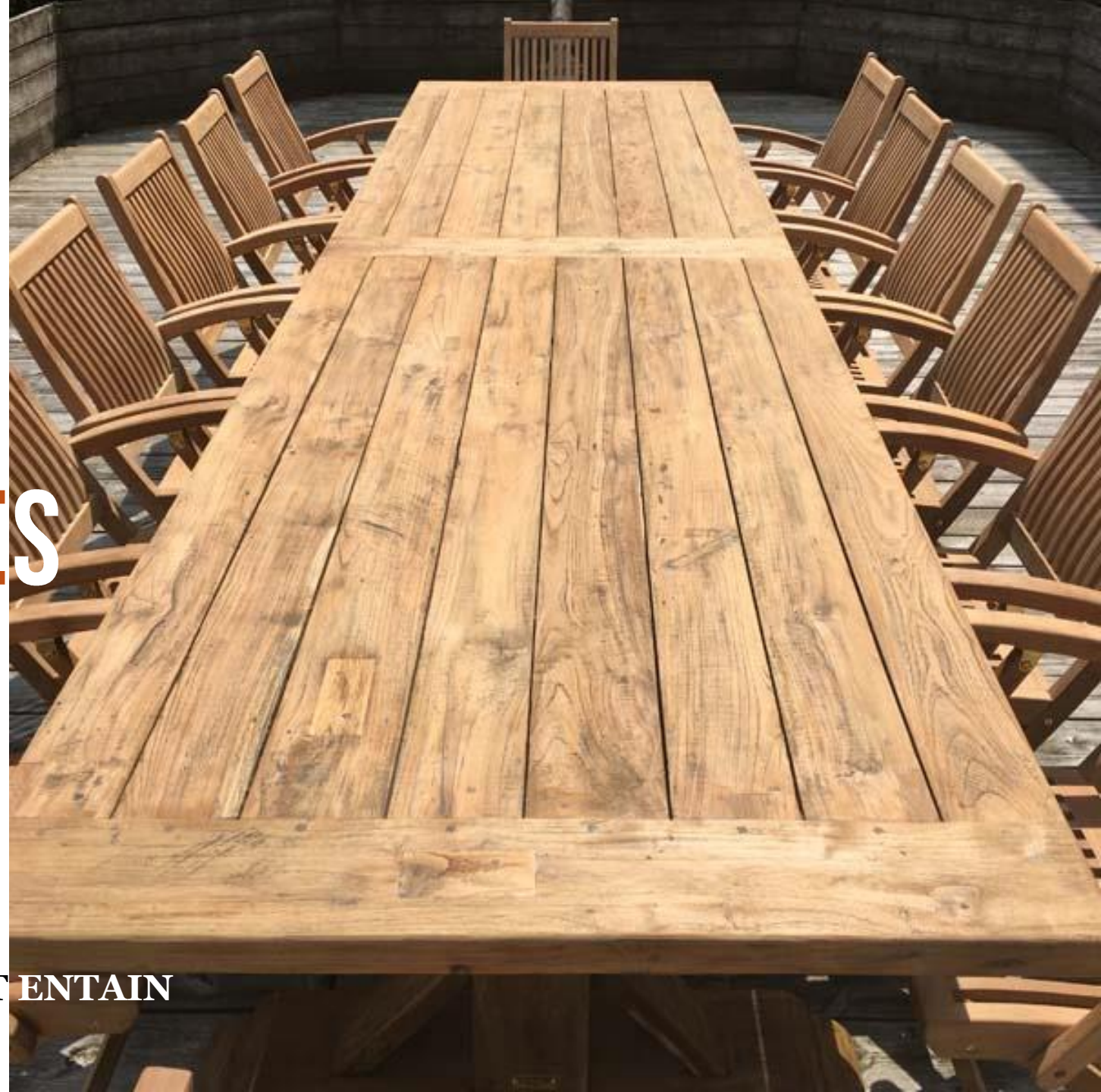
sqlbits
2024

DEALING WITH VERY LARGE TABLES IN SQL SERVER



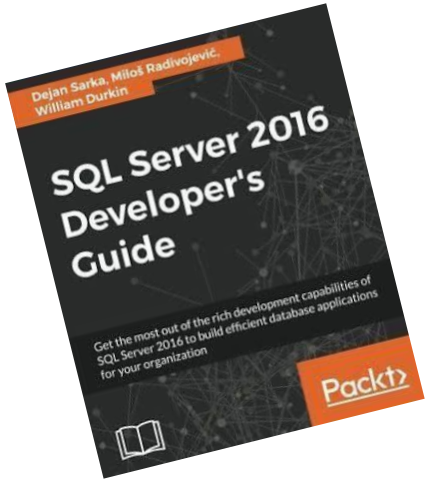
MILOŠ RADIVOJEVIĆ

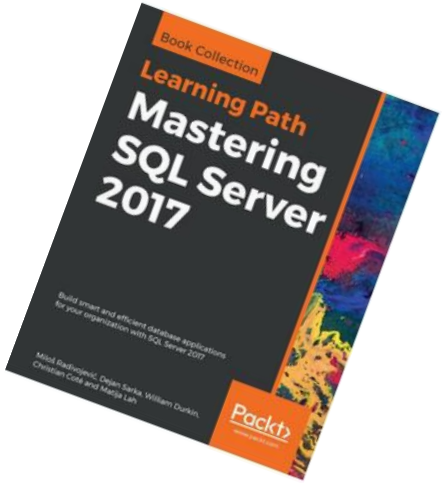
HEAD OF MS SQL DATABASE ENGINEERING AT **ENTAIN**

VIENNA, AUSTRIA

MILOŠ RADIVOJEVIĆ

- Head of Database Engineering at Entain
- Vienna, Austria
- Conference speaker, book author
- Email: **milos.radivojevic@chello.at**
- LinkedIn: **milossql**





SESSION FEEDBACK

DEALING WITH VERY LARGE TABLES

- What is a large or very large table?
  - A large table is a table where you see behavior and issues **that you do not see with other tables**
  - Large tables require **special treatment**
  - Rule of thumb **100M+ rows**

- Challenges with very large tables
  - Maintenance
  - Archiving
  - **Performance**

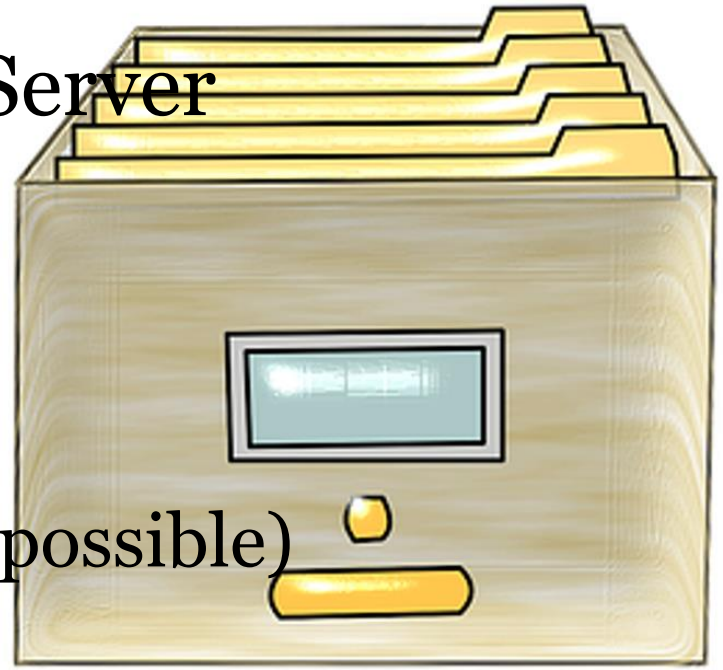
PARTITIONING

- This is not a talk about partitioning

- For partitioning, I would highly recommend sessions from **Uwe Ricken**, **Magi Naumova** and other authors at this and previous SQLBits and other conferences

ARCHIVING

- The concept **"data remains in the main table until the end of the world"**
- All data remain in the main database, **no hot/cold data splitting**, you are operating on **less than 1%** of all data
- No out of-box solution for archiving in SQL Server
- If there is no splitting or archiving
  - It is harder for data management
  - Longer backup and restore
  - Problems with linear scaling queries
  - New indexes are more expensive (sometimes impossible)



ALL INC. PROBLEM

- OLTP queries, transactions and reports from the same database

- **Report** like queries from the main database
- Aggregations, table scans, high CPU utilization
- **Unnecessary "flexible"** reports and searches
- **From** and **To** dates w/o restriction
  and optional parameters

- Hard (or impossible) to **scale**!

