

# Indeksy, optymalizator

## Lab 2

---

---

**Imię i nazwisko: Kacper Cienkosz, Miłosz Dubiel**

---

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
```

Przydatne wywołania w MSSQL:

```
SET STATISTICS IO ON  
SET STATISTICS TIME ON  
  
-- Use this when you want to clear the cache  
-- Recommended after each query  
CHECKPOINT;  
GO  
DBCC DROPCLEANBUFFERS
```

# 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:

Zapytanie 1

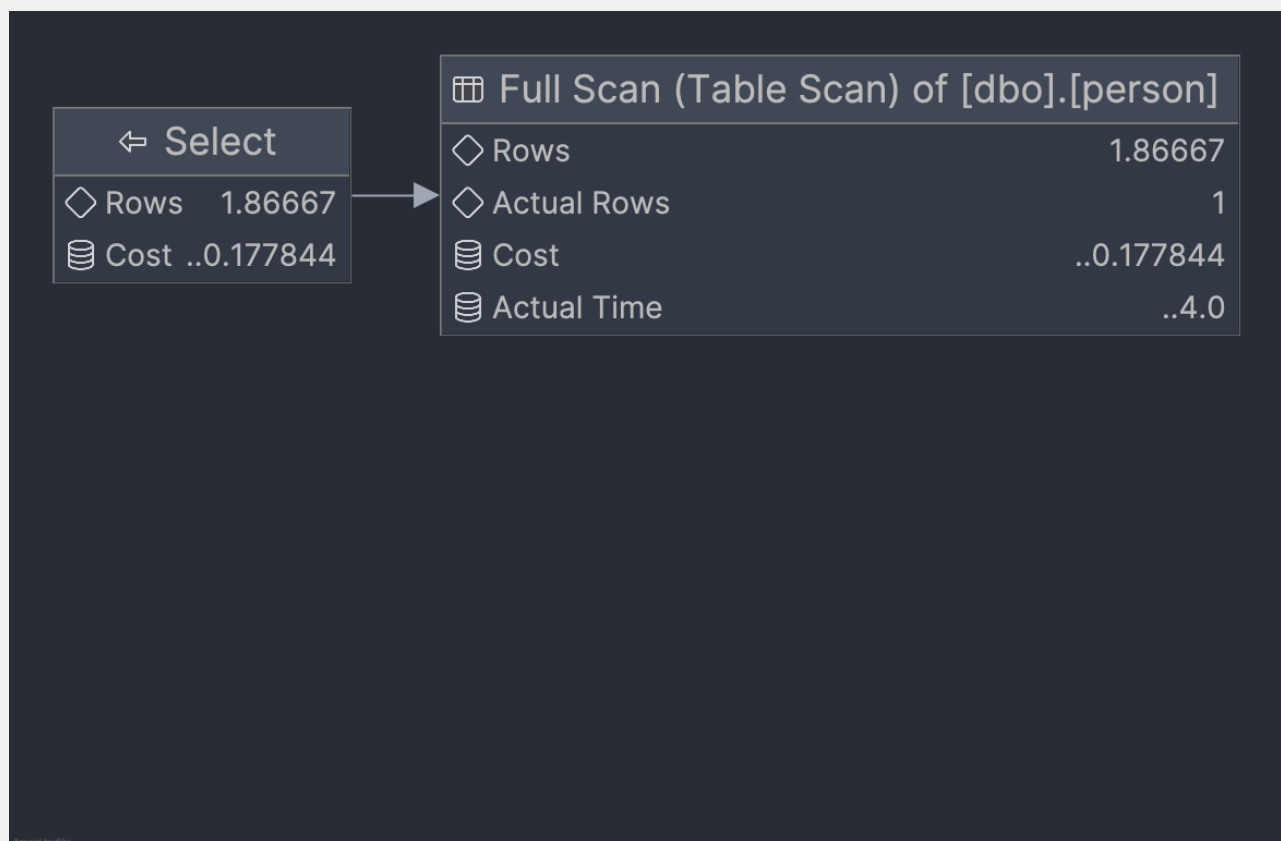
```
SELECT *  
FROM person  
WHERE lastname = 'Agbonile'  
[S0000][3613] SQL Server parse and compile time:  
CPU time = 47 ms, elapsed time = 50 ms.
```

```

[S0000][3613] SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 0 ms.
[S0000][3615]
Table 'person'.
Scan count 1,
logical reads 207,
physical reads 0,
page server reads 0,
read-ahead reads 0,
page server read-ahead reads 0,
lob logical reads 0,
lob physical reads 0,
lob page server reads 0,
lob read-ahead reads 0,
lob page server read-ahead reads 0.
[S0000][3612] SQL Server Execution Times:
CPU time = 4 ms, elapsed time = 3 ms.
1 row retrieved starting from 1 in 97 ms (execution: 59 ms, fetching: 38
ms)

```

Plan zapytania:



Zapytanie 2

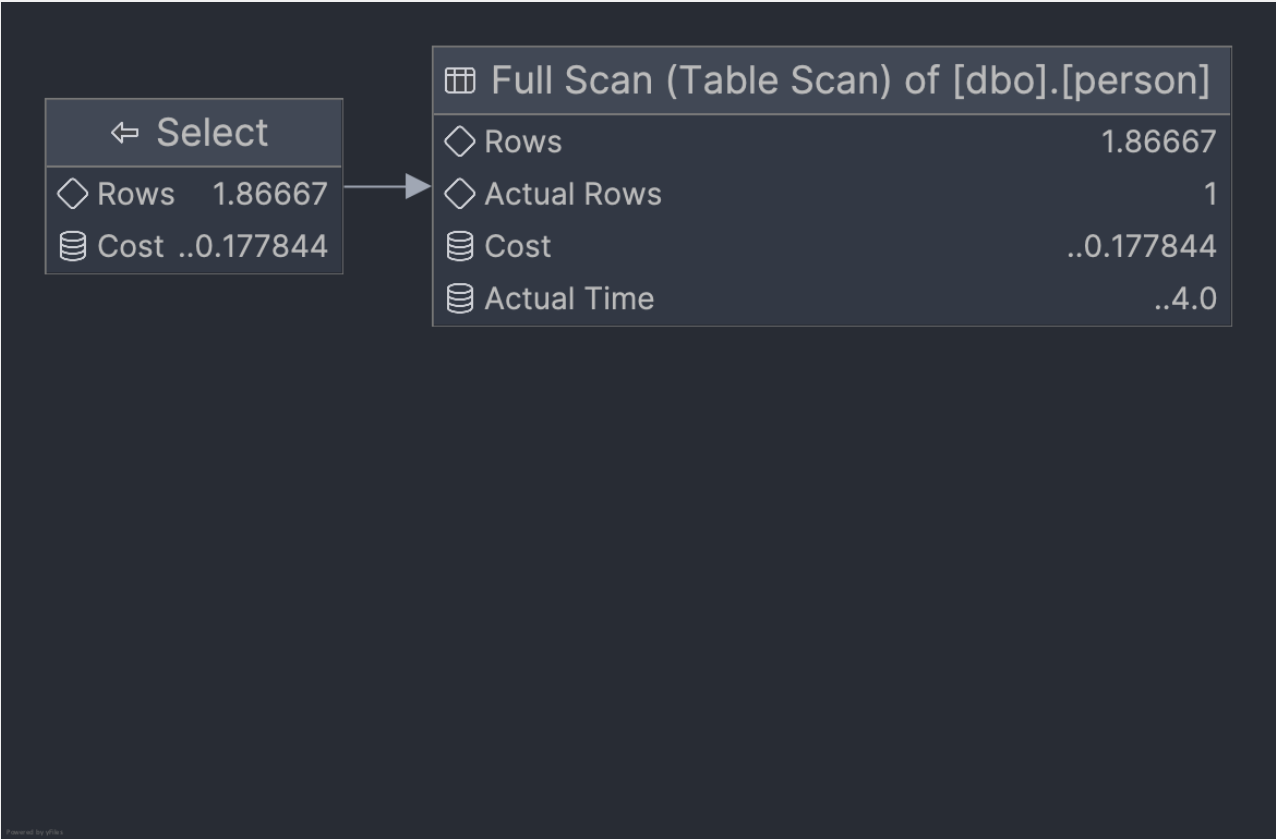
```

lab2> SELECT *
      FROM person
     WHERE lastname = 'Agbonile'
        AND firstname = 'Osarumwense'

```

```
[S0000][3613] SQL Server parse and compile time:
CPU time = 46 ms, elapsed time = 46 ms.
[S0000][3613] SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 0 ms.
[S0000][3615]
Table 'person'.
Scan count 1,
logical reads 207,
physical reads 0,
page server reads 0,
read-ahead reads 0,
page server read-ahead reads 0,
lob logical reads 0,
lob physical reads 0, lob page server reads 0,
lob read-ahead reads 0,
lob page server read-ahead reads 0.
[S0000][3612] SQL Server Execution Times:
CPU time = 3 ms, elapsed time = 2 ms.
1 row retrieved starting from 1 in 75 ms (execution: 54 ms, fetching: 21 ms)
```

Plan zapytania:

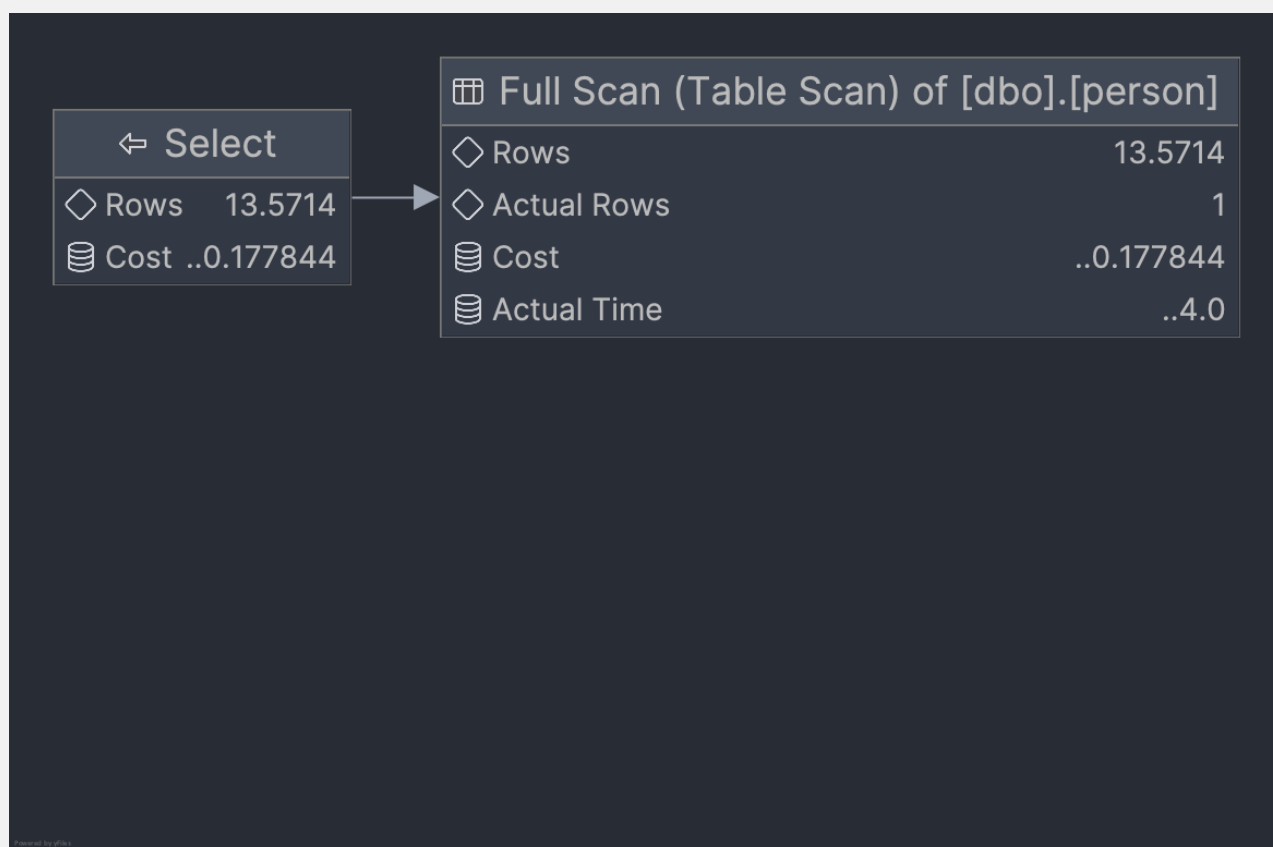


Zapytanie 3

