2022-2023

OPTIMALISATIE DOSSIER

DATA PERSISTENCY

DERBOVEN MAXIM & JONAS LEIJZEN
IAO3

Inhoud

2 Logische Indexed View

2.1 Inleiding

Een logische indexed view is een virtuele tabel die fysiek (logisch) wordt opgeslagen is op de harde schijf, dus niet virtueel. Hierdoor moeten we bij het ophalen van de view niet telkens weer alle data uit de basis tabel ophalen. Dit wil ook zeggen dat het aanpassen van de data in de indexed view niet performant is. Pas deze optimalisatie dus alleen toe in systemen die infrequent of zelden geupdate worden met nieuwe rijen, zoals een datawarehouse!

Wanneer uw basis tabel veel veranderd is het niet aangeraden een indexed view te gebruiken.

2.2 Query keuze

Queries met veel joins en veel rijen die niet vaak geupdate moeten worden / up to date data moeten hebben een grotere kans op optimalisatie voordelen.

Voor dit voorbeeld hebben we gebruik gemaakt van een query die joins deed op 3 verschillende tabellen.

We gaan volgende query proberen te optimaliseren.

```
SELECT c.ZipCode, c.City, c.CountryCode, COUNT_BIG(*) AS 'Aantal Ritten'
FROM dbo.factRide f
JOIN dbo.dimCustomers c on f.DIM_USER_SK = c.userRepSK
GROUP BY c.ZipCode, c.City, c.CountryCode
ORDER BY 4 DESC
```

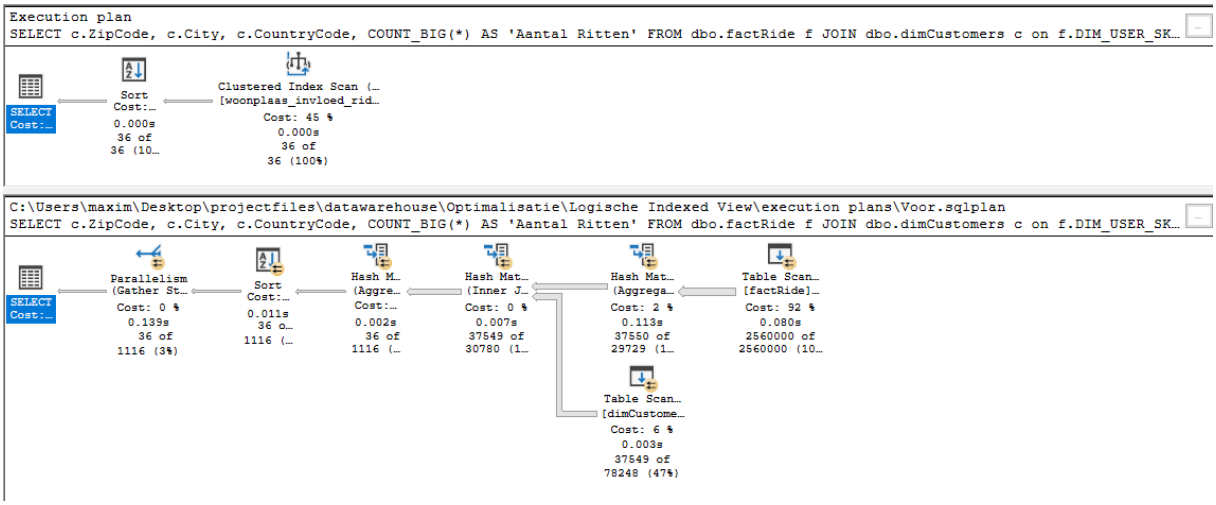
2.3 Werkwijze

Met volgende statements maken we een logische view aan met een cluster index.

