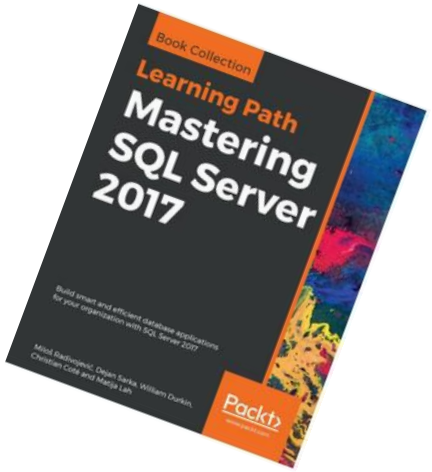
INTELLIGENT QUERY PROCESSING IN SQL SERVER – PROS AND CONS

Miloš Radivojević

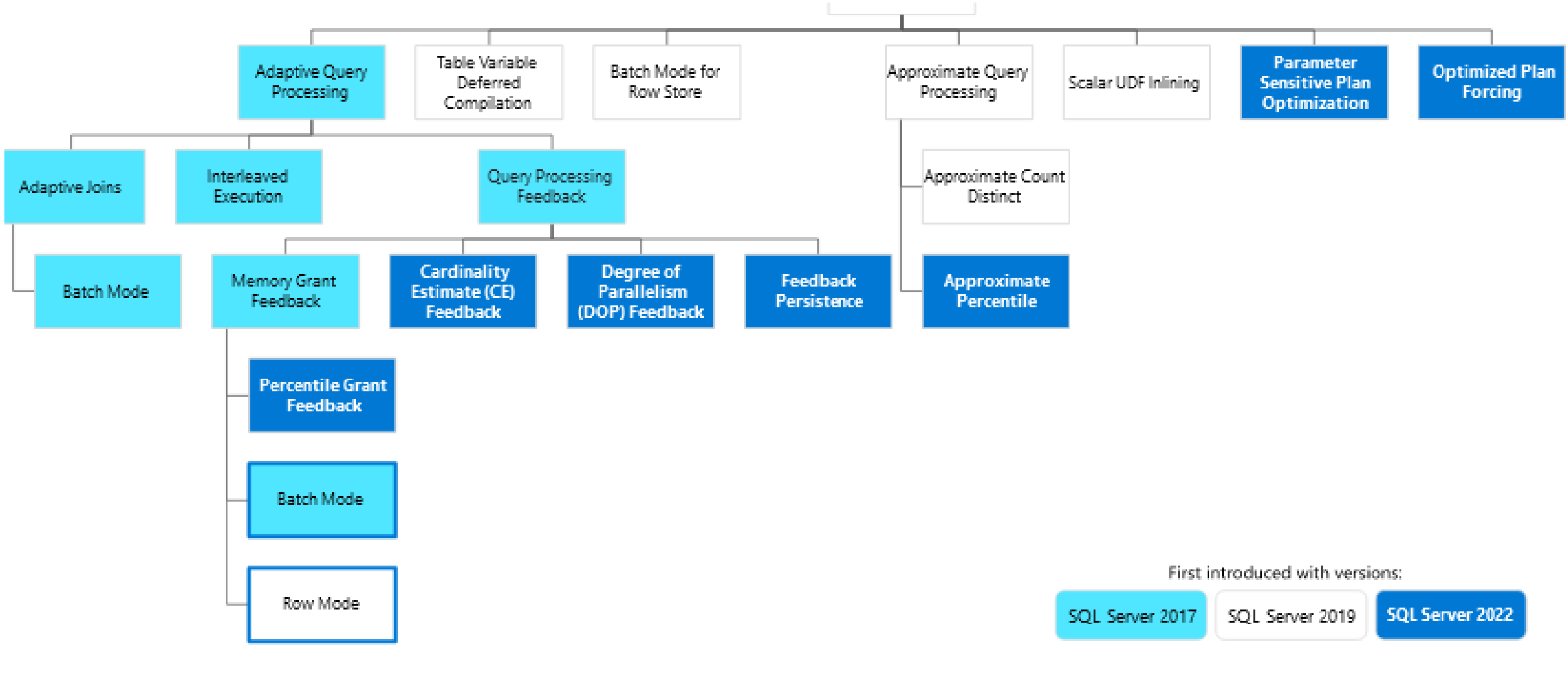
- Head of MSSQL Database Engineering at Entain, Austria
- Conference speaker
- Book author
- LinkedIn: **milossql**



Why Me?

- I am responsible for
  - a prod environment with more than 200 servers with SQL Server 2019 (with the latest compatibility level)
  - huge databases and high workloads
  - a real 24/7 environment
  - about 20 servers with SQL Server 2022
- 20+ years experience with SQL Server and performance troubleshooting and tuning
- Many internal and external workshops and sessions about IQP
- I had and have opportunity to talk with the corresponding product teams

Intelligent Query Processing in SQL Server 2022



Intelligent Query Processing in SQL Server

- A marketing name for 7 independent features in SQL Server 2019 or 11 independent features in SQL Server 2022
- There are no IQP settings, only settings for individual features
- When your database is at the latest compatibility level, all available IQP features are ON by default

Inefficient Execution Plans in SQL Server

- Execution plans are sometimes suboptimal for these queries
  - Queries using table variables
  - Queries with scalar user-defined functions
  - Queries referencing multi-statement table valued functions
  - Complex queries
  - Queries with tables with skew data distribution

- Common serious issues:
  - Inappropriate operator choice
  - Memory Grant under- or overestimation

Execution plans and execution statistics for one query

from SQL Server 2012 until SQL Server 2022

Calling an MSTV user-defined function

```sql
--funkcija
CREATE OR ALTER FUNCTION Fact.ufn_WhatIfOutlierEventQuantity(
@event VARCHAR(30), @beginOrderDateKey DATE, @endOrderDateKey DATE)
RETURNS @OutlierEventQuantity TABLE (
    [Order Key] [bigint],
    [City Key] [int] NOT NULL,
    [Customer Key] [int] NOT NULL,
    [Stock Item Key] [int] NOT NULL,
    [Order Date Key] [date] NOT NULL,
    [Picked Date Key] [date] NULL,
    [Salesperson Key] [int] NOT NULL,
    [Picker Key] [int] NULL,
    [OutlierEventQuantity] [int] NOT NULL)
AS
BEGIN
    IF @event = 'Mild Recession'
    INSERT  @OutlierEventQuantity
    SELECT [o].[Order Key], [o].[City Key], [o].[Customer Key],
           [o].[Stock Item Key], [o].[Order Date Key], [o].[Picked Date Key],
           [o].[Salesperson Key], [o].[Picker Key],
           CASE
            WHEN [o].[Quantity] > 2 THEN [o].[Quantity] * .5
            ELSE [o].[Quantity]
           END
    FROM [Fact].[Order] AS [o]
    INNER JOIN [Dimension].[City] AS [c]
        ON [c].[City Key] = [o].[City Key]

    IF @event = 'Hurricane - South Atlantic'
    INSERT  @OutlierEventQuantity
    SELECT [o].[Order Key], [o].[City Key], [o].[Customer Key],
           [o].[Stock Item Key], [o].[Order Date Key], [o].[Picked Date Key],
           [o].[Salesperson Key], [o].[Picker Key],
```

```sql
--procedura
CREATE OR ALTER PROCEDURE Fact.TestMSTVF
AS
BEGIN
    SELECT o.[Order Key], o.[Description], o.[Package], o.[Quantity], f.[Outli
    FROM [Fact].[Order] AS o
    INNER JOIN Fact.ufn_WhatIfOutlierEventQuantity('Mild Recession','20130101'
        ON o.[Order Key] = f.[Order Key]
        AND o.[City Key] = f.[City Key]
        AND o.[Customer Key] = f.[Customer Key]
        AND o.[Stock Item Key] = f.[Stock Item Key]
        AND o.[Order Date Key] = f.[Order Date Key]
        AND o.[Picked Date Key] = f.[Picked Date Key]
        AND o.[Salesperson Key] = f.[Salesperson Key]
        AND o.[Picker Key] = f.[Picker Key]
    INNER JOIN [Dimension].[Stock Item] AS si
        ON o.[Stock Item Key] = [si].[Stock Item Key]
    WHERE [si].[Lead Time Days] > 0 AND o.[Quantity] > 50;
END
GO
```

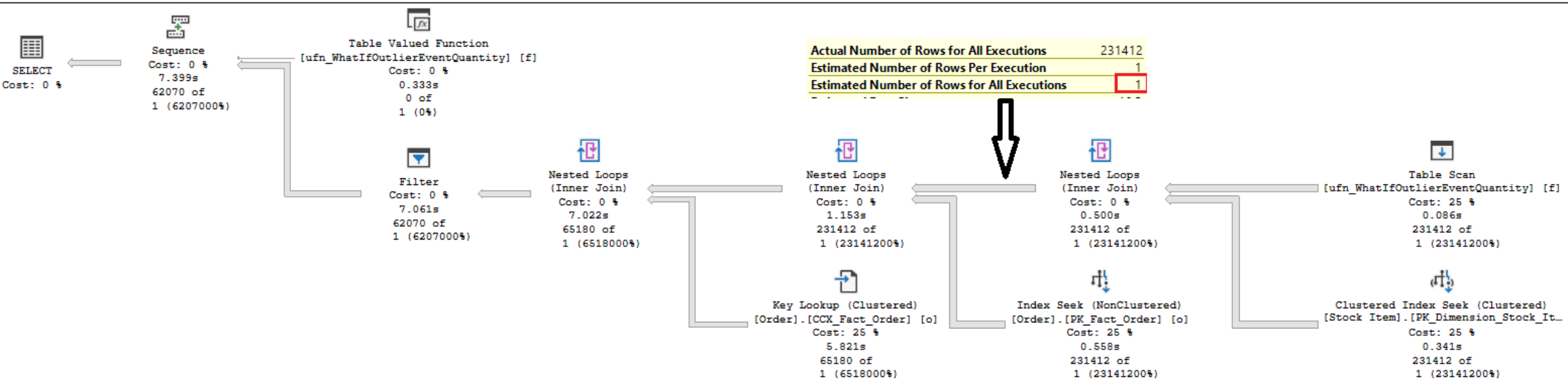
SQL Server 2012 / CL 110

```
ALTER DATABASE WideWorldImportersDW SET COMPATIBILITY_LEVEL = 110;
GO
EXEC Fact.TestMSTVF;
GO
/*
 SQL Server Execution Times:
   CPU time = 7485 ms,  elapsed time = 7473 ms.
*/
```

33 %

Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT o.[Order Key], o.[Description], o.[Package], o.[Quantity], f.[OutlierEventQuantity] FROM [Fact].[Order] AS o INNER JOIN Fact.ufn_WhatIfOutlierEventQuantity('Mild Recession','20130101
```



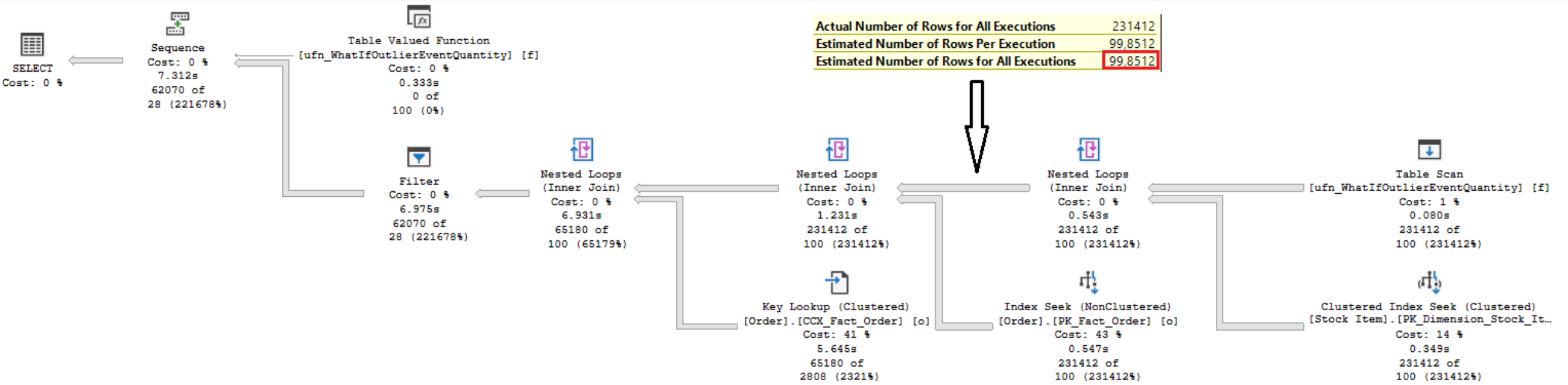
SQL Server 2014 / CL 120

```
ALTER DATABASE WideWorldImportersDW SET COMPATIBILITY_LEVEL = 120;
GO
EXEC Fact.TestMSTVF;
GO
/*
 SQL Server Execution Times:
   CPU time = 7375 ms,  elapsed time = 7384 ms.
*/
```

33 %

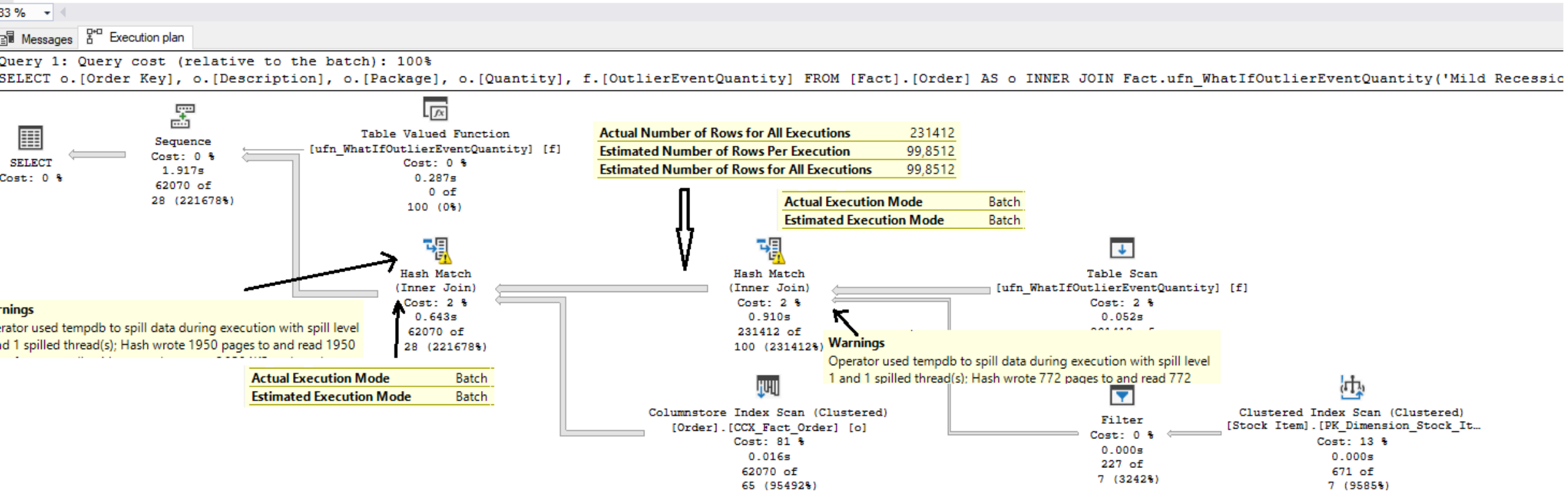
Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT o.[Order Key], o.[Description], o.[Package], o.[Quantity], f.[OutlierEventQuantity] FROM [Fact].[Order] AS o INNER JOIN Fact.ufn_WhatIfOutlierEventQuantity('Mild Recession','2013
```