PERFORMANCE PROBLEMS WITH VLT

- Missing or inappropriate **indexes**

- **Poorly written** Transact-SQL queries
  - Non SARG-able predicates
  - Local variables
  - Scalar and MSTV functions
  - Implicit data type conversions

- **Cardinality estimation** issues due to inaccurate **statistics**

MISSING INDEX

```
select * from t[illegible] where Inserted = '2024-01-23 08:22:22.5501681'
```

33 %

Results | Messages | Execution plan

1	469908380	136284123	1538	18578805	1	0	2024-01-23 08:22:22.5501681	CT2C	0.681241278486	1171255356	0	806aee0d-b9c8-11ee-a58e-9
2	469908381	136284123	1539	18924157	1	0	2024-01-23 08:22:22.5501681	CT2C	0.216543813766	1171255356	1	806aee0d-b9c8-11ee-a58e-9
3	469908382	136284123	1540	13521410	1	0	2024-01-23 08:22:22.5501681	CT2C	0.101986610754	1171255356	2	806aee0d-b9c8-11ee-a58e-9

A query returns three rows, but approp. index does not exist

Table size: **20B+** rows **6,4 TB**

CPU cores: **72**

Memory: **300 GB**

Query Duration: **?**

MISSING INDEX – QUERY DURATION

1. **9** minutes
2. **28** minutes
3. **44** minutes
4. **122** minutes

```
select * from t[illegible] where Inserted = '2024-01-23 08:22:22.5501681'
```

133 %

Results | Messages | Execution plan

1	469908380	136284123	1538	18578805	1	0	2024-01-23 08:22:22.5501681	CT2C	0.681241278486	1171255356	0	806aee0d-b9c8-11ee-a58e-92c7e
2	469908381	136284123	1539	18924157	1	0	2024-01-23 08:22:22.5501681	CT2C	0.216543813766	1171255356	1	806aee0d-b9c8-11ee-a58e-92c7e
3	469908382	136284123	1540	13521410	1	0	2024-01-23 08:22:22.5501681	CT2C	0.101986610754	1171255356	2	806aee0d-b9c8-11ee-a58e-92c7e

[illegible] | [illegible]\mradivojevic (431) | [illegible] | 00:28:02 | 3 rows

INS

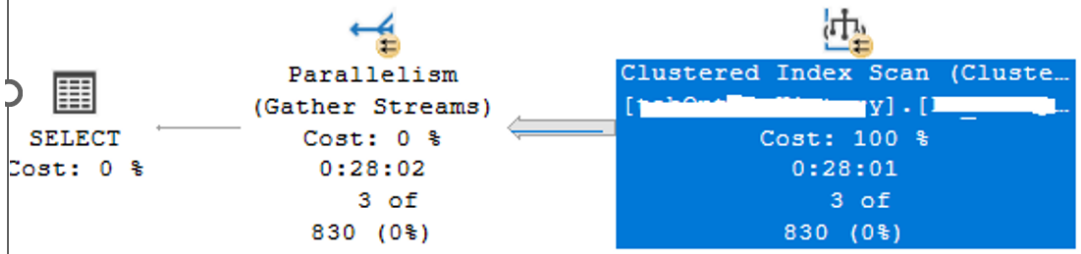
```
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 3 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

(3 rows affected)
Table 'tabOptionHistory'. Scan count 65, logical reads 620491208, physical reads 524, page server reads 0, read-ahead reads 720813199, pa

(1 row affected)

 SQL Server Execution Times:
   CPU time = 8422957 ms,  elapsed time = 1682276 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
```

```
-- table_name    row_count    used_space_GB    percent_of_total_size    cum
--                            20837987847 6355.194     76.10    76.10

select * from                                  where Inserted = '2024-01-23 08
```

33 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [t                    y] WHERE [
Missing Index (Impact 100): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,
```



Properties

Clustered Index Scan (Clustered)

Thread 4	326715268
Thread 40	329499606
Thread 41	320835487
Thread 42	323300728
Thread 43	322364087
Thread 44	321738129
Thread 45	325512394
Thread 46	319026987
Thread 47	319512986
Thread 48	296922320
Thread 49	319944305
Thread 5	314469209
Thread 50	319060042
Thread 51	324924456
Thread 52	330412323
Thread 53	320401696
Thread 54	319516693
Thread 55	313187702
Thread 56	319525784
Thread 57	322858126
Thread 58	321229771
Thread 59	312357566
Thread 6	327256234
Thread 60	321567799
Thread 61	317195124
Thread 62	319158584
Thread 63	314207138
Thread 64	309639654
Thread 7	323074800

INDEXING LARGE TABLES

- Better soon than later
- If you are 24/7 and you have Enterprise Edition you can
  - rebuild indexes online with **ONLINE = ON**
  - use **RESUMABLE** to pause and continue later
- You can use **WAIT_AT_LOW_PRIORITY**
- Use **filtered indexes** for most recent data
- Do not rebuild just because an index is fragmented, **rebuild only if performance suffers**

POORLY WRITTEN QUERIES

FUNCTIONS IN WHERE CLAUSE

```
USE AdventureWorks2019;
GO
SELECT * FROM Sales.SalesOrderHeader WHERE ABS(CustomerID) = 11020
GO
SELECT * FROM Sales.SalesOrderHeader WHERE CustomerID IN (-11020, 11020)
GO
```

133 %

Results | Messages

```
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 parse and compile time:
   CPU time = 0 ms, elapsed time = 16 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'SalesOrderHeader'. Scan count 1, logical reads 689, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logic

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 6 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 2 ms.
Table 'SalesOrderHeader'. Scan count 2, logical reads 7, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical

 SQL Server Execution Times:
```

FUNCTIONS IN WHERE CLAUSE

```
USE xSQLPASS2022;
GO
SELECT * FROM dbo.Orders WHERE ABS(CustomerID) = 11020
GO
SELECT * FROM dbo.Orders WHERE CustomerID IN (-11020, 11020)
GO
```

```
 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 parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'Orders'. Scan count 9, logical reads 1481635, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, l

 SQL Server Execution Times:
   CPU time = 7375 ms,  elapsed time = 943 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'Orders'. Scan count 2, logical reads 12, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob ph

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 14 ms.
SQL Server parse and compile time:
```

```sql
USE AdventureWorks2019;
GO
SELECT * FROM Sales.SalesOrderHeader WHERE CustomerID + 1 = 11021
GO
SELECT * FROM Sales.SalesOrderHeader WHERE CustomerID = 11021 - 1
GO
```

ARITHMETIC OPERATIONS

33 %

Results | Messages

```
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 parse and compile time:
   CPU time = 0 ms, elapsed time = 5 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'SalesOrderHeader'. Scan count 1, logical reads 60, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 1 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 2 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'SalesOrderHeader'. Scan count 1, logical reads 5, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical

 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
USE xSQLPASS2022;
GO
SELECT * FROM dbo.Orders WHERE CustomerID + 1 = 11021
GO
SELECT * FROM dbo.Orders WHERE CustomerID = 11021 - 1
GO
```

ARITHMETIC OPERATIONS

33 %

Results | Messages | Execution plan

```
 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 parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'Orders'. Scan count 9, logical reads 178818, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob phy
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 physi

 SQL Server Execution Times:
   CPU time = 6750 ms,  elapsed time = 871 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'Orders'. Scan count 1, logical reads 8, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical

 SQL Server Execution Times:
   CPU time = 1 ms,  elapsed time = 7 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
```

CONVERSION

```sql
USE WideWorldImporters;
GO
SELECT * FROM Sales.Orders WHERE CustomerPurchaseOrderNumber = 17913
GO
SELECT * FROM Sales.Orders WHERE CustomerPurchaseOrderNumber = '17913'
GO
```

33 %

Results | Messages | Execution plan

```
 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 parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'Orders'. Scan count 1, logical reads 692, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 30 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'Orders'. Scan count 1, logical reads 38, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 10 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
```

CONVERSION

```
USE TransactSQLTips;
GO
SELECT * FROM dbo.Orders WHERE CustId = 17913
GO
SELECT * FROM dbo.Orders WHERE CustId = '17913'
GO
```

33 %

Results | Messages | Execution plan

```
 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 parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'Orders'. Scan count 9, logical reads 721205, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical rea

