

Indeksy, optymalizator

Lab1

Imiona i nazwiska: Damian Torbus, Adam Woźny

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
- można dołączyć plik .md albo .sql

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

Stwórz swoją bazę danych o nazwie lab4.

```
create database lab1
go

use lab1
go
```

Część 1

Celem tej części ćwiczenia jest zapoznanie się z planami wykonania zapytań (execution plans) oraz narzędziem do automatycznego generowania indeksów.

Dokumentacja/Literatura

Przydatne materiały/dokumentacja. Proszę zapoznać się z dokumentacją:

- <https://docs.microsoft.com/en-us/sql/tools/dta/tutorial-database-engine-tuning-advisor>
- <https://docs.microsoft.com/en-us/sql/relational-databases/performance/start-and-use-the-database-engine-tuning-advisor>
- <https://www.simple-talk.com/sql/performance/index-selection-and-the-query-optimizer>
- <https://blog.quest.com/sql-server-execution-plan-what-is-it-and-how-does-it-help-with-performance-problems/>

Operatory (oraz reprezentujące je piktogramy/lkonki) używane w graficznej prezentacji planu zapytania opisane są tutaj:

- <https://docs.microsoft.com/en-us/sql/relational-databases/showplan-logical-and-physical-operators-reference>

Wykonaj poniższy skrypt, aby przygotować dane:

```
select * into [salesorderheader]
from [adventureworks2017].sales.[salesorderheader]
go

select * into [salesorderdetail]
from [adventureworks2017].sales.[salesorderdetail]
go
```

Zadanie 1 - Obserwacja

Wpisz do MSSQL Managment Studio (na razie nie wykonuj tych zapytań):

```
-- zapytanie 1
select *
from salesorderheader sh
inner join salesorderdetail sd on sh.salesorderid = sd.salesorderid
where orderdate = '2008-06-01 00:00:00.000'
go

-- zapytanie 1.1
select *
from salesorderheader sh
inner join salesorderdetail sd on sh.salesorderid = sd.salesorderid
where orderdate = '2013-01-28 00:00:00.000'
go

-- zapytanie 2
select orderdate, productid, sum(orderqty) as orderqty,
       sum(unitpricediscount) as unitpricediscount, sum(linetotal)
from salesorderheader sh
inner join salesorderdetail sd on sh.salesorderid = sd.salesorderid
group by orderdate, productid
having sum(orderqty) >= 100
go

-- zapytanie 3
select salesordernumber, purchaseordernumber, duedate, shipdate
from salesorderheader sh
inner join salesorderdetail sd on sh.salesorderid = sd.salesorderid
where orderdate in ('2008-06-01', '2008-06-02', '2008-06-03', '2008-06-04', '2008-
06-05')
go

-- zapytanie 4
select sh.salesorderid, salesordernumber, purchaseordernumber, duedate, shipdate
from salesorderheader sh
```

```
inner join salesorderdetail sd on sh.salesorderid = sd.salesorderid
where carriertrackingnumber in ('ef67-4713-bd', '6c08-4c4c-b8')
order by sh.salesorderid
go
```

Włącz dwie opcje: **Include Actual Execution Plan** oraz **Include Live Query Statistics**:

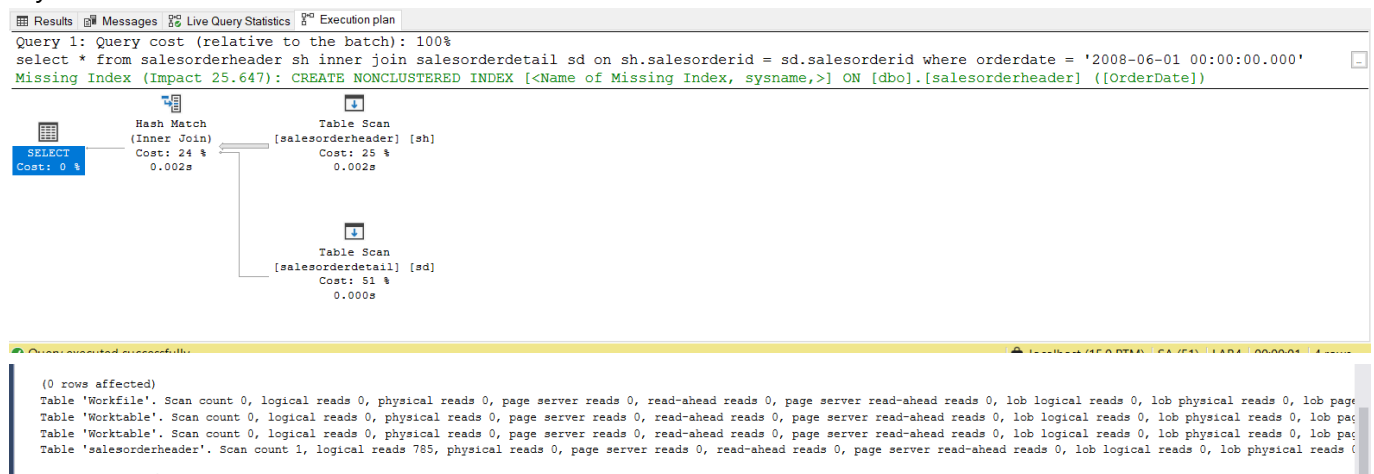


Teraz wykonaj poszczególne zapytania (najlepiej każde analizuj oddzielnie). Co można o nich powiedzieć? Co sprawdzają? Jak można je zoptymalizować?

Wyniki:

Zapytanie 1

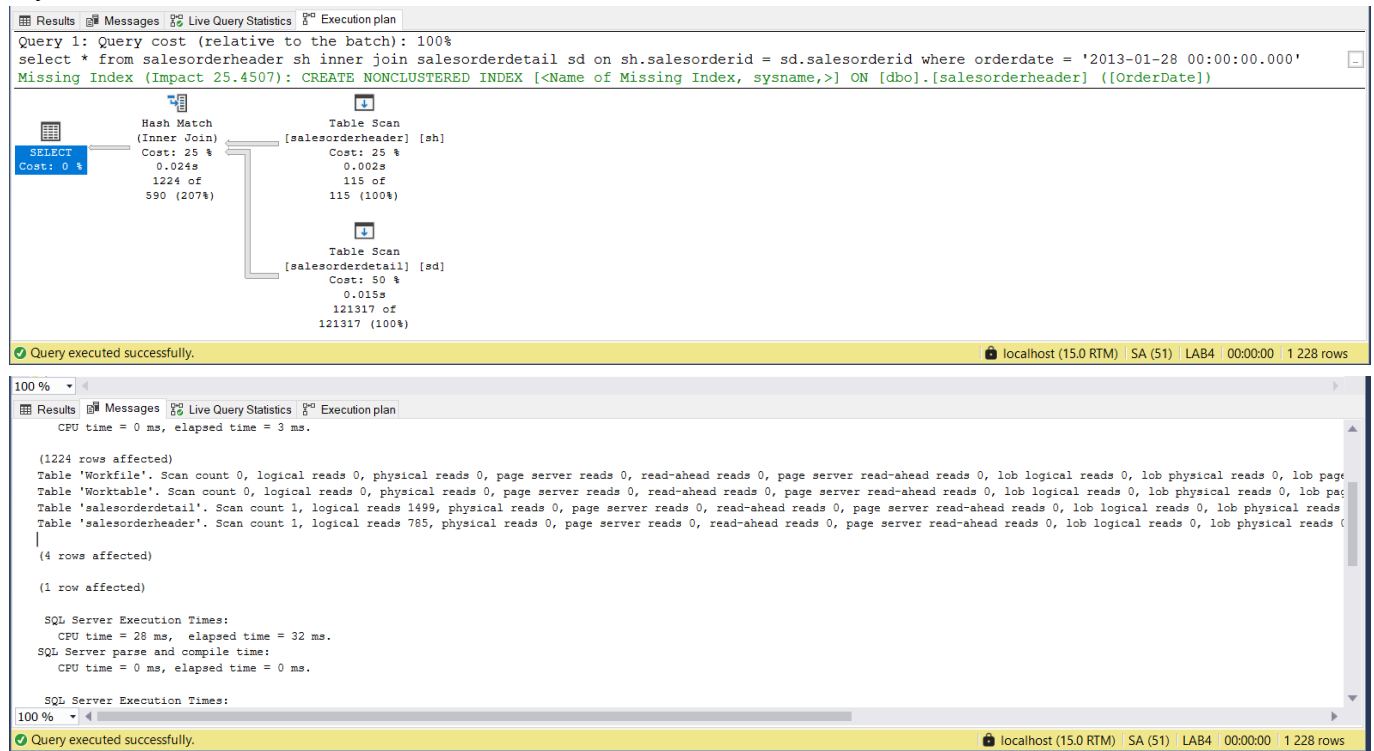
Wyniki:



Zapytanie zwracające dane z wszystkich kolumn z połączonych za pomocą `salesorderid` tabel `salesorderheader` oraz `salesorderdetail`, które zostało wykonane 2008-06-01 (nie istnieje takie). Jako, że nie ma tu żadnego indeksu nastąpiło pełne odczytanie tabeli `salesorderheader`, ale, że nie został znaleziony rekord odpowiadający klauzuli `WHERE` nie nastąpiło pełne odczytanie drugiej tabeli. Z racji na brak indeksów zapytanie jest bardzo nieefektywne. Proponowanym rozwiązaniem tego problemu jest założenie indeksu na klastrowego na kolumnie `salesorderid` (w tym konkretnym przypadku to nie jest bardzo ważne) oraz nieklastrowego na polu `orderdate`, żeby móc przyspieszyć wybieranie konkretnych ID oraz konkretnych danych. Indeksy również mogłyby być stworzone z klauzulą `include` zawierającą odpowiednie kolumny, bardzo rzadko są potrzebne wszystkie.