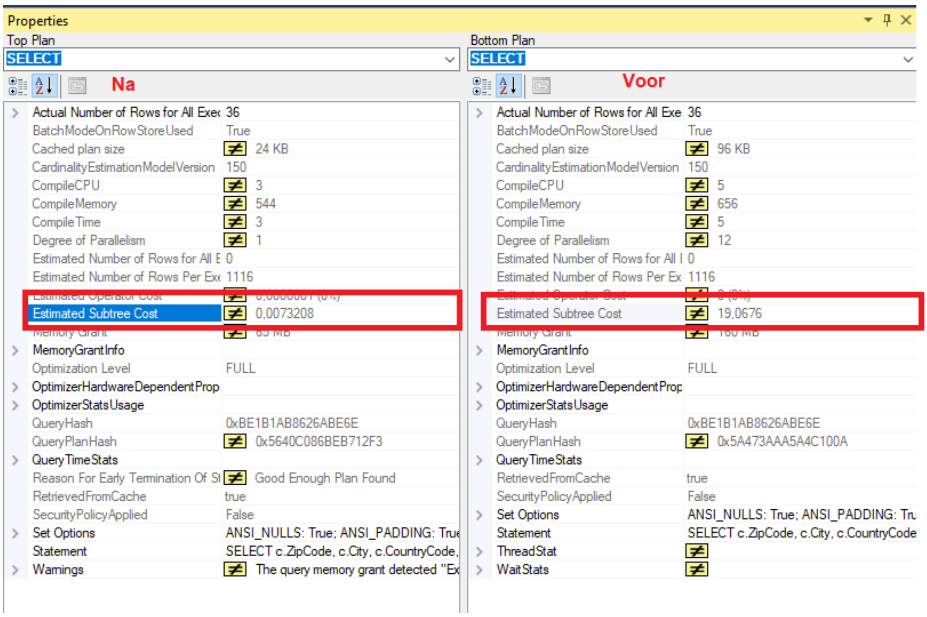
```
SET NUMERIC_ROUNDABORT OFF;
SET ANSI_PADDING, ANSI_WARNINGS, CONCAT_NULL_YIELDS_NULL, ARITHABORT,
   QUOTED_IDENTIFIER, ANSI_NULLS ON;

DROP VIEW woonplaas_invloed_rides;

--Create view with SCHEMABINDING.

GO
CREATE VIEW woonplaas_invloed_rides
   WITH SCHEMABINDING
   AS
   SELECT c.ZipCode, c.City, c.CountryCode, COUNT_BIG(*) AS 'Aantal Ritten'
FROM dbo.factRide f
JOIN dbo.dimCustomers c on f.DIM_USER_SK = c.userRepSK
GROUP BY c.ZipCode, c.City, c.CountryCode
GO

CREATE UNIQUE CLUSTERED INDEX INDEX_1
   ON  woonplaas_invloed_rides (ZipCode, City, CountryCode);
GO
```

2.3.1 Verschil voor en na

Explain plan



Subtree Cost



2.4 Conclusie

Door het aanmaken van een logische view was het niet meer nodig om de join uit te voeren tijdens de select statement. Het verschil was vooral te zien in de estimated subtree cost. Voor de optimalisatie was deze 19,076 en nadien 0,07.

We zien in dit voorbeeld dat een query die wordt uitgevoerd op een tabel die gepartitioneerd is, **minder** stappen nodig heeft.

3 Partitionering

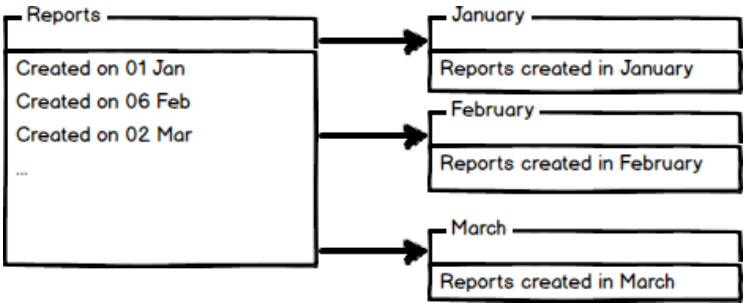
3.1 Inleiding

Partitioneren is het splitsten van tabellen in subtabellen op basis van een bepaalde voorwaarde.

Als er in de where clausule de voorwaarde verwerkt zit dan wordt enkel de relevante subtabel aangesproken. Het doel van partitioneren is het verbeteren van de efficiëntie en prestaties van het laden van gegevens.