 SQL Server Execution Times:
   CPU time = 2016 ms,  elapsed time = 342 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'Orders'. Scan count 1, logical reads 365, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 14 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
```

POORLY WRITTEN QUERIES

- Transact-SQL performance tips for developers
  - Do not use **functions** and **arithmetic operations** in filter conditions
  - Be careful with **local variables**
  - Avoid **non-SARG-able** predicates
  - Avoid **scalar** and **MSTV functions**, use **inline TVF** instead
  - Either **avoid data type conversions** or use **explicit** conversions

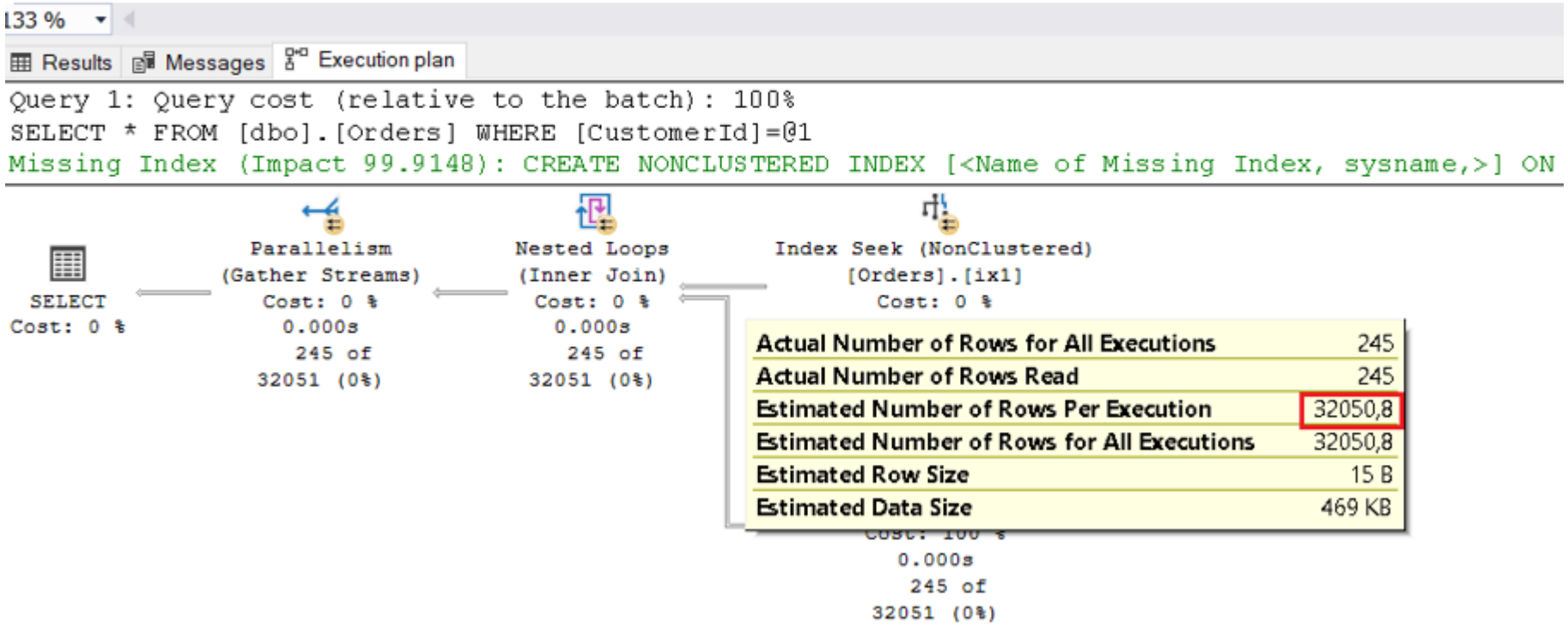
- If you want to have good performance with large tables, you have to **learn Transact-SQL**

CARDINALITY ESTIMATION ISSUES

CARDINALITY ESTIMATION ISSUES

- Overestimation – single table

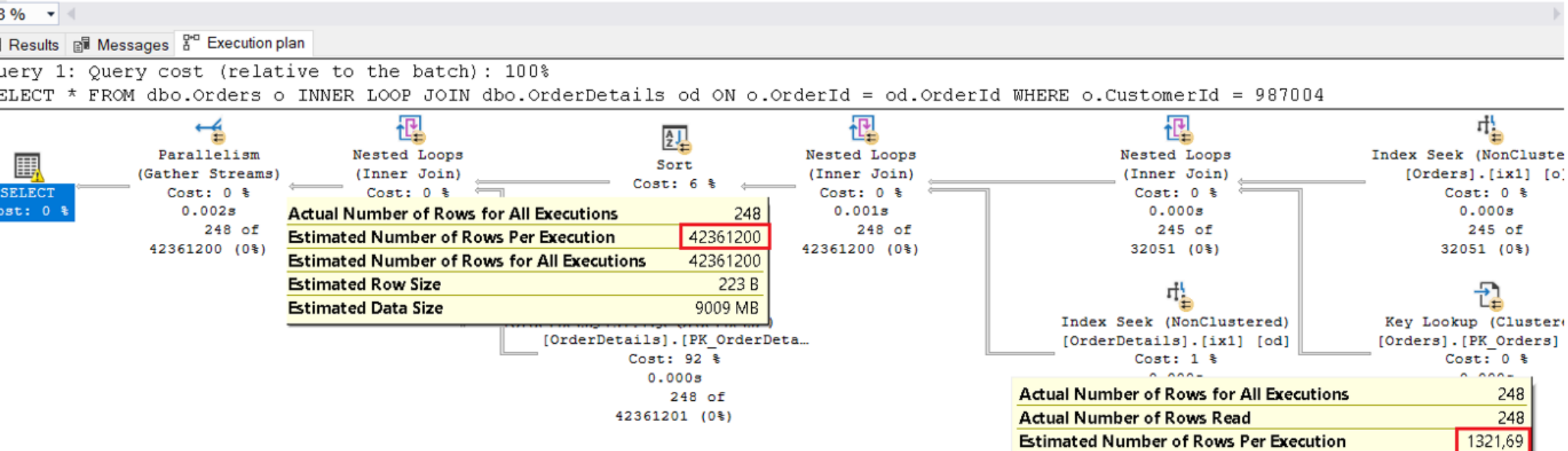
```
SELECT * FROM dbo.Orders
WHERE CustomerId = 987004
```



CARDINALITY ESTIMATION ISSUES

- Overestimation – two tables

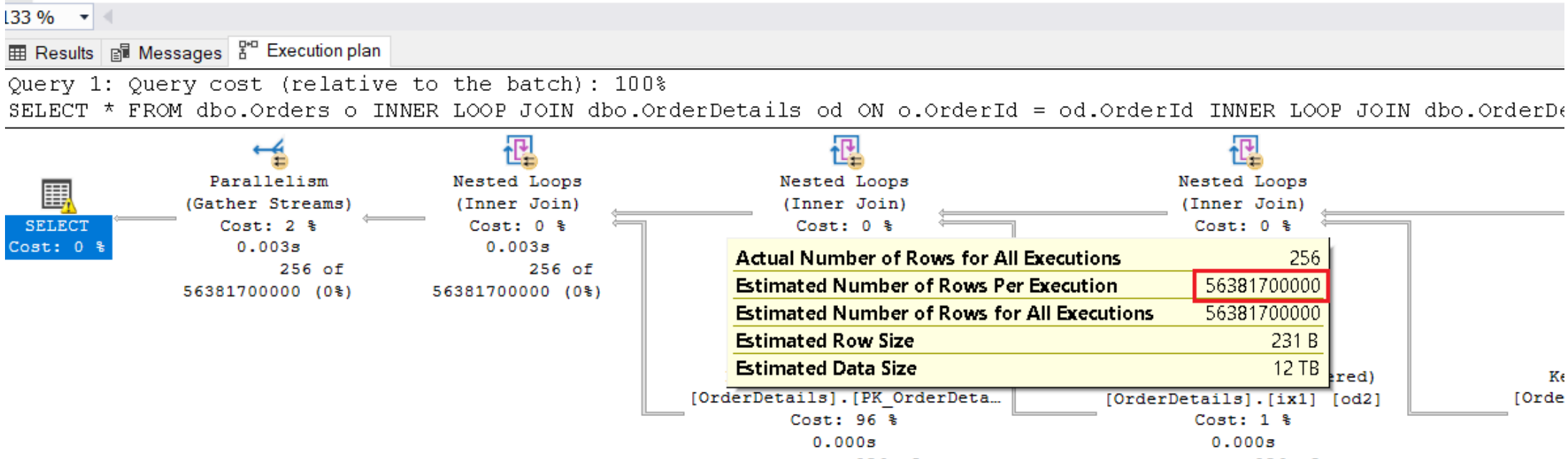
```
SELECT * FROM dbo.Orders o
INNER LOOP JOIN dbo.OrderDetails od ON o.OrderId = od.OrderId
WHERE o.CustomerId = 987004
```



CARDINALITY ESTIMATION ISSUES

- Overestimation – three tables

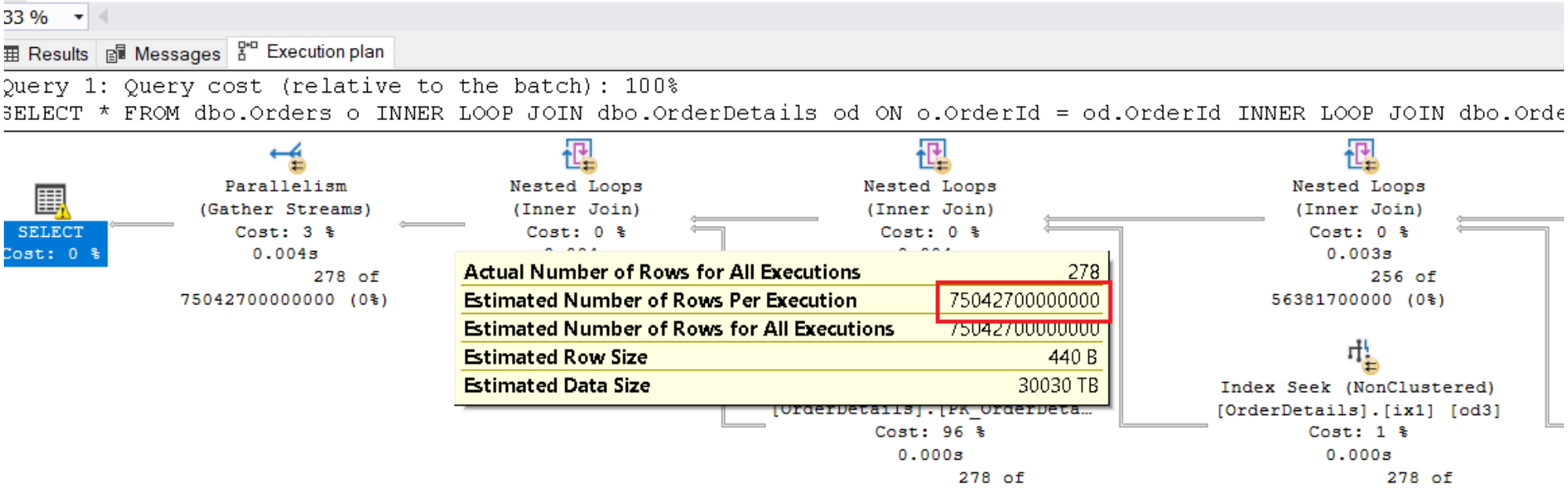
```
SELECT * FROM dbo.Orders o
INNER LOOP JOIN dbo.OrderDetails od ON o.OrderId = od.OrderId
INNER LOOP JOIN dbo.OrderDetails od2 ON od2.OrderId = od.OrderId
WHERE o.CustomerId = 987004
```



CARDINALITY ESTIMATION ISSUES

- Overestimation – four tables

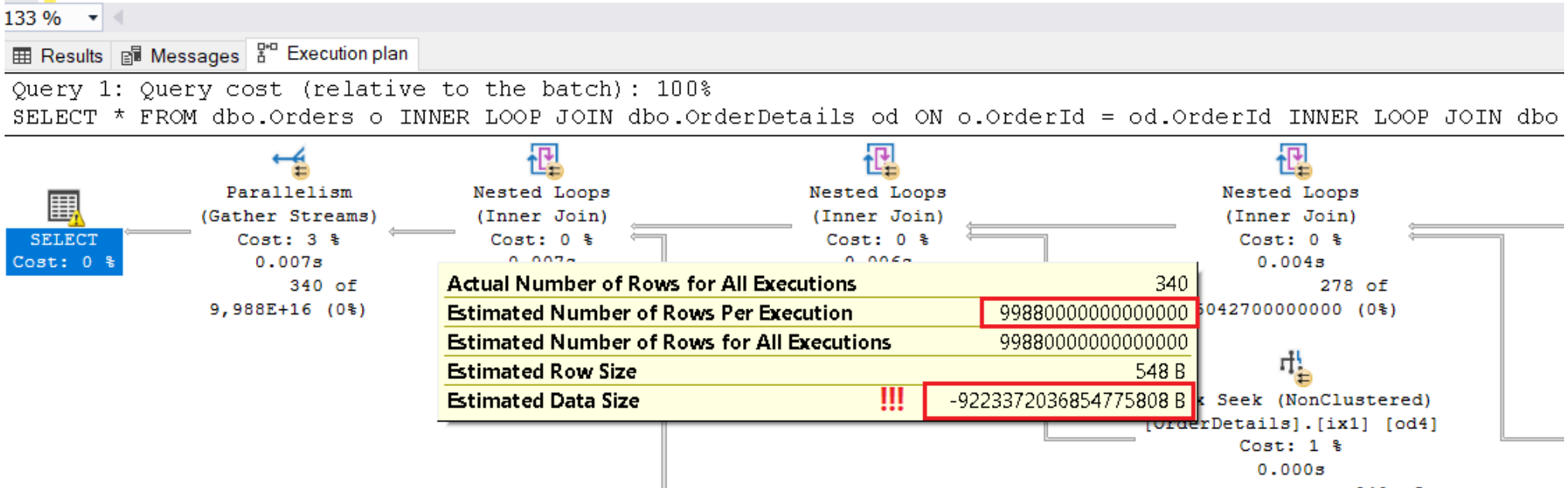
```
SELECT * FROM dbo.Orders o
INNER LOOP JOIN dbo.OrderDetails od ON o.OrderId = od.OrderId
INNER LOOP JOIN dbo.OrderDetails od2 ON od2.OrderId = od.OrderId
INNER LOOP JOIN dbo.OrderDetails od3 ON od2.OrderId = od3.OrderId
WHERE o.CustomerId = 987004
```



CARDINALITY ESTIMATION ISSUES

- Overestimation – five tables