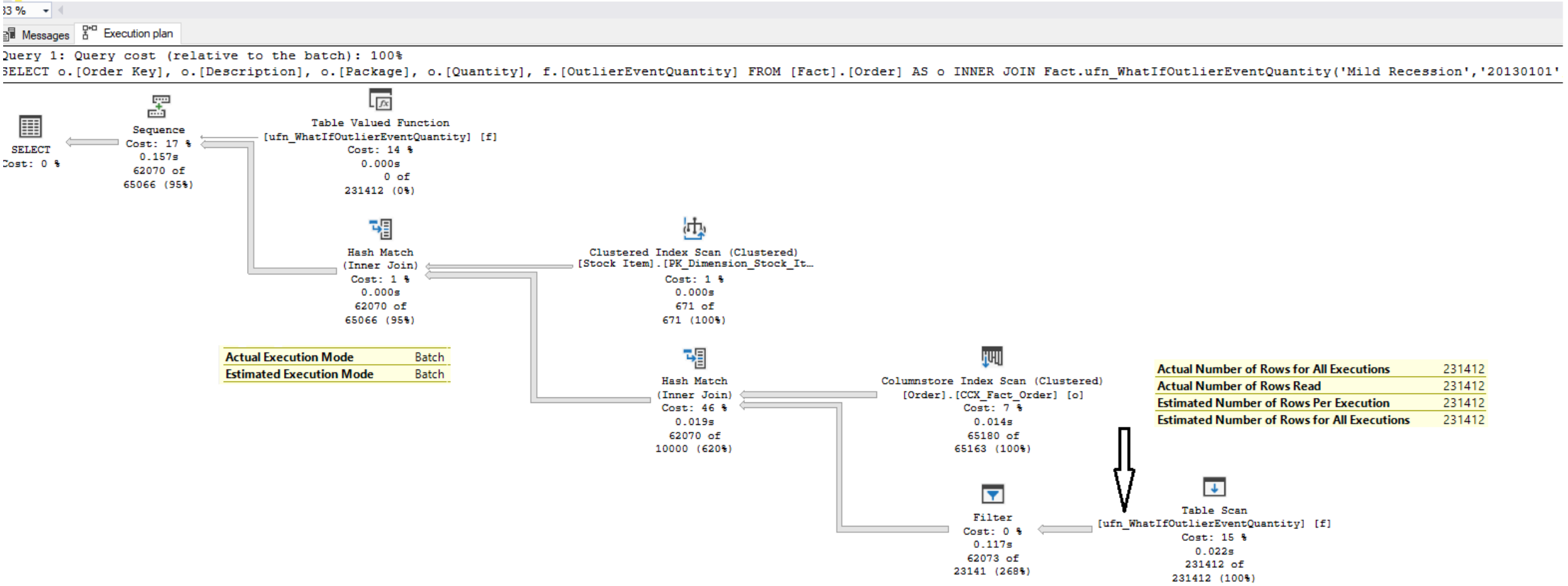
SQL Server 2016 / CL 130

```
ALTER DATABASE WideWorldImportersDW SET COMPATIBILITY_LEVEL = 130;
GO
EXEC Fact.TestMSTVF;
GO
/*
 SQL Server Execution Times:
   CPU time = 594 ms,  elapsed time = 2015 ms.
*/
```



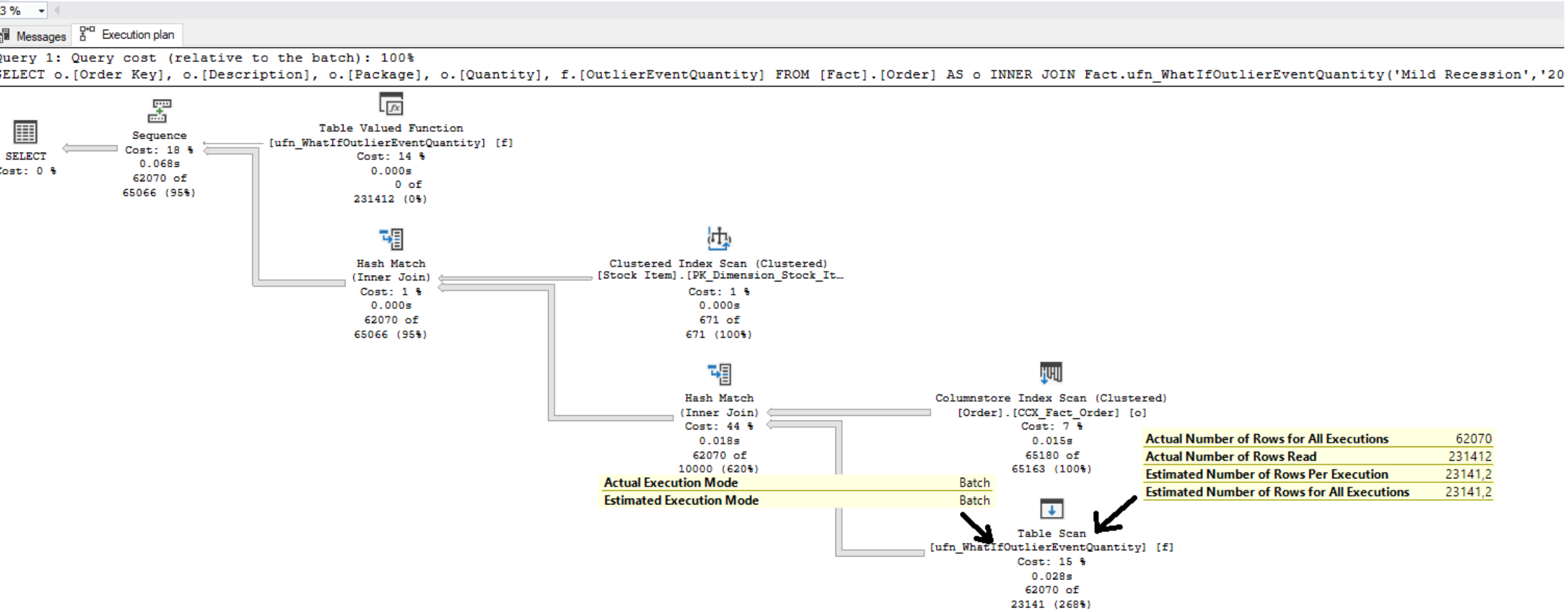
SQL Server 2017 / CL 140

```
ALTER DATABASE WideWorldImportersDW SET COMPATIBILITY_LEVEL = 140;
GO
EXEC Fact.TestMSTVF;
GO
/*
 SQL Server Execution Times:
   CPU time = 515 ms,  elapsed time = 539 ms.
*/
```



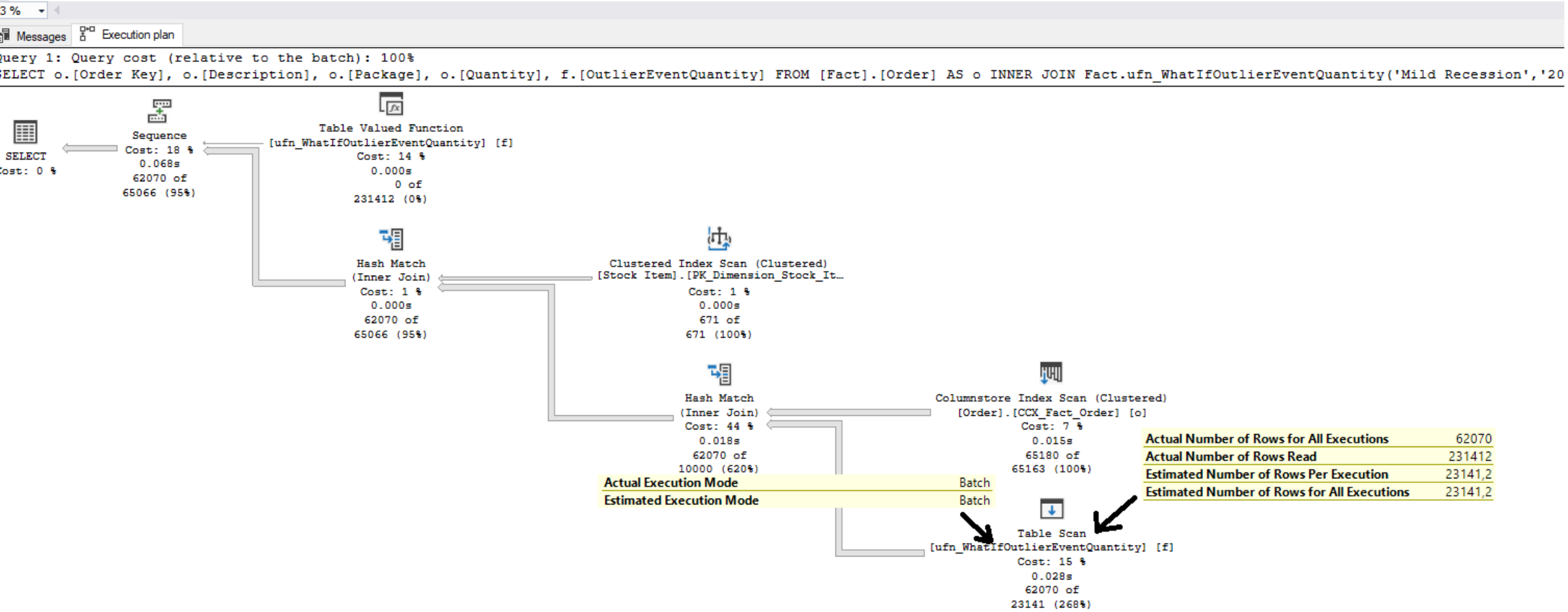
SQL Server 2019 / CL 150

```
ALTER DATABASE WideWorldImportersDW SET COMPATIBILITY_LEVEL = 150;
GO
EXEC Fact.TestMSTVF;
GO
/*
 SQL Server Execution Times:
   CPU time = 421 ms,  elapsed time = 432 ms.
*/
```



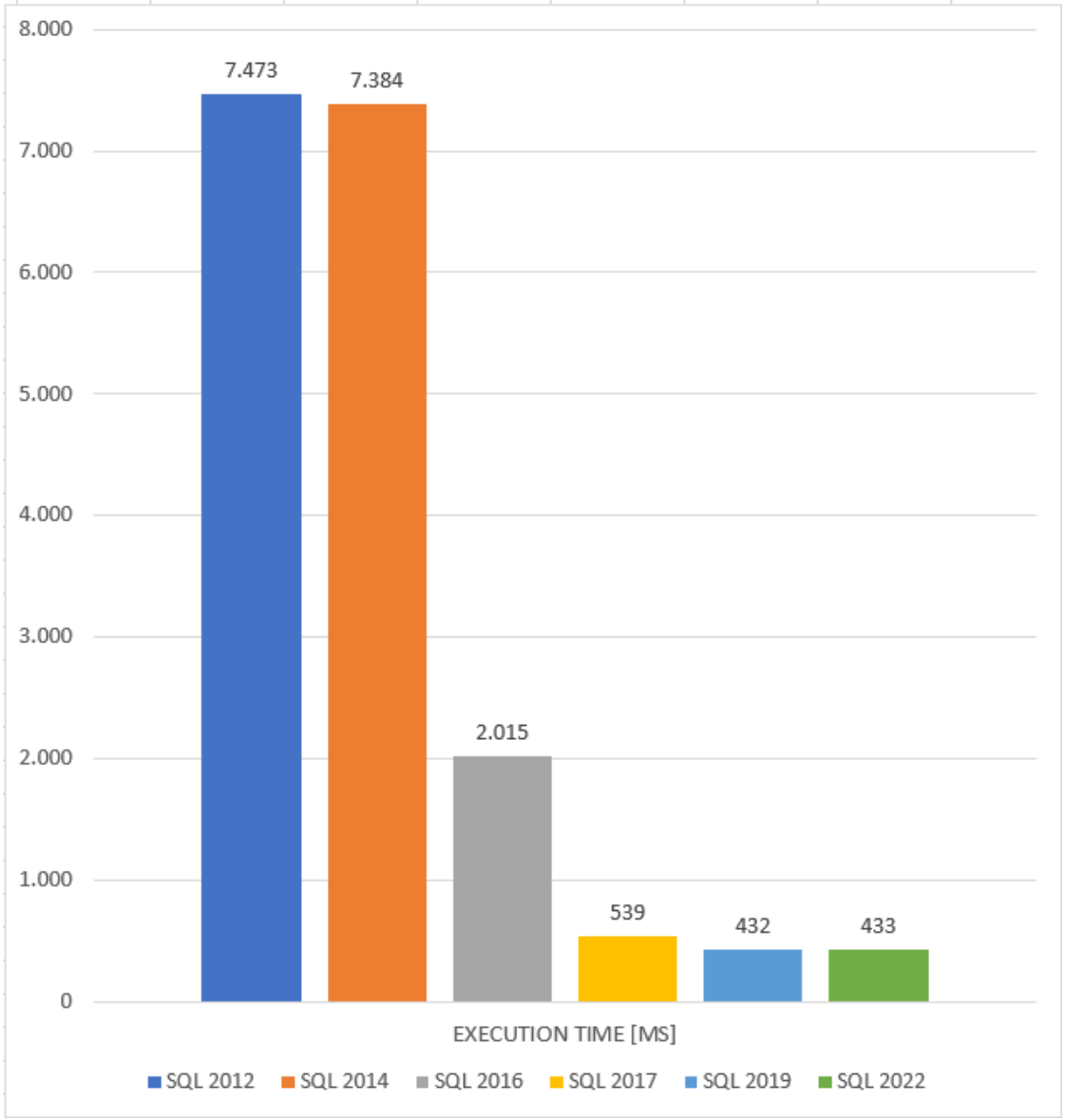
SQL Server 2022 / CL 160

```
ALTER DATABASE WideWorldImportersDW SET COMPATIBILITY_LEVEL = 150;
GO
EXEC Fact.TestMSTVF;
GO
/*
 SQL Server Execution Times:
   CPU time = 421 ms,  elapsed time = 432 ms.
*/
```



Journey: SQL Server 2012 – SQL Server 2022

- Execution time comparison
- No code changes – only upgrading the compatibility level



Intelligent Query Processing in SQL Server

- Most beneficial for those
  - who did not tune their queries
  - who have a lot of databases

- Some features could be even danger
  - For those whose database performance are critical for business

1

Interleaved Execution

Queries with MSTVFs

- It could be beneficial only for queries using multi-statement table-valued functions (MSTVF)
- And even not for all of them, rather for a small part

- On the other side, the feature is revolutionary
  - For the first time in SQL Server history, a part of a query is executed during the plan generation

Interleaved Execution

- Prior to SQL Server 2017
  - Estimated Number of Rows is too low (1 or 100)
  - an execution plan with the Nested Loops Join operator
  - insufficient memory allocated => tempdb spills

- In SQL Server 2019 and 2022 (with Interleaved Execution)
  - Estimated Number of Rows = Actual Number of Rows
  - optimizer can choose the appropriate operator
  - proper memory allocation => no tempdb spills