```
SELECT *
FROM person
WHERE firstname = 'Osarumwense'
```

```
[S0000][3613] SQL Server parse and compile time:
CPU time = 6 ms, elapsed time = 7 ms.
[S0000][3613] SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 0 ms.
[S0000][3615]
Table 'person'.
Scan count 1,
logical reads 207,
physical reads 0,
page server reads 0,
read-ahead reads 0,
page server read-ahead reads 0,
lob logical reads 0,
lob physical reads 0,
lob page server reads 0,
lob read-ahead reads 0,
lob page server read-ahead reads 0.
[S0000][3612] SQL Server Execution Times:
CPU time = 7 ms, elapsed time = 10 ms.
1 row retrieved starting from 1 in 44 ms (execution: 22 ms, fetching: 22 ms)
```

#### Plan zapytania:



Wszystkie trzy zapytania wykonują pełne skanowanie tabeli **person**, ponieważ nie ma indeksu, który mógłby przyspieszyć wyszukiwanie. W każdym przypadku liczba odczytów logicznych jest taka sama (207), co wskazuje na to, że SQL Server przeszukuje całą tabelę.

Przygotuj indeks obejmujący te zapytania:

```
CREATE INDEX person_first_last_name_idx
ON person(lastname, firstname)
```

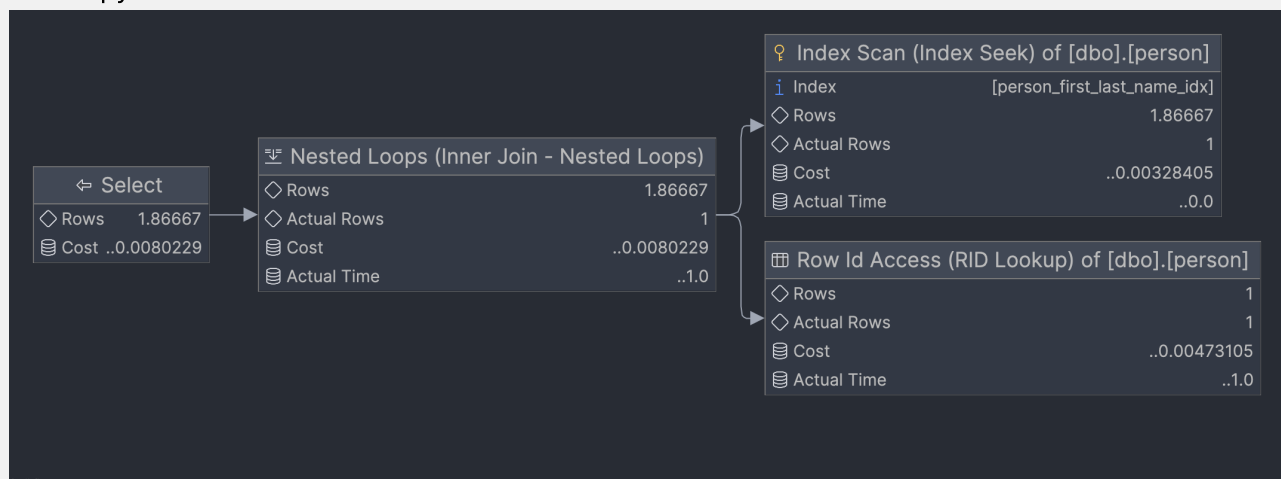
Sprawdź plan zapytania. Co się zmieniło?

Wyniki:

## Zapytanie 1

```
SELECT *
FROM person
WHERE lastname = 'Agbonile'
```

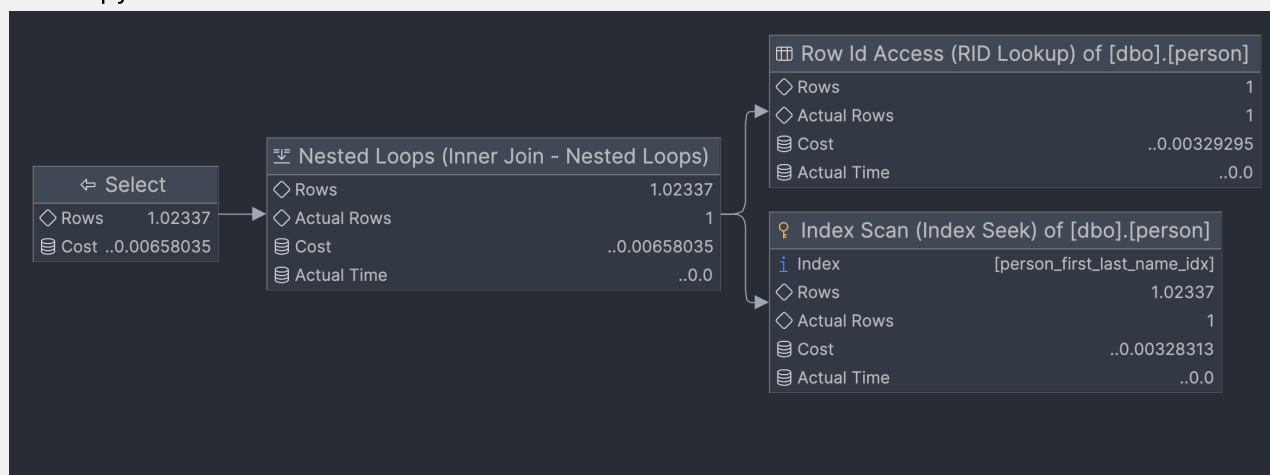
Plan zapytania:



## Zapytanie 2

```
SELECT *
FROM person
WHERE lastname = 'Agbonile'
AND firstname = 'Osarumwense'
```

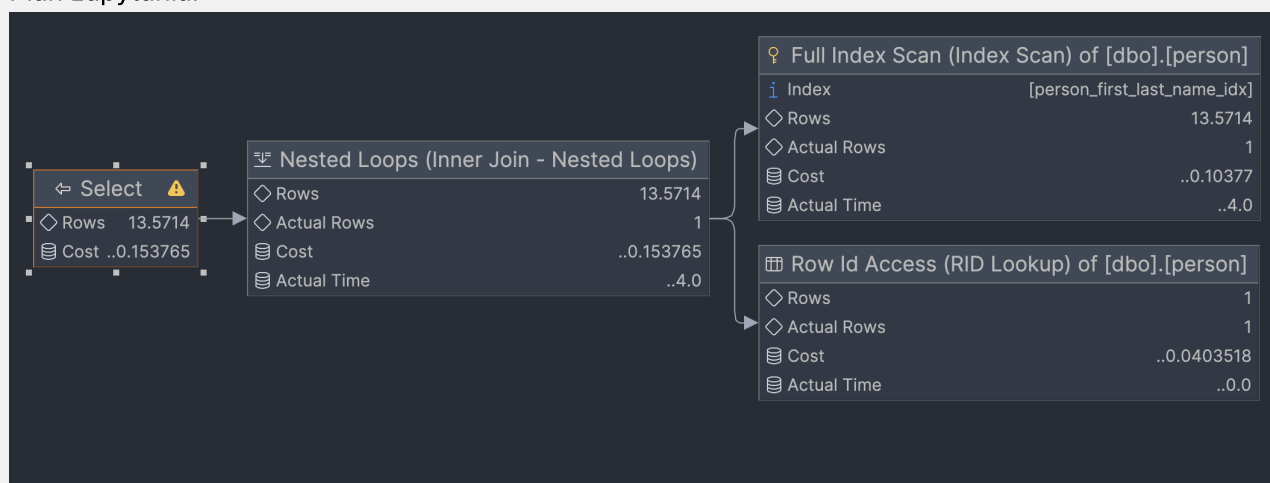
## Plan zapytania:



## Zapytanie 3

```
SELECT *
FROM person
WHERE firstname = 'Osarumwense'
```

## Plan zapytania:



Po dodaniu indeksu `person_first_last_name_idx`, plany zapytań uległy znaczącej zmianie. Teraz SQL Server korzysta z indeksu.

W przypadku pierwszego zapytania serwer wykonuje **INDEX SEEK**, co oznacza, że bezpośrednio przeszukuje indeks, zamiast skanować całą tabelę.

Drugie zapytanie również korzysta z indeksu, ale wykonuje **INDEX SEEK** z dodatkowymi warunkami na kolumnę `firstname`, co jest bardziej efektywne niż pełne skanowanie tabeli. W tym zapytaniu wykorzystywany jest cały indeks, co znacznie poprawia wydajność. Widać to poprzez koszt zapytania.

Trzecie zapytanie również korzysta z indeksu, ale w tym przypadku SQL Server wykonuje **FULL INDEX SCAN**, ponieważ nie ma warunku na kolumnę `lastname`. To oznacza, że serwer przeszukuje wszystkie wiersze w indeksie, ale nadal jest to bardziej efektywne niż pełne skanowanie tabeli.



Wynika to z zasady **lefft-prefix**. Mówi ona, że żeby zapytanie z indeksem było efektywne, musi wykorzystywać kolumny indeksu w kolejności, w jakiej zostały zdefiniowane.

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:

## Zapytanie 1

```
SELECT *  
FROM person  
WHERE lastname = 'Price'
```

Plan zapytania:

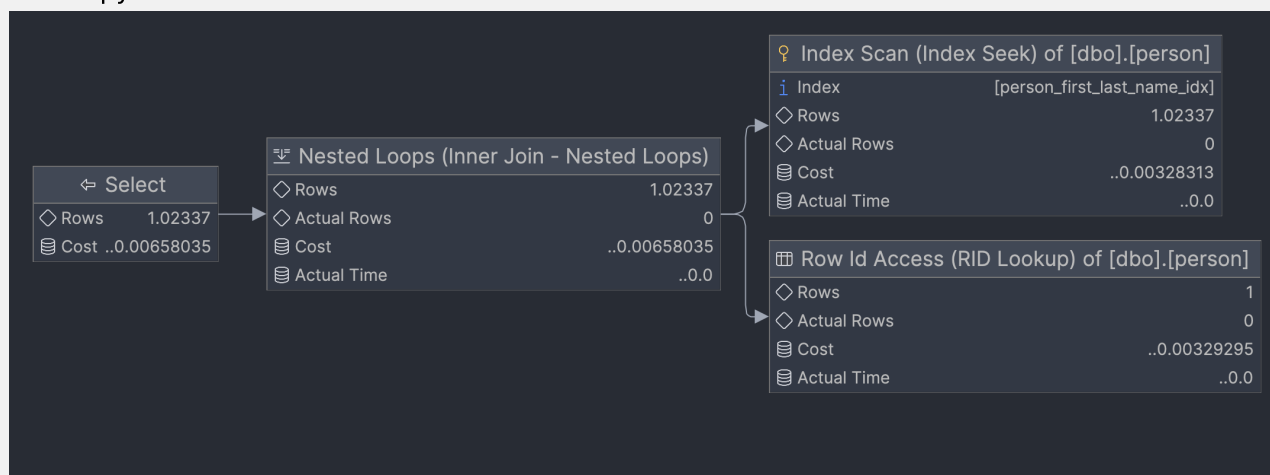
↶ Select	
◇ Rows	84
🗄 Cost	..0.177844

🗄 Full Scan (Table Scan) of [dbo].[person]	
◇ Rows	84
◇ Actual Rows	84
🗄 Cost	..0.177844
🗄 Actual Time	..2.0

## Zapytanie 2