```
SELECT * FROM dbo.Orders o
INNER LOOP JOIN dbo.OrderDetails od ON o.OrderId = od.OrderId
INNER LOOP JOIN dbo.OrderDetails od2 ON od2.OrderId = od.OrderId
INNER LOOP JOIN dbo.OrderDetails od3 ON od2.OrderId = od3.OrderId
INNER LOOP JOIN dbo.OrderDetails od4 ON od4.OrderId = od3.OrderId
WHERE o.CustomerId = 987004
```



Results | Messages

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows	Persisted Sample Percent
1	ix1	Mar 18 2024 5:32PM	100000000	441324	182	0.1770705	8	NO	NULL	100000000	0

HEADER

STATISTICS

	All density	Average Length	Columns
1	1.295387E-05	4	CustomerId
2	9.835855E-09	8	CustomerId, OrderId

DENSITY VECTOR

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	8759	0	359.7731	0	1
2	9260	1359.646	888.7393	1	1349.259
3	342011	1210538	7222.71	1296	934.0764
4	414959	414918.5	10156.94	376	1102.196
5	448081	208252.4	12865.45	154	1355.874
6	470984	250174.8	13091.16	164	1528.894
7	514446	344896.8	12865.45	301	1145.233
8	539865	200774.3	12865.45	184	1092.898
9	638477	688207.3	12865.45	570	1206.979
10	703544	439165.5	11736.9	315	1393.237
11	736760	358493.3	10156.94	269	1332.513
12	776393	340817.8	2031.387	273	1248.185
13	839199	478368.7	12865.45	404	1185.398
14	897495	451629	10382.65	292	1546.018
15	940707	354187.7	6319.871	246	1440.098
16	987004	429421.4	32050.78	310	1384.367
17	1024497	315437.8	13091.16	215	1468.33
18	1059029	371409.9	12865.45	353	1051.084
19	1105219	488792.6	4739.903	428	1142.991
20	1136246	401322.1	8802.678	306	1310.748
21	1158481	289151.3	11736.9	196	1477.112
22	1198004	436899.5	12865.45	336	1299.162

HISTOGRAM

AUTO UPDATE STATISTICS

- When are statistics automatically updated?

Table type	Table cardinality (*n*)	Recompilation threshold (# modifications)
	<= SQL SERVER 2014	
Temporary or permanent	*n* > 500	500 + (0.20 * *n*)
	>= SQL SERVER 2016	
Temporary or permanent	*n* > 500	MIN (500 + (0.20 * *n*), SQRT(1,000 * *n*))

- Until SQL Server 2014 => 20% of rows
- From SQL Server 2016 => $\sqrt{(1000 * Table\ Card)}$

AUTO UPDATE STATISTICS

- Old vs. New Threshold

ROWS	OLD THRESHOLD	NEW THRESHOLD
100.000	20.000	10.000
1.000.000	200.000	31.622
10.000.000	2.000.000	100.000
100.000.000	20.000.000	316.227
1.000.000.000	200.000.000	1.000.000

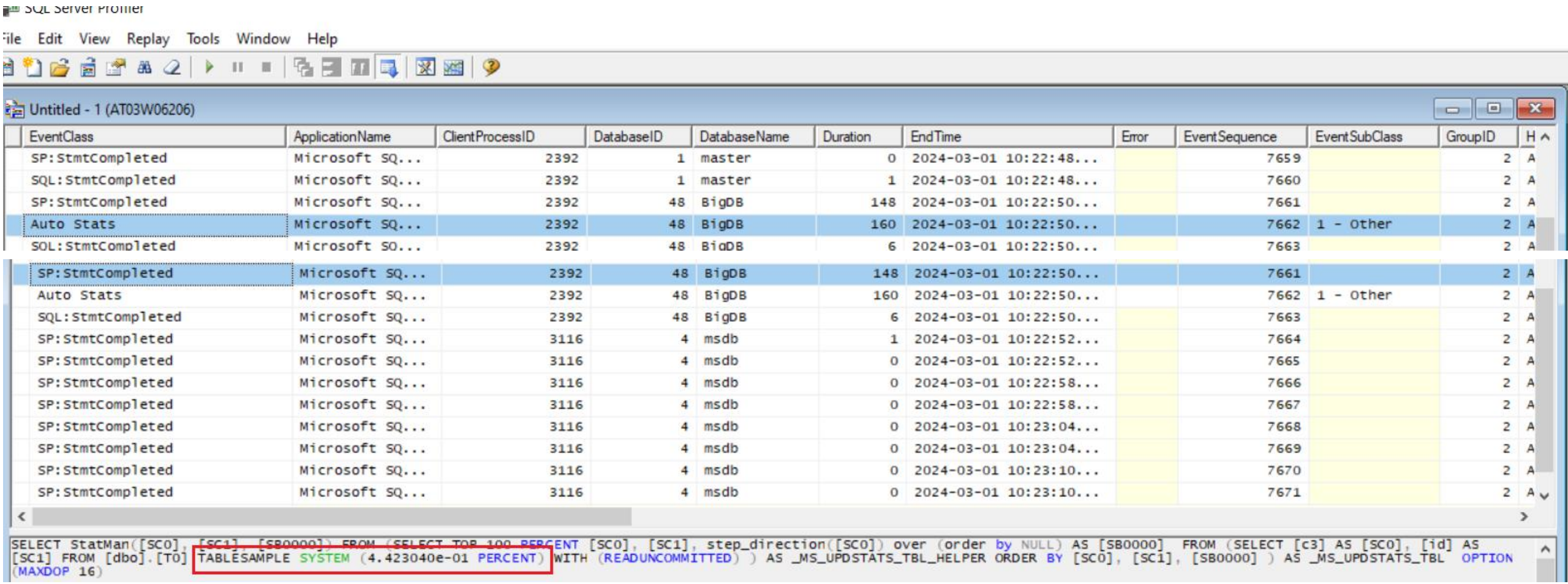
- for a billion rows table, the stats is updated after **one million changes**

AUTO UPDATE STATISTICS

- NOT after million rows have been changed, **but after million changes**
  - **1.000 rows are updated 1.000 times** => 1M changes
  - **one row is updated 1M times** => 1M changes
  - 1.000 rows is updated 1.000 times in a transaction and the **transaction is rolled back** => 1M changes

AUTO UPDATE STATISTICS

- What happens, when stats is auto updated?



EventClass	ApplicationName	ClientProcessID	DatabaseID	DatabaseName	Duration	EndTime	Error	EventSequence	EventSubClass	GroupID	H
SP:StmtCompleted	Microsoft SQ...	2392	1	master	0	2024-03-01 10:22:48...		7659		2	A
SQL:StmtCompleted	Microsoft SQ...	2392	1	master	1	2024-03-01 10:22:48...		7660		2	A
SP:StmtCompleted	Microsoft SQ...	2392	48	BigDB	148	2024-03-01 10:22:50...		7661		2	A
Auto Stats	Microsoft SQ...	2392	48	BigDB	160	2024-03-01 10:22:50...		7662	1 - Other	2	A
SQL:StmtCompleted	Microsoft SQ...	2392	48	BigDB	6	2024-03-01 10:22:50...		7663		2	A
SP:StmtCompleted	Microsoft SQ...	2392	48	BigDB	148	2024-03-01 10:22:50...		7661		2	A
Auto Stats	Microsoft SQ...	2392	48	BigDB	160	2024-03-01 10:22:50...		7662	1 - Other	2	A
SQL:StmtCompleted	Microsoft SQ...	2392	48	BigDB	6	2024-03-01 10:22:50...		7663		2	A
SP:StmtCompleted	Microsoft SQ...	3116	4	msdb	1	2024-03-01 10:22:52...		7664		2	A
SP:StmtCompleted	Microsoft SQ...	3116	4	msdb	0	2024-03-01 10:22:52...		7665		2	A
SP:StmtCompleted	Microsoft SQ...	3116	4	msdb	0	2024-03-01 10:22:58...		7666		2	A
SP:StmtCompleted	Microsoft SQ...	3116	4	msdb	0	2024-03-01 10:22:58...		7667		2	A
SP:StmtCompleted	Microsoft SQ...	3116	4	msdb	0	2024-03-01 10:23:04...		7668		2	A
SP:StmtCompleted	Microsoft SQ...	3116	4	msdb	0	2024-03-01 10:23:04...		7669		2	A
SP:StmtCompleted	Microsoft SQ...	3116	4	msdb	0	2024-03-01 10:23:10...		7670		2	A
SP:StmtCompleted	Microsoft SQ...	3116	4	msdb	0	2024-03-01 10:23:10...		7671		2	A