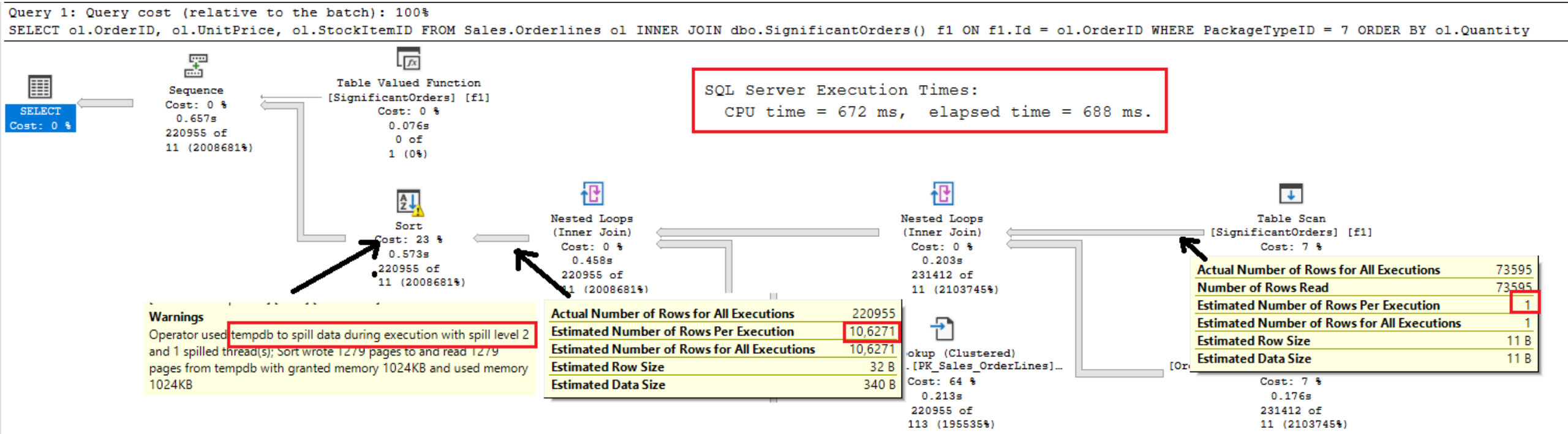
Demo – a sample MSTVF

```
CREATE OR ALTER FUNCTION dbo.SignificantOrders()
RETURNS @T TABLE (ID INT NOT NULL)
AS
BEGIN
    INSERT INTO @T SELECT OrderId FROM Sales.Orders
    RETURN
END
```

- It returns about 73K rows

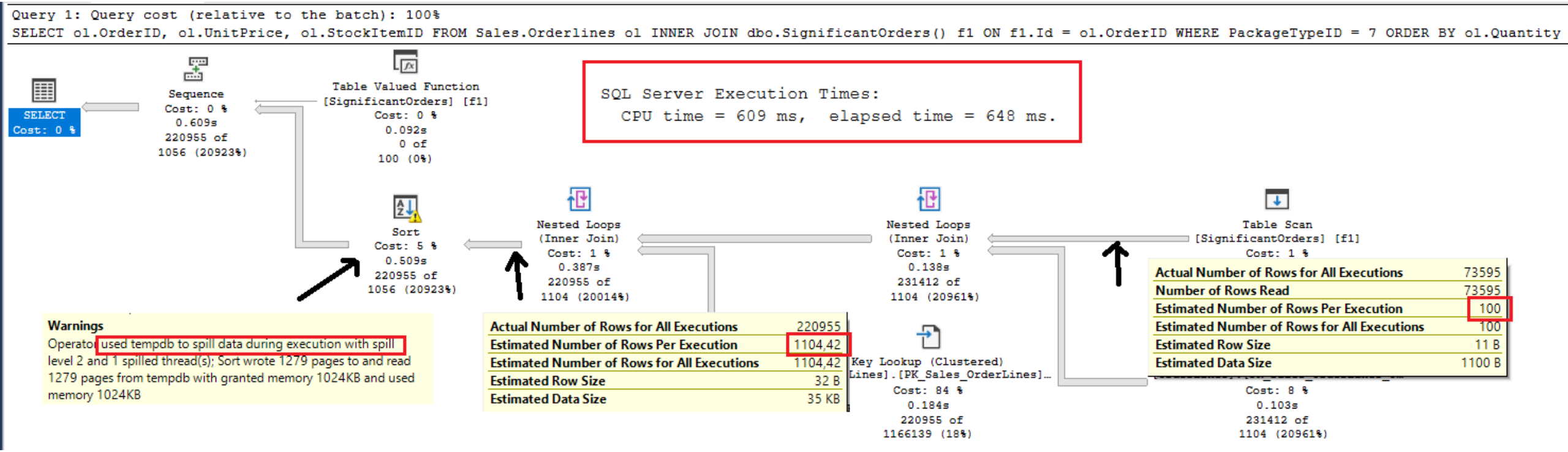
Demo – Query under SQL Server 2012

```
ALTER DATABASE WideWorldImporters SET COMPATIBILITY_LEVEL = 110;
GO
SELECT ol.OrderID, ol.UnitPrice, ol.StockItemID
FROM Sales.Orderlines ol
INNER JOIN dbo.SsignificantOrders() f1 ON f1.Id = ol.OrderID
WHERE PackageTypeID = 7 ORDER BY ol.Quantity;
```



Demo – Query under SQL Server 2014

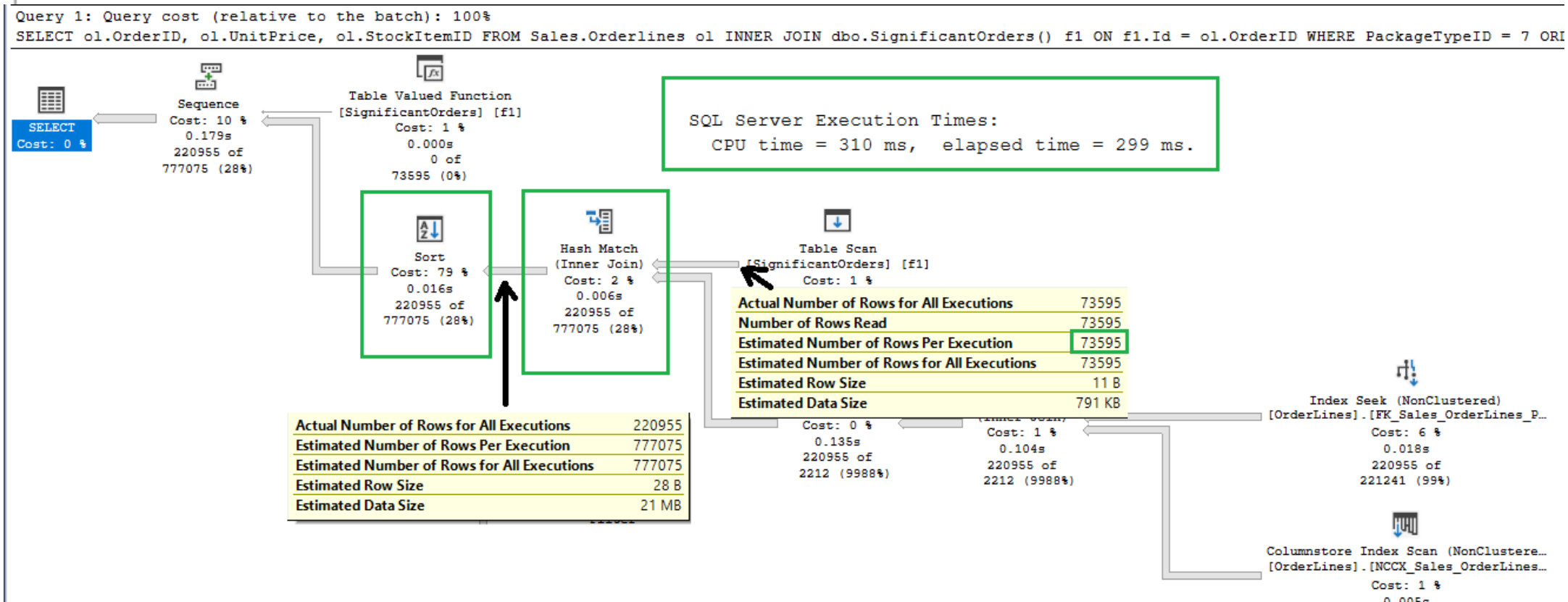
```
ALTER DATABASE WideWorldImporters SET COMPATIBILITY_LEVEL = 120;
GO
SELECT ol.OrderID, ol.UnitPrice, ol.StockItemID
FROM Sales.Orderlines ol
INNER JOIN dbo.SsignificantOrders() f1 ON f1.Id = ol.OrderID
WHERE PackageTypeID = 7 ORDER BY ol.Quantity;
```



Demo – Query under SQL Server 2017

2x

```
ALTER DATABASE WideWorldImporters SET COMPATIBILITY_LEVEL = 140;
GO
SELECT ol.OrderID, ol.UnitPrice, ol.StockItemID
FROM Sales.Orderlines ol
INNER JOIN dbo.SignificantOrders() f1 ON f1.Id = ol.OrderID
```

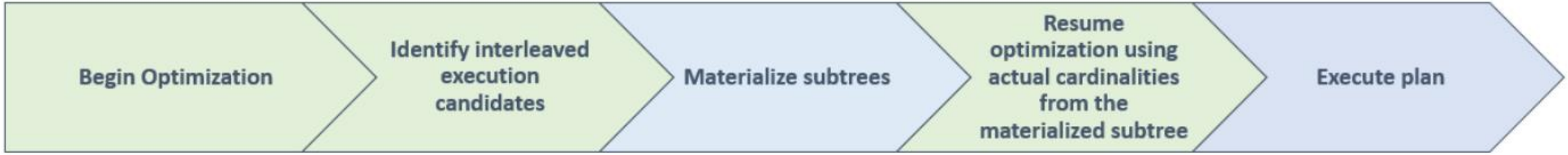


How it Works?

- This is the first time that part of the query is executed during compilation
  - Therefore, the feature can be described as revolutionary

- Improved cardinality estimation allows better selection of operators and better estimation of storage requirements

How it Works?



- If an MSTVF is called, the optimization will be interrupted
- MSTVF is invoked and the result is materialized
- The optimization process continues with the cardinality of the materialized rowset
- Execution Engine does not need to re-run the function

How to achieve ILE in previous versions?

```
SELECT ol.OrderID, ol.UnitPrice, ol.StockItemID
FROM Sales.Orderlines ol
INNER JOIN dbo.SsignificantOrders() f1 ON f1.Id = ol.OrderID WHERE PackageTypeID = 7;
```

```
DROP TABLE IF EXISTS #T;
SELECT Id INTO #T FROM dbo.SignificantOrders();

SELECT ol.OrderID, ol.UnitPrice, ol.StockItemID
FROM Sales.Orderlines ol
INNER JOIN #T f1 ON f1.Id = ol.OrderID WHERE PackageTypeID = 7;
```

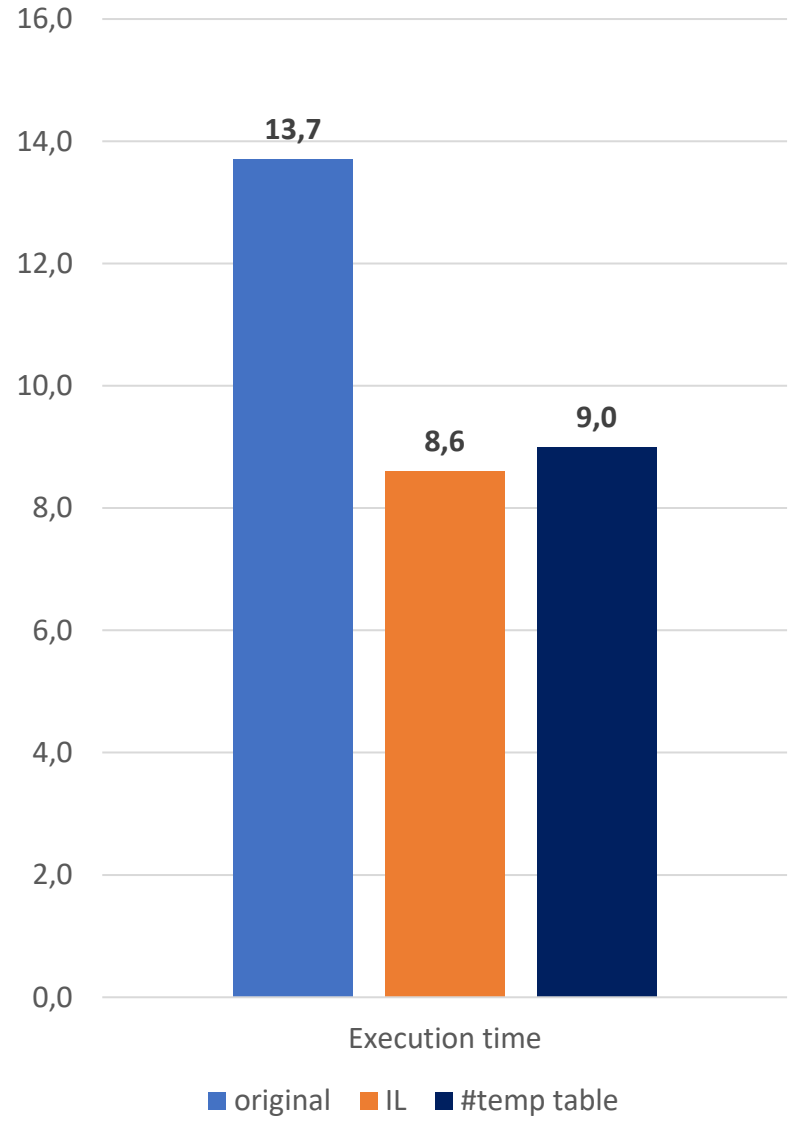
Comparing Solutions

- Query 1: original (CL 130)
- Query 2: with Interleaved Execution (CL 140)
- Query 3: Workaround with a temp table

```
C:\github\IT-Tage\Workshop\IL\ostress>cmd1

12/04/20 23:05:59.819 [0x00025768] Total IO waits: 0, Total IO wait time: 0 (ms)
12/04/20 23:05:59.819 [0x00025768] OSTRESS exiting normally, elapsed time: 00:00:13.696
12/04/20 23:05:59.820 [0x00025768] RsFx I/O completion thread ended.
```

```
C:\github\IT-Tage\Workshop\IL\ostress>cmd2

12/04/20 23:06:14.530 [0x000256C8] Total IO waits: 0, Total IO wait time: 0 (ms)
12/04/20 23:06:14.530 [0x000256C8] OSTRESS exiting normally, elapsed time: 00:00:08.574
12/04/20 23:06:14.531 [0x000256C8] RsFx I/O completion thread ended.
```

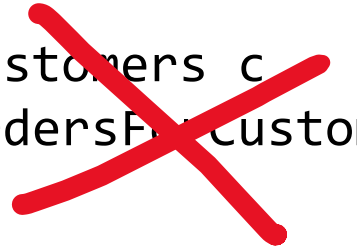
```
C:\github\IT-Tage\Workshop\IL\ostress>cmd3

12/04/20 23:06:30.607 [0x0001D14C] Total IO waits: 0, Total IO wait time: 0 (ms)
12/04/20 23:06:30.607 [0x0001D14C] OSTRESS exiting normally, elapsed time: 00:00:09.044
12/04/20 23:06:30.608 [0x0001D14C] RsFx I/O completion thread ended.
```



Interleaved Execution - Limitations

- It works only for functions with fixed arguments (or no arguments)

```
CREATE OR ALTER FUNCTION dbo.Top2OrdersForCustomer(@CustomerId INT)
RETURNS @T TABLE
(ID    INT  NOT NULL)
AS
BEGIN
    INSERT INTO @T SELECT TOP (2) OrderId FROM Sales.Orders WHERE
    CustomerID = @CustomerId
    RETURN
END
GO
SELECT * FROM Sales.Customers c
CROSS APPLY dbo.Top2OrdersFor Customer(c.CustomerID) x
```



Interleaved Execution - Limitations

- It works only for SELECT statements, for INSERT no ILE

```
SELECT ol.OrderID,  ol.UnitPrice, ol.StockItemID
INTO #o
FROM Sales.Orderlines ol
INNER JOIN dbo.SignificantOrders() f1 ON f1.Id = ol.OrderID WHERE PackageTypeID = 7;
```

	name	timestamp	estimated ...	actual card
▶	interleaved_exec_disabled_reason	2020-04-29 13:39:53.3569494	NULL	NULL
	interleaved_exec_status	2020-04-29 13:39:34.8877182	NULL	NULL
	interleaved_exec_stats_update	2020-04-29 13:39:34.8850858	100	73595

Event:interleaved_exec_disabled_reason (2020-04-29 13:39:53.3569494)

Details

Field	Value
disabled_reason	SelectQueryOnly
sql_text	SELECT ol.OrderID, ol.UnitPrice, ol.StockItemID into #o FROM Sales.Orderlines ol INNER JOIN dbo.SignificantOr...