Zapytanie 1.1

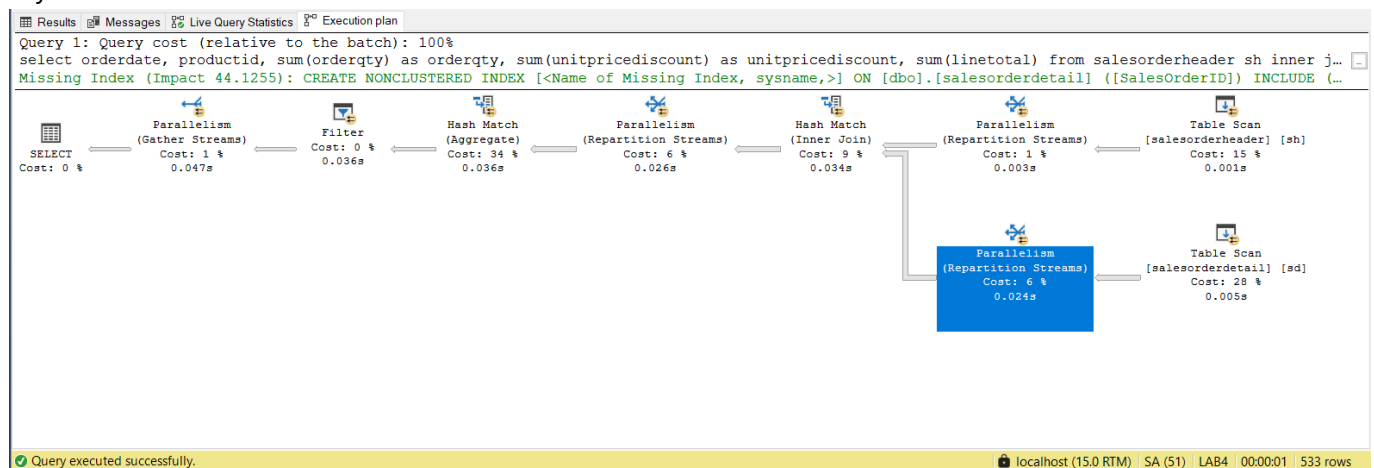
Wyniki:

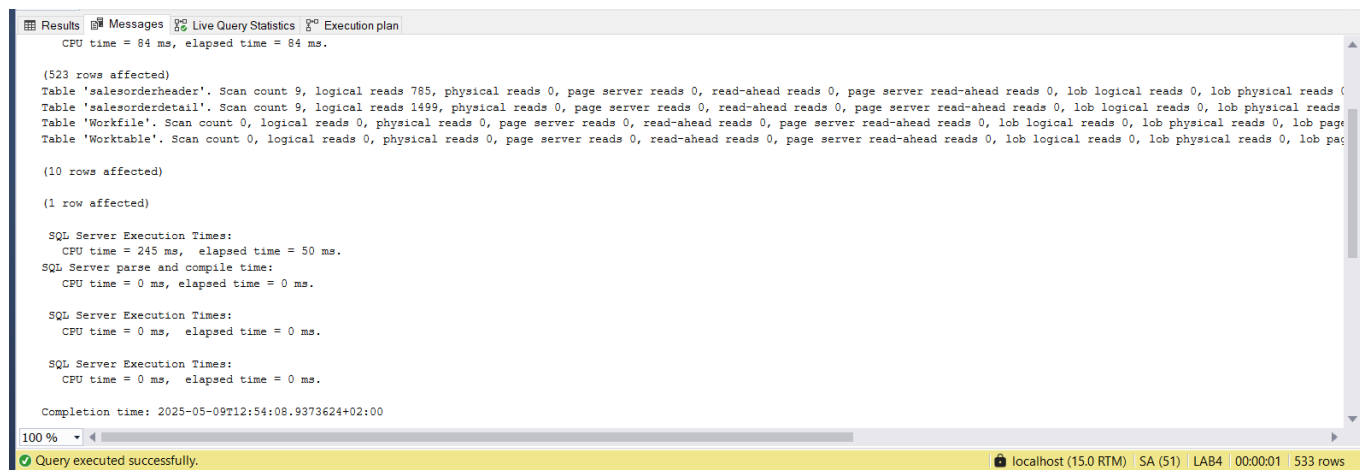


Zapytanie o bardzo podobnej charakterystyce co poprzednie, z tą różnicą, że ono zwraca rekordy, więc następuje też odczyt z drugiej tabeli (1499 logicznych operacji). Proponowane usprawnienia są również takie same jak w poprzednim przypadku.

Zapytanie 2

Wyniki:

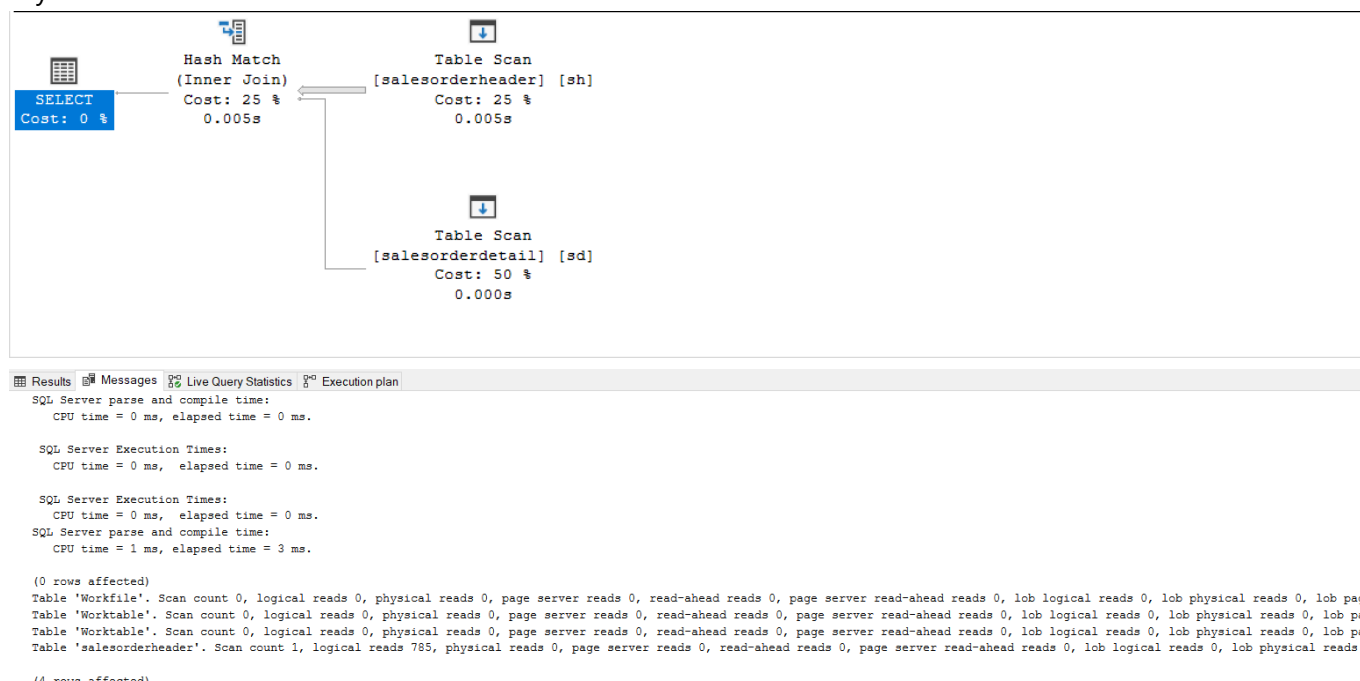




Zapytanie zwraca dla każdego produktu z tabeli `salesorderheader` sumę kupionych przedmiotów, sumaryczną cenę oraz sumę atrybutu `linetotal`. Korzysta z tabel `salesorderheader` oraz `salesorderdetail` połączonych za pomocą `salesorderid`. Wyniki są następnie sortowane po wartościach funkcji agregującej `sum(orderqty) >= 100`. Z powodu braku indeksów nastąpił pełny odczyt z obydwu tabel (785 + 1499 odczytów logicznych), co również jest bardzo nieefektywne. Można zaobserwować również, że agregacja wyników była bardzo kosztowna, co również może być spowodowane brakiem jakichkolwiek indeksów. Proponowanym usprawnieniem byłoby założenie indeksu klastrowego w obu tabelach na kolumnie `salesorderid` oraz nieklastrowych na kolumnach `orderdate`, `productid`.

Zapytanie 3

Wyniki:

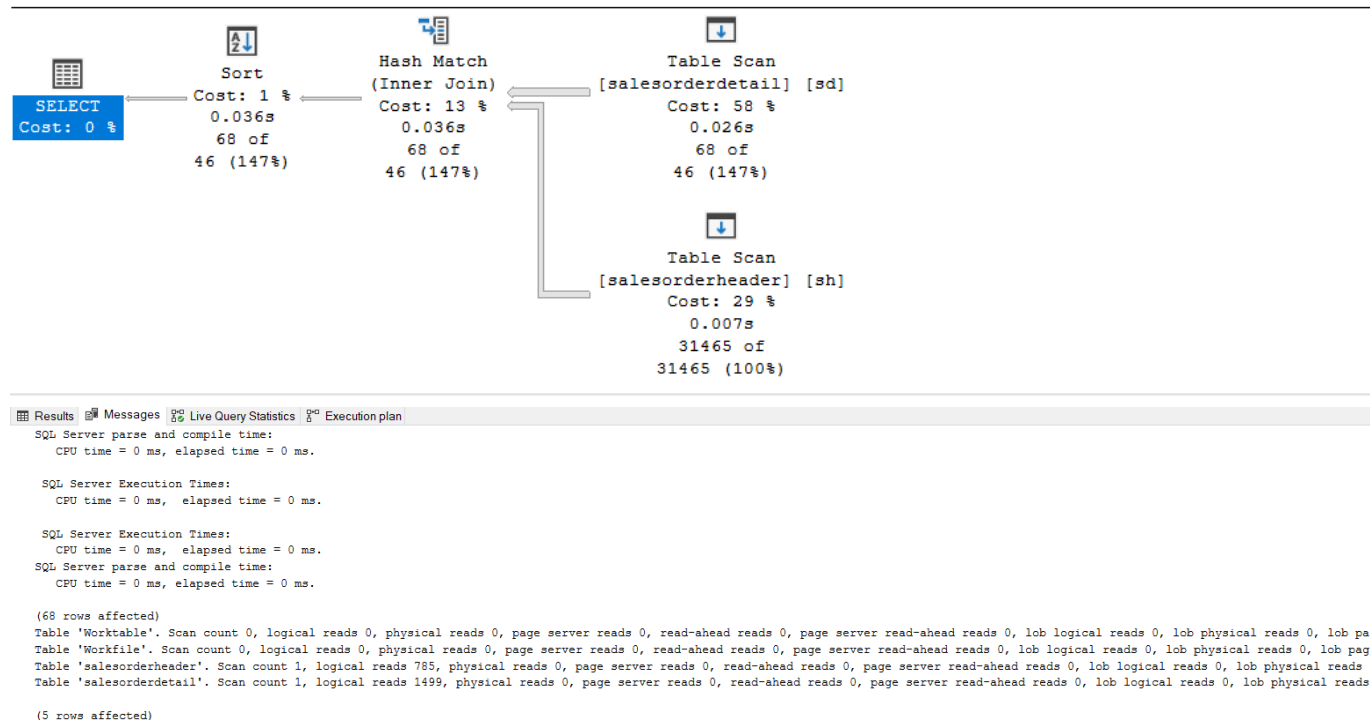


Zapytanie podobnie jak w zap.1 zwraca dane z połączonych za pomocą `salesorderid` tabel `salesorderheader` oraz `salesorderdetail`. Z tą różnicą, że zwracane są tylko dane z konkretnych kolumn oraz szukane są zamówienia wykonane w ciągu dat (również w żadnej z nich żadne zamówienie nie zostało wykonane) Jako, że nie ma tu żadnego indeksu nastąpiło pełne odczytanie tabeli `salesorderheader`, ale, że nie został znaleziony rekord odpowiadający klauzuli `WHERE` nie nastąpiło pełne odczytanie drugiej tabeli. Z racji na brak indeksów zapytanie jest bardzo nieefektywne . Proponowanym rozwiązaniem tego problemu jest założenie indeksu na klastrowego na kolumnie `salesorderid` (w tym konkretnym przypadku to nie jest

bardzo ważne) oraz nieklastrowego na polu `orderid`, żeby móc przyspieszyć wybieranie konkretnych ID oraz konkretnych datach.