```
SELECT StatMan([SC0], [SC1], [SB0000]) FROM (SELECT TOP 100 PERCENT [SC0], [SC1], step_direction([SC0]) over (order by NULL) AS [SB0000] FROM (SELECT [c3] AS [SC0], [id] AS
[SC1] FROM [dbo].[T0] TABLESAMPLE SYSTEM (4.423040e-01 PERCENT) WITH (READUNCOMMITTED) ) AS _MS_UPDSTATS_TBL_HELPER ORDER BY [SC0], [SC1], [SB0000] ) AS _MS_UPDSTATS_TBL OPTION
(MAXDOP 16)
```

AUTO UPDATE STATISTICS

- For small and moderate tables, stats is updated by using values from the column **in all rows**
- For large tables, SQL Server uses a **sample of rows**
  - the larger the table, the smaller the sample
- In a case of ascending columns, this leads to **(huge) overestimations**

AUTO UPDATE STATISTICS

rows	percent	estimated	actual	ratio
10.000	100,0	20	20	1,0
50.000	100,0	37	37	1,0
90.000	82,2	73	27	2,7
100.000	72,7	66	37	1,8
200.000	39,3	64	20	3,2
500.000	16,5	994	99	10,0
1.000.000	9,6	614	44	14,0
5.000.000	2,7	11 236	864	13,0
10.000.000	1,7	13 597	432	31,5
50.000.000	0,6	69 747	1 640	42,5
100.000.000	0,4	132 954	2 968	44,8

MY EXAMPLE: ORDERS

- 100M rows – overestimation 130x

```
select top (1) * from sys.dm_db_stats_histogram (OBJECT_ID('Orders'), 2) order by equal_rows desc
select COUNT(1) from dbo.Orders o where o.CustomerId = 987004
```

133 %

Results | Messages

	object_id	stats_id	step_number	range_high_key	range_rows	equal_rows	distinct_range_rows	average_range_rows
1	581577110	2	15	987004	918665,9	32050,77	627	1464,146

overestimated 130x

	(No column name)
1	245

MY EXAMPLE: ORDERDETAILS

- 335M rows – overestimation 400x

```
dbcc show_statistics('orderdetails','ix1') with density_vector
select 3.973378E-06*335305134
select (select count(1) from OrderDetails)*1.0/(select COUNT (distinct OrderId) from OrderDetails)
```

133 %

Results | Messages | Execution plan

	All density	Average Length	Columns
1	3,973378E-06	4	OrderId
2	3,035861E-09	8	OrderId, OrderDetailsId

	(No column name)
1	1332,29404272265

1332,3 items per order

≈ 400x

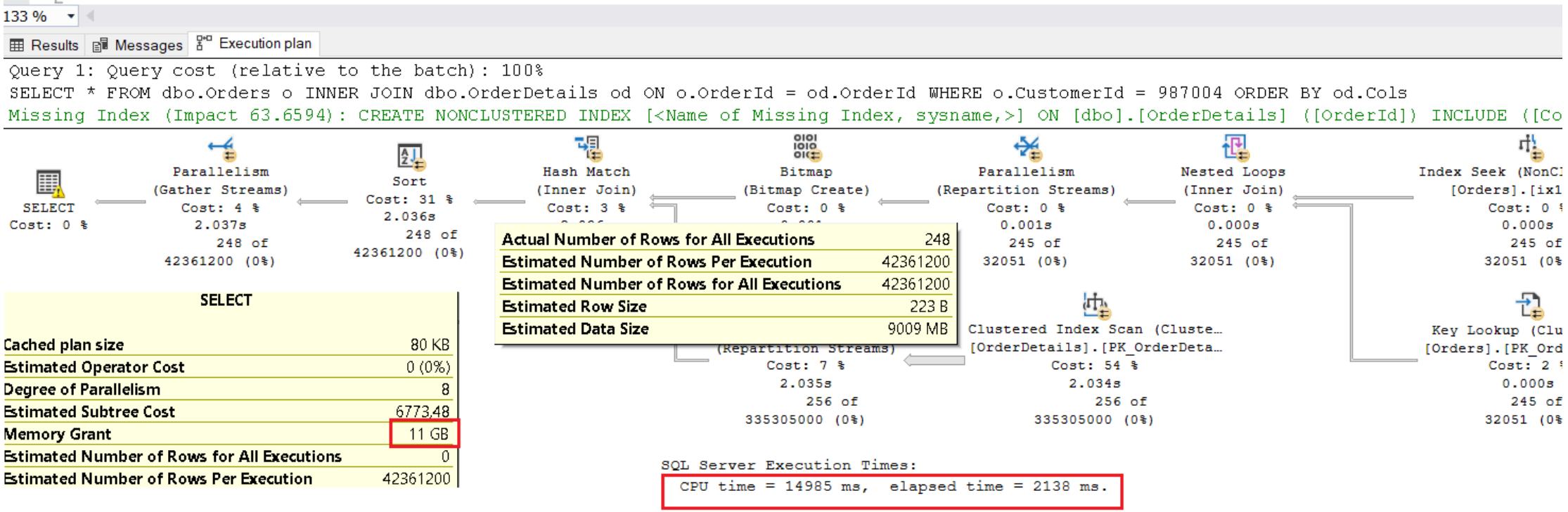
	(No column name)
1	3.359168722159

3,4 items per order

MY EXAMPLE: EPILOGUE

- Slow execution, a lot of CPU, Hash Join, a huge Memory Grant

```
SELECT * FROM dbo.Orders o
INNER JOIN dbo.OrderDetails od ON o.OrderId = od.OrderId
WHERE o.CustomerId = 987004
ORDER BY od.Cols;
```



MY EXAMPLE: SOLUTION

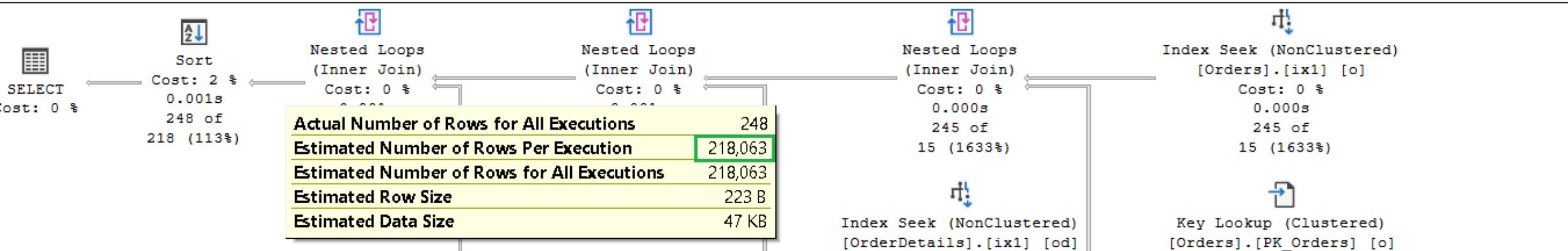
```
UPDATE STATISTICS Orders ix1 WITH SAMPLE 50 PERCENT, MAXDOP = 8
UPDATE STATISTICS OrderDetails ix1 WITH SAMPLE 20 PERCENT, MAXDOP = 8
GO

SELECT * FROM dbo.Orders o
INNER JOIN dbo.OrderDetails od ON o.OrderId = od.OrderId
WHERE o.CustomerId = 987004
ORDER BY od.Cols;
```

33 %

Results | Messages | Execution plan

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

SELECT * FROM dbo.Orders o INNER JOIN dbo.OrderDetails od ON o.OrderId = od.OrderId WHERE o.CustomerId = 987004 ORDER BY od.Cols



SOLUTION

- Do not let SQL Server to update stats with a default sample rate, **choose a higher value as a sample**!
- Use **PERSIST_SAMPLE_PERCENT = ON** option!
  - SQL Server 2016 SP1 CU4, SQL Server 2017 CU1
- Otherwise, SQL Server will **overwrite** it with a default sample rate with the next auto update