```
SELECT *  
FROM person  
WHERE lastname = 'Price'  
AND firstname = 'Angela'
```

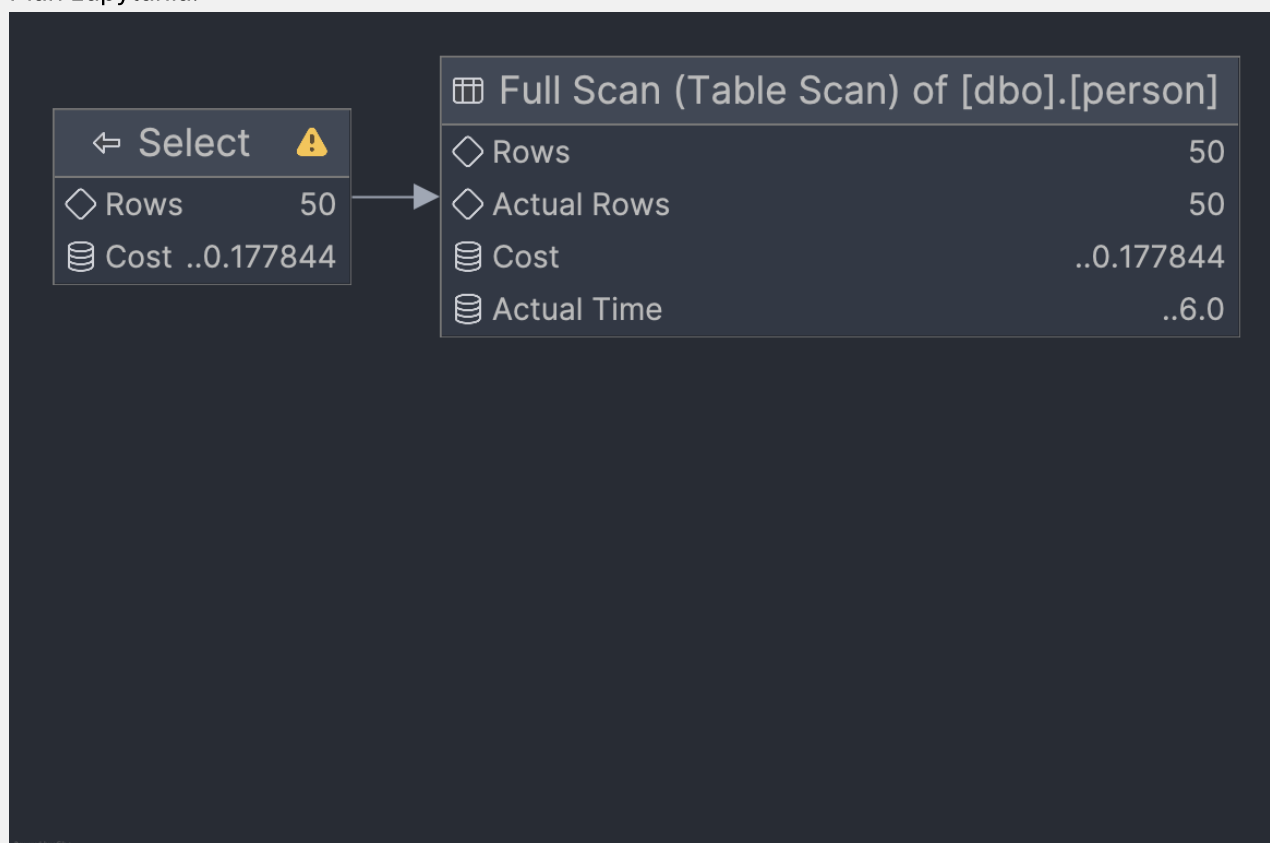
## Plan zapytania:



## Zapytanie 3

```
SELECT *
FROM person
WHERE firstname = 'Angela'
```

## Plan zapytania:



W przypadku zapytań z parametrami `FirstName = 'Angela'` i `LastName = 'Price'` serwer nie zdecydował się wykorzystać indeksu w przypadku pierwszego i trzeciego zapytania. Jest tak ponieważ zarówno imię Angela, jak i nazwisko Price występują w tabeli wiele razy, więc SQL Server uznał, że pełne skanowanie tabeli będzie bardziej efektywne niż korzystanie z indeksu, który ustalony jest według nazwiska i imieni.

## Zadanie 2

Skopiuj tabelę Product do swojej bazy danych:

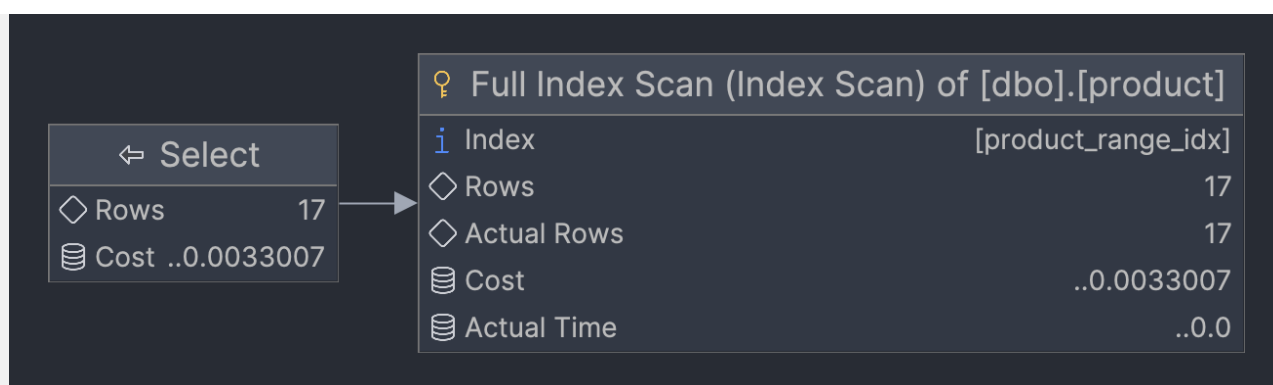
```
SELECT * INTO product
FROM adventureworks2017.production.product
```

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
```



Select	
Rows	17
Cost	..0.0033007

Full Index Scan (Index Scan) of [dbo].[product]	
Index	[product_range_idx]
Rows	17
Actual Rows	17
Cost	..0.0033007
Actual Time	..0.0

Tak, został wykorzystany indeks.

Sprawdź, czy indeks jest użyty w zapytaniu, który jest dopełnieniem zbioru:

```
SELECT name,
       productsubcategoryid,
       listprice
FROM product
WHERE productsubcategoryid < 27
    OR productsubcategoryid > 36
```

↔ Select	
◇ Rows	278
🗄 Cost	..0.0127253

🗄 Full Scan (Table Scan) of [dbo].[product]	
◇ Rows	278
◇ Actual Rows	278
🗄 Cost	..0.0127253
🗄 Actual Time	..0.0

Nie został wykorzystany indeks.

Skomentuj oba zapytania. Czy indeks został użyty w którymś zapytaniu, dlaczego? Jak działają indeksy z warunkiem?

Indeksy z warunkiem są używane tylko wtedy, gdy zapytanie pasuje do warunku. W przeciwnym razie, SQL Server nie użyje indeksu, ponieważ nie przyniesie to korzyści w wydajności. W naszym przypadku, w pierwszym zapytaniu warunek był spełniony, więc indeks został użyty. W drugim zapytaniu warunek nie był spełniony, więc SQL Server zdecydował się na pełne skanowanie tabeli.

Indeks z warunkiem działa w ten sposób, że tylko te wiersze, które spełniają określony warunek, są indeksowane. Dzięki temu można zaoszczędzić miejsce i przyspieszyć wyszukiwanie danych, ale tylko dla zapytań, które pasują do tego warunku.

Wyniki:

-- ...

## Zadanie 3

Skopiuj tabelę `PurchaseOrderDetail` do swojej bazy danych:

```
SELECT *
INTO purchaseorderdetail
FROM adventureworks2017.purchasing.purchaseorderdetail
```

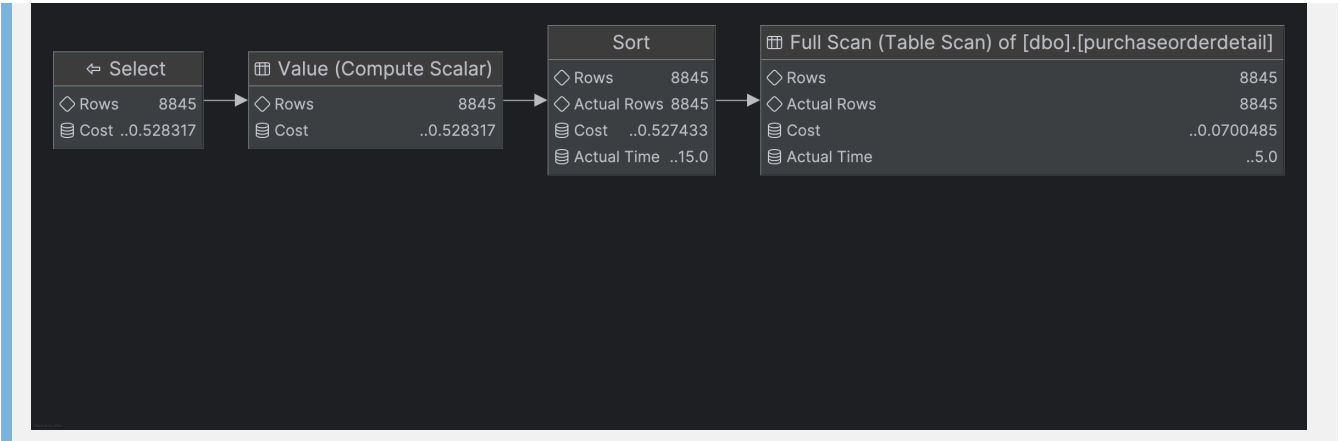
Wykonaj analizę zapytania:

```
SELECT rejectedqty,
       ((rejectedqty / orderqty) * 100) AS rejectionrate,
       productid,
       duedate
FROM purchaseorderdetail
ORDER BY rejectedqty DESC, productid ASC
```

## Wyniki:

```
lab05> SELECT rejectedqty,
           ((rejectedqty / orderqty) * 100) AS rejectionrate,
           productid,
           duedate
           FROM purchaseorderdetail
           ORDER BY rejectedqty DESC, productid ASC
[2025-05-28 17:22:53] [S0000][3613] SQL Server parse and compile time:
[2025-05-28 17:22:53] CPU time = 2 ms, elapsed time = 3 ms.
[2025-05-28 17:22:53] [S0000][3615] Table 'Worktable'.
    Scan count 0,
    logical reads 0,
    physical reads 0,
    page server reads 0,
    read-ahead reads 0,
    page server read-ahead reads 0,
    lob logical reads 0,
    lob physical reads 0,
    lob page server reads 0,
    lob read-ahead reads 0,
    lob page server read-ahead reads 0.
[2025-05-28 17:22:53] [S0000][3615] Table 'purchaseorderdetail'.
    Scan count 1,
    logical reads 78,
    physical reads 0,
    page server reads 0,
    read-ahead reads 78,
    page server read-ahead reads 0,
    lob logical reads 0,
    lob physical reads 0,
    lob page server reads 0,
    lob read-ahead reads 0,
    lob page server read-ahead reads 0.
[2025-05-28 17:22:53] [S0000][3612] SQL Server Execution Times:
[2025-05-28 17:22:53] CPU time = 21 ms, elapsed time = 22 ms.
[2025-05-28 17:22:53] completed in 29 ms
```

## Plan zapytania:



Która część zapytania ma największy koszt?

Patrząc na surowy rezultat planu zapytania, dostępny w DataGrip w formie pseudo-tabelki, którą przytoczymy tutaj w okrojonej formie, zawierającej tylko potrzebne informacje:

Step	EstimatedCPU	EstimatedIO	EstimatedTotalSubtreeCost	Estimated step cost (własne obliczenia)
Full table scan	0.009808	0.0602405	0.0700485	0.0700485
Sort	0.446123	0.0112613	0.527433	0.4573845
Value	0.0008845	0.0	0.528317	0.000884
Select	N/A	N/A	0.528317	0.0

Z tabelki wynika, że największy koszt wiązał się z posortowaniem rekordów, który był ponad 6-krotnie wyższy od kosztu wykonania pełnego skanu tabeli.