3.2 Horizontal Partitioning

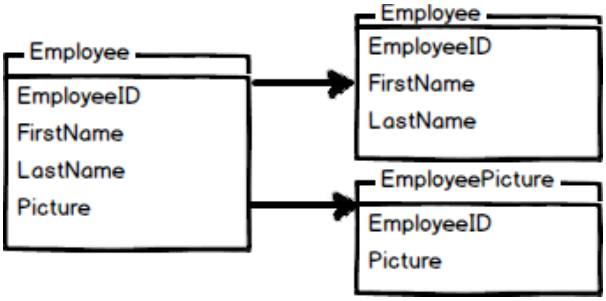
Horizontale partitionering verdeelt een tabel in meerdere tabellen die hetzelfde aantal kolommen bevatten, maar minder rijen. Op deze manier verwijzen zoekopdrachten die gegevens voor een specifiek jaar vereisen, alleen naar de juiste tabel. Tabellen moeten zo worden gepartitioneerd dat query's naar zo weinig mogelijk tabellen verwijzen.



3.3 Vertical Partitioning

Verticale tabelpartitionering wordt meestal gebruikt om de prestaties van SQL Server te verbeteren.

- kolommen ophalen die zeer brede tekst- of BLOB-kolommen bevat
- beperken van de toegang tot gevoelige gegevens



3.4 Query keuze

Bij partitionering haal je er het meeste voordeel uit als je een query neemt die zich beperkt tot een bepaald deel van de data.

We gaan volgende Query optimaliseren:

```
SELECT TOP 1 c.Name AS 'Name',
SUM(f.DURATION_MV / 60000) as 'totaal tijd (min).',
count(*) as 'Totaal ritten',
SUM(f.DURATION_MV / 60000) /  count(*) as 'Gemiddelde tijd per rit (min)'
from factRide f
join dimCustomers c on f.DIM_USER_SK = c.userRepSK
WHERE f.STARTTIME_MV BETWEEN '2015-09-01' AND '2015-09-30'
GROUP BY f.DIM_USER_SK, c.Name
ORDER BY COUNT(f.DURATION_MV) DESC
```

3.5 Werkwijze

Om partities aan te maken gebruiken we in dit hoofdstuk SQL Server Management Studio.

We zullen opteren voor een SQL-Syntax optie i.p.v. een grafische omgeving. Zo kunnen we meer in detail gaan op de werking van partities zoals vorig jaar gezien bij Oracle.

Filegroepen aanmaken voor het opslaan van de partitie op de harde schijf.

```
-- een filegroep aanmaken
ALTER DATABASE velo_dwh
ADD FILEGROUP fg1;

ALTER DATABASE velo_dwh
ADD FILEGROUP fg2;

ALTER DATABASE velo_dwh
ADD FILEGROUP fg3;

ALTER DATABASE velo_dwh
ADD FILEGROUP fg4;
```

Voor elke filegroep een bestand

```
-- per filegroep een bestand
ALTER DATABASE velo_dwh
ADD FILE (
    NAME = dat1,
    FILENAME = '/var/opt/mssql/data/dat1.ndf',
        SIZE = 5 MB,
        MAXSIZE = UNLIMITED,
        FILEGROWTH = 1024KB
        )
TO FILEGROUP fg1;

ALTER DATABASE velo_dwh
ADD FILE (
    NAME = dat3,
    FILENAME = '/var/opt/mssql/data/dat3.ndf',
        SIZE = 5 MB,
        MAXSIZE = UNLIMITED,
        FILEGROWTH = 1024KB
        )
TO FILEGROUP fg3;
```

```
ALTER DATABASE velo_dwh
ADD FILE (
    NAME = dat2,
    FILENAME = '/var/opt/mssql/data/dat2.ndf',
        SIZE = 5 MB,
        MAXSIZE = UNLIMITED,
        FILEGROWTH = 1024KB
        )
TO FILEGROUP fg2;

ALTER DATABASE velo_dwh
ADD FILE (
    NAME = dat4,
    FILENAME = '/var/opt/mssql/data/dat4.ndf',
        SIZE = 5 MB,
        MAXSIZE = UNLIMITED,
        FILEGROWTH = 1024KB
        )
TO FILEGROUP fg4;
```

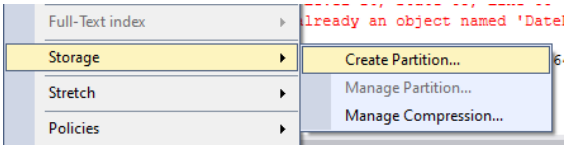
Ranges opstellen per maand / jaar (in dit geval simpel gehouden omdat we even puur een test willen doen voor die oktober)