Interleaved Execution - Conclusion

- Useful feature but with very limited scope
- It improves cardinality estimations and helps producing better execution plans
- Only for queries that use MSTVF with fixed parameters
- Nobody will migrate because of Interleaved Execution

2

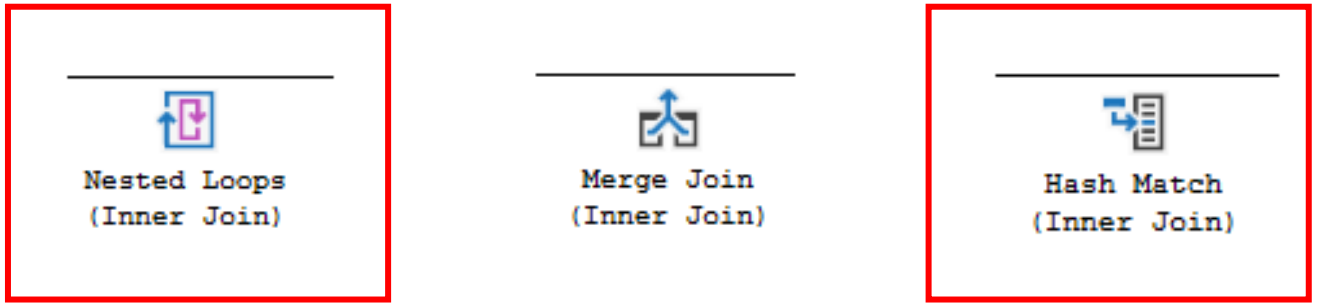
Batch Mode Adaptive Join

Implementation of a JOIN operation

- Logical operation

```
SELECT * FROM A INNER JOIN B ON
```

- Physical operators



Sample parameter sensitive stored procedure

```sql
CREATE OR ALTER PROCEDURE dbo.GetOrderDetails
    @UnitPrice DECIMAL(18,2)
AS
SELECT o.OrderID, o.OrderDate, ol.OrderLineID, ol.Quantity, ol.UnitPrice
FROM Sales.OrderLines ol
INNER JOIN Sales.Orders o ON ol.OrderID = o.OrderID
WHERE ol.UnitPrice = @UnitPrice;
```

- For parameter **112**, the stored proc returns **1.004** rows
- For parameter **36**, the stored proc returns **33** rows

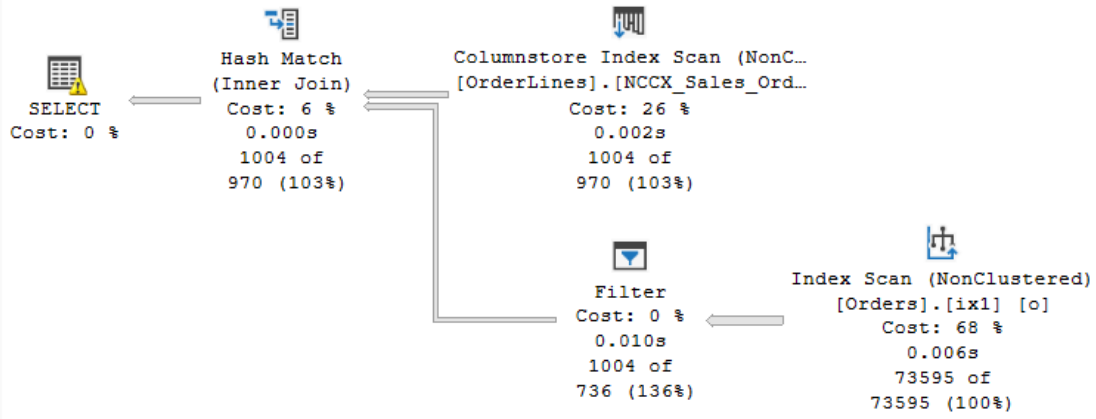
First call with param 112

```
EXEC dbo.GetOrderDetails 112;
EXEC dbo.GetOrderDetails 36;
```

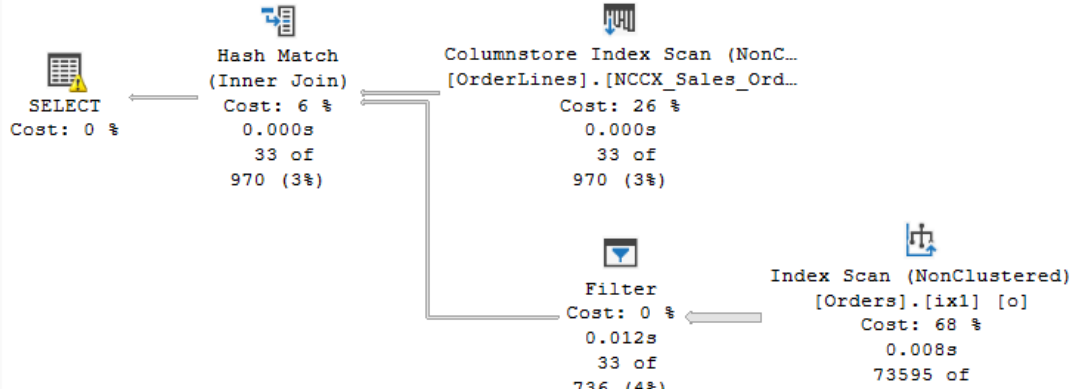
176 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 50%
SELECT o.OrderID, o.OrderDate, ol.OrderLineID, ol.Quantity, ol.UnitPrice FROM Sales.OrderLines ol INNER JOIN Sales.Orders o ON ol.OrderID = o.OrderID WHERE ol.UnitPric
Missing Index (Impact 24.092): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales].[OrderLines] ([UnitPrice]) INCLUDE ([OrderID],[Quantity])
```




```
Query 2: Query cost (relative to the batch): 50%
SELECT o.OrderID, o.OrderDate, ol.OrderLineID, ol.Quantity, ol.UnitPrice FROM Sales.OrderLines ol INNER JOIN Sales.Orders o ON ol.OrderID = o.OrderID WHERE ol.UnitPric
Missing Index (Impact 24.092): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales].[OrderLines] ([UnitPrice]) INCLUDE ([OrderID],[Quantity])
```



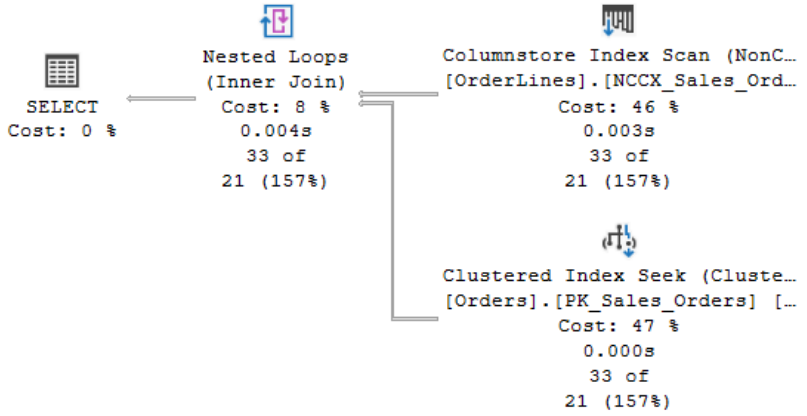
First call with param 36

```
EXEC dbo.GetOrderDetails 36;
EXEC dbo.GetOrderDetails 112;
```

176 %

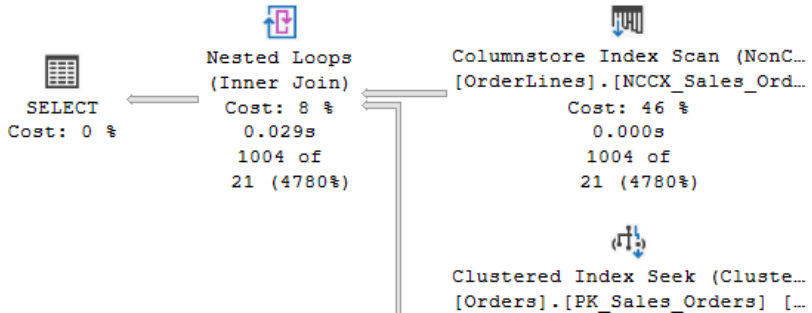
Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 50%
SELECT o.OrderID, o.OrderDate, ol.OrderLineID, ol.Quantity, ol.UnitPrice FROM Sales.OrderLines ol INNER JOIN Sales.Orders o ON ol.OrderID = o.OrderID WHERE o
Missing Index (Impact 42.8516): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales].[OrderLines] ([UnitPrice]) INCLUDE ([OrderID],[Quant
```



Bad Parameter Sniffing!

```
Query 2: Query cost (relative to the batch): 50%
SELECT o.OrderID, o.OrderDate, ol.OrderLineID, ol.Quantity, ol.UnitPrice FROM Sales.OrderLines ol INNER JOIN Sales.Orders o ON ol.OrderID = o.OrderID WHERE o
Missing Index (Impact 42.8516): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales].[OrderLines] ([UnitPrice]) INCLUDE ([OrderID],[Quant
```



Can we have both Joins in a single plan?

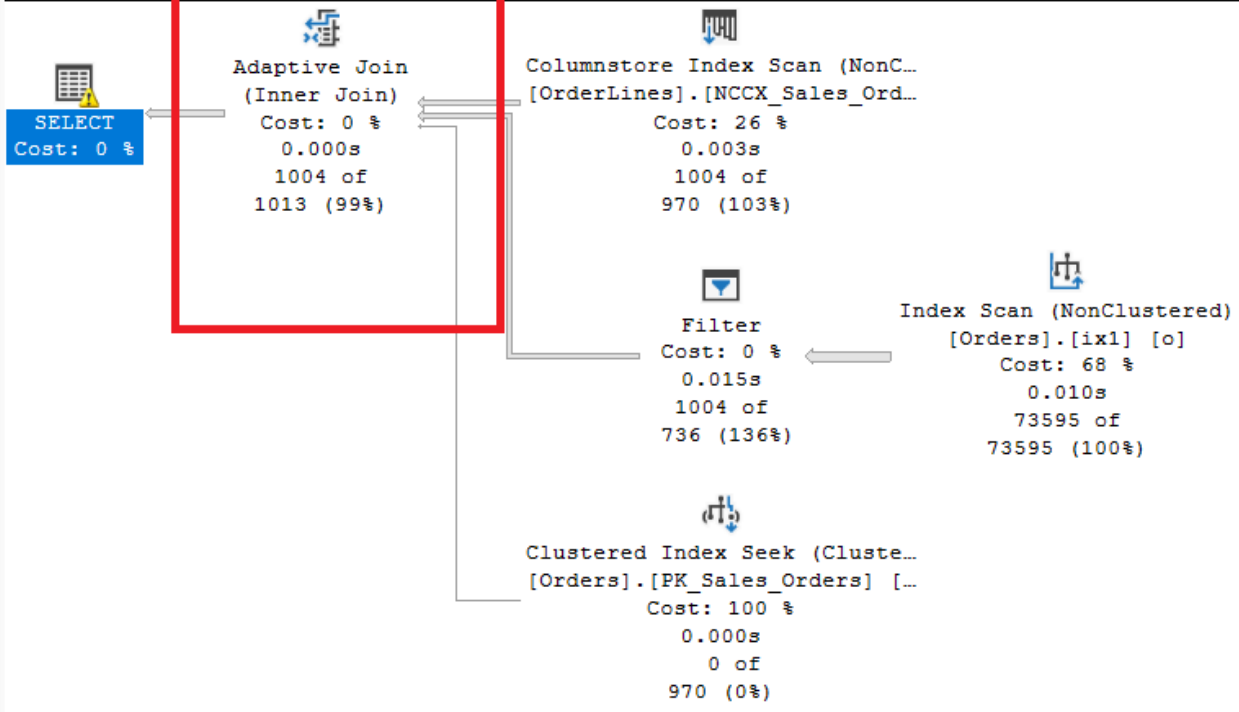
- No, but we can have both in a single operator – Adaptive Join!

```
EXEC dbo.GetOrderDetails 112;
GO
```

176 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT o.OrderID, o.OrderDate, ol.OrderLineID, ol.Quantity, ol.UnitPrice FROM Sales.OrderLines ol INNER JOIN Sales.Orders o ON ol.OrderID = o.OrderID W
Missing Index (Impact 24.0916): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales].[OrderLines] ([UnitPrice]) INCLUDE ([OrderID],
```



Batch Mode Adaptive Join

- New operator - **Adaptive Join**
- It has both Hash Join and Nested Loops Join operators in it
- It allows to choose between Hash Join and Nested Loop Join at runtime
- It starts as Hash Join and switches to Nested Loop Join when number of rows < Threshold

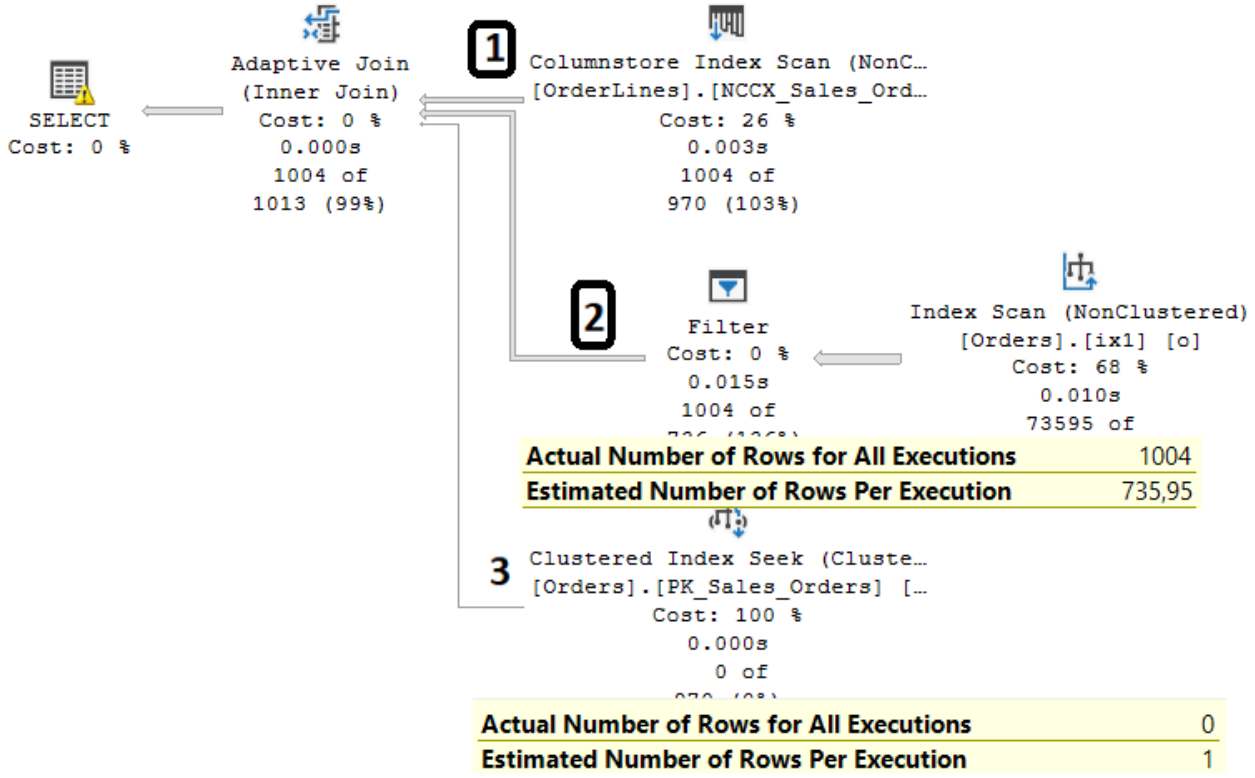
Batch Mode Adaptive Join

```
EXEC dbo.GetOrderDetails 112;
GO
```

176 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT o.OrderID, o.OrderDate, ol.OrderLineID, ol.Quantity, ol.UnitPrice FROM Sales.OrderLines ol INNER JOIN Sales.Orders o ON ol.OrderID = o.Order
Missing Index (Impact 24.0916): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales].[OrderLines] ([UnitPrice]) INCLUDE ([Order
```



Adaptive Join

Chooses dynamically between hash join and nested loops.