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

Do optymalizacji tego zapytania można zastosować **CLUSTERED** indeks zawierający kolumny, które występują w klauzuli **ORDER BY**, razem z zastosowaniem takiej samej kolejności (sortowania rekordów) jak w zapytaniu.

```
CREATE CLUSTERED INDEX idx_purchaseorderdetail_rejectedqty_productid
ON purchaseorderdetail (rejectedqty DESC, productid ASC);

SELECT rejectedqty,
       ((rejectedqty / orderqty) * 100) AS rejectionrate,
       productid,
       duedate
FROM purchaseorderdetail WITH(
    INDEX(idx_purchaseorderdetail_rejectedqty_productid)
)
ORDER BY rejectedqty DESC, productid ASC
```

## Wyniki:

Stworzenie indeksu `idx_purchaseorderdetail_rejectedqty_productid`:

```

lab05> CREATE CLUSTERED INDEX
idx_purchaseorderdetail_rejectedqty_productid
      ON purchaseorderdetail (rejectedqty DESC, productid ASC)
[2025-05-28 18:09:59] [S0000][3613] SQL Server parse and compile time:
[2025-05-28 18:09:59] CPU time = 1 ms, elapsed time = 1 ms.
[2025-05-28 18:09:59] [S0000][3615] Table 'purchaseorderdetail'.
      Scan count 1,
      logical reads 87,
      physical reads 0,
      page server reads 0,
      read-ahead reads 87,
      page server read-ahead reads 0,
      lob logical reads 0,
      lob physical reads 0,
      lob page server reads 0,
      lob read-ahead reads 0,
      lob page server read-ahead reads 0.
[2025-05-28 18:09:59] [S0000][3612] SQL Server Execution Times:
[2025-05-28 18:09:59] CPU time = 30 ms, elapsed time = 39 ms.
[2025-05-28 18:09:59] completed in 46 ms

```

## Wykonanie zapytania w oparciu o indeks:

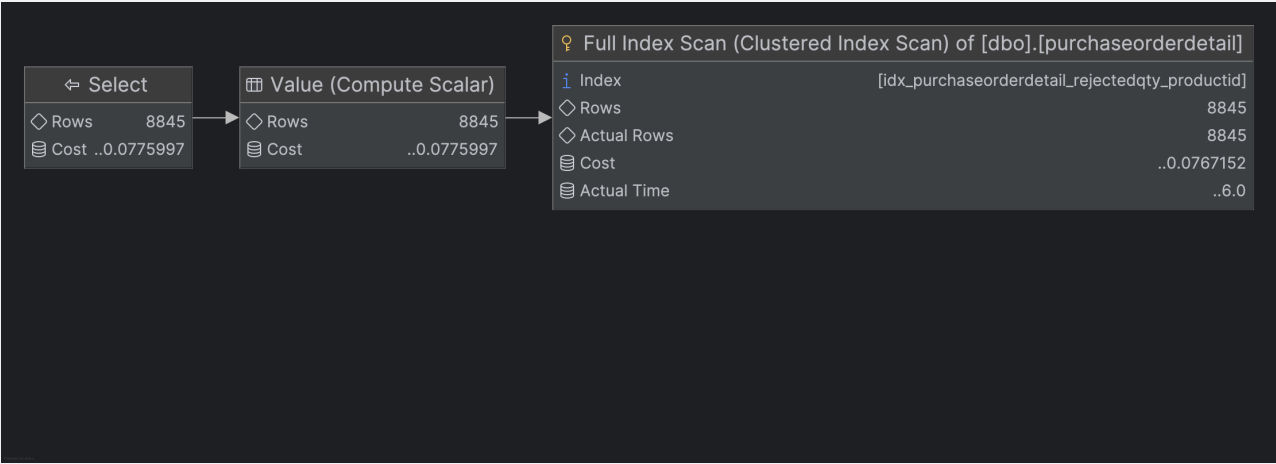
```

lab05> SELECT rejectedqty,
      ((rejectedqty / orderqty) * 100) AS rejectionrate,
      productid,
      duedate
      FROM purchaseorderdetail WITH(
      INDEX(idx_purchaseorderdetail_rejectedqty_productid)
      )
      ORDER BY rejectedqty DESC, productid ASC
[2025-05-28 18:11:23] [S0000][3613] SQL Server parse and compile time:
[2025-05-28 18:11:23] CPU time = 0 ms, elapsed time = 0 ms.
[2025-05-28 18:11:23] [S0000][3613] SQL Server parse and compile time:
[2025-05-28 18:11:23] CPU time = 6 ms, elapsed time = 7 ms.
[2025-05-28 18:11:23] [S0000][3615] Table 'purchaseorderdetail'.
      Scan count 1,
      logical reads 89,
      physical reads 1,
      page server reads 0,
      read-ahead reads 87,
      page server read-ahead reads 0,
      lob logical reads 0,
      lob physical reads 0,
      lob page server reads 0,
      lob read-ahead reads 0,

```

```
lob page server read-ahead reads 0.
[2025-05-28 18:11:23] [S0000][3612] SQL Server Execution Times:
[2025-05-28 18:11:23] CPU time = 11 ms, elapsed time = 16 ms.
[2025-05-28 18:11:23] completed in 28 ms
```

Plan zapytania:



Okrojona tabelka z rezultatem surowego planu zapytania:

Step	EstimatedCPU	EstimatedIO	EstimatedTotalSubtreeCost	Estimated step cost (własne obliczenia)
Full index scan	0.0098865	0.0668287	0.0767152	0.0767152
Value	0.0008845	0.0	0.0775997	0.0008845 (= EstimatedCPU)
Select	N/A	N/A	0.0775997	0.0

Porównując koszty z zapytania bez użycia indeksu oraz z jego użyciem, to widać dużą różnicę rzędu prawie 7-krotności kosztu z indeksem (0.528317 vs. 0.0775997). Pokazuje to, że zastosowanie w tym wypadku **CLUSTERED** indeksu daje bardzo dużą optymalizację kosztu zapytania.

Zapytanie z użyciem indeksu jest również szybsze czasowo:

	CPU time	Elapsed time	Completed
bez indeksu	21 ms	22 ms	29 ms
z indeksem	11 ms	16 ms	28 ms

Co prawda niewiele szybsze, ale jednak 😊 .

Drugą opcją jest stworzenie **NONCLUSTERED** indeksu, który zawiera klauzulę **INCLUDE** z kolumnami, które nie są związane z sortowaniem w tym zapytaniu, ale są "wyciągane" przy jego pomocy.



```

CREATE NONCLUSTERED INDEX
idx_purchaseorderdetail_rejectedqty_productid_include
ON purchaseorderdetail (rejectedqty DESC, productid ASC)
INCLUDE (duedate, orderqty);

SELECT rejectedqty,
        ((rejectedqty / orderqty) * 100) AS rejectionrate,
        productid,
        duedate
FROM purchaseorderdetail WITH(
    INDEX(idx_purchaseorderdetail_rejectedqty_productid_include)
)
ORDER BY rejectedqty DESC, productid ASC

```

Wyniki:

```

lab05> CREATE NONCLUSTERED INDEX
idx_purchaseorderdetail_rejectedqty_productid_include
        ON purchaseorderdetail (rejectedqty DESC, productid ASC)
        INCLUDE (duedate, orderqty)
[2025-05-28 18:24:58] [S0000][3613] SQL Server parse and compile time:
[2025-05-28 18:24:58] CPU time = 1 ms, elapsed time = 1 ms.
[2025-05-28 18:24:58] [S0000][3615] Table 'purchaseorderdetail'.
        Scan count 1,
        logical reads 87,
        physical reads 0,
        page server reads 0,
        read-ahead reads 87,
        page server read-ahead reads 0,
        lob logical reads 0,
        lob physical reads 0,
        lob page server reads 0,
        lob read-ahead reads 0,
        lob page server read-ahead reads 0.
[2025-05-28 18:24:58] [S0000][3612] SQL Server Execution Times:
[2025-05-28 18:24:58] CPU time = 22 ms, elapsed time = 29 ms.
[2025-05-28 18:24:58] completed in 34 ms

```

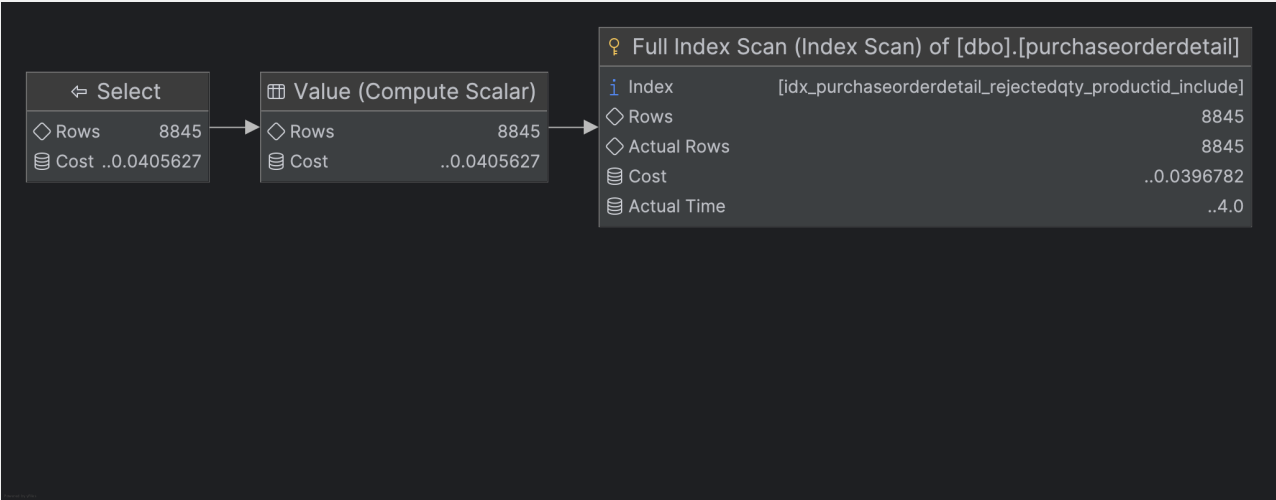
```

lab05> SELECT rejectedqty,
        ((rejectedqty / orderqty) * 100) AS rejectionrate,
        productid,
        duedate
FROM purchaseorderdetail WITH(
    INDEX(idx_purchaseorderdetail_rejectedqty_productid_include)
)
ORDER BY rejectedqty DESC, productid ASC
[2025-05-28 18:26:09] [S0000][3613] SQL Server parse and compile time:
[2025-05-28 18:26:09] CPU time = 0 ms, elapsed time = 0 ms.

```

```
[2025-05-28 18:26:09] [S0000][3613] SQL Server parse and compile time:
[2025-05-28 18:26:09] CPU time = 5 ms, elapsed time = 5 ms.
[2025-05-28 18:26:09] [S0000][3615] Table 'purchaseorderdetail'.
    Scan count 1,
    logical reads 39,
    physical reads 1,
    page server reads 0,
    read-ahead reads 37,
    page server read-ahead reads 0,
    lob logical reads 0,
    lob physical reads 0,
    lob page server reads 0,
    lob read-ahead reads 0,
    lob page server read-ahead reads 0.
[2025-05-28 18:26:09] [S0000][3612] SQL Server Execution Times:
[2025-05-28 18:26:09] CPU time = 12 ms, elapsed time = 14 ms.
[2025-05-28 18:26:09] completed in 23 ms
```

Plan zapytania:



Okrojona tabelka z rezultatem surowego planu zapytania:

Step	EstimatedCPU	EstimatedIO	EstimatedTotalSubtreeCost	Estimated step cost (własne obliczenia)
Full index scan	0.0098865	0.0297917	0.0396782	0.0396782
Value	0.0008845	0.0	0.0405627	0.0008845 (= EstimatedCPU)
Select	N/A	N/A	0.0405627	0.0

Użycie **NONCLUSTERED** indeksu pozwoliło na jeszcze lepszą optymalizację kosztu zapytania. Zeszliśmy z kosztu 0.528317 (bez indeksu), przez koszt 0.0775997 (z indeksem **CLUSTERED**) aż do kosztu 0.0405627 (z indeksem **NONCLUSTERED**).

Istotnym elementem tej optymalizacji przy użyciu indeksu **NONCLUSTERED** jest wykorzystanie klauzuli **INCLUDE** i zawarcie w niej kolumn, po których nie sortujemy, ale je "wyciągamy" – bez tej klauzuli skorzystanie z samego indeksu **NONCLUSTERED** dało nam wynik kosztu na ponad 3-krotnie gorszym poziomie niż bez indeksu (koszt ~1.5).

Zapytanie z użyciem indeksu **NONCLUSTERED** jest też szybsze czasowo niż dwa poprzednie (w aspekcie *Completed*):

	CPU time	Elapsed time	Completed
bez indeksu	21 ms	22 ms	29 ms
z indeksem <b>CLUSTERED</b>	11 ms	16 ms	28 ms
z indeksem <b>NONCLUSTERED</b>	12 ms	14 ms	23 ms

Oczywiście, ta przewaga jest tak minimalna, że ciężko to będzie odczuć gołym okiem, a dodatkowo nie wiemy, czy nie mieści się w błędzie pomiarowym związanym z innymi procesami następującymi w środowisku testowym.

Warto jeszcze zwrócić uwagę na liczbę odczytów logicznych i fizycznych wykonanych zapytań:

	logical reads	physical reads	read-ahead reads
bez indeksu	78	0	78
z indeksem <b>CLUSTERED</b>	89	1	87
z indeksem <b>NONCLUSTERED</b>	39	1	37

Wszystkie z tych wyników zostały obliczone po wyczyszczeniu cache'u.