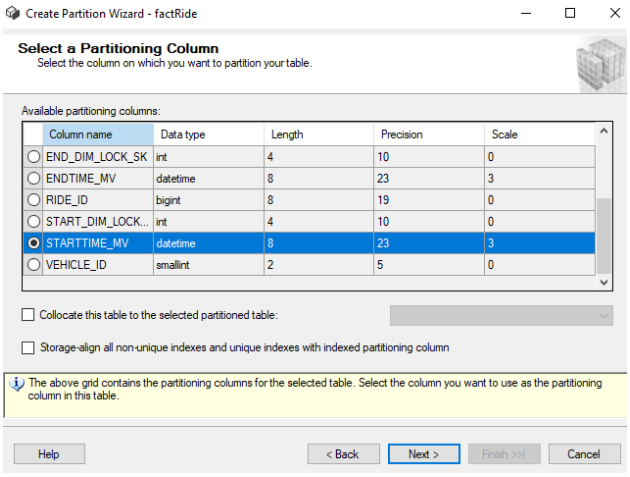
```
CREATE PARTITION FUNCTION DateRangesMonths (datetime) AS RANGE RIGHT
FOR VALUES (N'2015-08-01T00:00:00.000',N'2015-09-01T00:00:00.000', N'2015-10-01T00:00:00.000');
```

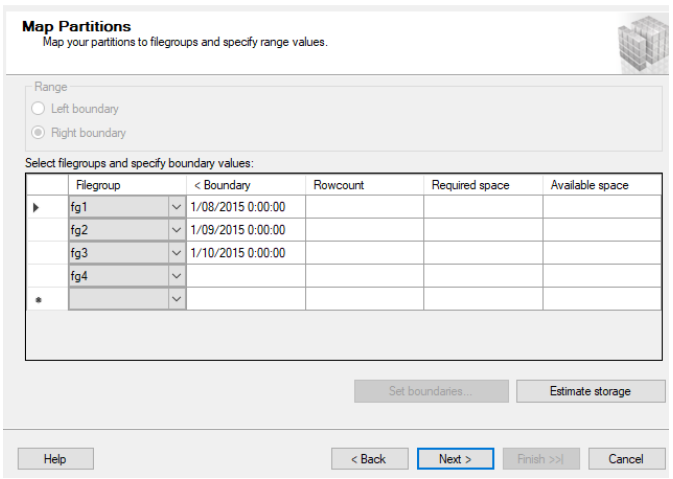
Partities toewijzen aan de filegroepen

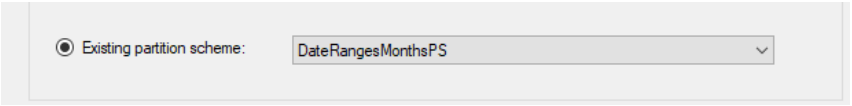
```
CREATE PARTITION SCHEME DateRangesMonthsPS
AS PARTITION DateRangesMonths
TO (fg1,fg2,fg3,fg4);
```

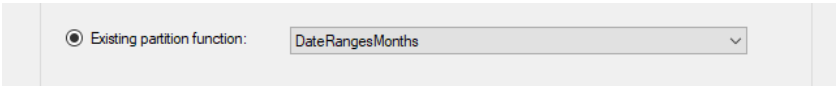
De tabel indelen in deze partities kunnen we doen met de MSMS wizard:





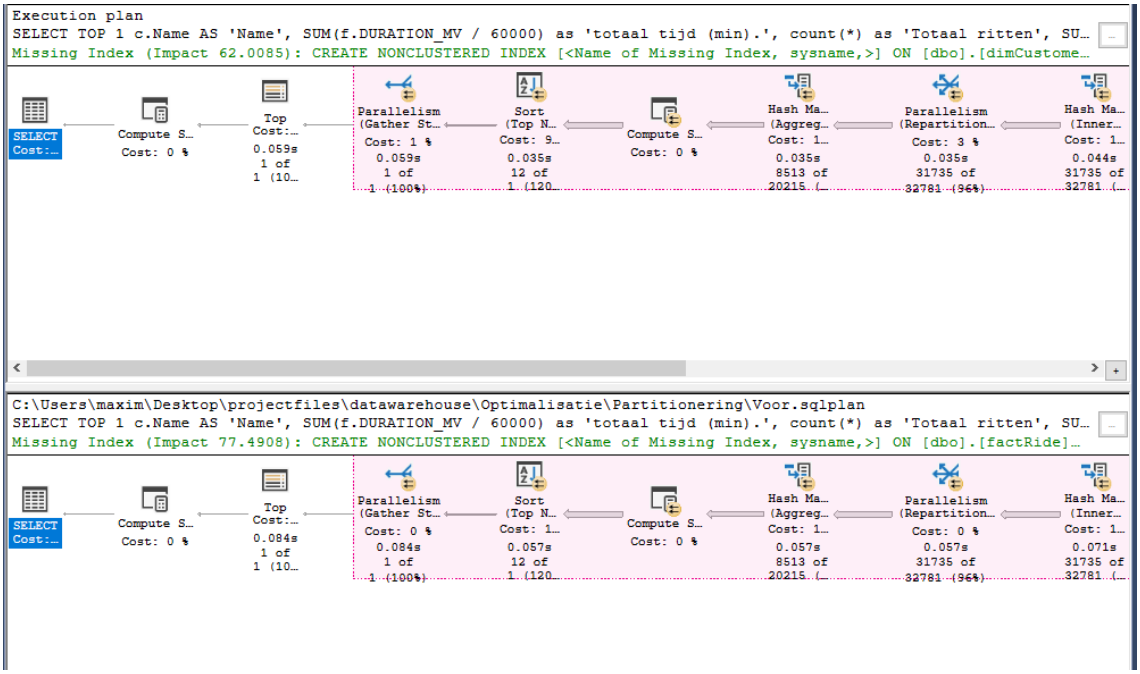




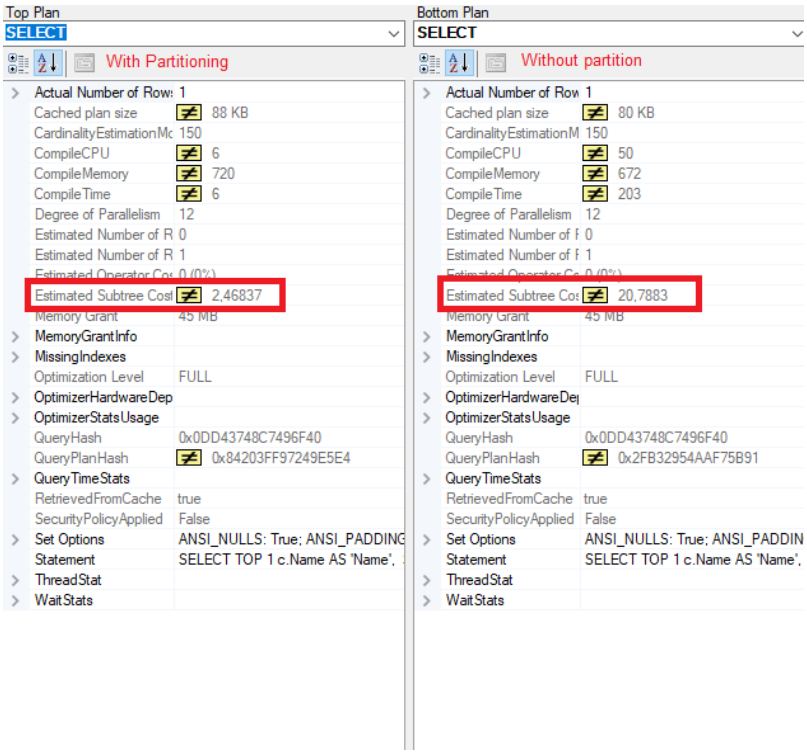


3.5.1 Verschil voor en na

Explain plan



Cost



3.6 Conclusie

We zien een duidelijke verbetering met een cost van 18, als we dit op grotere schaal gaan uitvoeren halen we hier veel voordelen uit.

4 Column storage

4.1 Inleiding

Het idee is simpel: Repitieve data in een kolom zoeken en unieke informatie als rij of page of column store index opslagen.

Columnstore indexen zijn standaard voor het opslaan en bevragen van grote gegevenstabellen voor datawarehousing. Deze index maakt gebruik van kolomgebaseerde gegevensopslag en queryverwerking om in de datawarehouse tot 10 keer betere queryprestaties te bereiken dan met traditionele rijgeoriënteerde opslag.