Physical Operation	Adaptive Join
Logical Operation	Inner Join
Actual Join Type	HashMatch
Actual Execution Mode	Batch
Adaptive Threshold Rows	97,309
Is Adaptive	True
Estimated Execution Mode	Batch
Estimated Join Type	HashMatch
Actual Number of Rows for All Executions	1004

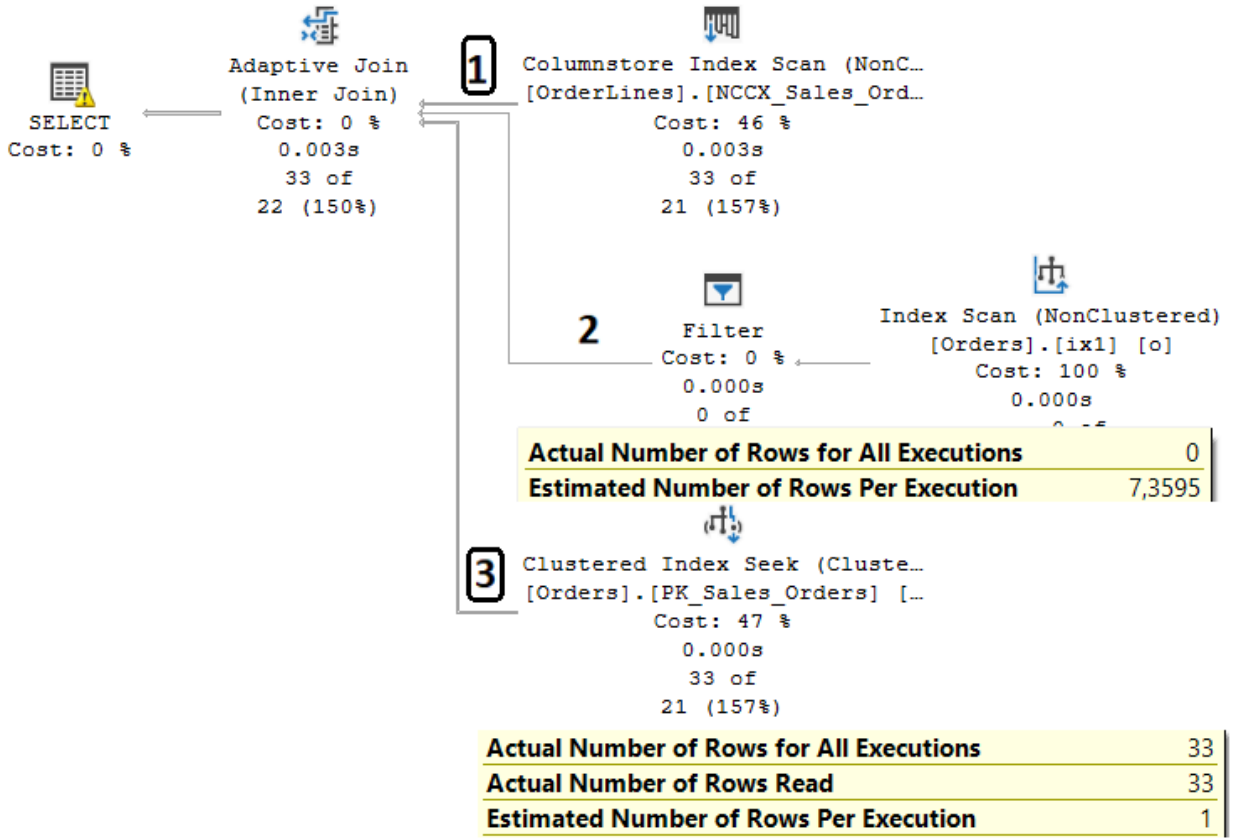
Batch Mode Adaptive Join

```
EXEC dbo.GetOrderDetails 36;
GO
```

176 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT o.OrderID, o.OrderDate, ol.OrderLineID, ol.Quantity, ol.UnitPrice FROM Sales.OrderLines ol INNER JOIN Sales.Orders o ON ol.OrderID = o.Orde
Missing Index (Impact 42.8394): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales].[OrderLines] ([UnitPrice]) INCLUDE ([Orde
```



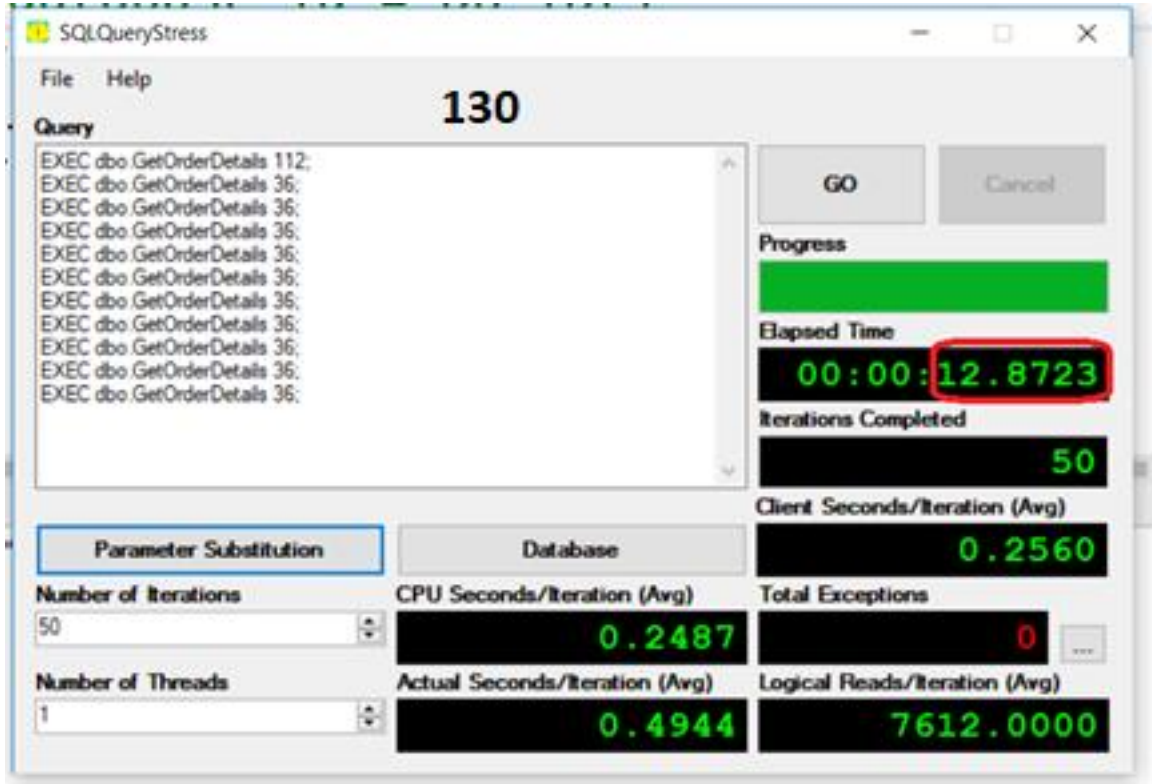
Adaptive Join

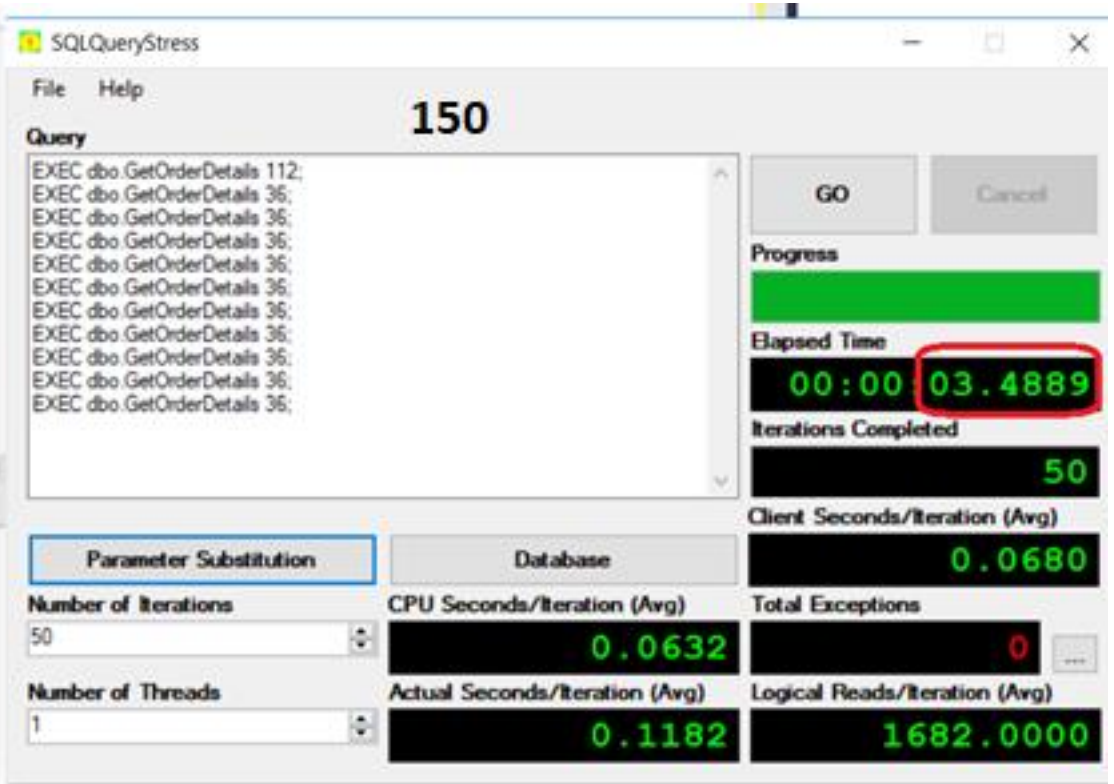
Chooses dynamically between hash join and nested loops.

Physical Operation	Adaptive Join
Logical Operation	Inner Join
Actual Join Type	NestedLoops
Actual Execution Mode	Row
Estimated Join Type	NestedLoops
Is Adaptive	True
Estimated Execution Mode	Row
Adaptive Threshold Rows	56,6024
Actual Number of Rows for All Executions	33

Batch Mode Adaptive Join

Q: Is it better with new Adaptive Join operator?



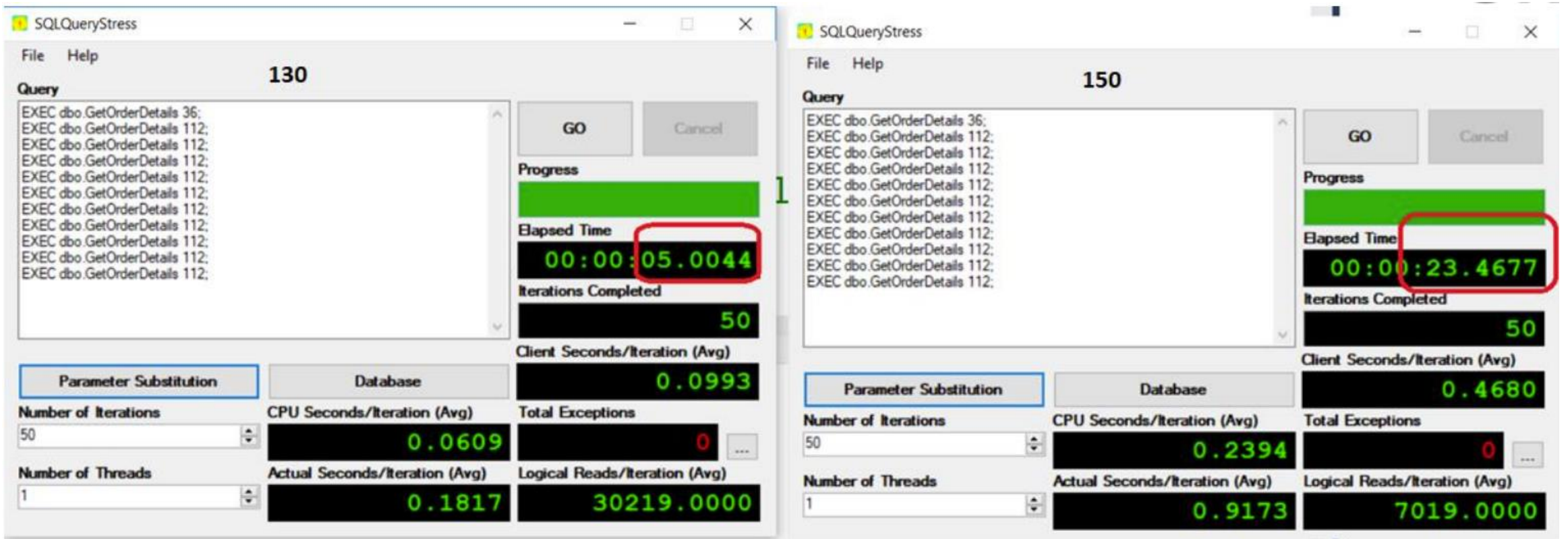


A: Yes!!! Under the CL 150, the query runs 4x faster!

Batch Mode Adaptive Join

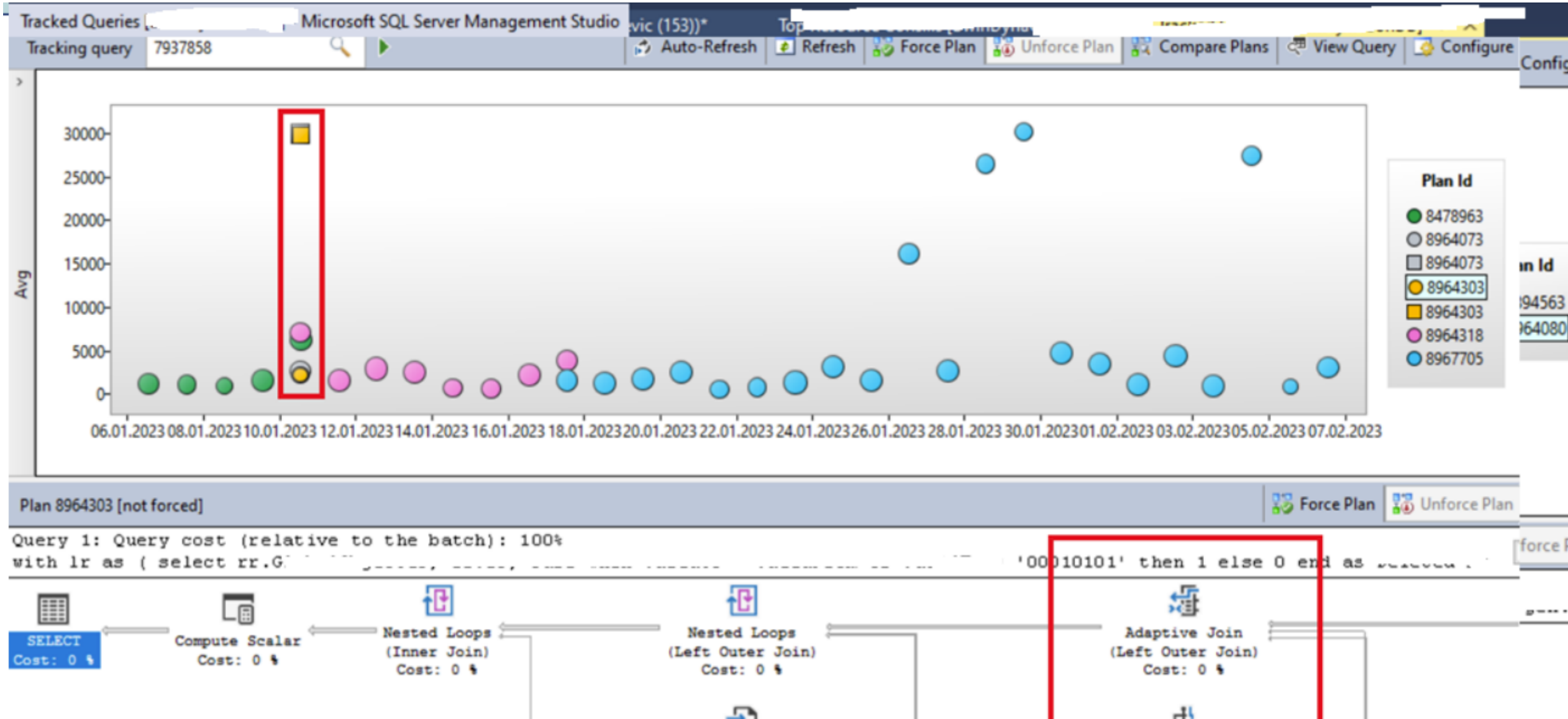
IT DEPENDS!

Q: Is it better with new Adaptive Join operator?



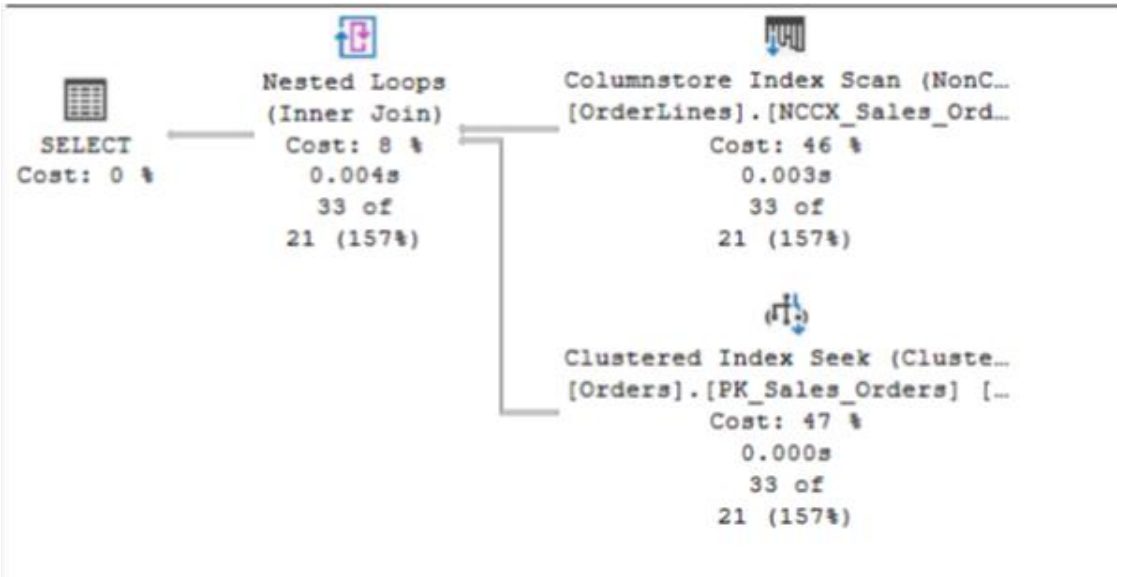
A: Actually NO - under the CL 150, the query runs 5x slower!

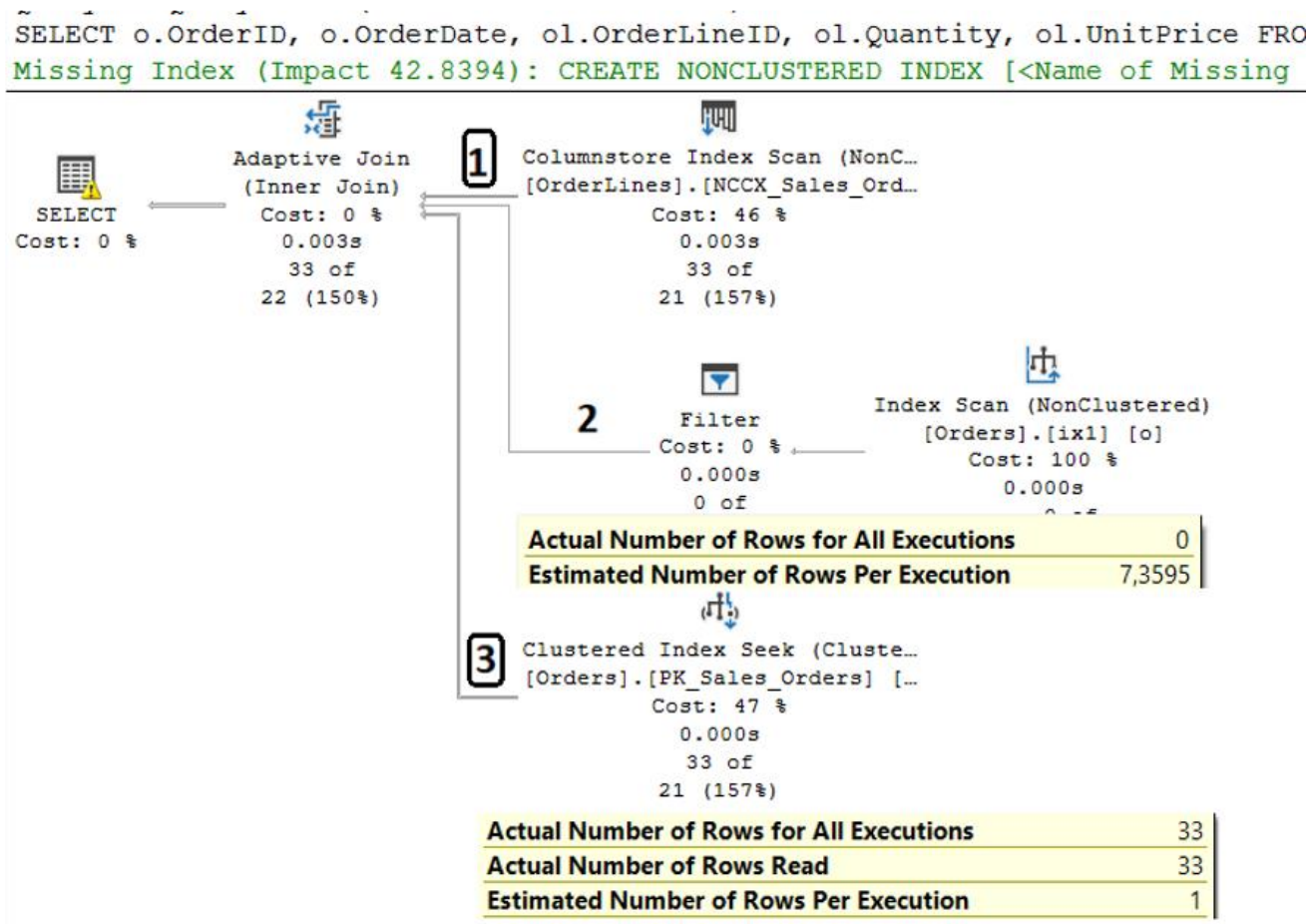
Regressions from prod. system



Batch Mode Adaptive Join - Summary

- Nested Loops Join is significantly cheaper than Adaptive Join acting as Nested Loops





Batch Mode Adaptive Join - Conclusion

- It could be beneficial for some workloads to reduce Parameter Sniffing issues for sensitive queries
- It could be good for some databases, but very bad for other
- You must test, test and test
- You might even disable it for some databases completely
  - especially for pure OLTP workloads
- It works in batch mode only
- Enterprise feature

3

Memory Grant Feedback

Memory Grant in SQL Server

- Significantly inaccurate estimates + Sort or Hash operators in the execution plan

OVERESTIMATION

- Unnecessarily large memory grants
- waste of memory
- RESOURCE_SEMAPHORE waits in high-concurrency workloads

UNDERESTIMATION

- Insufficient memory grants
- Tempdb spills
- Slow execution

Memory Grant Feedback

- Adjusts the memory grant parameter in the execution plan AFTER creating the plan (after a few executions)
- Memory is recalculated when:
  - The query is using less than 50% allocated memory
  - Has come to tempdb spills
- Batch mode (SQL Server 2017)
  - Batch mode operators required - aka Columnstore Index
- Row mode (SQL Server 2019)
- Enterprise Edition feature

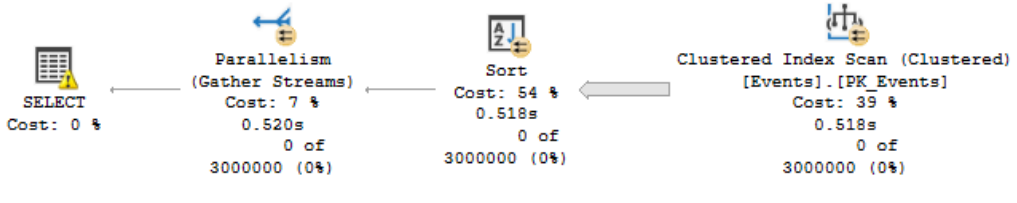
MGF Example

- A stored procedure with constant memory overestimation (by design)