Zapytanie 4

Wyniki:



Zapytanie zwraca dane z wybranych kolumn z połączonych za pomocą `salesorderid` tabel `salesorderheader` oraz `salesorderdetail`, które mają jedno z 2 wybranych `carriertrackingnumber`. Jako, że istnieją takie rekordy i nie ma żadnych indeksów to nastąpił pełny odczyt z obu tabel. Warto zauważyć również, że w tym zapytaniu występuje klauzula `order by`, która co prawda nie jest zbyt kosztowna (być może dlatego, że po filtrze zostaje tylko 68 rekordów), ale mogłaby być przeprowadzona znacznie efektywniej gdyby zastosować indeksy. Proponowanym usprawnieniem byłoby założenie indeksu klastrowego na kolumnie `salesorderid` oraz nieklastrowego na kolumnie `carriertrackingnumber`.

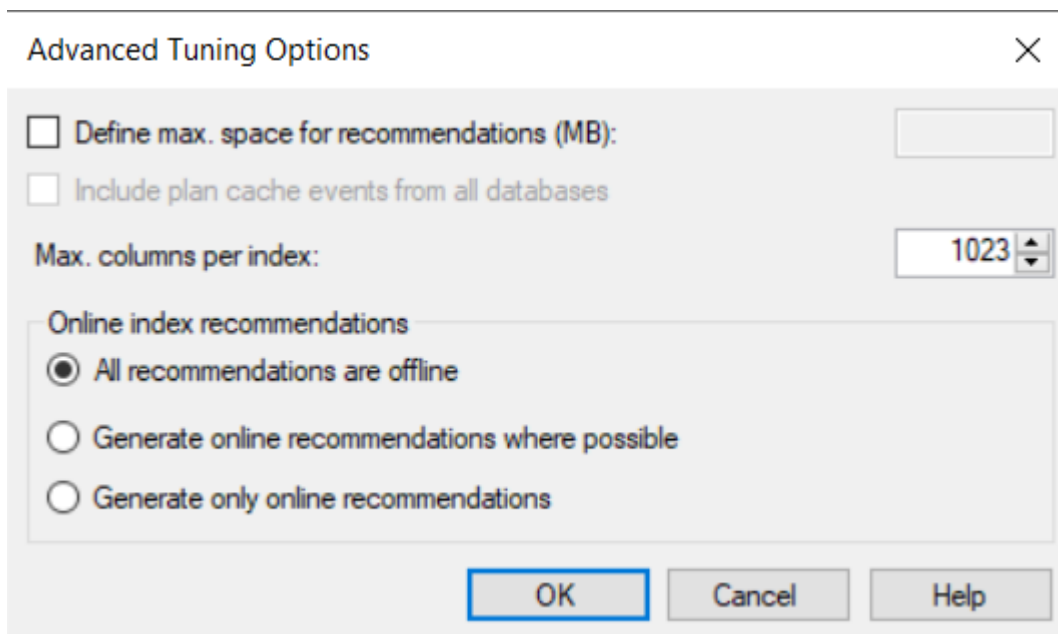
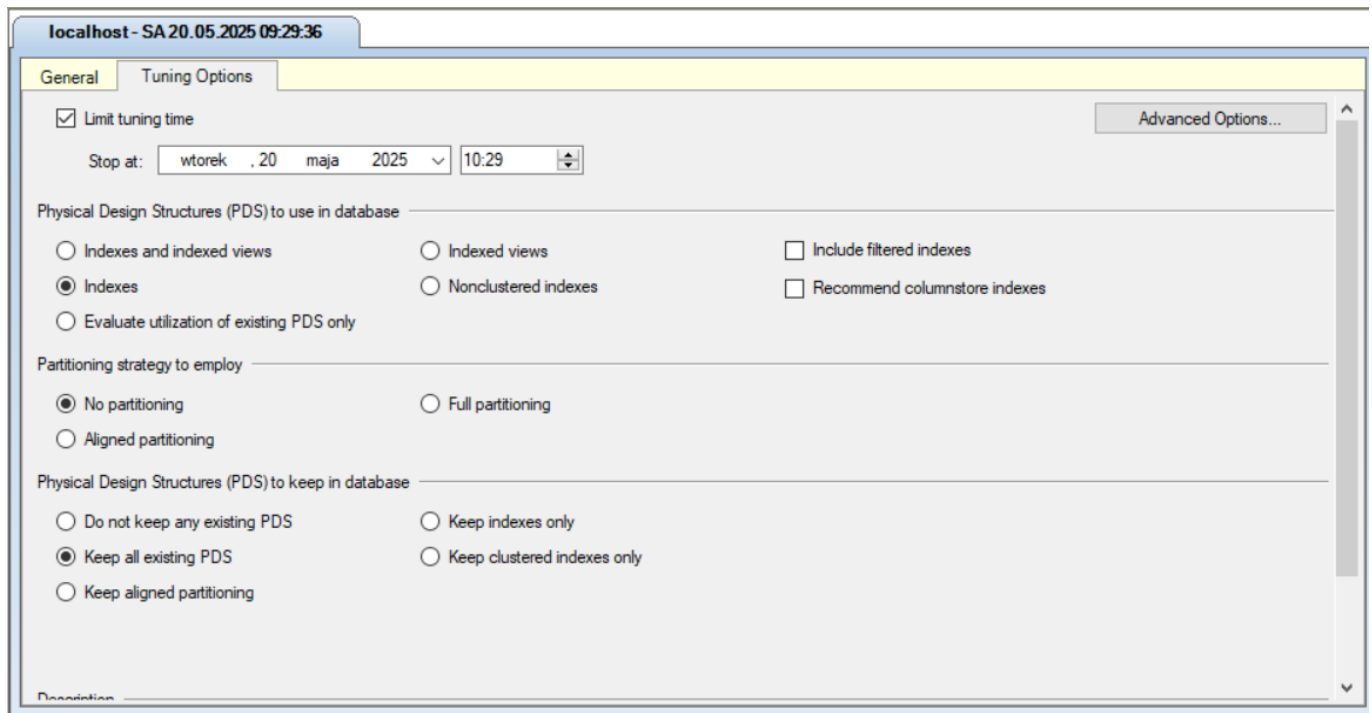
Zadanie 2 - Dobór indeksów / optymalizacja

Do wykonania tego ćwiczenia potrzebne jest narzędzie SSMS

Zaznacz wszystkie zapytania, i uruchom je w **Database Engine Tuning Advisor**:



Sprawdź zakładkę **Tuning Options**, co tam można skonfigurować?



W zakładce **Tuning Options** jest możliwość skonfigurowania

- limitu czasu tunowania - jak długo maksymalnie może potrwać proces ulepszania i rekomendacji
- obiekty, które są rozważane do utworzenia
 - indeksy i widoki indeksowane,
 - indeksy (klastrowe i nieklastrowe)
 - indeksy nieklastrowe
 - widoki indeksowane (nie ma bezpośrednich zmian w tabelach)
 - ewaluacja (nic nie zmienia, tylko patrzy jak dobrze wykorzystywane są już istniejące)
- obiekty, które są nie rozważane do zmiany
 - brak ograniczeń
 - brak modyfikacji
 - ograniczenie tylko do indeksów klastrowych
- podejście do partycjonowania
 - brak zmian

- pełne przeprojektowania
- przeprojektowanie z jak najmniejszym wkładem w to
- opcje zaawansowane
 - maksymalna ilość kolumn w indeksie
 - maksymalna ilość nowych struktur

Dla naszego przypadku (laboratorium z indeksów) została wybrana opcja żeby rozważyć stworzenie nowych indeksów, nie zostały nałożone żadne ograniczenia co do modyfikowania istniejących struktur (bo aktualnie żadne nie istnieją) oraz opcja **no partitioning**, ponieważ partycjonowania nie jest bezpośrednim tematem tego laboratorium.

Użyj **Start Analysis**:



Zaobserwuj wyniki w **Recommendations**.

Przejdź do zakładki **Reports**. Sprawdź poszczególne raporty. Główną uwagę zwróć na koszty i ich poprawę:

Wynikowe rekomendacje:

localhost - SA 20.05.2025 09:29:36

General

Tuning Options

Progress

Recommendations

Reports

Estimated improvement: 74%

Partition Recommendations

Index Recommendations

Database Name	Object Name	Recommendation	Target of Recommendation
LAB4	[dbo].[salesorderdetail]	create	_dta_index_salesorderdetail_7_597577167__K3_K1
LAB4	[dbo].[salesorderdetail]	create	_dta_index_salesorderdetail_7_597577167__K1_K5_4_8_9
LAB4	[dbo].[salesorderdetail]	create	_dta_index_salesorderdetail_7_597577167__K1_2_3_4_5_6_7_8_9_10_11
LAB4	[dbo].[salesorderheader]	create	_dta_index_salesorderheader_7_581577110__K3_1
LAB4	[dbo].[salesorderheader]	create	_dta_index_salesorderheader_7_581577110__K1_4_5_8_9
LAB4	[dbo].[salesorderheader]	create	_dta_index_salesorderheader_7_581577110__K3_K1_4_5_8_9
LAB4	[dbo].[salesorderheader]	create	_dta_index_salesorderheader_7_581577110__K3_K1_2_4_5_6_7_8_9_10_11_12_13_14_15_16_
LAB4	[dbo].[salesorderheader]	create	_dta_stat_581577110_1_3

☐ Show existing objects [See Reports for sizes of existing objects](#)

localhost - SA 20.05.2025 09:29:36

General

Tuning Options

Progress

Recommendations

Reports

Estimated improvement: 74%

Partition Recommendations

Index Recommendations

Index Name	Size (KB)	Definition
	2312	[(CarrierTrackingNumber) asc, (SalesOrderID) asc]
	4768	[(SalesOrderID) asc, (ProductID) asc] include ((OrderQty), (UnitPriceDiscount), (LineTotal))
	12000	[(SalesOrderID) asc] include ((SalesOrderDetailID), (CarrierTrackingNumber), (OrderQty), (ProductID), (SpecialOfferID), (UnitPrice), (UnitPriceDiscount), (LineTotal), (rowguid), (ModifiedDate))
	416	[(OrderDate) asc] include ((SalesOrderID))
	1336	[(SalesOrderID) asc] include ((DueDate), (ShipDate), (SalesOrderNumber), (PurchaseOrderNumber))
	1624	[(OrderDate) asc, (SalesOrderID) asc] include ((DueDate), (ShipDate), (SalesOrderNumber), (PurchaseOrderNumber))
	6288	[(OrderDate) asc, (SalesOrderID) asc] include ((RevisionNumber), (DueDate), (ShipDate), (Status), (OnlineOrderFlag), (SalesOrderNumber), (PurchaseOrderNumber), (AccountNumber), (CustomerID), (SalesPersonID), (OrderDate))