4.2 Query keuze

We gaan proberen deze query proberen te optimaliseren

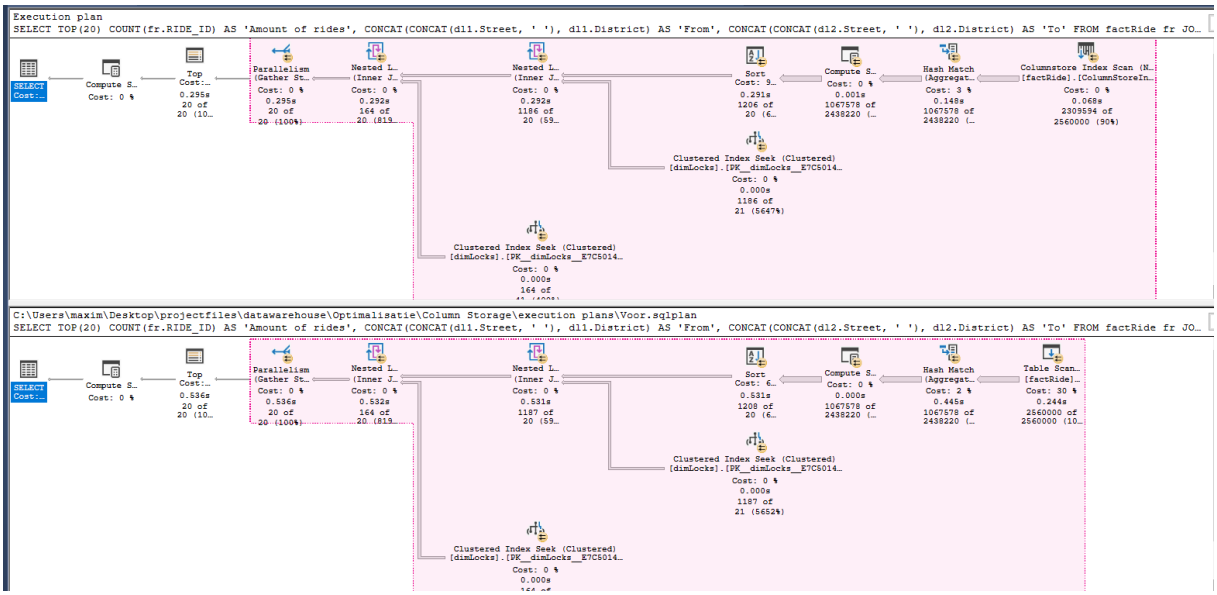
```
  SELECT TOP(20) COUNT(fr.RIDE_ID) AS 'Amount of rides',
  CONCAT(CONCAT(dl1.Street, ' '), dl1.District) AS 'From',
  CONCAT(CONCAT(dl2.Street, ' '), dl2.District) AS 'To'
  FROM factRide fr
  JOIN dimLocks dl1 ON fr.START_DIM_LOCK_SK = dl1.LockSK
  JOIN dimLocks dl2 ON fr.END_DIM_LOCK_SK = dl2.LockSK
  WHERE dl1.StationId IS NOT NULL
  AND dl1.Street != 'ONTBREKEND'
  AND dl2.Street != 'ONTBREKEND'
  AND dl1.StationId != dl2.StationId
  GROUP BY fr.START_DIM_LOCK_SK, fr.END_DIM_LOCK_SK, dl1.Street, dl1.District,
dl2.Street, dl2.District
  ORDER BY COUNT(RIDE_ID) DESC;
```