```
CREATE OR ALTER PROCEDURE dbo.GetEvents
@EventDate DATETIME
AS
BEGIN
    DECLARE @now DATETIME = @EventDate;
    SELECT * FROM dbo.Events
    WHERE EventDate >= @now
    ORDER BY Note DESC;
END
```

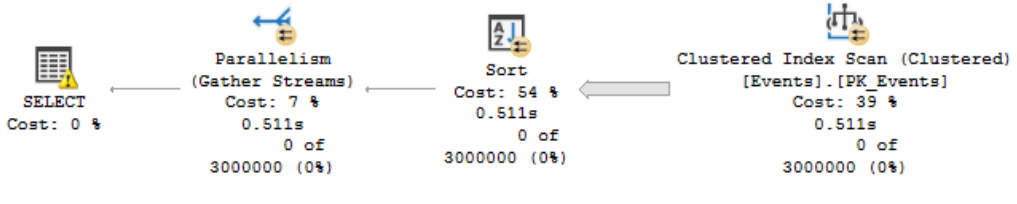
```
--ensure that the database runs under CL 130 (SQL Server 2016)
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 130;
GO
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
EXEC dbo.GetEvents '20180101';
GO 3
```

Query 1: Query cost (relative to the batch): 33%
SELECT * FROM dbo.Events WHERE EventDate >= @now ORDER BY Note DESC
Missing Index (Impact 26.1061): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Event



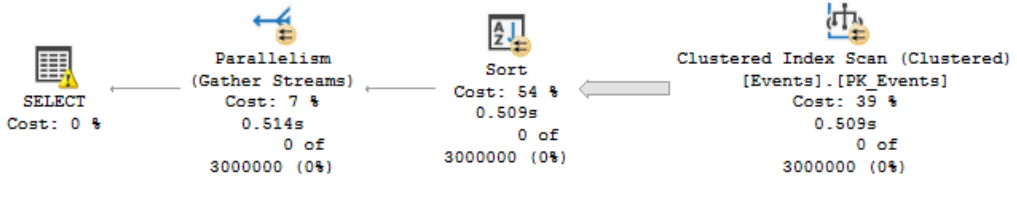
SELECT	
Cached plan size	24 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	8
Estimated Subtree Cost	329.792
Memory Grant	591424
Estimated Number of Rows	3000000

Query 2: Query cost (relative to the batch): 33%
SELECT * FROM dbo.Events WHERE EventDate >= @now ORDER BY Note DESC
Missing Index (Impact 26.1061): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Event



SELECT	
Cached plan size	24 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	8
Estimated Subtree Cost	329.792
Memory Grant	591424
Estimated Number of Rows	3000000

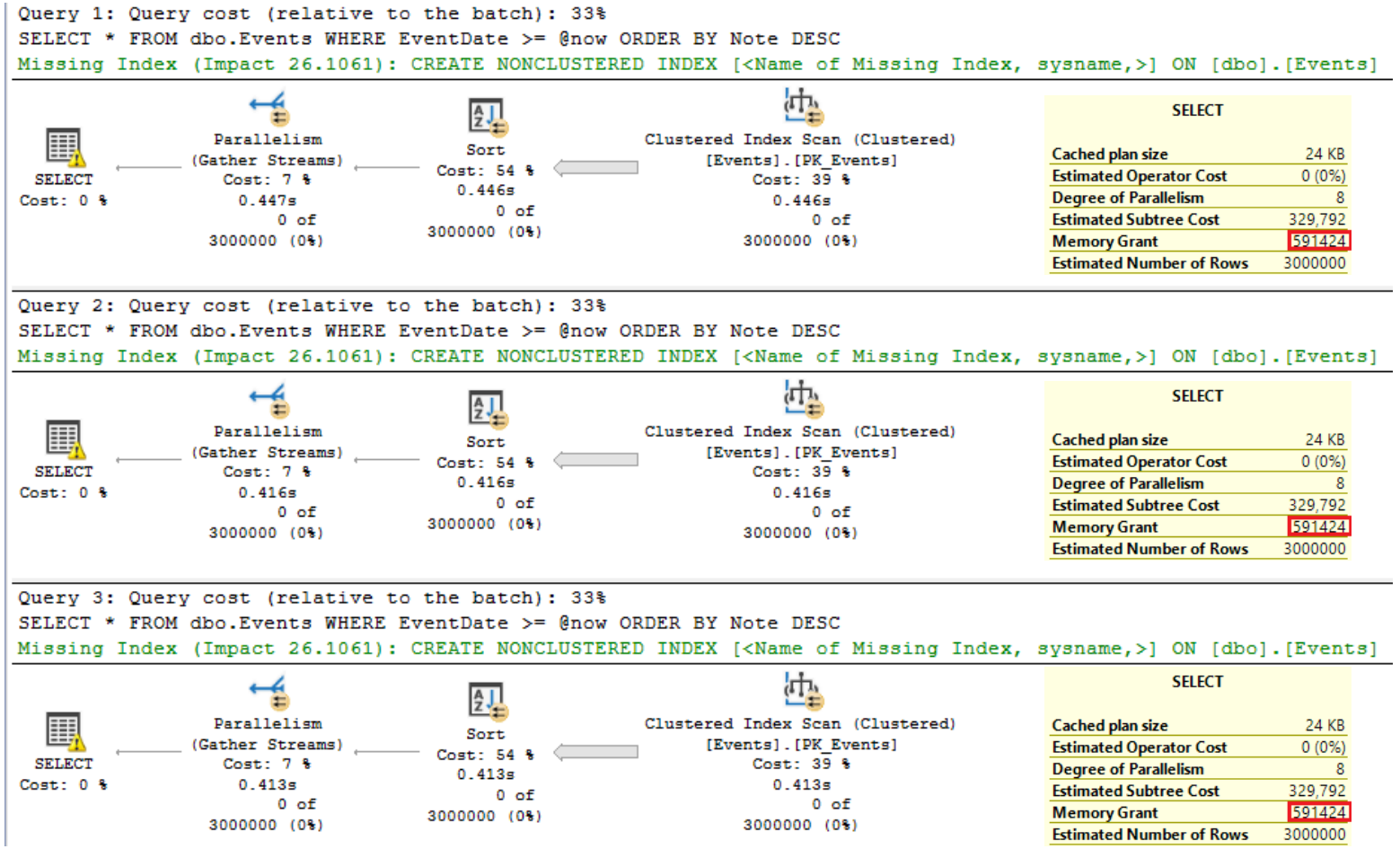
Query 3: Query cost (relative to the batch): 33%
SELECT * FROM dbo.Events WHERE EventDate >= @now ORDER BY Note DESC
Missing Index (Impact 26.1061): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Event



SELECT	
Cached plan size	24 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	8
Estimated Subtree Cost	329.792
Memory Grant	591424
Estimated Number of Rows	3000000

The same memory grant

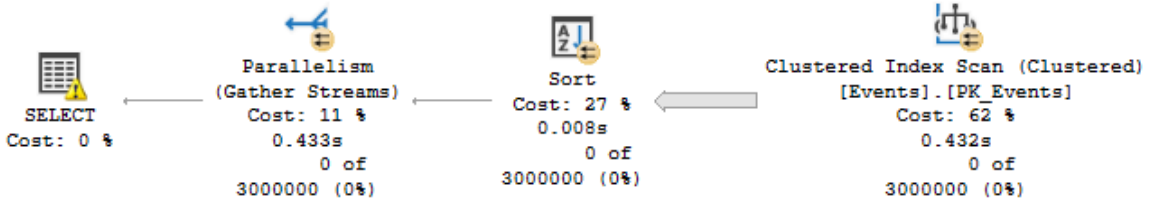
```
--ensure that the database runs under CL 140 (SQL Server 2017)
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 140;
GO
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
EXEC dbo.GetEvents '20180101';
GO 3
```



Batch mode Memory Grant
in SQL Server 2017
requires a columnstore index

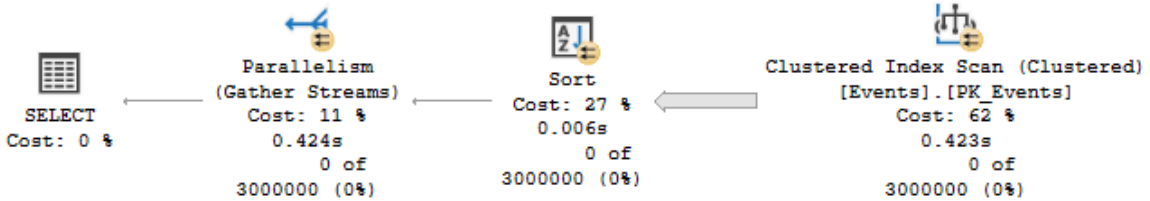
```
CREATE NONCLUSTERED COLUMNSTORE INDEX ixc ON dbo.Events(id, EventType,EventDate, Note) WHERE id  = -4;
GO
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
EXEC dbo.GetEvents '20180101';
GO 3
```