☐ Show existing objects [See Reports for sizes of existing objects](#)

Reporty

Tuning Reports

Select report: Statement cost report

Statement Id	Statement String	Percent Improvement	Statement Type
4	-- zapytanie 3 select salesordemumbe...	99.74	Select
1	-- zapytanie 1 select * from salesorder...	99.73	Select
5	-- zapytanie 4 select sh.salesorderid, ...	93.13	Select
2	-- zapytanie 1.1 select * from salesord...	85.07	Select
3	-- zapytanie 2 select orderdate, produ...	28.59	Select

Tuning Reports

Select report: Index usage report (recommended)

Database Name	Schema Name	Table/View Name	Index Name	Number of references	Percent Usage
LAB4	dbo	salesorderdetail	_dta_index_salesorderdetail_7_5975...	3	60.00
LAB4	dbo	salesorderheader	_dta_index_salesorderheader_7_581...	2	40.00
LAB4	dbo	salesorderheader	_dta_index_salesorderheader_7_581...	1	20.00
LAB4	dbo	salesorderheader	_dta_index_salesorderheader_7_581...	1	20.00
LAB4	dbo	salesorderdetail	_dta_index_salesorderdetail_7_5975...	1	20.00
LAB4	dbo	salesorderdetail	_dta_index_salesorderdetail_7_5975...	1	20.00
LAB4	dbo	salesorderheader	_dta_index_salesorderheader_7_581...	1	20.00



Zapisz poszczególne rekomendacje:

Uruchom zapisany skrypt w Management Studio.

Opisz, dlaczego dane indeksy zostały zaproponowane do zapytań:

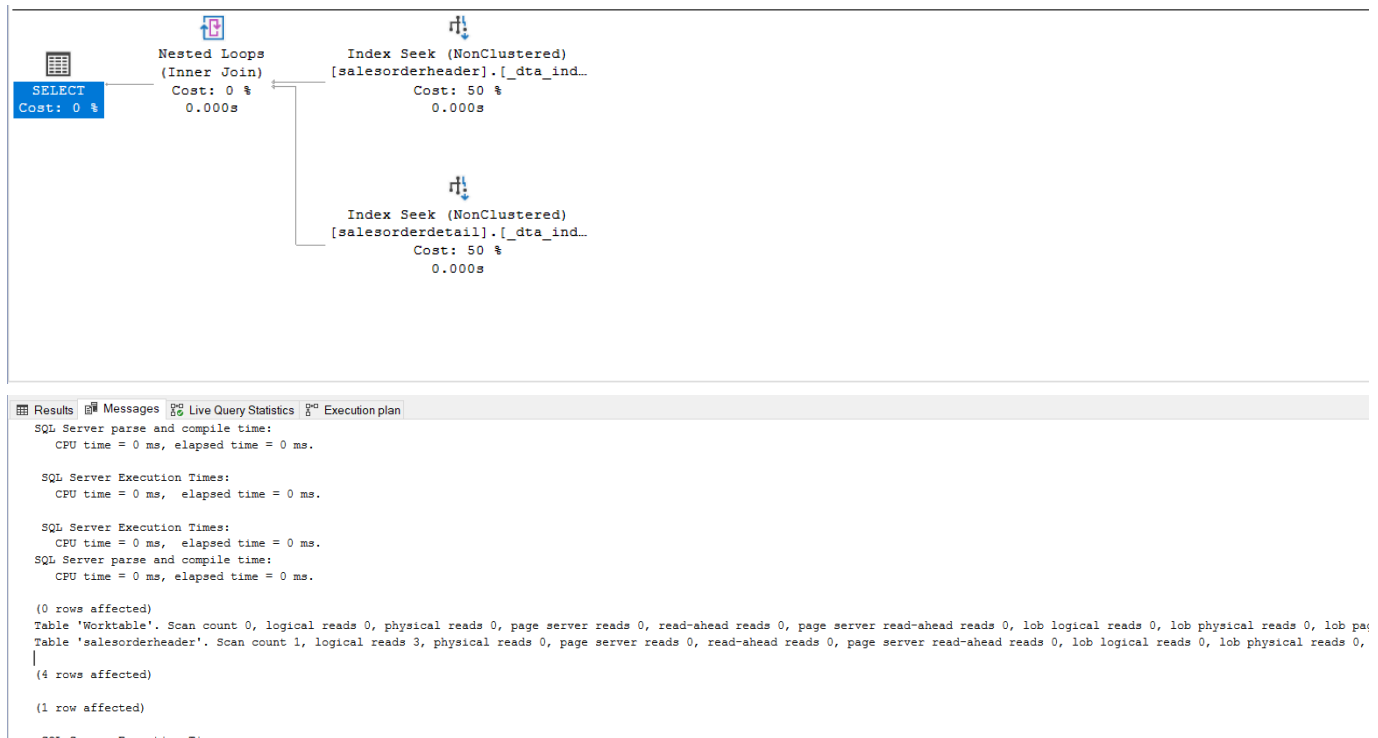
Zostało zaproponowane, aby stworzyć 7 indeksów nieklastrowych. Wygląda na to, że każdy z indeksów został stworzony specjalnie z myślą o konkretnym zapytaniu. Dla każdej konfiguracji zwracanych kolumn oraz kolumn do filtrowania, agregacji, joinów został stworzony osobny indeks. Możemy zoobserwować duży rozmiar indeksów spowodowanych użyciem `select * ...` ponieważ powoduje to stworzeniem indeksu zawierającego wszystkie kolumny. Indeksy zwykle są robione według szablonu (`_wszystkie kolumny do filtrowania, agregacji, joinów`) include (`_wszystkie zwracane kolumny`)

Można zauważyć, że następują kosmiczne wręcz zmniejszenie kosztów zapytań. Dzieje się tak, ponieważ dzięki zastosowaniu indeksów nie ma potrzeby wykonywania bardzo drogich operacji `scan` (które skanują każdą stronę) tylko można je zastąpić operacjami `seek` (które skanują tylko wybrane strony).

Sprawdź jak zmieniły się Execution Plans. Opisz zmiany:

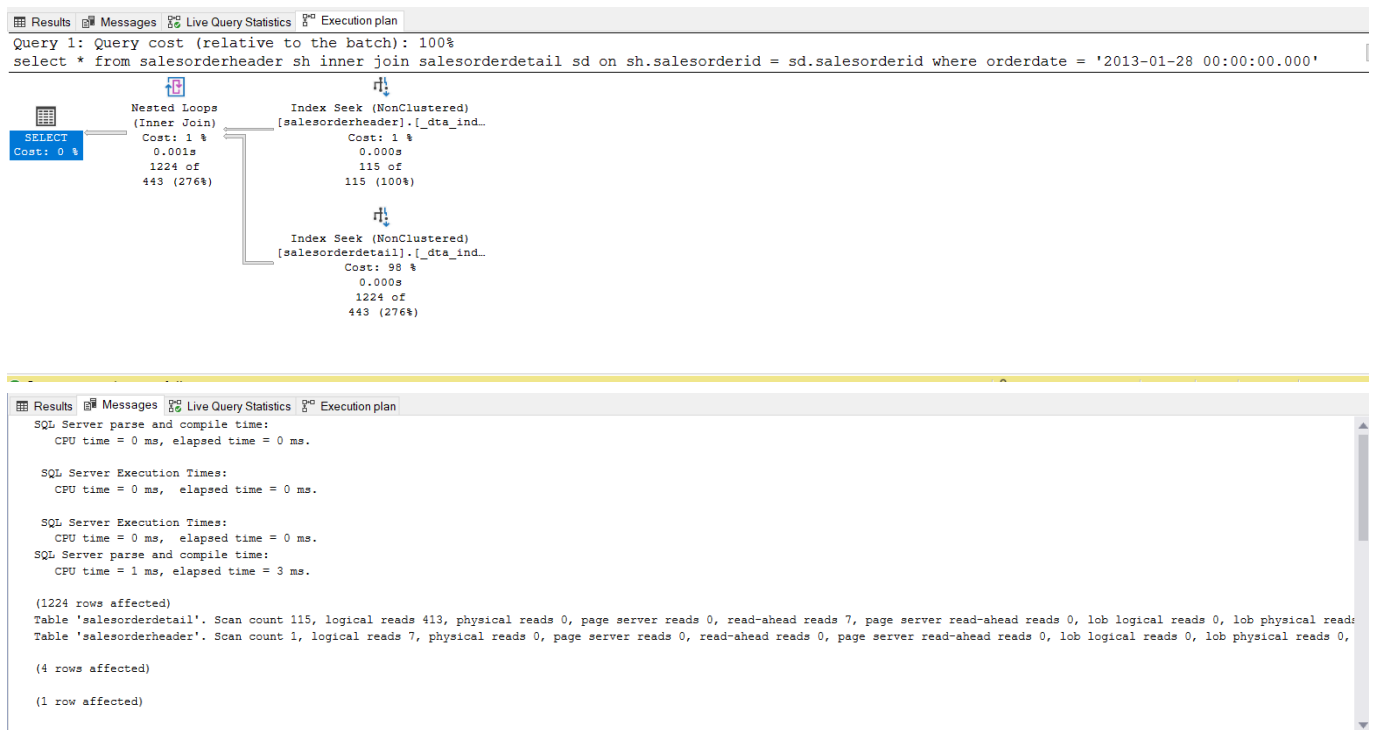
Wyniki: [[lab2-index-opt]]