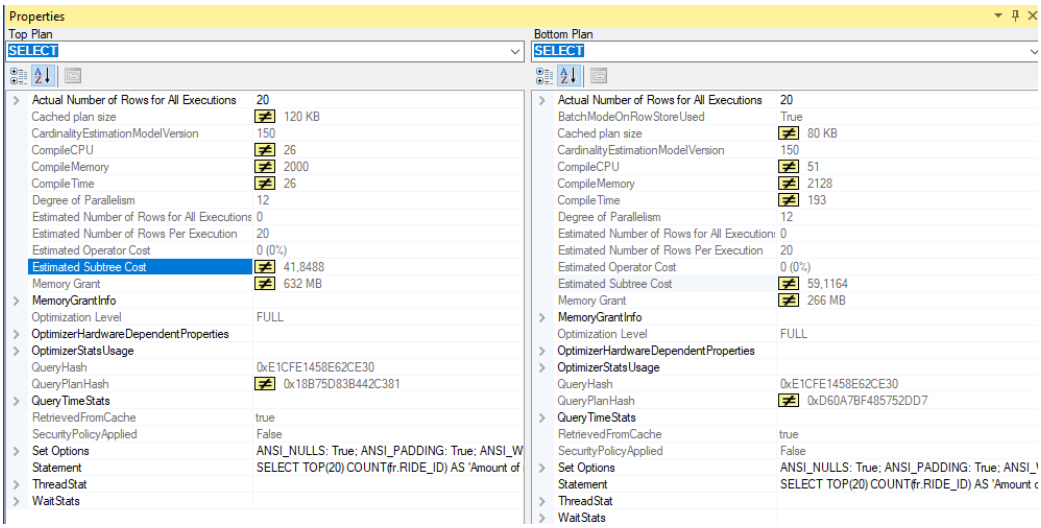
4.3 Werkwijze

Met volgende statement maken we een columnstore index genaamd ColumnStoreIndexOnLocks, die de lock SKs gaat indexen.

```
CREATE COLUMNSTORE INDEX ColumnStoreIndexOnlocks
ON dbo.factRide (START_DIM_LOCK_SK, END_DIM_LOCK_SK);

of

CREATE COLUMNSTORE INDEX ColumnStoreIndexOnlockSKs
ON dbo.dimLocks (LockSK);
```

4.3.1 Verschil voor en na





4.4 Conclusie

Als we kijken naar het Explain Plan dan zien we dat deze nu gebruik maakt van een columnstore index scan i.p.v. een FULL Table Scan. Dit is een simpel voorbeeld. In de realiteit ga je meestal kolommen nemen die vaak samen voorkomen en daarop een index toepassen.

5 Table Compression

5.1 Inleiding

Moderne compressie-algoritmen kunnen de voetafdruk op de schijf met 40-60% of zelfs meer verminderen, afhankelijk van het type gegevens.

Het comprimeren van uw SQL Server-gegevens is een eenvoudige manier om meer in uw beperkte schijfruimte te proppen

5.2 Query keuze

Bij het kiezen van een tabel om compressie op toe te passen is het handig te kijken naar het aantal rijen in de tabel. Grote tabellen met veel rijen hebben het meeste voordeel bij compressie. Compressie heeft natuurlijk wel een relatief grote impact op CPU time, aangezien de data ingepakt/uitgepakt moet worden bij vele database operaties.

We proberen volgend statement te optimaliseren.

```
SELECT c.ZipCode AS 'Zipcode Costumer', COUNT(*) AS 'Ritten', COUNT(*) /
(SELECT COUNT(*) FROM dimCustomers c1
       WHERE c1.Zipcode = c.Zipcode
) AS 'Gemiddelde ritten per persoon'
FROM factRide f
JOIN dimCustomers c on f.DIM_USER_SK = c.userRepSK
GROUP BY c.ZipCode
ORDER BY 3 DESC;
```

5.3 Werkwijze

Met volgende statement pas je een data compressie toe

```
ALTER TABLE dbo.dimCustomers REBUILD WITH (DATA_COMPRESSION=ROW)
```