```
UPDATE STATISTICS Orders ix1
    WITH SAMPLE 50 PERCENT,
         MAXDOP = 8,
         PERSIST_SAMPLE_PERCENT = ON
```

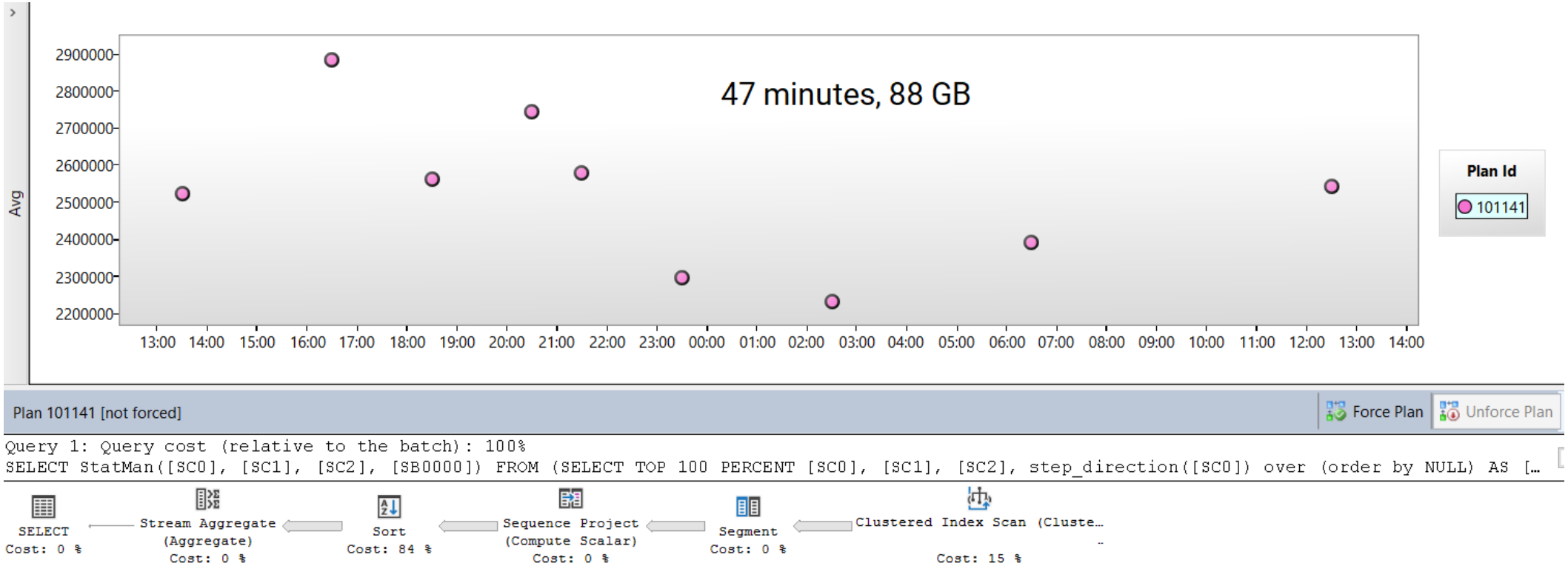
WARNING

- By increasing sample rate percent, you will increase IO traffic and CPU and memory usage too!
- Some stats update are useless i.e. you end up with the (almost) same stats histogram and same execution plans!
- If so, use **NORECOMPUTE** to exclude this stats from the auto-update routine

```
UPDATE STATISTICS Orders ix1
    WITH SAMPLE 50 PERCENT,
         MAXDOP = 8,
         PERSIST_SAMPLE_PERCENT = ON,
         NORECOMPUTE
```

AN EXAMPLE

- **9x daily**, avg. duration **47 minutes**, it uses **88 GB** memory
- All exec plans remain the same after each stats update!



KEY TAKEAWAYS

- Very large tables are spatial, and they require spatial treatment

- Do not let SQL Server to update statistics object on very large tables by using default sample rate – choose a higher sample rate

- Use **NORECOMPUTE** in case of frequent stats updates and take care about them manually

- When nonclustered columnstore index exists (including the column from stats), stats are calculated differently

Results | Messages

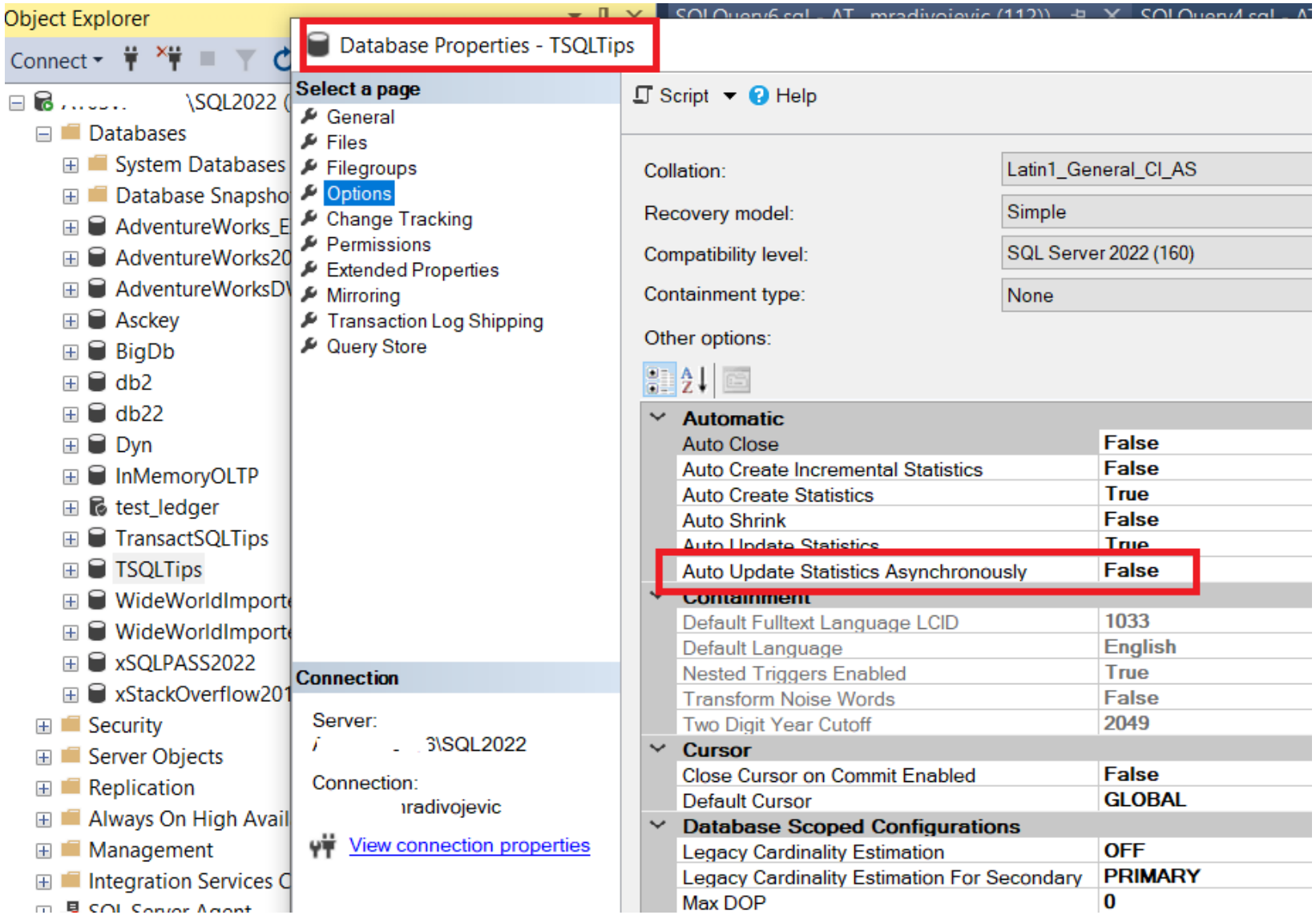
	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS
1	8759	0	359,773
2	9256	5,483388	1354,258
3	342011	1075705	8351,258
4	379147	208931,9	8125,548
5	448081	561759,4	11059,77
6	470984	235218,3	11059,77
7	514446	349655	12639,74
8	539865	194655,6	10834,06
9	626335	615918,6	10834,06
10	659775	243602,8	14896,84
11	736760	629515	6997
12	778541	360305,6	2708,516
13	839199	462958,7	11059,77
14	882769	331753,1	9028,387
15	987004	918665,9	32050,77
16	1024497	318609,8	12865,45
17	1059029	382059,9	12639,74
18	1109950	531620,7	10834,06
19	1158481	617731,4	9028,387
20	1198004	478368	11059,77
21	1220076	233858,7	4965,613
22	1279533	649683,1	11059,77
23	1382505	865866,4	11059,77

Results | Messages

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS
1	878	0	1
2	212333	576739,9	1
3	334560	542344,2	870,3508
4	432163	609804,3	870,3508
5	502337	537684,1	870,3508
6	565152	439156,8	870,3508
7	645569	644643,9	870,3508
8	750168	962638,7	626,8761
9	861347	898285,3	626,8761
10	922718	617127,3	1105,196
11	1035633	1139056	870,3508
12	1173131	1224713	870,3508
13	1244714	714988,9	870,3508
14	1378599	1145047	1105,196
15	1420398	475105,9	870,3508
16	1465395	414746,8	626,8761
17	1506831	567863,6	870,3508
18	1585759	891184,2	1105,196
19	1621563	641537,3	870,3508
20	1677584	646419,2	870,3508
21	1718505	788884,4	870,3508
22	1774540	554771,1	870,3508
23	1805644	397216,1	870,3508

UNDOCUMENTED, NO PROOF

AUTO UPDATE STATS ASYNC



- Default is **False**
- It should be set to **True**

AUTO CREATE STATS

- Auto create stats cannot be done asynchronuosly

- You can
  - execute it in some other window
  - create statistics on some other machine and import it (this is not documented or supported, but it works and it s not danger as it sounds)

AUTO CREATE STATS

```
select * from  veryLargeTable where fData0 = '1072141477' and UpdatedOn= '2024-03-03'
```

33 %

Results | Messages | Execution plan

```
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 92297 ms, elapsed time = 93245 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'atBAWInternLogging'. Scan count 1, logical reads 56, physical reads 0, page server reads 0, read-ahead reads 0, page server read-a

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 1 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.