Zapytanie 1



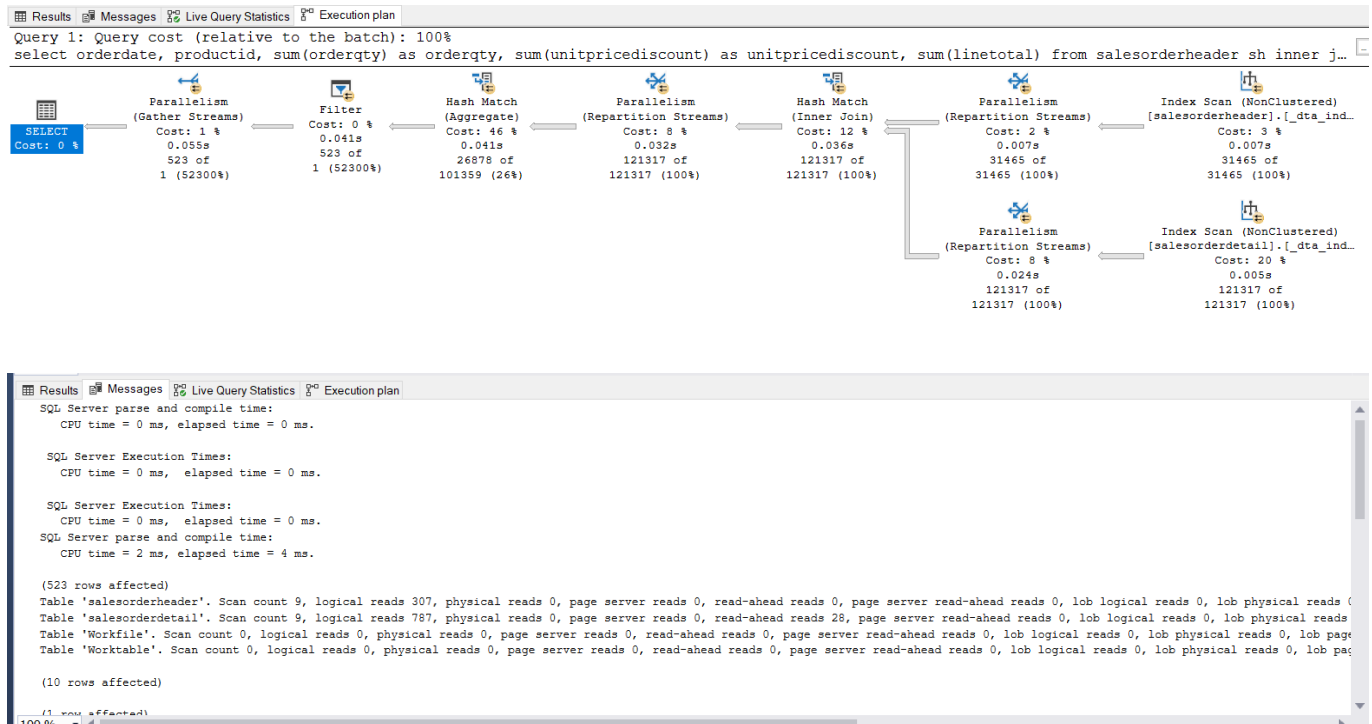
Można zauważyć zmianę operacji **scan** na **seek**, oznacza to, że nie są przeszukiwane wszystkie strony tabeli. Z tego powodu nastąpiła znaczna poprawa liczby przeczytanych stron z 785 do tylko 3. Jako, że nie istnieje rekord o podanych warunkach również nastąpił odczyt z tylko jednej tabeli.

Zapytanie 1.1



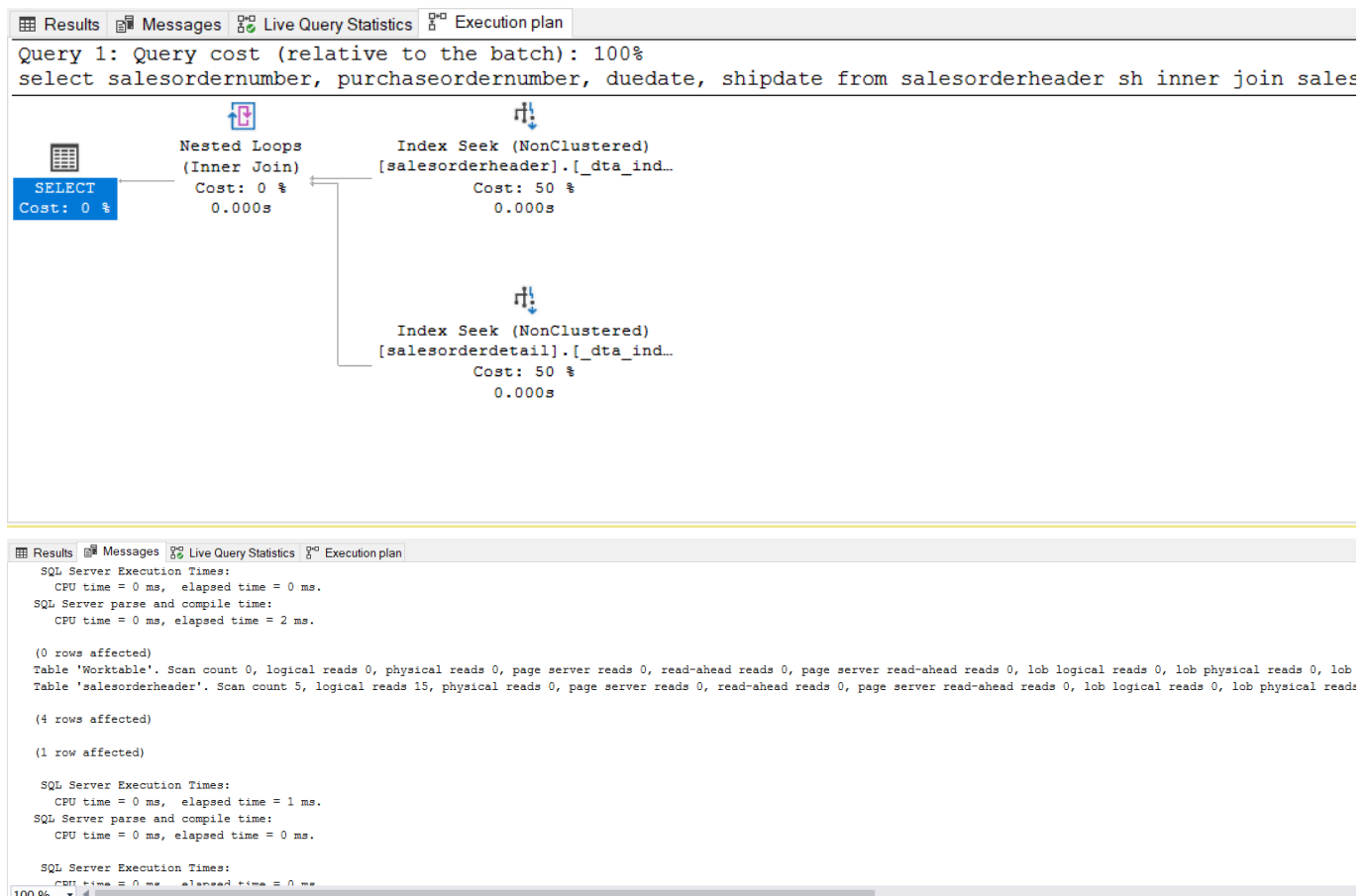
Tutaj również nastąpiła zmiana operacji z **scan** na **seek**, ale jako, że już takie rekordy istnieją to nastąpił też odczyt z drugiej tabeli w której w prawdzie nastąpiło zmniejszenie liczby odczytanych stron, ale już nie takie spektakularne (z 1499 na 476)

Zapytanie 2



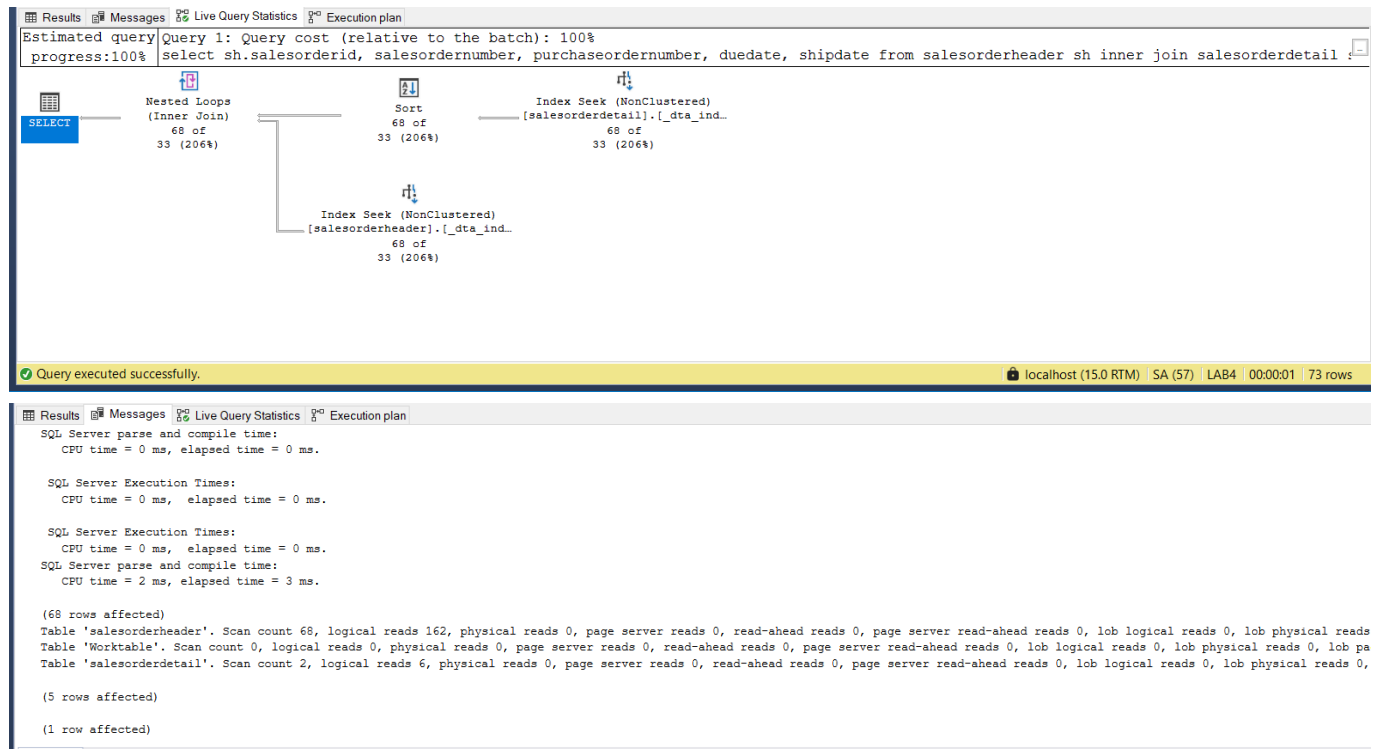
Po dodaniu indeksów widoczna jest zmiana operacji z **scan** na **seek**, co znacząco ograniczyło koszt zapytania. Dzięki temu operacja Group By miała mniej danych do przetworzenia – liczba logicznych odczytów spadła z 2284 (785 + 1499) do zaledwie kilkudziesięciu. Koszt agregacji również się zmniejszył, ponieważ dane były już częściowo posortowane i zoptymalizowane pod kątem wykonania przez silnik SQL Server.

Zapytanie 3



Podobnie jak w poprzednich przypadkach, operacja scan została zamieniona na seek dzięki indeksowi na kolumnie orderdate. Pomimo braku wyników odpowiadających warunkowi WHERE, SQL Server przeszukał tylko odpowiednie strony danych, co ograniczyło zużycie zasobów. Liczba odczytów w tabeli salesorderheader spadła z kilkuset do kilku, natomiast z racji braku pasujących rekordów nie nastąpił odczyt z tabeli salesorderdetail.

