartDate = '2021-01-01';
```




**Zapytanie 7** - wykorzystuje indeks klastrowy, ponieważ w tym przypadku nie można użyć indeksu warunkowego - nie działa na dopełnienie warunku.

```sql
SELECT COUNT(IsCurrent)
FROM EmployeeSalaryHistory
WHERE IsCurrent = 0;
```



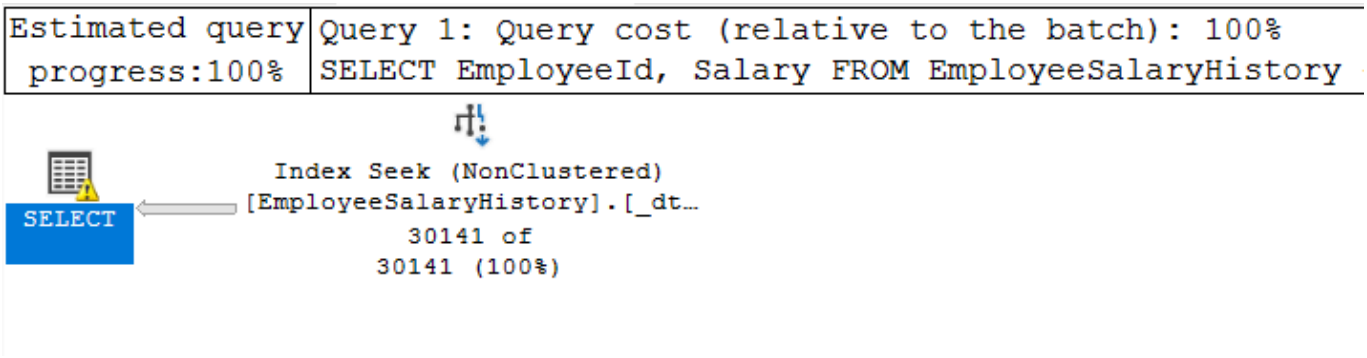

**Database Engine Tuning Advisor** - po analizie zaproponował stworzenie dwóch dodatkowych indeksów

```sql
use [lab4]
go

CREATE NONCLUSTERED INDEX [_dta_index_EmployeeSalaryHistory_5_1237579447__K9_2_5]
ON [dbo].[EmployeeSalaryHistory]
(
    [IsCurrent] ASC
)
```

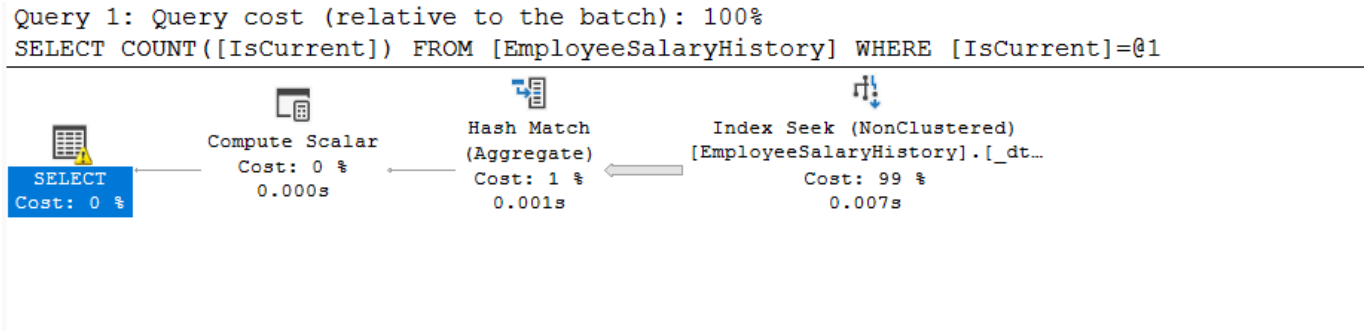
```
INCLUDE([EmployeeId],[Salary])
  WITH (SORT_IN_TEMPDB = OFF, DROP_EXISTING = OFF, ONLINE = OFF) ON [PRIMARY]
go

CREATE NONCLUSTERED INDEX [_dta_index_EmployeeSalaryHistory_5_1237579447__K9]
ON [dbo].[EmployeeSalaryHistory]
(
    [IsCurrent] ASC
)
WITH (SORT_IN_TEMPDB = OFF, DROP_EXISTING = OFF, ONLINE = OFF) ON [PRIMARY]
go
```

Oba indeksy dotyczą IsCurrent, pierwszy indeks ma zoptymalizować zapytanie 4, a drugi zapytanie 7. Sprawdzimy ich działanie.

Zapytanie 4 - zapytanie wykonuje tylko index seek po nowym indeksie. Koszt zmalał z 0.68 do 0.11.




Zapytanie 7 - to zapytanie również korzysta z nowego indeksu, jednak plan nie zmienił się wyraźnie. Koszt zmalał z 0.74 do 0.2.




| zadanie | pkt |
|---|---|
| 1 | 2 |
| 2 | 2 |
| 3 | 2 |
| 4 | 2 |
| 5 | 5 |
| razem | 15 |