```
Query 1: Query cost (relative to the batch): 33%
SELECT * FROM dbo.Events WHERE EventDate >= @now ORDER BY Note DESC
Missing Index (Impact 41.6303): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Events] ([Ev
```



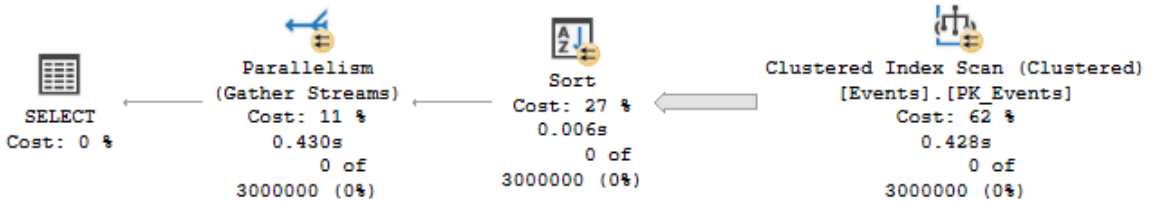
SELECT	
Cached plan size	24 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	8
Estimated Subtree Cost	206.811
Memory Grant	670408
Estimated Number of Rows	3000000

```
Query 2: Query cost (relative to the batch): 33%
SELECT * FROM dbo.Events WHERE EventDate >= @now ORDER BY Note DESC
Missing Index (Impact 41.6303): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Events] ([Ev
```



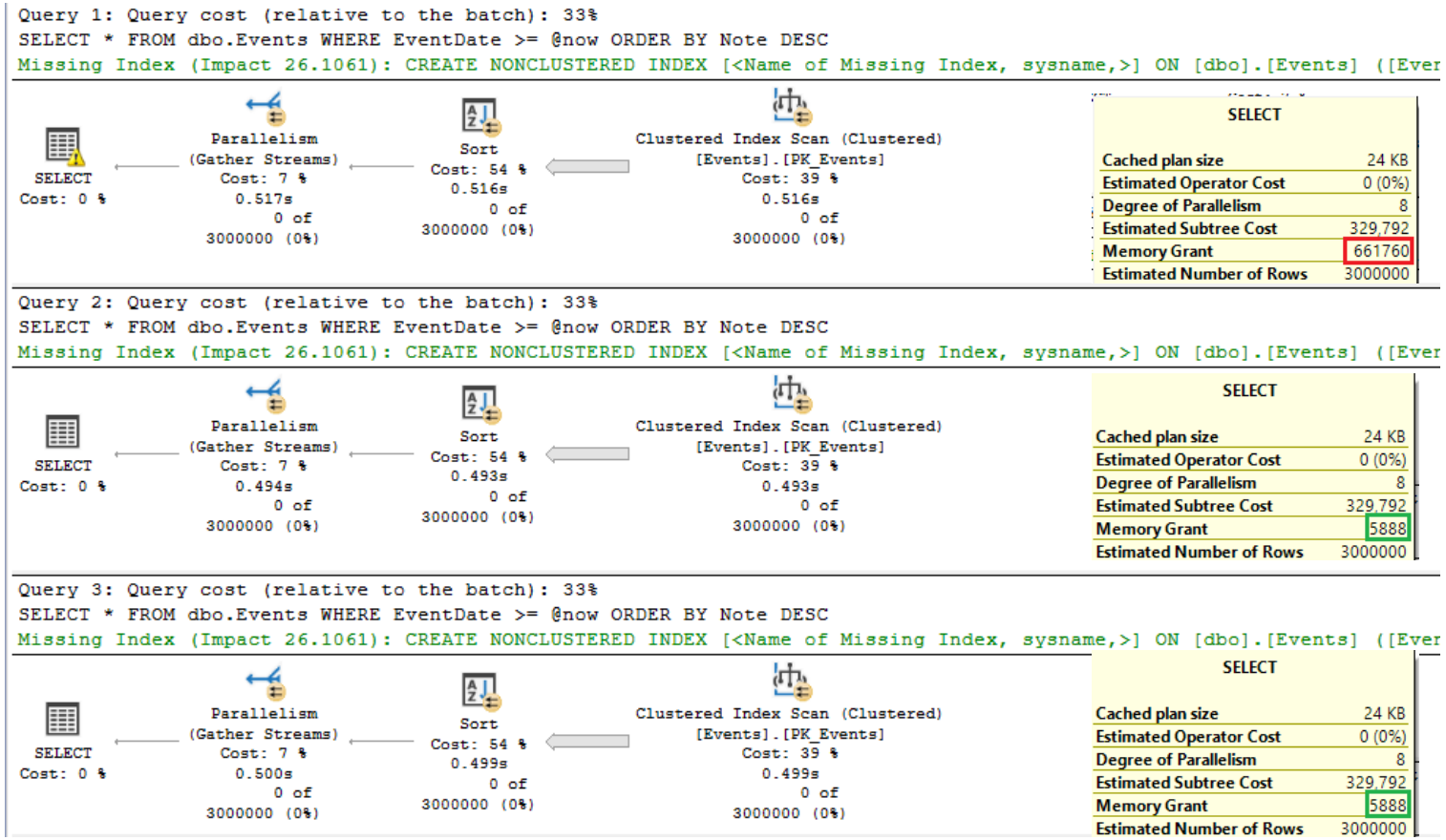
SELECT	
Cached plan size	24 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	8
Estimated Subtree Cost	206.811
Memory Grant	2440
Estimated Number of Rows	3000000

```
Query 3: Query cost (relative to the batch): 33%
SELECT * FROM dbo.Events WHERE EventDate >= @now ORDER BY Note DESC
Missing Index (Impact 41.6303): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Events] ([Ev
```



SELECT	
Cached plan size	24 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	8
Estimated Subtree Cost	206.811
Memory Grant	2440
Estimated Number of Rows	3000000

```
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 150;
GO
--remove the columnstore index
DROP INDEX ixc ON dbo.Events;
GO
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
EXEC dbo.GetEvents '20180101';
GO 3
```



Row mode Memory Grant
in SQL Server 2019
No columnstore index required

Issues with Memory Grant in SQL Server

- It works only with cached plans and memory grants > 1MB
- It does not work with OPTION (RECOMPILE)
- It is not persisted if the plan is removed from cache
- If memory granted value oscillates, the feature is auto-disabled

Issues with Memory Grant in SQL Server

- It works only with cached plans and memory grants > 1MB
- It does not work with OPTION (RECOMPILE)
- It is not persisted if the plan is removed from cache
- If memory granted value oscillates, the feature is auto-disabled

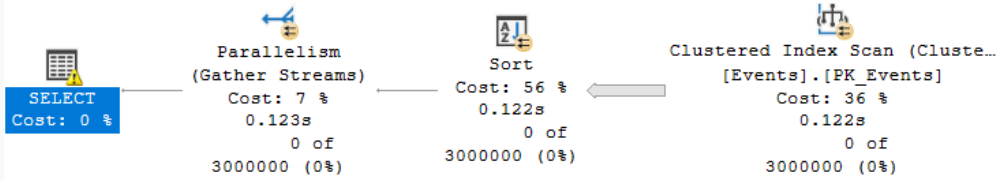
Memory Grant Feedback Disabled

```
EXEC dbo.GetEvents '20230101';
EXEC dbo.GetEvents '20180101';
GO 4
```

176 %

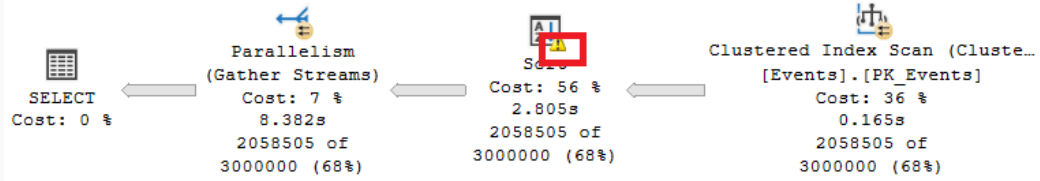
Messages | Execution plan

Query 1: Query cost (relative to the batch): 12%
SELECT * FROM dbo.Events WHERE EventDate >= @now ORDER BY Note DESC
Missing Index (Impact 22.9925): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysnam DE ([EventType],[Note])



SELECT	
Cached plan size	24 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	8
Estimated Subtree Cost	316,539
Memory Grant	1013 MB

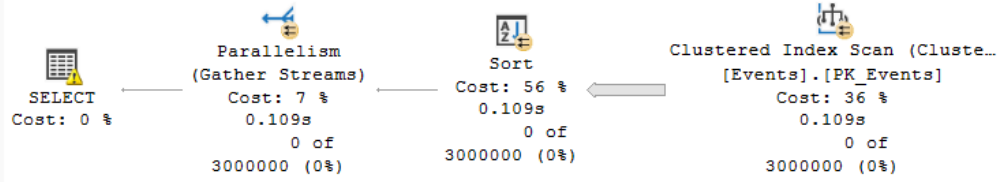
Query 2: Query cost (relative to the batch): 12%
SELECT * FROM dbo.Events WHERE EventDate >= @now ORDER BY Note DESC
Missing Index (Impact 22.9925): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname DE ([EventType],[Note])



SELECT	
Cached plan size	24 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	8
Estimated Subtree Cost	316,539
Memory Grant	5888 KB

Warnings
Operator used tempdb to spill data during execution with spill level 2 and 8 spilled thread(s); Sort wrote 85856 pages to and read 85856 pages from tempdb with granted memory 5120KB and used memory 5120KB

Query 3: Query cost (relative to the batch): 12%
SELECT * FROM dbo.Events WHERE EventDate >= @now ORDER BY Note DESC
Missing Index (Impact 22.9925): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] [EventType],[Note])



SELECT	
Cached plan size	24 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	8
Estimated Subtree Cost	316,539
Memory Grant	1012 MB

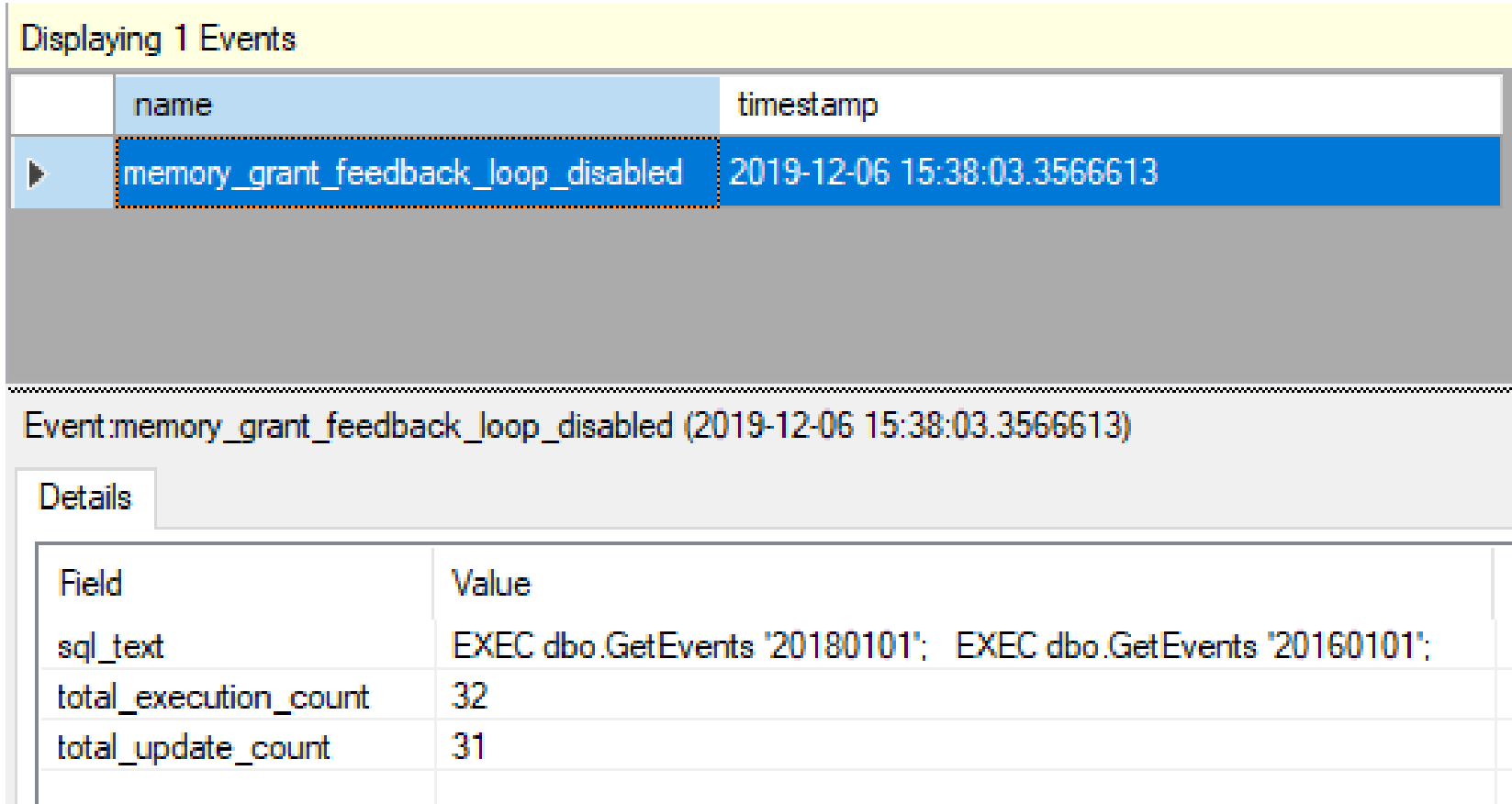
SELECT	
Cached plan size	24 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	8
Estimated Subtree Cost	316,539
Memory Grant	5888 KB

Query 4: Query cost (relative to the batch): 12%
SELECT * FROM dbo.Events WHERE EventDate >= @now ORDER BY Note DESC
Missing Index (Impact 22.9925): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Events] ([EventDate]) INCLUDE ([E

Memory Grant Feedback Disabled

- If memory grant memory values oscillates, the feature is disabled