Zapytanie 4



W wyniku utworzenia indeksu nieklastrowego na kolumnie carriertrackingnumber, zapytanie przeszło z pełnego scan na wydajny seek. Dodatkowo, ponieważ ORDER BY salesorderid odpowiada naturalnemu porządkowi indeksu klastrowego (jeśli salesorderid to klucz klastra), operacja sortowania była tańsza lub wręcz pominięta. Liczba odczytanych stron w obu tabelach znacznie się zmniejszyła (z 785 i 1499 do kilkudziesięciu lub mniej), co wpłynęło na bardzo zauważalne przyspieszenie działania tego zapytania

Część 2

Celem ćwiczenia jest zapoznanie się z różnymi rodzajami indeksów oraz możliwością ich wykorzystania

Dokumentacja/Literatura

Przydatne materiały/dokumentacja. Proszę zapoznać się z dokumentacją:

- <https://docs.microsoft.com/en-us/sql/relational-databases/indexes/indexes>
- <https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide>
- <https://www.simple-talk.com/sql/performance/14-sql-server-indexing-questions-you-were-too-shy-to-ask/>
- <https://www.sqlshack.com/sql-server-query-execution-plans-examples-select-statement/>

Zadanie 3 - Indeksy klastrowane I nieklastrowane

Skopiuj tabelę **Customer** do swojej bazy danych:

```
select * into customer from adventureworks2017.sales.customer
```

Wykonaj analizy zapytań:

```
select * from customer where storeid = 594
```

```
select * from customer where storeid between 594 and 610
```

Zanotuj czas zapytania oraz jego koszt koszt:

Wyniki: Każde z zapytań wybiera wartość wszystkich kolumn z pewnymi warunkami dla wierszy, które są różne dla zapytania.

Zapytanie 1

The screenshot displays the SQL Server Enterprise Manager interface. At the top, the query text is shown: "Query 1: Query cost (relative to the batch): 100% SELECT * FROM [customer] WHERE [storeid]=@1". Below the query, the execution plan is visualized, showing a "Table Scan [customer]" operation with a cost of 100 and 0.001s. The "Results" tab is active, showing the query results. The results are summarized as follows:

- (1 row affected)
- Table 'customer'. Scan count 1, logical reads 155, 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.
- (2 rows affected)
- (1 row affected)

Below the results, the SQL Server Execution Times are displayed:

- SQL Server Execution Times: CPU time = 0 ms, elapsed time = 2 ms.
- SQL Server parse and compile time: CPU time = 0 ms, elapsed time = 0 ms.
- SQL Server Execution Times: CPU time = 0 ms, elapsed time = 0 ms.
- SQL Server Execution Times: CPU time = 0 ms, elapsed time = 0 ms.

The status bar at the bottom indicates "Query executed successfully." and "localhost (15.0 RTM) | SA (52) | LAB4 | 00:00:01 | 3 rows".

Zapytanie zostało zrealizowane poprzez kosztowną operację **scan**, ponieważ gdy nie ma indeksu, żeby zastosować warunek z klauzuli **WHERE** trzeba przejrzeć każdy rekord. Został zwrócony 1 rekord z 8 kolumnami. Czas trwania wynosił 2ms, a jego koszt 0,1391581. Zostało wykonane 155 logicznych odczytów.

Zapytanie 2

Results

Messages

Live Query Statistics

Execution plan

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

SELECT * FROM [customer] WHERE [storeid]>=@1 AND [storeid]<=@2

Table Scan

[customer]

Cost: 100 %

0.001s

SELECT

Cost: 0 %

Results

Messages

Live Query Statistics

Execution plan

(16 rows affected)

Scan count 1, logical reads 155, 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, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob page server reads 0, read-ahead reads 0, page server read-ahead reads 0

(2 rows affected)

(1 row affected)

SQL Server Execution Times:

CPU time = 2 ms, elapsed time = 2 ms.

SQL Server parse and compile time:

CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 0 ms.

100 %

Zapytanie zostało tak samo jak poprzednie zrealizowane przez kosztowną operację **scan**, z tych samych powodów, również aby zebrać wszystkie rekordy spełniające klauzulę **WHERE** trzeba je wszystkie przeczytać. Czas również wynosił 2ms, koszt 0,1391581. Również zostało wykonane 155 logicznych odczytów.

Dodaj indeks:

```
create index customer_store_cls_idx on customer(storeid)
```

Jak zmienił się plan i czas? Czy jest możliwość optymalizacji?

Wyniki:

Zapytanie 1

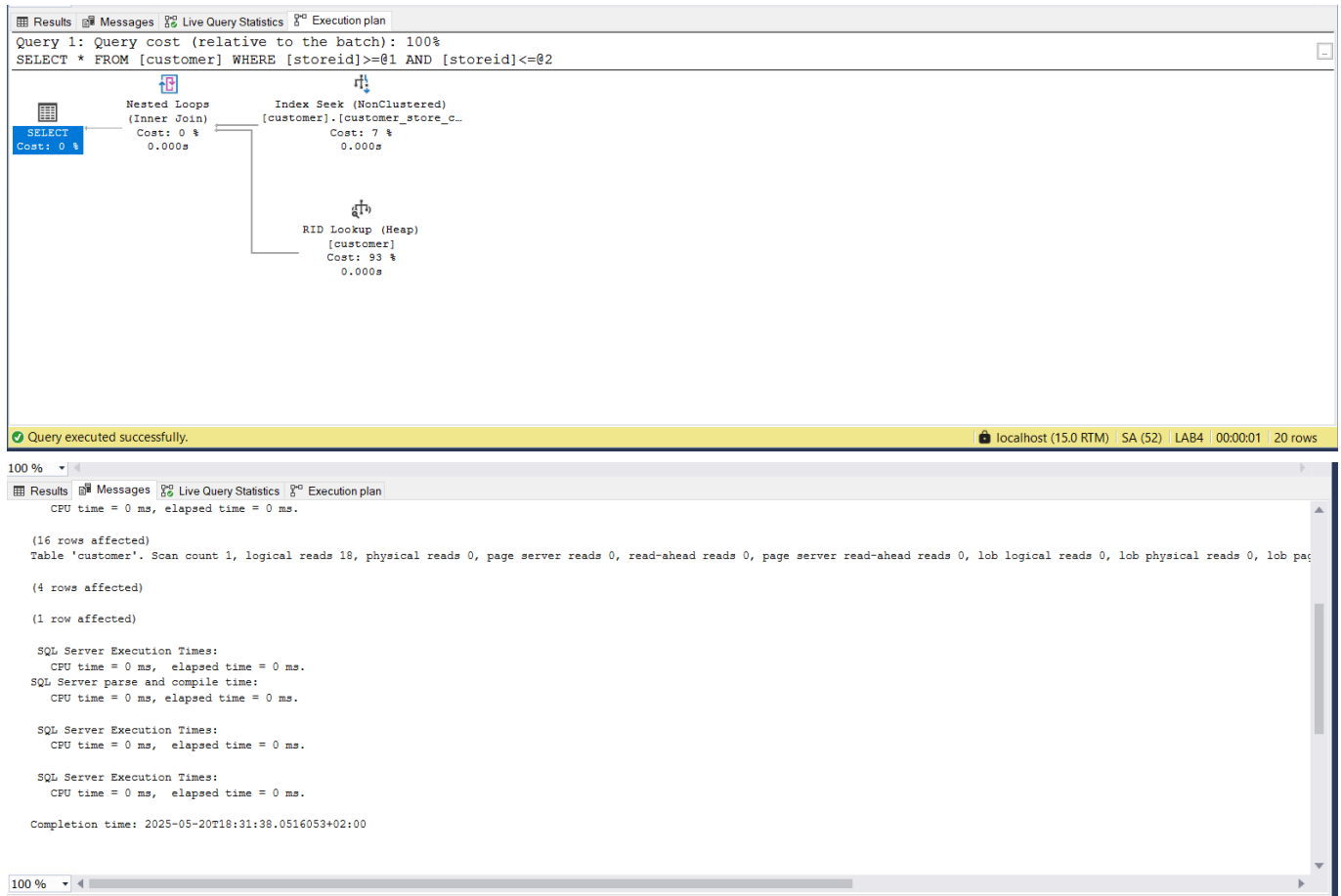
The screenshot displays the SQL Server Enterprise Manager interface. The top pane shows the query execution plan for the query: `SELECT * FROM [customer] WHERE [storeid]=@1`. The plan consists of a **SELECT** operator (Cost: 0%, 0.000s) connected to a **Nested Loops (Inner Join)** operator (Cost: 0%, 0.000s). The **Nested Loops** operator is connected to an **Index Seek (NonClustered)** operator (Cost: 50%, 0.000s) and an **RID Lookup (Heap)** operator (Cost: 50%, 0.000s). The **Index Seek** operator is connected to the **RID Lookup** operator.

The bottom pane shows the results of the query. It indicates that 1 row was affected by the query. The results show the table 'customer' with 1 logical read and 3 physical reads. The completion time is 2025-05-20T18:19:10.2202690+02:00.

Query executed successfully. localhost (15.0 RTM) | SA (52) | LAB4 | 00:00:01 | 5 rows

Można zauważyć znacząco różny plan wykonania zapytania, tym razem zamiast operacji **scan** została wykonana operacja **seek** oraz **rid lookup**. Możemy zaobserwować również znaczący spadek kosztu (0,1391581 => 0,00657038), ilości odczytów logicznych (155 => 3) oraz czasu (0,002 => 0,0001581). Dzieje się tak dlatego, że dzięki założeniu indeksu, nie ma już potrzeby odczytania każdego rekordu, żeby znaleźć wszystkie spełniające klauzulę **WHERE**, a w zasadzie wystarczy tylko z +-jednej strony (plus też koszt dojścia do tej strony). Potencjalnym polem do optymalizacji zapytania jest wyleminowanie potencjalnie drogiej operacji **rid lookup**, która odpowiada za dołączenie do wyniku zapytania wartości kolumn innych niż ten na których jest założony (lub tych, które uwzględnia w swojej strukturze) indeks nieklastrowy, poprzez zredukowanie liczby zwracanych kolumn, dołożenie poprzez klauzulę **include** kolumn do indeksu, które chcemy zobaczyć w wyniku, ewentualnie założyć indeks na kolumnach które chcemy zobaczyć w wyniku (to nie jest zbyt dobry pomysł, indeks będzie bardzo duży, a wcale nie filtrujemy po innych kolumnach) lub zastosowanie indeksu klastrowanego

Zapytanie 2



Ogólny plan zapytania zmienił się podobnie jak w punkcie wyżej, ale tutaj możemy zaobserwować o wiele mniej spektakularne poprawy kosztów: czasu z 0,002 => 0,0001746, kosztu z 0,1391581 => 0,0507122 oraz ilości odczytów logicznych 155 => 18. Dzieje się tak za sprawą wysokiego kosztu operacji **rid lookup** (wykonuje się ona dla rekordów, które spełniają klauzulę **WHERE**, których w tym przypadku jest o wiele 16x więcej niż w poprzednim). W tym przypadku opisane wyżej metody optymalizacji pozwolą na jeszcze większy postęp, bo element zapytania, który jest przez nie redukowany zajmuje procentowo o wiele większą część całego wykonania zapytania.