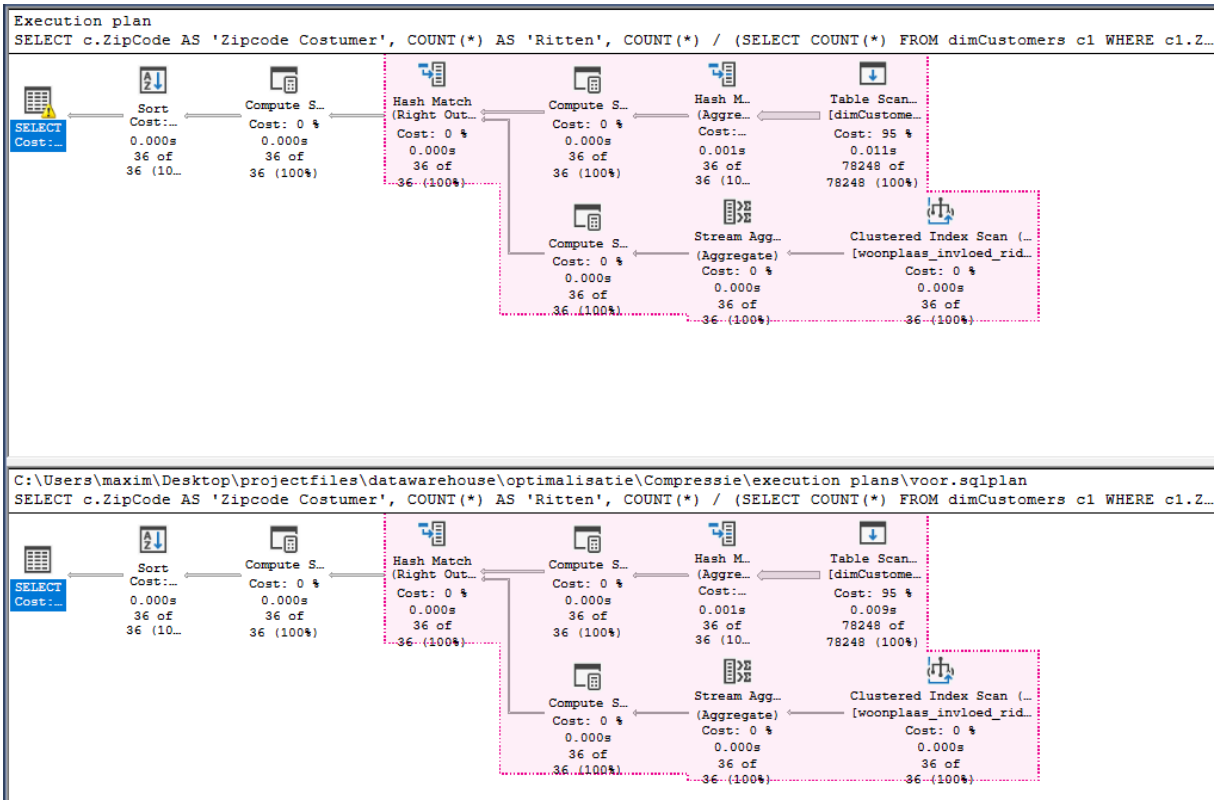
Of

	Partition no.	Compression type	Boundary	Row count	Current space	Requested compressed space
▶	1	Page		78248	10,898 MB	6,891 MB

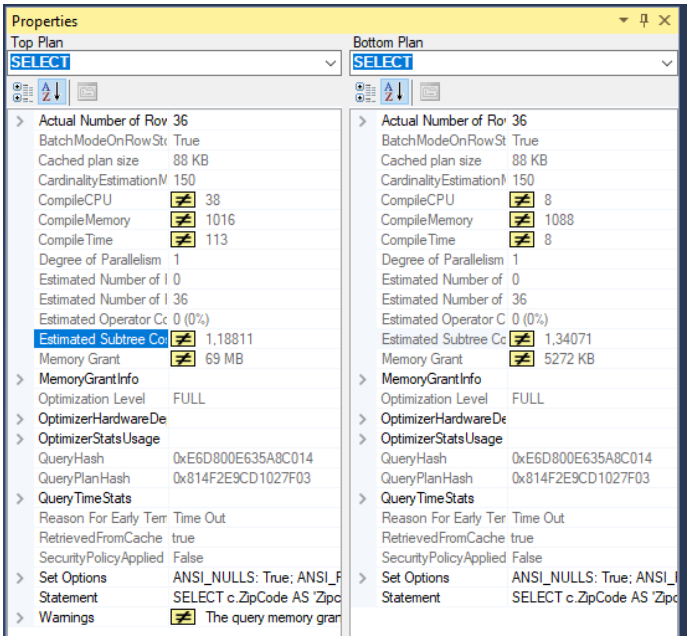
Hiermee kunnen we de opslag van 10MB naar 6MB brengen.

5.3.1 Verschil voor en na

Explain plan



Cost



5.4 Conclusie

Het verschil in subtree cost is minimaal. Maar de grootte van de tabel is serieus verminderd. Op lange termijn en met grote hoeveelheden data kan dit zeker een positief effect hebben.