Widać tutaj ogromną różnicę zapytania z indeksem **NONCLUSTERED**, które jest wygranym w tym pojedynku. Warto natomiast zauważyć, że użycie indeksu **CLUSTERED** zwiększyło zarówno ilość odczytów logicznych, jak i fizycznych, względem zapytania bez indeksu.

## 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  
)
```

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 obecny indeks:

```
SELECT productid,  
       SUM(unitprice),  
       AVG(unitprice),  
       SUM(orderqty),  
       AVG(orderqty)  
FROM saleshistory  
GROUP BY productid  
ORDER BY productid
```

Założ 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 columns store? Jak działają? (poszukaj materiałów w internecie/literaturze)

---

Wyniki:

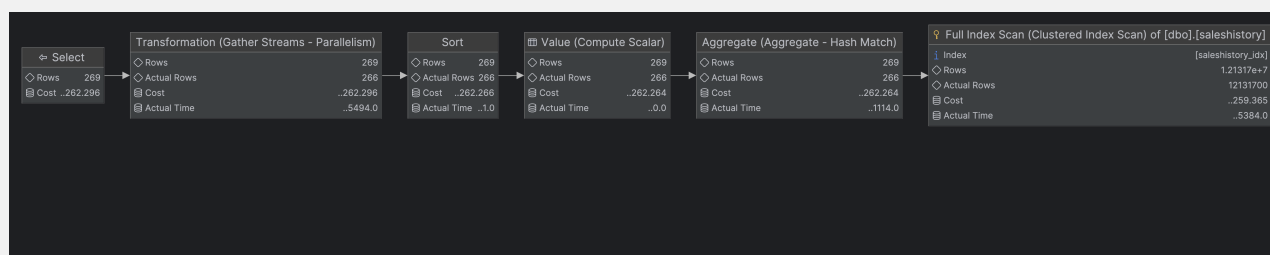
Zapytanie z indeksem **saleshistory\_idx**

```

lab05> SELECT productid,
           SUM(unitprice),
           AVG(unitprice),
           SUM(orderqty),
           AVG(orderqty)
           FROM saleshistory WITH(INDEX(saleshistory_idx))
           GROUP BY productid
           ORDER BY productid
[2025-05-28 19:10:27] [S0000][3613] SQL Server parse and compile time:
[2025-05-28 19:10:27] CPU time = 19 ms, elapsed time = 22 ms.
[2025-05-28 19:10:27] [S0000][3615] Table 'saleshistory'.
    Scan count 5,
    logical reads 343119,
    physical reads 3,
    page server reads 0,
    read-ahead reads 343327,
    page server read-ahead reads 0,
    lob logical reads 0,
    lob physical reads 0,
    lob page server reads 0,
    lob read-ahead reads 0,
    lob page server read-ahead reads 0.
[2025-05-28 19:10:27] [S0000][3615] Table 'Worktable'.
    Scan count 0,
    logical reads 0,
    physical reads 0,
    page server reads 0,
    read-ahead reads 0,
    page server read-ahead reads 0,
    lob logical reads 0,
    lob physical reads 0,
    lob page server reads 0,
    lob read-ahead reads 0,
    lob page server read-ahead reads 0.
[2025-05-28 19:10:27] [S0000][3612] SQL Server Execution Times:
[2025-05-28 19:10:27] CPU time = 2383 ms, elapsed time = 5511 ms.
[2025-05-28 19:10:27] completed in 5 s 544 ms

```

### Plan zapytania:



Okrojona tabelka z rezultatem surowego planu zapytania:

Step	EstimatedCPU	EstimatedIO	EstimatedTotalSubtreeCost	Estimated step cost (własne obliczenia)
Full index scan	6.67251	252.693	259.365	259.365
Aggregate	2.89842	0.0	262.264	2.899
Value	0.0	0.0	262.264	0.0
Sort	0.000219373	0.00187688	262.266	0.002
Transformation	0.0305796	0.0	262.296	0.03
Select	N/A	N/A	262.296	0.0

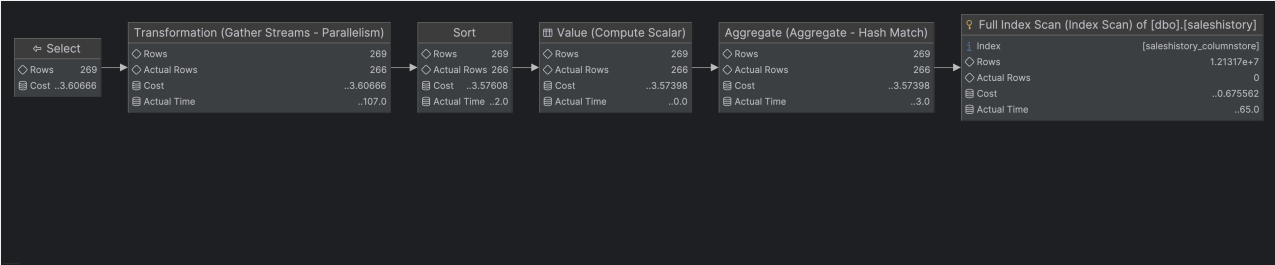
### Zapytanie z indeksem `saleshistory_columnstore`

```
lab05> SELECT productid,
           SUM(unitprice),
           AVG(unitprice),
           SUM(orderqty),
           AVG(orderqty)
FROM saleshistory WITH(INDEX(saleshistory_columnstore))
GROUP BY productid
ORDER BY productid
```

[2025-05-28 19:10:55] [S0000][3613] SQL Server parse and compile time:  
[2025-05-28 19:10:55] CPU time = 35 ms, elapsed time = 38 ms.  
[2025-05-28 19:10:55] [S0000][3615] Table 'saleshistory'.  
Scan count 8,  
logical reads 0,  
physical reads 0,  
page server reads 0,  
read-ahead reads 0,  
page server read-ahead reads 0,  
lob logical reads 3761,  
lob physical reads 42,  
lob page server reads 0,  
lob read-ahead reads 7905,  
lob page server read-ahead reads 0.  
[2025-05-28 19:10:55] [S0000][3642] Table 'saleshistory'. Segment reads  
13, segment skipped 0.  
[2025-05-28 19:10:55] [S0000][3615] Table 'Worktable'.  
Scan count 0,  
logical reads 0,  
physical reads 0,  
page server reads 0,  
read-ahead reads 0,  
page server read-ahead reads 0,  
lob logical reads 0,  
lob physical reads 0,  
lob page server reads 0,

```
lob read-ahead reads 0,  
lob page server read-ahead reads 0.  
[2025-05-28 19:10:55] [S0000][3612] SQL Server Execution Times:  
[2025-05-28 19:10:55] CPU time = 131 ms, elapsed time = 114 ms.  
[2025-05-28 19:10:55] completed in 160 ms
```

Plan zapytania:



Okrojona tabelka z rezultatem surowego planu zapytania:

Step	EstimatedCPU	EstimatedIO	EstimatedTotalSubtreeCost	Estimated step cost (własne obliczenia)
Full index scan	0.667251	0.00831019	0.675562	0.675562
Aggregate	2.89842	0.0	3.57398	2.898418
Value	0.0	0.0	3.57398	0.0
Sort	0.000219373	0.00187688	3.57608	0.0021
Transformation	0.0305796	0.0	3.60666	0.03058
Select	N/A	N/A	3.60666	0.0

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

Porównując ze sobą wyniki z dwóch powyższych tabel, widać bardzo wyraźnie, że wszystkie kroki oprócz pierwszego (Aggregate, Value, Sort, Transformation, Select) są prawie identyczne w obydwóch zapytaniach.

Jedyna różnica (ogromna) wynika z dokonanego pełnego skanu tabeli za pomocą dwóch różnych indeksów. "Klasyczny" indeks **CLUSTERED** wiąże się z ogromnym kosztem wykonania (259.365). Ten bardzo duży koszt robi się jeszcze bardziej drastyczny, gdy spojrzymy na koszt wykonania pełnego skanu za pomocą indeksu **COLUMNSTORE** – wtedy koszt wyniósł 0.675562 i jest on prawie 400-krotnie mniejszy od kosztu dla indeksu **CLUSTERED**.

Różnica w kosztach tych dwóch zapytań niesie ze sobą również różnicę w czasie wykonania zapytań:

	CPU time	Elapsed time	Completed
indeks <b>CLUSTERED</b>	2383 ms	5511 ms	5544 ms

	CPU time	Elapsed time	Completed
indeks <b>COLUMNSTORE</b>	131 ms	114 ms	160 ms

Różnice w czasach są kolosalne: ponad 5 sekund dla zapytania z indeksem **CLUSTERED** względem 160 milisekund dla zapytania z indeksem **COLUMNSTORE**!

Co to są indeksy columns store? Jak działają? (poszukaj materiałów w internecie/literaturze)

Indeksy typu **COLUMNSTORE** zostały zaprojektowane aby przyspieszyć analizy dużych zbiorów danych (np. w hurtowniach danych). Charakteryzują się tym, że przechowują dane w formacie kolumnowym, a nie wierszowym.

Pozwala to na dużo szybsze wykonywanie zapytań analitycznych, które wykorzystują funkcje agregujące. Ich dodatkową zaletą jest również dużo łatwiejsza i lepsza kompresja danych, co pozwala na zmniejszenie rozmiaru danych przechowywanych. Ponadto, indeksy **COLUMNSTORE** pozwalają na tzw. *Batch execution*, czyli przetwarzanie wsadowe/wektorowe, które pozwala przetwarzać wiele wierszy jednocześnie.

Minusem dla tego typu indeksów jest z pewnością częste wykonywanie operacji **INSERT/UPDATE/DELETE** na pojedynczych rekordach oraz dostęp do wielu kolumn z jednego wiersza (co jest dosyć zrozumiałe, skoro dane są trzymane w formacie kolumnowym).

Źródło: [Microsoft's article: Columnstore indexes: overview](#)

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

#### Wyniki:

W pakiecie przesyłamy dwa pliki: `iot_device_readings-table-init.sql` oraz `create-indexes.sql`. Te pliki zawierają pełne stworzenie tabeli oraz uzupełnienie jej danymi, jak również stworzenie wszystkich indeksów.

#### Schemat tabeli:

```
CREATE TABLE iot_device_readings (  
  id BIGINT NOT NULL,  
  unix_timestamp BIGINT NOT NULL,  
  peripheral_id INT NOT NULL,  
  connection_type INT NOT NULL,  
  peripheral_status INT NOT NULL,  
  firmware_version INT NOT NULL,  
  packet_counter INT NOT NULL,  
  battery_voltage FLOAT NOT NULL,  
  internal_temperature FLOAT NOT NULL,  
  external_temperature FLOAT NOT NULL,  
  acceleration_x FLOAT NOT NULL,  
  acceleration_y FLOAT NOT NULL,  
  acceleration_z FLOAT NOT NULL,  
  pressure FLOAT NOT NULL,  
  humidity FLOAT NOT NULL,  
  tension FLOAT NOT NULL  
);
```

#### Opis danych (ich rozmiar, zawartość, statystyki)

1 milion losowo wygenerowanych rekordów o strukturze:

- `unix_timestamp` - losowy stempel czasowy
- `peripheral_id` - liczba z zakresu (1, 101)
- `connection_type` - 'Bluetooth' lub 'LoRa'
- `peripheral_status` - 'Disabled', 'Event' lub 'Time Domain'
- `firmware_version` - wartość z zakresu [1, 10]
- `packet_counter` - wartość z zakresu (1, 255)
- `battery_voltage` - wartość w V z zakresu (2.5, 4.2)
- `internal_temperature` - wartość w stopniach Celsjusza z zakresu (-10, 60)
- `external_temperature` - wartość w stopniach Celsjusza z zakresu (-20, 50)
- `acceleration_x` - wartość z zakresu (-10, 10)
- `acceleration_y` - wartość z zakresu (-10, 10)

- **acceleration\_z** - wartość z zakresu (-10, 10)
- **pressure** - wartość w hPa z zakresu (950, 1050)
- **humidity** - wartość w % z zakresu (0, 100)
- **tension** - wartość w kN z zakresu (0, 100)

## Indeksy

- Klastrowany nie na kluczu głównym:

```
CREATE CLUSTERED INDEX  
idx_iot_device_readings_peripheral_id_clustered  
ON iot_device_readings (peripheral_id);
```

Zrobiony na **peripheral\_id**, ponieważ to też jest unikalny atrybut.

### Zapytanie bez indeksu