Dodaj indeks klastrowany:

```
create clustered index customer_store_cls_idx on customer(storeid) -- nie
zadziała, zdublowanie nazwy
```

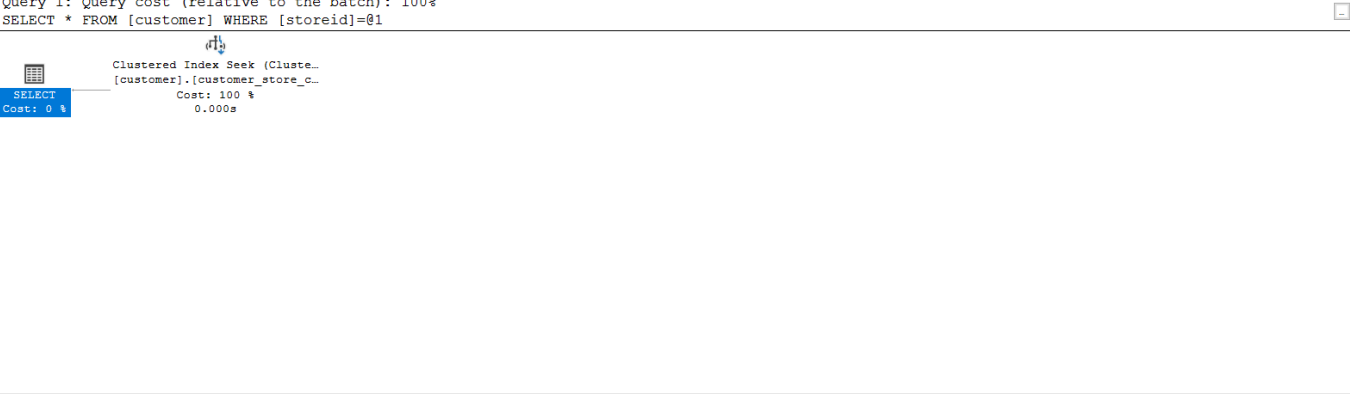
Czy zmienił się plan/koszt/czas? Skomentuj dwa podejścia w wyszukiwaniu krotek.

Wyniki:

Zapytanie 1

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

```
SELECT * FROM [customer] WHERE [storeid]=@1
```



Clustered Index Seek (Clustered Index)

[customer].[customer_store_c]

Cost: 100 %

0.000s

Results Messages Live Query Statistics Execution plan

CPU time = 0 ms, elapsed time = 0 ms.

(1 row affected)

Table 'customer'. Scan count 1, logical reads 2, 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

(2 rows affected)

(1 row affected)

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 0 ms.

SQL Server parse and compile time:

CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2025-05-20T18:45:21.0046995+02:00

100 %

Znowu możemy zaobserwować znaczącą zmianę planu zapytania, zamiast operacji **seek** oraz **rid lookup** została wykonana operacja **seek**, ale na indeksie klastrowym. Możemy zaobserwować zmniejszenie się czasu 0,0001746 => 0,0001581, kosztu 0,00657038 => 0,0032831. Ilość logicznych odczytów stron wynosi 2. Spadek kosztów wynika z użycia indeksu klastrowanego, który z racji na swoją budowę (w zasadzie z racji na to, że fizycznie jest tabelą z danymi) eliminuje potrzebę dołączenia do wyniku zapytania wartości kolumn innych niż ten na których założony jest indeks, w indeksie fizycznie znajduje się wartość wszystkich kolumn.

Zapytanie 2

The screenshot displays the SQL Server Enterprise Manager interface. At the top, the 'Execution plan' tab is selected, showing a 'Clustered Index Seek (Clustered Index)' operation. Below the plan, the 'Results' tab shows the query: `SELECT * FROM [customer] WHERE [storeid]>=01 AND [storeid]<=02`. The query executed successfully, returning 18 rows. The execution statistics at the bottom show: CPU time = 0 ms, elapsed time = 0 ms. The completion time is 2025-05-20T18:53:53.8082691+02:00.

Możemy zaobserwować dokładnie taką samą zmianę planu zapytań co w przypadku wyżej, z tym, że w tej sytuacji zmiana przynosi o wiele bardziej spektakularne efekty, ponieważ w poprzedniej formie zapytania droga operacja **rid lookup**, która teraz została wyeliminowana, została wykonana dla 16-krotnie większej ilości rekordów. Czas pozostał ten sam, koszt zmalał z 0,0507122 => 0,0032996 oraz ilość odczytów logicznych z 18 => 2.

Podejścia w wyszukiwaniu krotek

Zastosowane podejścia wyżej w wyszukiwaniu krotek: indeks klastrowany oraz indeks nieklastrowany, znacząco poprawiają efektywność zapytań w porównaniu do ich braku, przy zastosowaniu warunku z klauzulą **WHERE**. Pozwalają na nieprzeszukiwanie wszystkich rekordów w celu sprawdzenia klauzuli. Jednakże występują znaczące różnice pomiędzy nimi, w indeksie nieklastrowym bezpośredni dostęp jest do kolumn na których założony jest indeks oraz tych uwzględnionych w klauzuli **include** do reszty kolumn wymagana jest operacja **rid lookup**, która jest bardziej kosztowna. Z racji na to, że indeks klastrowany możemy założyć tylko jeden na całą tabelę, trzeba bardzo rozważnie podchodzić do jego tworzenia. W przypadku kiedy nie mamy pewności, że będzie najlepszym rozwiązaniem można zastosować indeks nieklastrowy wraz z klauzulą **include** zawierającą porządkane kolumny - je także powinniśmy dobierać rozważnie, rzadko kiedy potrzebne są wszystkie

Zadanie 4 - dodatkowe kolumny w indeksie

Celem zadania jest porównanie indeksów zawierających dodatkowe kolumny.

Skopiuj tabelę **Address** do swojej bazy danych:

```
select * into address from adventureworks2017.person.address
```

W tej części będziemy analizować następujące zapytanie:

```
select addressline1, addressline2, city, stateprovinceid, postalcode
from address
where postalcode between n'98000' and n'99999'
```

```
create index address_postalcode_1
on address (postalcode)
include (addressline1, addressline2, city, stateprovinceid);
go

create index address_postalcode_2
on address (postalcode, addressline1, addressline2, city, stateprovinceid);
go
```

Czy jest widoczna różnica w planach/kosztach zapytań?

- w sytuacji gdy nie ma indeksów
- przy wykorzystaniu indeksu: - address_postalcode_1 - address_postalcode_2 Jeśli tak to jaka?

Aby wymusić użycie indeksu użyj `WITH(INDEX(Address_PostalCode_1))` po `FROM`

Wyniki:

Bez indeksu

The first screenshot shows the execution plan for the query. It is a 'Table Scan' for the 'address' table. The cost is 100% and the estimated execution time is 0.007s. The query is: `SELECT [addressline1],[addressline2],[city],[stateprovinceid],[postalcode] FROM [address] WHERE [postalcode]>=98000 AND [postalcode]<=99999`. The status bar indicates 'Query executed successfully.' and 'localhost (15.0 RTM) | SA (51) | LAB4 | 00:00:01 | 2 640 rows'.

The second screenshot shows the results of the query. It displays the execution times for the query: 'SQL Server Execution Times: CPU time = 0 ms, elapsed time = 10 ms.' and 'SQL Server parse and compile time: CPU time = 0 ms, elapsed time = 0 ms.' The status bar indicates 'Query executed successfully.' and 'localhost (15.0 RTM) | SA (51) | LAB4 | 00:00:01 | 2 640 rows'.

Komentarz

Z address_postalcode_1

Komentarz

Pokrywający indeks z kolumnami w INCLUDE pozwala na bardzo selektywne Index Seek bez konieczności odczytywania całej tabeli ani dokonywania Key Lookups. W efekcie logical reads spadają z 344 do zaledwie 32 (ponad 10× oszczędności), a zapytanie wykonuje się blisko 3× szybciej (3 ms vs 10 ms w scenariuszu bez indeksu), przy zerowych odczytach fizycznych.

Z address_postalcode_2

Results

Messages

Live Query Statistics

Execution plan

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

select addressline1, addressline2, city, stateprovinceid, postalcode from address WITH(INDEX(Address_PostalCode_2)) where postalcode between '98000'...

SELECT

Cost: 0 %

Index Seek (NonClustered)

[address].[address_postalcode]

Cost: 100 %

0.001s

Query executed successfully.

localhost (15.0 RTM) | SA (51) | LAB4 | 00:00:01 | 2 640 rows

Results

Messages

Live Query Statistics

Execution plan

(2638 rows affected)

Table 'address'. Scan count 1, logical reads 34, 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

(2 rows affected)

(1 row affected)

SQL Server Execution Times:

CPU time = 5 ms, elapsed time = 3 ms.

SQL Server parse and compile time:

CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 0 ms.

100 %

Query executed successfully.

localhost (15.0 RTM) | SA (51) | LAB4 | 00:00:01 | 2 640 rows

Results

Messages

Live Query Statistics

Execution plan

	addressline1	addressline2	city	stateprovinceid	postalcode
1	225 South 314th Street	NULL	Federal Way	79	98003
2	108 Lakeside Court	NULL	Bellevue	79	98004
3	1343 Prospect St	NULL	Bellevue	79	98004
4	1648 Eastgate Lane	NULL	Bellevue	79	98004
5	2284 Azalea Avenue	NULL	Bellevue	79	98004
6	25915 140th Ave Ne	NULL	Bellevue	79	98004
7	2681 Eagle Peak	NULL	Bellevue	79	98004
8	2947 Vine Lane	NULL	Bellevue	79	98004
9	3067 Maya	NULL	Bellevue	79	98004

dValues	EstimateRows	EstimateIO	EstimateCPU	AvgRowSize	TotalSubtreeCost	OutputList	Warnings	Type	Parallel	EstimateExecutions	
1	2693,25	NULL	NULL	0,0284668	NULL	NULL		SELECT	0	NULL	
2	[.dbo].[address].[AddressLine1], [LAB4].[d...	2693,25	0,02534722	0,003119575	180	0,0284668	[LAB4].[dbo].[address].[AddressLine1], [LAB4].[d...	NULL	PLAN_ROW	0	1

Komentarz

Indeks kompozytowy, zawierający wszystkie wymagane kolumny w kluczu, pozwala na bezpośredni Index Seek i sekwencyjny odczyt stron. Logical Reads wzrastają minimalnie do 34 (vs 32 w wersji z INCLUDE), ale Elapsed Time pozostaje na poziomie 3 ms. Wyższy CPU Time (5 ms) to efekt większej liczby porównań w głębszym drzewie indeksu. Ten indeks oferuje najlepsze fizyczne właściwości dostępu (najbardziej zagęszczone drzewo), kosztem nieznacznie wyższego obciążenia CPU.

Wnioski końcowe

Jeśli potrzebujemy najniższych logical reads i najmniejszego CPU, najlepszy będzie indeks z INCLUDE (postalcode_1).

Gdy za to zależy nam na najbardziej zwartym drzewie i sekwencyjnym dostępie (np. przy klastracji lub bardzo dużych zakresach), warto zastosować composite key (postalcode_2), akceptując nieco wyższy CPU.