Completion time: 2024-03-03T21:45:23.1120896+01:00
```

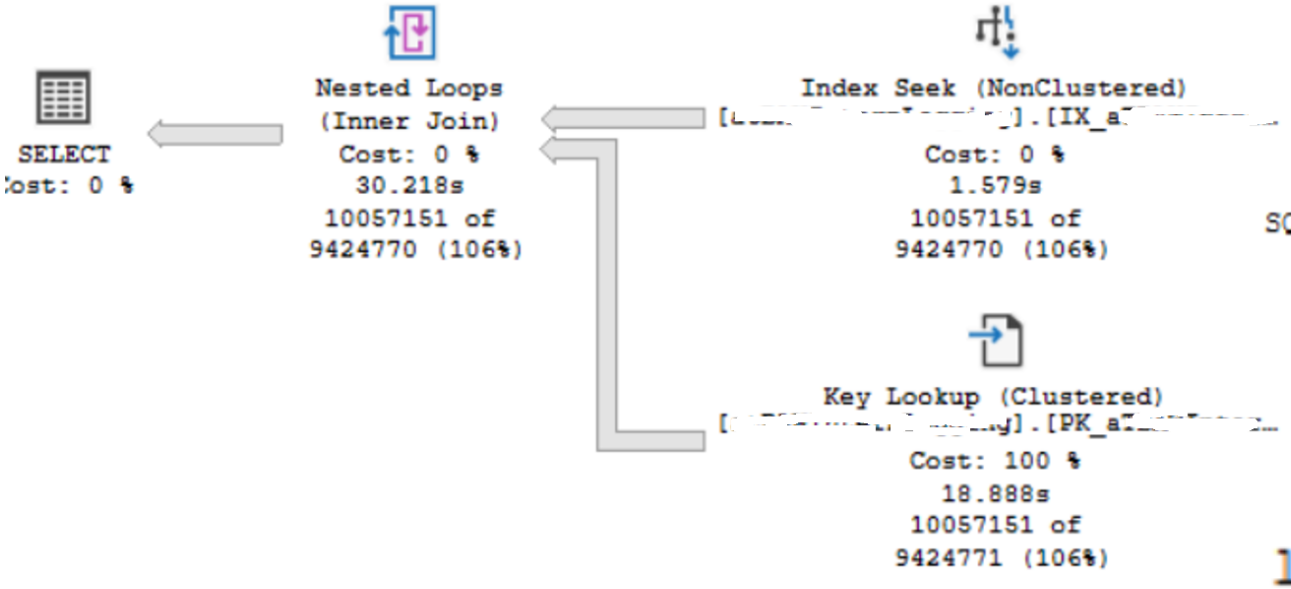
QUERYING A LARGE LOG TABLE

- Forcing **Clustered Index Seek**

```
SELECT fId, fTimestamp, fDescription FROM veryLargeTable
WHERE fTimestamp >='20110716' AND  fTimestamp <  '20110719'
GO

DECLARE @left BIGINT = (SELECT TOP 1 fID FROM veryLargeTable
WHERE fTimestamp < '20110716' ORDER BY fTimestamp DESC, fID DESC)

DECLARE @right BIGINT = (SELECT TOP 1 fID FROM veryLargeTable
WHERE fTimestamp < '20110719'ORDER BY fTimestamp DESC, fID DESC)

SELECT fId, fTimestamp, fDescription FROM veryLargeTable
WHERE fID > @left and fID <= @right and fTimestamp >='20110716' AND  fTimestamp < '20110719'
OPTION(RECOMPILE)
```

QUERYING A LARGE LOG TABLE

```
Query 1: Query cost (relative to the batch): 99%
SELECT [fId],[fTimestamp],[fDescription] FROM [...] WHERE [fTimestamp]>=@1 AND [fTimestamp]<@2
```

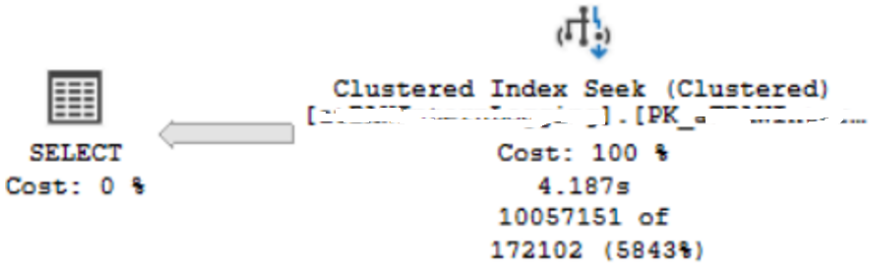



```
SQL Server Execution Times:
  CPU time = 33063 ms,  elapsed time = 34572 ms.
```

```
logical reads 50938186
```

>5x faster

```
Query 4: Query cost (relative to the batch): 1%
SELECT fId, fTimestamp, fDescription FROM aT... WHERE fID > @left and fID <= @right and fTimTimestamp DESC, fID DESC)
```




```
SQL Server Execution Times:
  CPU time = 5125 ms,  elapsed time = 6513 ms.
```

```
logical reads 305302
```

ASCENDING KEY PROBLEM

- It can be a huge problem if CL<=110, auto update stats threshold is 20%
- It looks better if CL>=120, but it is not perfect

- Example: table has 100M rows
  - CL = 110 -> threshold 20M rows
  - CL = 150 -> threshold 316.228 rows

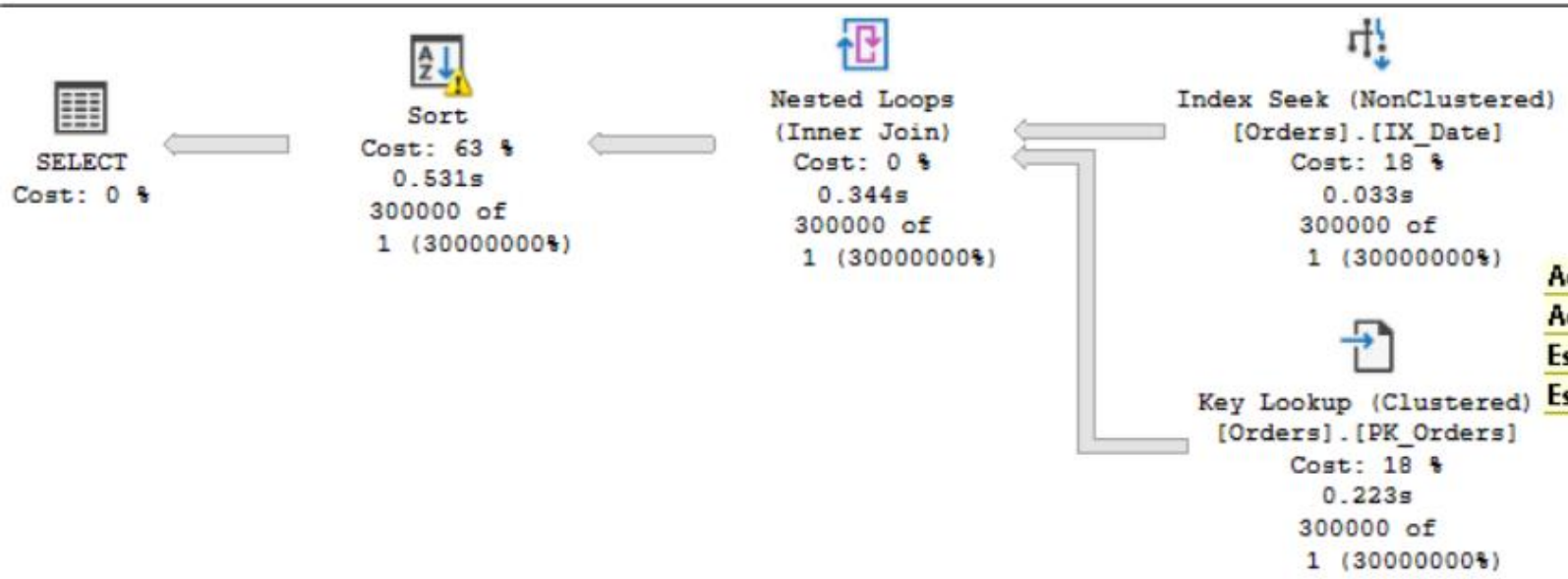
ASCENDING KEY PROBLEM

- Orders table has 100M rows
- Adding 300K rows -> no stats update

```sql
SELECT * FROM dbo.Orders
WHERE OrderDate >= '20240101' AND OrderDate < '20250101' ORDER BY amount