```
iot> SELECT *  
      FROM iot_device_readings  
      WHERE peripheral_id = 55  
[2025-05-28 22:26:49] [S0000][3613] SQL Server parse and compile  
time:  
[2025-05-28 22:26:49] CPU time = 0 ms, elapsed time = 0 ms.  
[2025-05-28 22:26:49] [S0000][3613] SQL Server parse and compile  
time:  
[2025-05-28 22:26:49] CPU time = 0 ms, elapsed time = 0 ms.  
[2025-05-28 22:26:49] [S0000][3613] SQL Server parse and compile  
time:  
[2025-05-28 22:26:49] CPU time = 110 ms, elapsed time = 147 ms.  
[2025-05-28 22:26:49] [S0000][3613] SQL Server parse and compile  
time:  
[2025-05-28 22:26:49] CPU time = 0 ms, elapsed time = 0 ms.  
[2025-05-28 22:26:49] [S0000][3615] Table 'iot_device_readings'.  
Scan count 5,  
logical reads 15152,  
physical reads 0,  
page server reads 0,  
read-ahead reads 11816,  
page server read-ahead reads 0,  
lob logical reads 0,  
lob physical reads 0,  
lob page server reads 0,  
lob read-ahead reads 0,  
lob page server read-ahead reads 0.  
[2025-05-28 22:26:49] [S0000][3612] SQL Server Execution Times:  
[2025-05-28 22:26:49] CPU time = 102 ms, elapsed time = 77 ms.  
[2025-05-28 22:26:49] completed in 232 ms
```

Plan zapytania:

↶ Select ⚠	Transformation (Gather Streams - Parallelism)	Full Scan (Table Scan) of [dbo].[iot_device_readings]
◇ Rows 10437.7	◇ Rows 10437.7	◇ Rows 10437.7
📄 Cost ..12.1391	◇ Actual Rows 9927	◇ Actual Rows 9927
	📄 Cost ..12.1391	📄 Cost ..11.7762
	📄 Actual Time ..67.0	📄 Actual Time ..70.0

### Zapytanie z indeksem

```

iot> SELECT *
      FROM iot_device_readings
      WHERE peripheral_id = 55
[2025-05-28 22:29:24] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 22:29:24] CPU time = 0 ms, elapsed time = 0 ms.
[2025-05-28 22:29:24] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 22:29:24] CPU time = 0 ms, elapsed time = 0 ms.
[2025-05-28 22:29:24] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 22:29:24] CPU time = 6 ms, elapsed time = 6 ms.
[2025-05-28 22:29:24] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 22:29:24] CPU time = 0 ms, elapsed time = 0 ms.
[2025-05-28 22:29:24] [S0000][3615] Table 'iot_device_readings'.
      Scan count 1,
      logical reads 164,
      physical reads 2,
      page server reads 0,
      read-ahead reads 161,
      page server read-ahead reads 0,
      lob logical reads 0,
      lob physical reads 0,
      lob page server reads 0,
      lob read-ahead reads 0,
      lob page server read-ahead reads 0.
[2025-05-28 22:29:24] [S0000][3612] SQL Server Execution Times:
[2025-05-28 22:29:24] CPU time = 16 ms, elapsed time = 24 ms.
[2025-05-28 22:29:24] completed in 37 ms

```

Plan zapytania:

↔ Select	
◇ Rows	9927
📄 Cost	..0.132078

♀ Index Scan (Clustered Index Seek) of [dbo].[iot_device_readings]	
i Index	[idx_iot_device_readings_peripheral_id_clustered]
◇ Rows	9927
◇ Actual Rows	9927
📄 Cost	..0.132078
📄 Actual Time	..18.0

Komentarz: indeks klastrowany faktycznie zadziałał i zmniejszył koszt oraz czas zapytania.

- Nieklastrowany:

```
CREATE NONCLUSTERED INDEX idx_iot_device_readings_connection_type
ON iot_device_readings (connection_type);
```

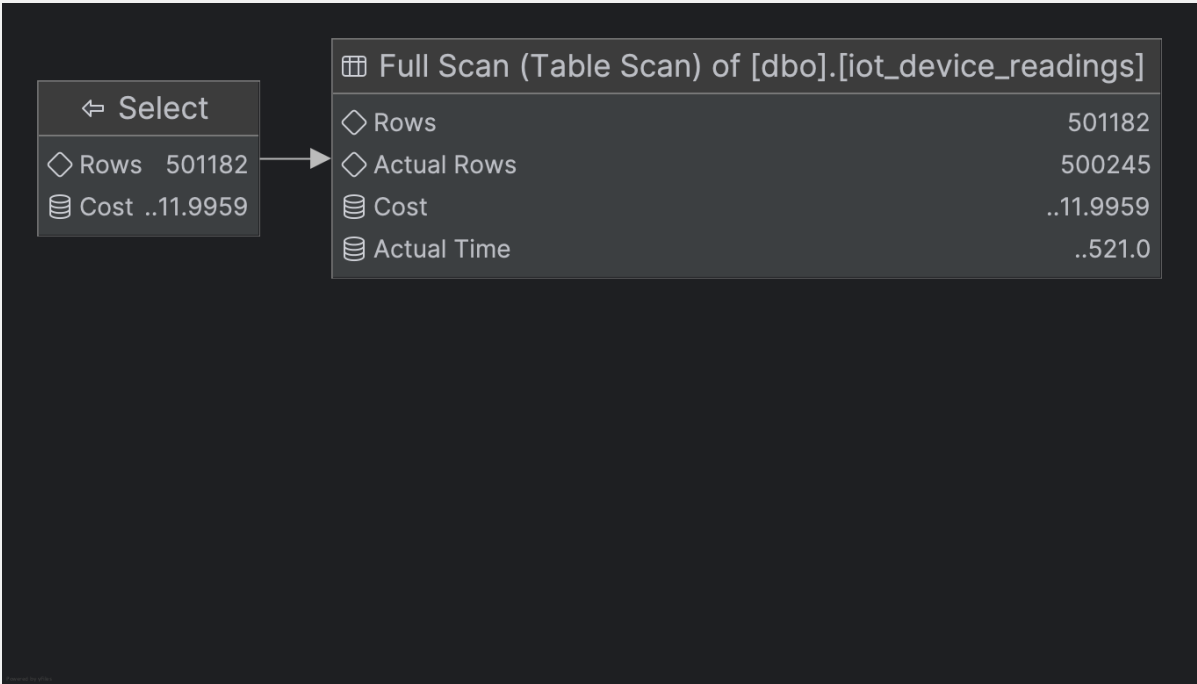
Zrobiony na `connection_type`, ponieważ jest to wartość dyskretna.

### Zapytanie bez indeksu

```
iot> SELECT *
      FROM iot_device_readings
      WHERE connection_type = 0
[2025-05-28 22:44:20] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 22:44:20] CPU time = 14 ms, elapsed time = 15 ms.
[2025-05-28 22:44:20] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 22:44:20] CPU time = 0 ms, elapsed time = 0 ms.
[2025-05-28 22:44:20] [S0000][3615] Table 'iot_device_readings'.
  Scan count 1,
  logical reads 14706,
  physical reads 0,
  page server reads 0,
  read-ahead reads 14692,
  page server read-ahead reads 0,
  lob logical reads 0,
  lob physical reads 0,
  lob page server reads 0,
  lob read-ahead reads 0,
  lob page server read-ahead reads 0.
[2025-05-28 22:44:20] [S0000][3612] SQL Server Execution Times:
```

```
[2025-05-28 22:44:20] CPU time = 490 ms, elapsed time = 648 ms.  
[2025-05-28 22:44:20] completed in 667 ms
```

Plan zapytania:



Zapytanie z indeksem

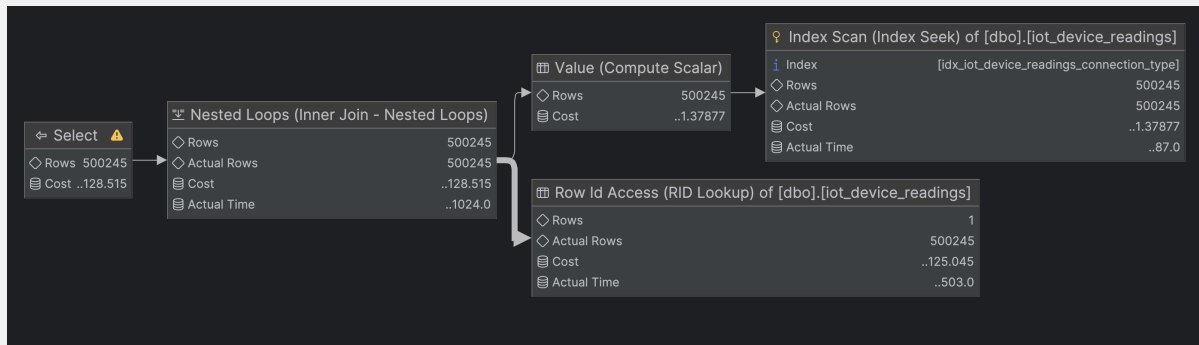
```
iot> SELECT *  
      FROM iot_device_readings WITH  
(INDEX(idx_iot_device_readings_connection_type))  
      WHERE connection_type = 0  
[2025-05-28 22:47:31] [S0000][3613] SQL Server parse and compile  
time:  
[2025-05-28 22:47:31] CPU time = 17 ms, elapsed time = 20 ms.  
[2025-05-28 22:47:31] [S0000][3615] Table 'iot_device_readings'.  
  Scan count 1,  
  logical reads 501367,  
  physical reads 1719,  
  page server reads 0,  
  read-ahead reads 2099,  
  page server read-ahead reads 0,  
  lob logical reads 0,  
  lob physical reads 0,  
  lob page server reads 0,  
  lob read-ahead reads 0,  
  lob page server read-ahead reads 0.  
[2025-05-28 22:47:31] [S0000][3615] Table 'Worktable'.  
  Scan count 0,  
  logical reads 0,  
  physical reads 0,  
  page server reads 0,  
  read-ahead reads 0,  
  page server read-ahead reads 0,
```

```

lob logical reads 0,
lob physical reads 0,
lob page server reads 0,
lob read-ahead reads 0,
lob page server read-ahead reads 0.
[2025-05-28 22:47:31] [S0000][3612] SQL Server Execution Times:
[2025-05-28 22:47:31] CPU time = 1047 ms, elapsed time = 1319
ms.
[2025-05-28 22:47:31] completed in 1 s 341 ms

```

Plan zapytania:



Komentarz: indeks nieklastrowany zdecydowanie pogorszył zarówno koszt jak i czas zapytania.

- Kilkuatrybutowy:

```

CREATE NONCLUSTERED INDEX
idx_iot_device_readings_peripheralid_timestamp
ON iot_device_readings (peripheral_id, unix_timestamp);

```

Zrobiony na parze `peripheral_id`, `unix_timestamp`, ponieważ często mogą pojawiać się zapytania związane z konkretnym urządzeniem i czasem.

**Zapytanie bez indeksu**

```

iot> SELECT *
      FROM iot_device_readings
      WHERE peripheral_id = 55
            AND unix_timestamp BETWEEN 45781 AND 45795
      ORDER BY unix_timestamp DESC
[2025-05-28 22:52:04] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 22:52:04] CPU time = 138 ms, elapsed time = 154 ms.
[2025-05-28 22:52:04] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 22:52:04] CPU time = 0 ms, elapsed time = 0 ms.
[2025-05-28 22:52:04] [S0000][3615] Table 'Worktable'.
      Scan count 0,
      logical reads 0,