Praca bez indeksu jest opłacalna jedynie przy bardzo rzadkich ad hoc-ach na małych tabelach - na dużych danych zawsze warto dodać odpowiedni indeks.

Sprawdź rozmiar Indeksów:

```
select i.name as indexname, sum(s.used_page_count) * 8 as indexsizekb
from sys.dm_db_partition_stats as s
inner join sys.indexes as i on s.object_id = i.object_id and s.index_id =
i.index_id
where i.name = 'address_postalcode_1' or i.name = 'address_postalcode_2'
group by i.name
go
```

Który jest większy? Jak można skomentować te dwa podejścia do indeksowania? Które kolumny na to wpływają?

Wyniki:

indexname	indexsizekb
address_postalcode_1	1784
address_postalcode_2	1808

Rows	Executes	StmtText	StmtId	NodeId	Parent	PhysicalOp	LogicalOp	Argument	DefinedValues
2	1	select i.name as indexname, sum(s.used_page_cou...	1	1	0	NULL	NULL	NULL	NULL
0	0	-Compute Scalar(DEFINE:([Expr1066]=[Expr1065]...	1	2	1	Compute Scalar	Compute Scalar	DEFINE:([Expr1066]=[Expr1065]*(8))	[Expr1066]=[Expr1065]*(8)
0	0	-Compute Scalar(DEFINE:([Expr1065]=CASE ...	1	3	2	Compute Scalar	Compute Scalar	DEFINE:([Expr1065]=CASE WHEN [Expr1073]=(0) THEN...	[Expr1065]=CASE WHEN [Expr1073]=(0) THEN...
2	1	-Stream Aggregate(GROUP BY:([i].[name]) ...	1	4	3	Stream Aggregate	Aggregate	GROUP BY:([i].[name])	[Expr1073]=COUNT_BIG(Uni...
2	1	-Nested Loops(Inner Join, OUTER REF...	1	5	4	Nested Loops	Inner Join	OUTER REFERENCES:([i].[id])	NULL
2	1	-Nested Loops(Inner Join, OUTER R...	1	6	5	Nested Loops	Inner Join	OUTER REFERENCES:([i].[id], [i].[indid])	NULL
2	1	-Nested Loops(Inner Join, OUTER...	1	8	6	Nested Loops	Inner Join	OUTER REFERENCES:([i].[id], [i].[indid])	NULL
2	1	-Filter(WHERE: (has_access('CO', [LAB4] [sys] [sysidstats] [id] ...	1	9	8	Filter	Filter	WHERE: (has_access('CO', [LAB4] [sys] [sysidstats] [id] ...	NULL
2	1	-Index Seek(OBJECT: ([LAB4] [sys] [sysidstats] [nc] AS [i]), SEEK: ([i] [...	1	10	9	Index Seek	Index Seek	OBJECT: ([LAB4] [sys] [sysidstats] [nc] AS [i]), SEEK: ([i] [...	[i].[id], [i].[indid], [i].[name]

Wnioski

Indeks address_postalcode_2 (1808 KB) jest nieco większy od address_postalcode_1 (1784 KB), ponieważ w composite key wszystkie kolumny klucza przechowywane są na każdym poziomie drzewa. W pokrywającym indeksie z INCLUDE dodatkowe kolumny znajdują się tylko w liściach, co ogranicza rozmiar struktury nawigacyjnej. Dzięki temu address_postalcode_1 oferuje pełne pokrycie zapytania bez kosztownych Key Lookupów przy minimalnym wzroście objętości danych. Composite key (address_postalcode_2) z kolei zapewnia pełne kluczowe ujęcie kolumn, co bywa przydatne przy wymuszaniu unikalności lub klastrowaniu, ale kosztem większego drzewa indeksu. W praktyce warto najczęściej sięgać po indeks z INCLUDE, a composite key stosować tam, gdzie struktura klucza wymaga fizycznego porządkowania wszystkich kolumn.

Zadanie 5 – Indeksy z filtrami

Celem zadania jest poznanie indeksów z filtrami.

Skopiuj tabelę `BillofMaterials` do swojej bazy danych:

```
select * into billofmaterials
from adventureworks2017.production.billofmaterials
```


W tej części analizujemy zapytanie:

```
select productassemblyid, componentid, startdate
from billofmaterials
where enddate is not null
    and componentid = 327
    and startdate >= '2010-08-05'
```

Zastosuj indeks:

```
create nonclustered index billofmaterials_cond_idx
on billofmaterials (componentid, startdate)
where enddate is not null
```

Sprawdź czy działa.

Przeanalizuj plan dla poniższego zapytania:

Czy indeks został użyty? Dlaczego?

Wyniki:

bez wymuszenia

ResultsMessagesLive Query StatisticsExecution plan

Estimated query progress:100%Query 1: Query cost (relative to the batch): 100%
SELECT [productassemblyid],[componentid],[startdate] FROM [billofmaterials] WHERE [enddate] IS NOT NULL AND [componentid]=@1 AND [start

Table Scan

[billofmaterials]

14 of 7 (200%)

Query executed successfully.localhost (15.0 RTM)SA (51)LAB400:00:0116 rows

ResultsMessagesLive Query StatisticsExecution plan

	productassemblyid	componentid	startdate
1	718	327	2010-08-05 00:00:00.000
2	725	327	2010-08-05 00:00:00.000
3	729	327	2010-08-05 00:00:00.000
4	731	327	2010-08-05 00:00:00.000
5	735	327	2010-08-05 00:00:00.000
6	738	327	2010-08-05 00:00:00.000
7	742	327	2010-08-05 00:00:00.000
8	745	327	2010-08-05 00:00:00.000
9	832	327	2010-08-05 00:00:00.000

	definedValues	EstimateRows	EstimateIO	EstimateCPU	AvgRowSize	TotalSubtreeCost	OutputList	Warnings	Type	Parallel	EstimateExecutions
1	JLL	6,643897	NULL	NULL	NULL	0,02030297	NULL	NULL	SELECT	0	NULL
2	AB4].[dbo].[billofmaterials].[ProductAssemblyl...	6,643897	0,01719907	0,0031039	31	0,02030297	[LAB4].[dbo].[billofmaterials].[ProductAssemblyl...	NULL	PLAN_ROW	0	1

100 %

ResultsMessagesLive Query StatisticsExecution plan

Table 'billofmaterials'. Scan count 1, logical reads 20, 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,

(2 rows affected)

(1 row affected)

SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2025-05-20T19:32:41.4533194+02:00

100 %

Komentarz

Pomimo istnienia filtrowanego indeksu billofmaterials_cond_idx, optymalizator wybrał pełny skan tabeli. Przy tak małej liczbie stron (20) i niewielkim wyniku (2 wiersze) koszt nawigacji po B-drzewie indeksu okazał się wyższy niż prosty TABLE SCAN, dlatego indeks nie został użyty.

Spróbuj wymusić indeks. Co się stało, dlaczego takie zachowanie?

Wyniki:

z wymuszeniem

ResultsMessagesLive Query StatisticsExecution plan

Estimated query
progress:100%
Query 1: Query cost (relative to the batch): 100%
select productassemblyid, componentid, startdate from billofmaterials WITH(INDEX(billofmaterials_cond_idx)) where enddate is not null

SELECT

Nested Loops
(Inner Join)
14 of
7 (200%)

Index Seek (NonClustered)
[billofmaterials].[billofmat...
14 of
59 (23%)

RID Lookup (Heap)
[billofmaterials]
14 of
59 (23%)

Query executed successfully. localhost (15.0 RTM) SA (51) LAB4 00:00:01 18 rows

ResultsMessagesLive Query StatisticsExecution plan

CPU time = 0 ms, elapsed time = 2 ms.

(14 rows affected)
Table 'billofmaterials'. Scan count 1, logical reads 16, 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,

(4 rows affected)

(1 row affected)

SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 0 ms.

Query executed successfully. localhost (15.0 RTM) SA (51) LAB4 00:00:01 18 rows

ResultsMessagesLive Query StatisticsExecution plan

	productassemblyid	componentid	startdate
1	832	327	2010-08-05 00:00:00.000
2	840	327	2010-08-05 00:00:00.000
3	890	327	2010-08-05 00:00:00.000
4	898	327	2010-08-05 00:00:00.000
5	902	327	2010-08-05 00:00:00.000
6	920	327	2010-08-05 00:00:00.000
7	718	327	2010-08-05 00:00:00.000
8	725	327	2010-08-05 00:00:00.000

	DefinedValues	EstimateRows	EstimateIO	EstimateCPU	AvgRowSize	TotalSubtreeCost	OutputList	Warnings	Type	Parallel	EstimateExecuti
1	NULL	6,643897	NULL	NULL	NULL	0,07235777	NULL	NULL	SELECT	0	NULL
2	NULL	6,643897	0	0,0002482486	23	0,07235777	[LAB4] [dbo] [billofmaterials] [ProductAssemblyID...	NULL	PLAN_ROW	0	1
3	[Bmk1000], [LAB4] [dbo] [billofmaterials] [Compo...	59,38963	0,003125	0,000223286	27	0,003347329	[Bmk1000], [LAB4] [dbo] [billofmaterials] [Compo...	NULL	PLAN_ROW	0	1
4	[LAB4] [dbo] [billofmaterials] [ProductAssemblyID]	1	0,003125	0,0001581	11	0,0687622	[LAB4] [dbo] [billofmaterials] [ProductAssemblyID]	NULL	PLAN_ROW	0	59,38963

26 / 27

Komentarz

Wymuszając hint, optymalizator rzeczywiście korzysta z filtrowanego indeksu (Index Seek), ale ponieważ potrzebna kolumna productassemblyid nie jest w nim pokryta, każda z 14 dopasowanych pozycji wymaga osobnego RID Lookup na oryginalnym heapie. To obniża logical reads z 20 do 16, ale kosztem wielu losowych odczytów, co przekłada się na wyższy Elapsed Time (2 ms vs 0 ms przy sekwencyjnym Table Scan). Początkowo algorytm pominął indeks, bo uznał, że taniej będzie przeskanować 20 stron naraz niż robić 14 pojedynczych look-upów. Wymuszenie indeksu obniża I/O, ale z uwagi na narzut losowych odczytów nie zawsze przekłada się na lepszą rzeczywistą wydajność.

Wnioski końcowe

- Optymalizator pominął filtrowany indeks i użył Table Scan (20 logical reads, 0 ms), bo przy tak małej liczbie stron sekwencyjny odczyt był tańszy niż nawigacja po drzewie i look-upy.
- Wymuszony Index Seek na billofmaterials_cond_idx zmniejszył logical reads do 16, ale generował 14 RID Lookup, co podniosło realny czas do 2 ms.
- Filtrowane indeksy najlepiej sprawdzają się, gdy zapytanie dokładnie pokrywa warunek filtra i indeks zawiera wszystkie potrzebne kolumny.
- Zwykle warto polegać na decyzji optymalizatora i unikać hintów, bo niesie to ryzyko dodatkowych kosztów losowego I/O.

Punktacja:

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