SELECT * FROM dbo.Orders
WHERE OrderDate >= '20240101' AND OrderDate < '20240101 00:05:00.000' ORDER BY amount
```

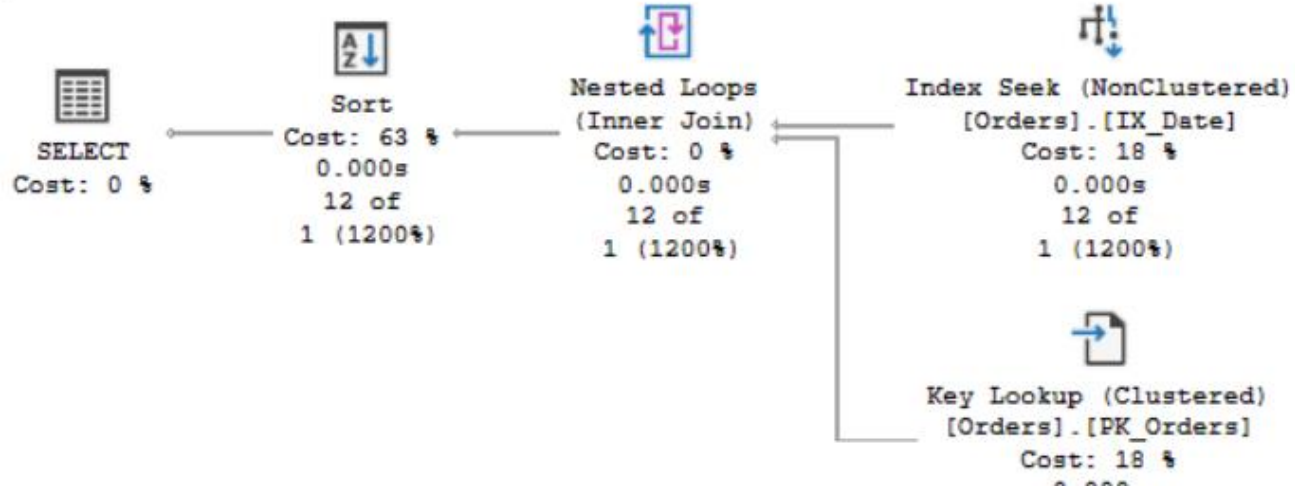
CL = 110

Query 1: Query cost (relative to the batch): 50%
SELECT * FROM [dbo].[Orders] WHERE [OrderDate]>=@1 AND [OrderDate]<@2 ORDER BY [amount] ASC



Actual Number of Rows for All Executions	300000
Actual Number of Rows Read	300000
Estimated Number of Rows Per Execution	1
Estimated Number of Rows for All Executions	1

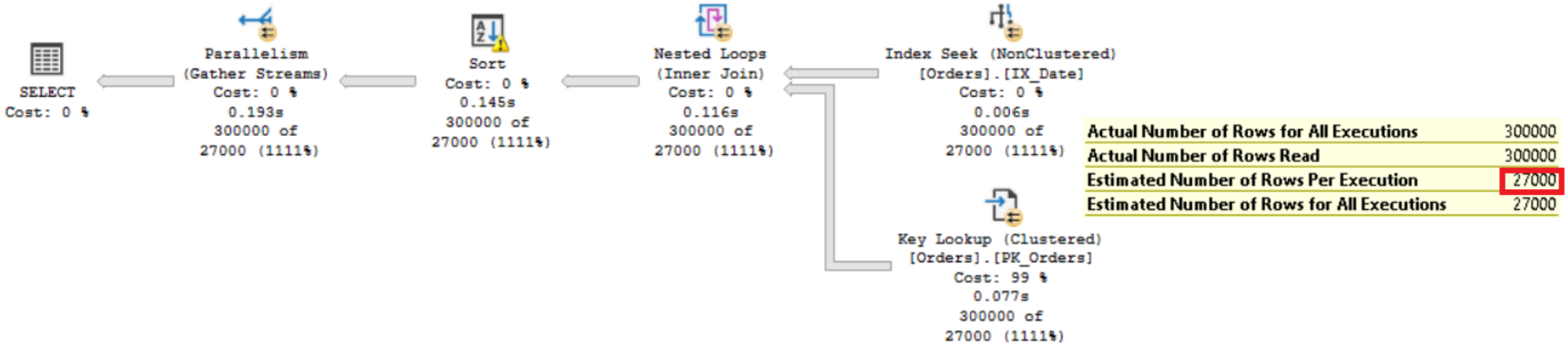
Query 2: Query cost (relative to the batch): 50%
SELECT * FROM [dbo].[Orders] WHERE [OrderDate]>=@1 AND [OrderDate]<@2 ORDER BY [amount] ASC



Actual Number of Rows for All Executions	12
Actual Number of Rows Read	12
Estimated Number of Rows Per Execution	1
Estimated Number of Rows for All Executions	1

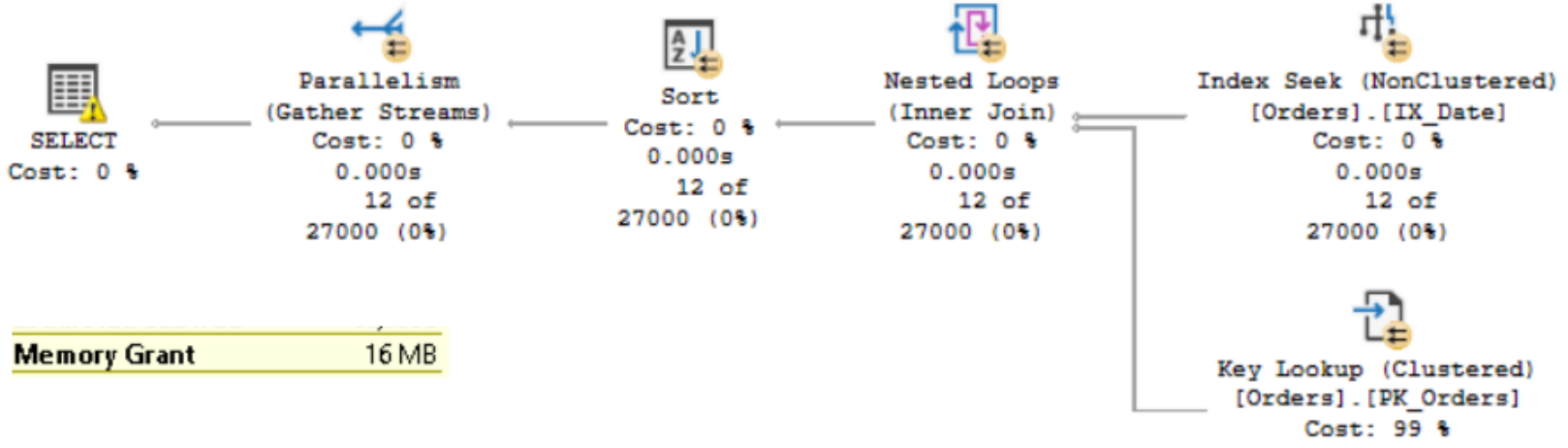
CL = 150

Query 1: Query cost (relative to the batch): 50%
SELECT * FROM [dbo].[Orders] WHERE [OrderDate]>=@1 AND [OrderDate]<@2 ORDER BY [amount] ASC



Actual Number of Rows for All Executions	300000
Actual Number of Rows Read	300000
Estimated Number of Rows Per Execution	27000
Estimated Number of Rows for All Executions	27000

Query 2: Query cost (relative to the batch): 50%
SELECT * FROM [dbo].[Orders] WHERE [OrderDate]>=@1 AND [OrderDate]<@2 ORDER BY [amount] ASC



Actual Number of Rows for All Executions	12
Actual Number of Rows Read	12
Estimated Number of Rows Per Execution	27000
Estimated Number of Rows for All Executions	27000

Memory Grant	16 MB

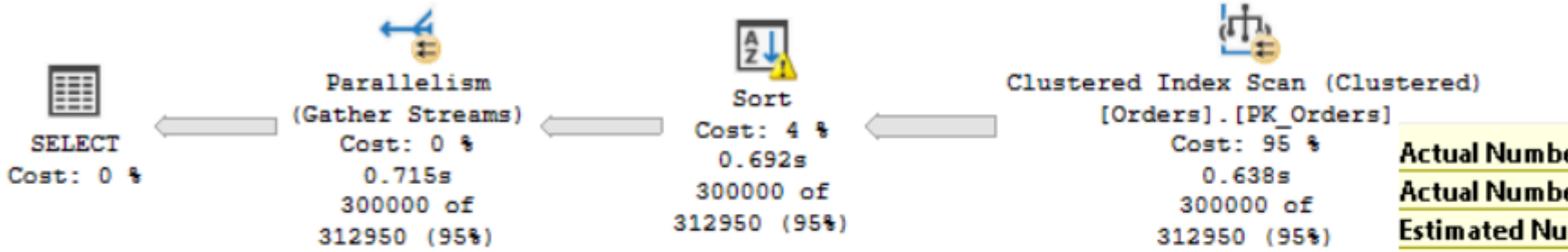
FILTERED STATISTICS

```
CREATE STATISTICS s1 ON Orders(OrderDate) WHERE OrderDate >= '20240101'
```

133 %

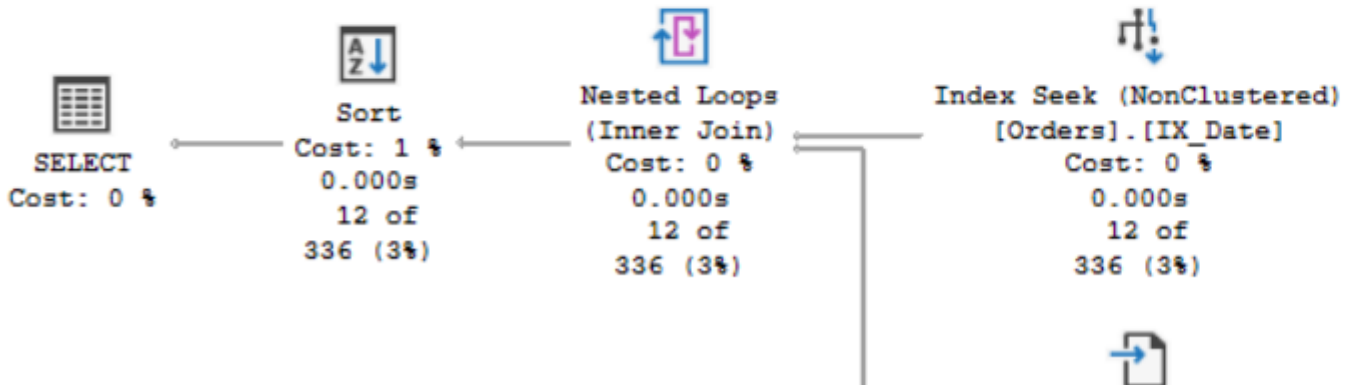
Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [dbo].[Orders] WHERE [OrderDate]>=@1 AND [OrderDate]<@2 ORDER BY [amount] ASC
Missing Index (Impact 64.7991): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Orders] ([orderda
```



Actual Number of Rows for All Executions	300000
Actual Number of Rows Read	100300000
Estimated Number of Rows Per Execution	312950
Estimated Number of Rows for All Executions	312950

```
Query 2: Query cost (relative to the batch): 0%
SELECT * FROM [dbo].[Orders] WHERE [OrderDate]>=@1 AND [OrderDate]<@2 ORDER BY [amount] ASC
```



Actual Number of Rows for All Executions	12
Actual Number of Rows Read	12
Estimated Number of Rows Per Execution	336,477
Estimated Number of Rows for All Executions	336,477

ADDING/REMOVING IDENTITY

- You cannot simply **remove the IDENTITY** property from a column
  - You need either to create and populate a **new column** or a **new table**
- But there is another option
  - To use **SWITCH** partition functionality
- The same applies to **adding IDENTITY**

TAKEAWAYS

- Do not let SQL Server to update statistics object on very large tables by using default sample rate – choose a higher sample rate
  - Use **NORECOMPUTE** in case of frequent stats updates and take care about them manually
- Set **Auto update stats asynchronously** to **True**
- Be careful when adding a new column to a large table (**create statistics for it manually**)
- Consider **index strategy** before the table becomes too large
- **Learn** how to write performant queries

SESSION FEEDBACK