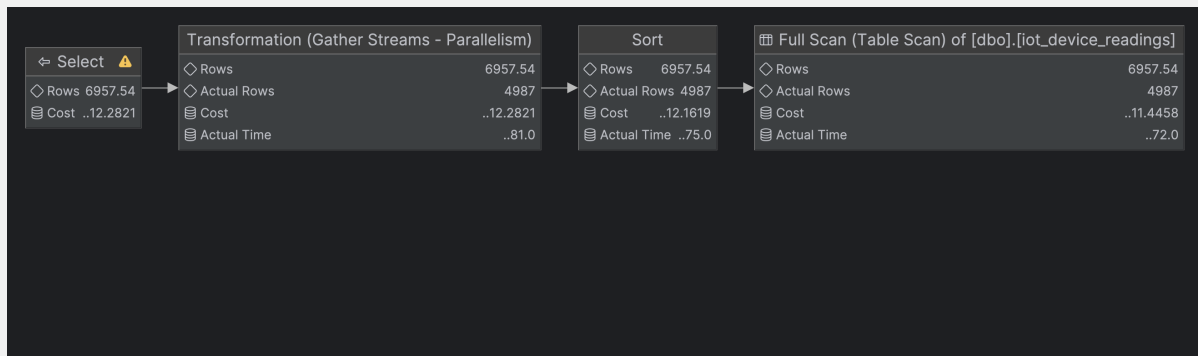
```

```

physical reads 0,
page server reads 0,
read-ahead reads 0,
page server read-ahead reads 0,
lob logical reads 0,
lob physical reads 0,
lob page server reads 0,
lob read-ahead reads 0,
lob page server read-ahead reads 0.
[2025-05-28 22:52:04] [S0000][3615] Table 'iot_device_readings'.
Scan count 5,
logical reads 14706,
physical reads 0,
page server reads 0,
read-ahead reads 11238,
page server read-ahead reads 0,
lob logical reads 0,
lob physical reads 0,
lob page server reads 0,
lob read-ahead reads 0,
lob page server read-ahead reads 0.
[2025-05-28 22:52:04] [S0000][3612] SQL Server Execution Times:
[2025-05-28 22:52:04] CPU time = 88 ms, elapsed time = 59 ms.
[2025-05-28 22:52:04] completed in 219 ms

```

Plan zapytania:



Zapytanie z indeksem

```

iot> SELECT *
      FROM iot_device_readings WITH
      (INDEX(idx_iot_device_readings_peripheralid_timestamp))
      WHERE peripheral_id = 55
         AND unix_timestamp BETWEEN 45781 AND 45795
      ORDER BY unix_timestamp DESC
[2025-05-28 22:53:36] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 22:53:36] CPU time = 11 ms, elapsed time = 12 ms.
[2025-05-28 22:53:36] [S0000][3615] Table 'iot_device_readings'.
Scan count 1,
logical reads 5007,

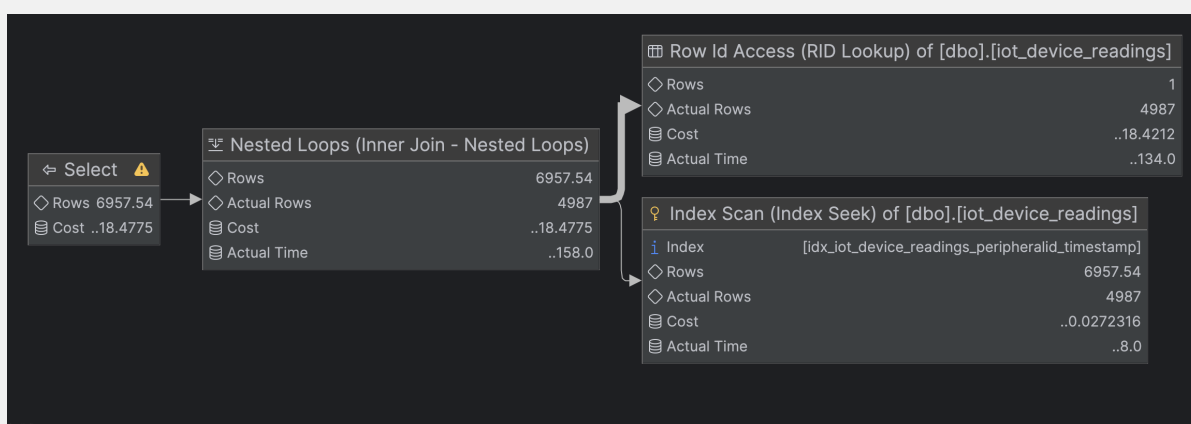
```

```

physical reads 1589,
page server reads 0,
read-ahead reads 1192,
page server read-ahead reads 0,
lob logical reads 0,
lob physical reads 0,
lob page server reads 0,
lob read-ahead reads 0,
lob page server read-ahead reads 0.
[2025-05-28 22:53:36] [S0000][3612] SQL Server Execution Times:
[2025-05-28 22:53:36] CPU time = 68 ms, elapsed time = 166 ms.
[2025-05-28 22:53:36] completed in 182 ms

```

Plan zapytania:



Komentarz: W tym wypadku jestem zdziwiony, że indeks kilkua trybutowy zwiększył koszt zapytania, ale na szczęście (przynajmniej na papierze) zmniejszył czas zapytania.

- Kilkua trybutowy z klazulą **INCLUDE**:

```

CREATE NONCLUSTERED INDEX
idx_iot_device_readings_peripheralid_timestamp_include_temp
ON iot_device_readings (peripheral_id, unix_timestamp)
INCLUDE (internal_temperature, external_temperature);

```

Pojawił się tutaj jako rozszerzenie powyższego, ale zawierając jeszcze w klazuli **INCLUDE** kolumny związane z temperaturą.

### Zapytanie bez indeksu

```

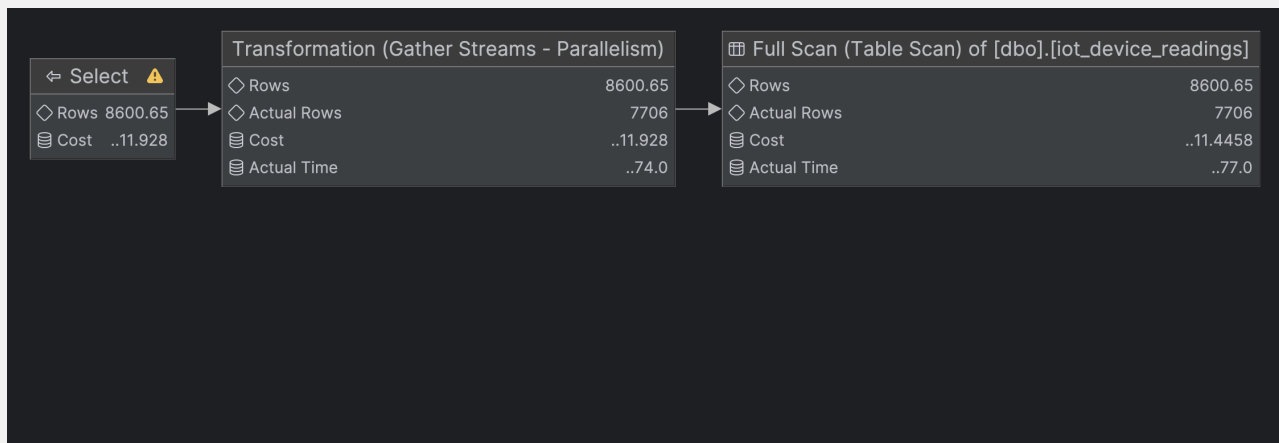
iot> SELECT internal_temperature, external_temperature
FROM iot_device_readings
WHERE peripheral_id = 55
AND unix_timestamp > 45781
[2025-05-28 22:57:58] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 22:57:58] CPU time = 13 ms, elapsed time = 15 ms.

```



```
[2025-05-28 22:57:58] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 22:57:58] CPU time = 0 ms, elapsed time = 0 ms.
[2025-05-28 22:57:58] [S0000][3615] Table 'iot_device_readings'.
    Scan count 5,
    logical reads 14706,
    physical reads 0,
    page server reads 0,
    read-ahead reads 14692,
    page server read-ahead reads 0,
    lob logical reads 0,
    lob physical reads 0,
    lob page server reads 0,
    lob read-ahead reads 0,
    lob page server read-ahead reads 0.
[2025-05-28 22:57:58] [S0000][3612] SQL Server Execution Times:
[2025-05-28 22:57:58] CPU time = 133 ms, elapsed time = 81 ms.
[2025-05-28 22:57:58] completed in 99 ms
```

Plan zapytania:



Zapytanie z indeksem

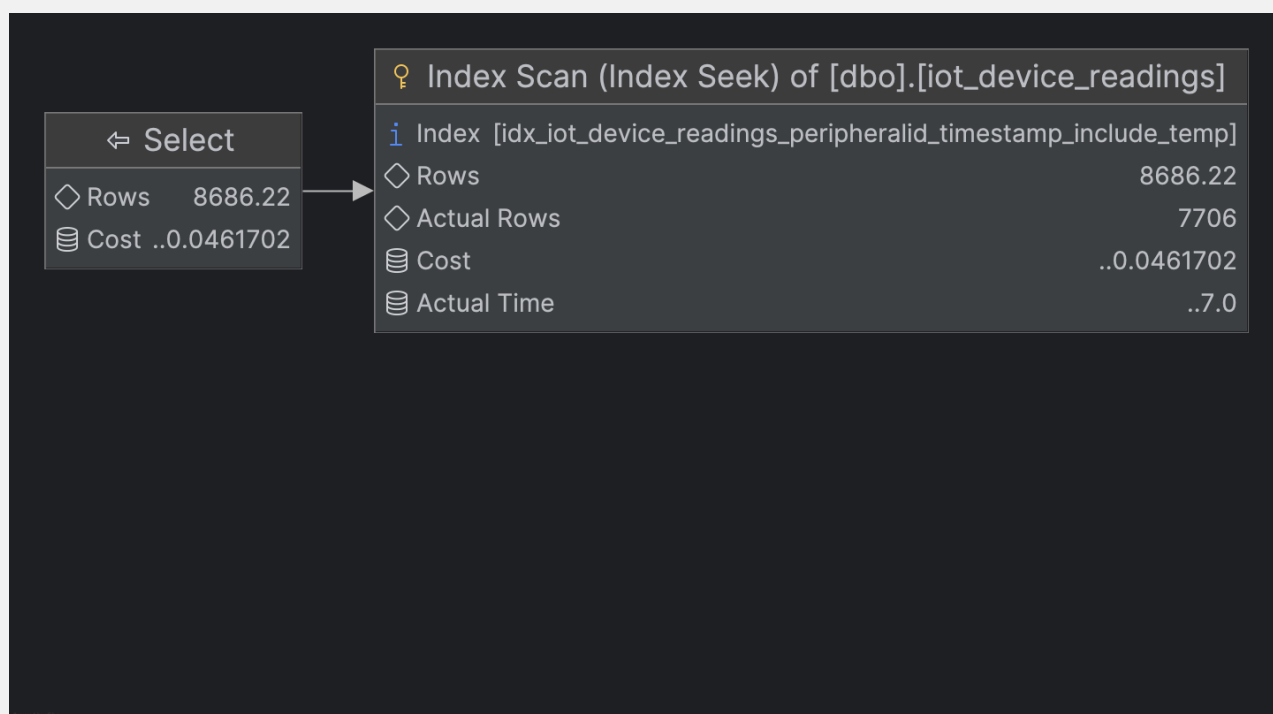
```
iot> SELECT internal_temperature, external_temperature
      FROM iot_device_readings WITH
      (INDEX(idx_iot_device_readings_peripheralid_timestamp_include_temp))
      WHERE peripheral_id = 55
         AND unix_timestamp > 45781
[2025-05-28 22:59:28] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 22:59:28] CPU time = 7 ms, elapsed time = 8 ms.
[2025-05-28 22:59:28] [S0000][3615] Table 'iot_device_readings'.
    Scan count 1,
    logical reads 44,
    physical reads 2,
    page server reads 0,
    read-ahead reads 41,
    page server read-ahead reads 0,
    lob logical reads 0,
```

```

lob physical reads 0,
lob page server reads 0,
lob read-ahead reads 0,
lob page server read-ahead reads 0.
[2025-05-28 22:59:28] [S0000][3612] SQL Server Execution Times:
[2025-05-28 22:59:28] CPU time = 10 ms, elapsed time = 11 ms.
[2025-05-28 22:59:28] completed in 23 ms

```

Plan zapytania:



Komentarz: w tym przypadku indeks kilkumtrybutowy z klauzulą **INCLUDE** zadziałał bardzo dobrze i kilkukrotnie zmniejszył czas zapytania oraz ogromnie zmniejszył jego koszt.

- Warunkowy:

```

CREATE NONCLUSTERED INDEX idx_iot_device_readings_status_event
ON iot_device_readings (peripheral_status)
WHERE peripheral_status = 1; -- 1 Means 'Event'

```

Związany z filtrowaniem po statusie urządzenia (równym 'Event').

### Zapytanie bez indeksu

```

iot> SELECT *
      FROM iot_device_readings
      WHERE peripheral_status = 1
[2025-05-28 23:01:30] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 23:01:30] CPU time = 10 ms, elapsed time = 11 ms.
[2025-05-28 23:01:30] [S0000][3613] SQL Server parse and compile

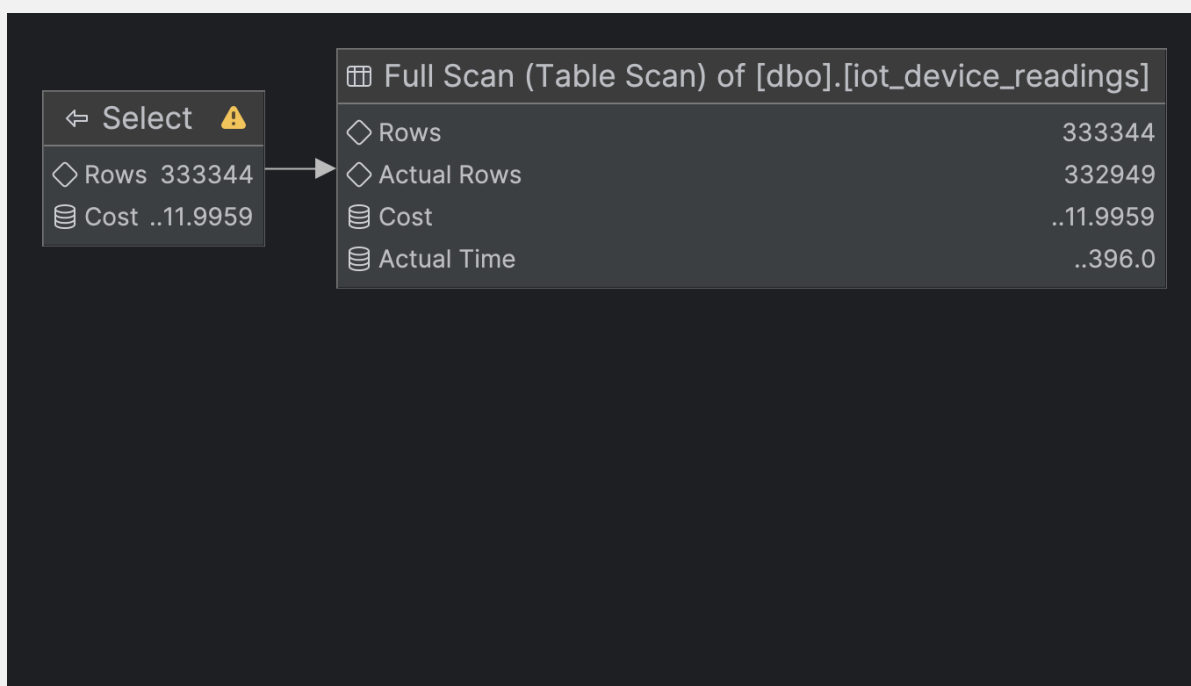
```

```

time:
[2025-05-28 23:01:30] CPU time = 0 ms, elapsed time = 0 ms.
[2025-05-28 23:01:30] [S0000][3615] Table 'iot_device_readings'.
    Scan count 1,
    logical reads 14706,
    physical reads 0,
    page server reads 0,
    read-ahead reads 14692,
    page server read-ahead reads 0,
    lob logical reads 0,
    lob physical reads 0,
    lob page server reads 0,
    lob read-ahead reads 0,
    lob page server read-ahead reads 0.
[2025-05-28 23:01:30] [S0000][3612] SQL Server Execution Times:
[2025-05-28 23:01:30] CPU time = 361 ms, elapsed time = 482 ms.
[2025-05-28 23:01:30] completed in 497 ms

```

Plan zapytania:



**Zapytanie z indeksem**

```

iot> SELECT *
      FROM iot_device_readings WITH
(INDEX(idx_iot_device_readings_status_event))
      WHERE peripheral_status = 1
[2025-05-28 23:02:44] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 23:02:44] CPU time = 17 ms, elapsed time = 18 ms.
[2025-05-28 23:02:44] [S0000][3615] Table 'iot_device_readings'.
    Scan count 1,
    logical reads 333695,
    physical reads 1739,

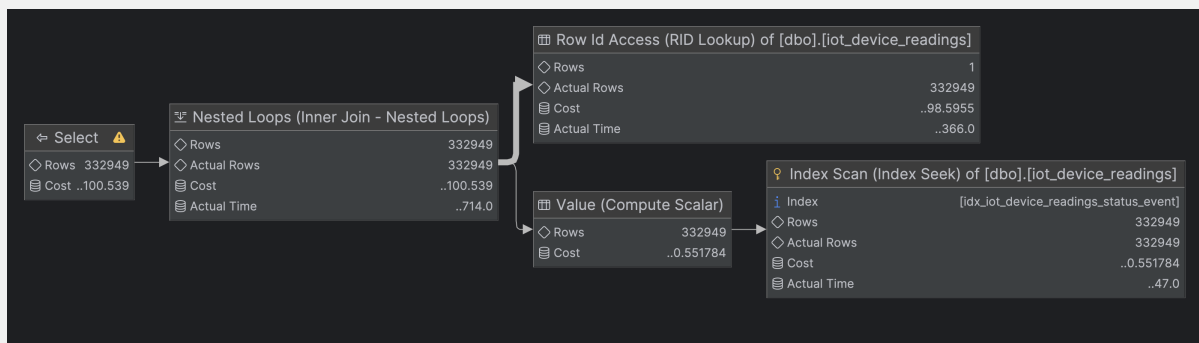
```

```

page server reads 0,
read-ahead reads 1550,
page server read-ahead reads 0,
lob logical reads 0,
lob physical reads 0,
lob page server reads 0,
lob read-ahead reads 0,
lob page server read-ahead reads 0.
[2025-05-28 23:02:44] [S0000][3615] Table 'Worktable'.
Scan count 0,
logical reads 0,
physical reads 0,
page server reads 0,
read-ahead reads 0,
page server read-ahead reads 0,
lob logical reads 0,
lob physical reads 0,
lob page server reads 0,
lob read-ahead reads 0,
lob page server read-ahead reads 0.
[2025-05-28 23:02:44] [S0000][3612] SQL Server Execution Times:
[2025-05-28 23:02:44] CPU time = 706 ms, elapsed time = 905 ms.
[2025-05-28 23:02:44] completed in 926 ms

```

Plan zapytania:



Komentarz: indeks warunkowy nie zadziałał w tym wypadku lepiej niż zwykłe zapytanie, pogarszając czas dwukrotnie i koszt zapytania około 10-krotnie.

- Kolumnowy:

```

CREATE NONCLUSTERED COLUMNSTORE INDEX
idx_iot_device_readings_columnstore
ON iot_device_readings (
    peripheral_id,
    battery_voltage,
    internal_temperature,
    external_temperature,
    acceleration_x,
    acceleration_y,

```

```

        acceleration_z
    );

```

Związany z zapytaniem analitycznymi dotyczącymi statystyk zbieranych przez urządzenia.

### Zapytanie bez indeksu

```

iot> SELECT
    peripheral_id,
    MAX(battery_voltage) AS max_battery_voltage,
    AVG(internal_temperature) AS avg_internal_temp,
    AVG(external_temperature) AS avg_external_temp,
    MIN(acceleration_x) AS min_acceleration_x,
    AVG(acceleration_y) AS avg_acceleration_y,
    MAX(acceleration_z) AS max_acceleration_z
FROM   iot_device_readings
GROUP BY peripheral_id
[2025-05-28 23:07:49] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 23:07:49] CPU time = 17 ms, elapsed time = 17 ms.
[2025-05-28 23:07:49] [S0000][3615] Table 'iot_device_readings'.
    Scan count 5,
    logical reads 14706,
    physical reads 0,
    page server reads 0,
    read-ahead reads 14692,
    page server read-ahead reads 0,
    lob logical reads 0,
    lob physical reads 0,
    lob page server reads 0,
    lob read-ahead reads 0,
    lob page server read-ahead reads 0.
[2025-05-28 23:07:49] [S0000][3615] Table 'Worktable'.
    Scan count 0,
    logical reads 0,
    physical reads 0,
    page server reads 0,
    read-ahead reads 0,
    page server read-ahead reads 0,
    lob logical reads 0,
    lob physical reads 0,
    lob page server reads 0,
    lob read-ahead reads 0,
    lob page server read-ahead reads 0.
[2025-05-28 23:07:49] [S0000][3612] SQL Server Execution Times:
[2025-05-28 23:07:49] CPU time = 298 ms, elapsed time = 138 ms.
[2025-05-28 23:07:49] completed in 160 ms

```

Plan zapytania:

↔ Select	Transformation (Gather Streams - Parallelism)	Value (Compute Scalar)	Aggregate (Aggregate - Hash Match)	Full Scan (Table Scan) of [dbo].[iot_device_readings]
↗ Rows 101	↗ Rows 101	↗ Rows 101	↗ Rows 101	↗ Rows 1e+6
↘ Actual Rows 101	↘ Actual Rows 101	↘ Actual Rows 101	↘ Actual Rows 101	↘ Actual Rows 1000000
⊞ Cost ..11.7154	⊞ Cost ..11.7154	⊞ Cost ..11.6865	⊞ Cost ..11.6865	⊞ Cost ..11.4458
⌚ Actual Time ..132.0	⌚ Actual Time ..132.0	⌚ Actual Time ..1.0	⌚ Actual Time ..29.0	⌚ Actual Time ..103.0

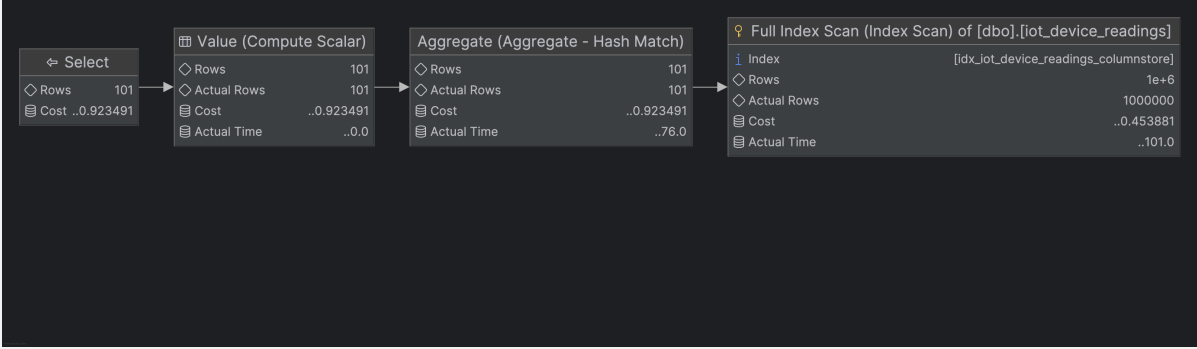
## Zapytanie z indeksem

```

iot> SELECT
    peripheral_id,
    MAX(battery_voltage) AS max_battery_voltage,
    AVG(internal_temperature) AS avg_internal_temp,
    AVG(external_temperature) AS avg_external_temp,
    MIN(acceleration_x) AS min_acceleration_x,
    AVG(acceleration_y) AS avg_acceleration_y,
    MAX(acceleration_z) AS max_acceleration_z
FROM iot_device_readings WITH
(INDEX(idx_iot_device_readings_columnstore))
GROUP BY peripheral_id
[2025-05-28 23:09:35] [S0000][3613] SQL Server parse and compile
time:
[2025-05-28 23:09:35] CPU time = 21 ms, elapsed time = 23 ms.
[2025-05-28 23:09:35] [S0000][3615] Table 'iot_device_readings'.
    Scan count 2,
    logical reads 0,
    physical reads 0,
    page server reads 0,
    read-ahead reads 0,
    page server read-ahead reads 0,
    lob logical reads 4011,
    lob physical reads 10,
    lob page server reads 0,
    lob read-ahead reads 12337,
    lob page server read-ahead reads 0.
[2025-05-28 23:09:35] [S0000][3642] Table 'iot_device_readings'.
    Segment reads 1,
    segment skipped 0.
[2025-05-28 23:09:35] [S0000][3612] SQL Server Execution Times:
[2025-05-28 23:09:35] CPU time = 165 ms, elapsed time = 201 ms.
[2025-05-28 23:09:35] completed in 230 ms

```

Plan zapytania:



Komentarz: indeks kolumnowy znacząco zmniejszył koszt zapytania, lecz zapytanie trwało delikatnie dłużej niż bez indeksu (co jest dla mnie dziwne).

Jestem za to pod wrażeniem jak szybko wykonało się zapytanie bez indeksu.

zadanie	pkt
1	2
2	2
3	2
4	2
5	5
razem	13