```
EXEC dbo.GetEvents '20180101';
EXEC dbo.GetEvents '20160101';
GO 30
```



Memory Grant Feedback in SQL Server 2022

- SQL Server 2022 MGF Percentiles
  - addresses the “optimized for the last execution” issue

- SQL Server 2022 MGF Persistence
  - Makes MGF resistant to plan cache eviction

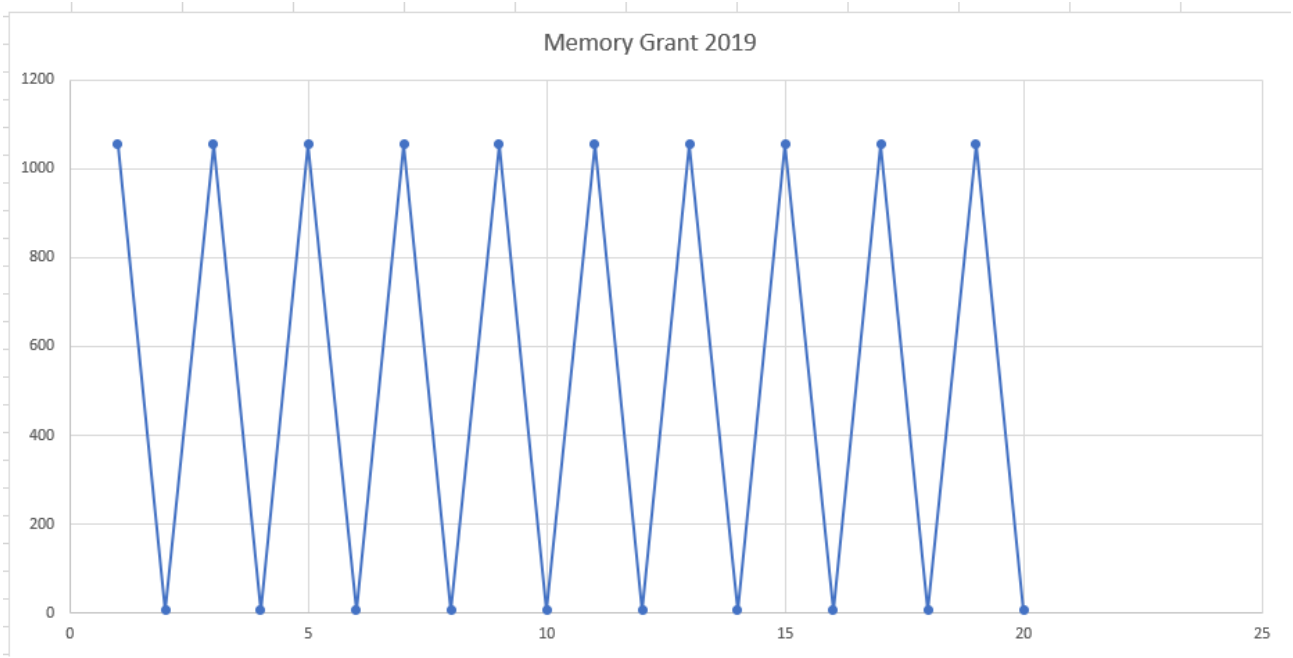
SQL Server 2022 MGF Percentiles

- It solves the problem with oscillations in memory requests

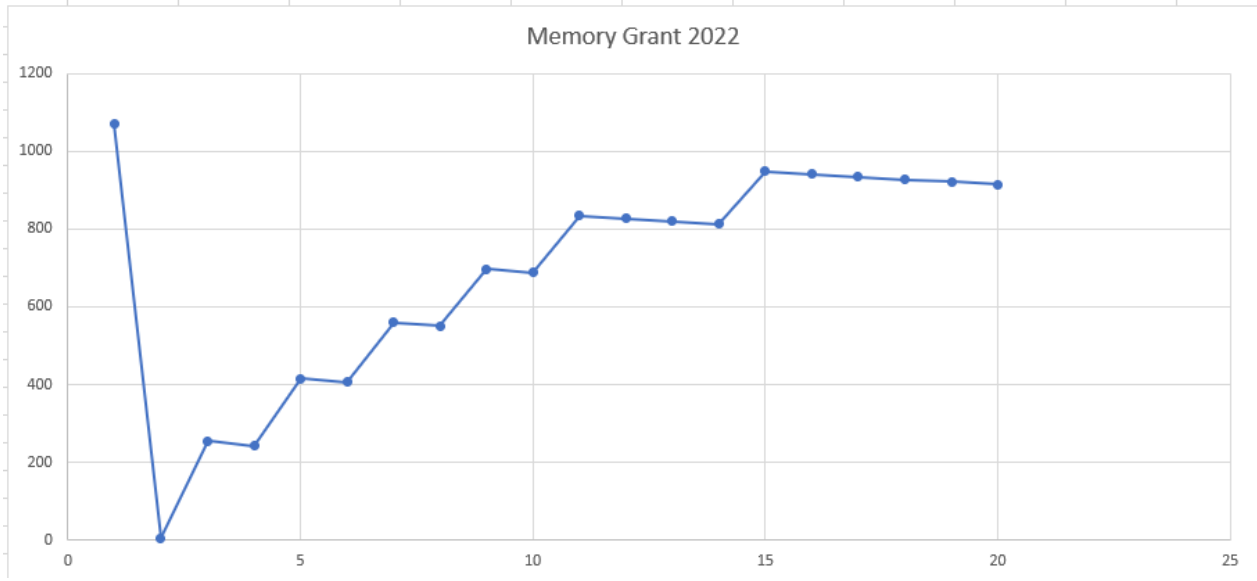
- In SQL Server 2019, it ended with disabling MGF for that plan
  - Several recent calls are considered for memory grant recalc (and not only the last call in the previous versions)
  - more memory will be allocated to avoid tempdb spills

SQL Server 2022 MGF Percentiles

- MGF in SQL Server 2019



- MGF in SQL Server 2022



SQL Server 2022 MGF Persistence

- Memory Grant Feedback is stored in the database and persists server crashes, failovers and other actions which remove cached plans

- Requires Query Store to be enabled

Memory Grant Feedback - Conclusion

- Excellent feature to deal with queries or SPs that constantly and consistently over- or underestimate memory

- Not so good for SPs having significantly different memory requirements for each execution
  - It looks better in SQL Server 2022

- Overhead of checking and recalculating MG is acceptable
  - Tested with SPs which are called 1000x per second

Memory Grant Feedback - Summary

- Potential issues
  - Additional dealing with parameter sniffing caused by memory might be needed

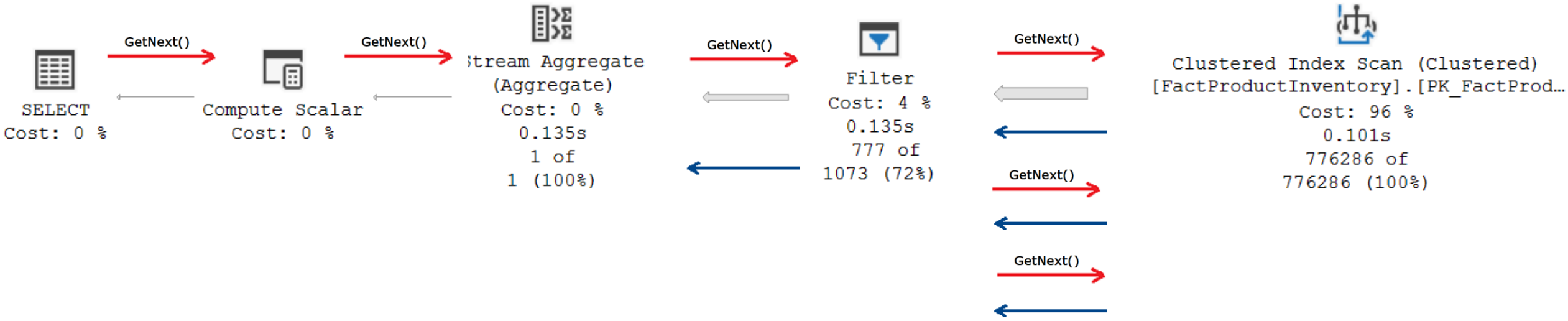
- Enterprise Edition feature
- For enhancements in SQL Server 2022 Query Store is required

- I have enabled it literally in all databases and did not have regressions (stored procs based workloads, a few hundreds calls per second, 30TB+ db size)

4

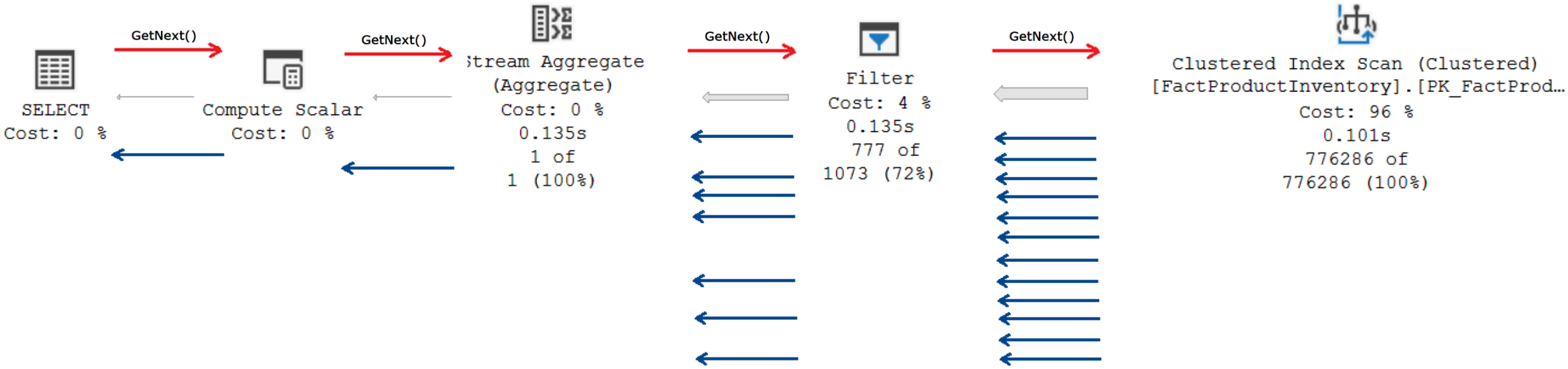
Batch Mode on RowStore

ROW Mode



Inefficient; the same instructions for every row, overhead of giving control to another operator and taking it back

BATCH Mode



batch of rows as working unit: up to 900 rows, depends on the number and size of columns and CPU L2 cache

What is Batch mode?

- Batch mode allows query operators to work on a batch of rows, instead of just one row at a time

- At the CPU level multiple rows processed at once instead of one row
- Number of processing instructions reduced

- Better CPU cache utilization and increased memory throughput

- Not exactly documented how to get the number of rows in batch (900 in all my tests)

- Can be beneficial for queries that are CPU bound

Batch Mode on Columnstore/Rowstore

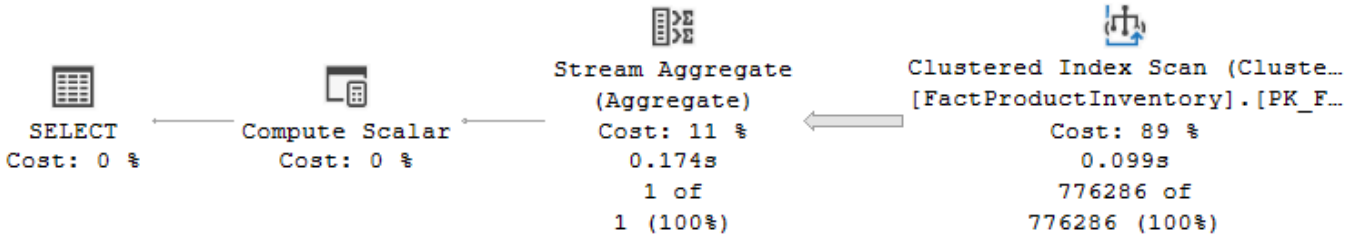
- Batch Mode on columstore introduced in SQL Server 2012
  - Improvements – up to 20x faster queries!

- Batch Mode on rowstore introduced in SQL Server 2019
  - Some queries could be significantly faster
  - In my examples 2-5x faster!

```
USE AdventureWorksDW2019;
GO
SET NOCOUNT ON SET STATISTICS TIME ON;
GO
SELECT COUNT(*), MAX(UnitsIn) FROM dbo.FactProductInventory OPTION (USE HINT ('QUERY_OPTIMIZER_COMPATIBILITY_LEVEL_140'));
GO
SELECT COUNT(*), MAX(UnitsIn) FROM dbo.FactProductInventory;
GO
```

Query 1: Query cost (relative to the batch): 53%

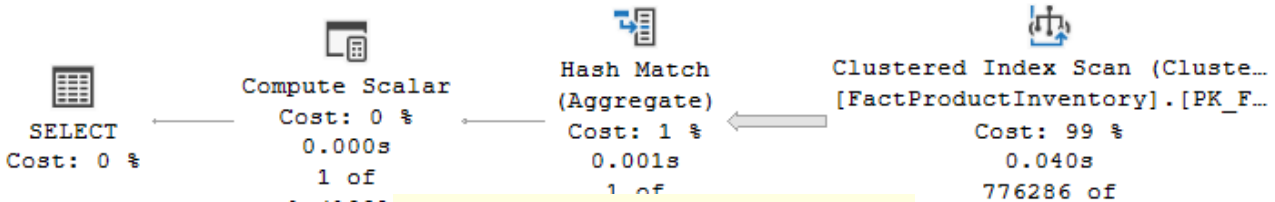
```
SELECT COUNT(*), MAX(UnitsIn) FROM dbo.FactProductInventory OPTION (USE HINT ('QUERY_OPTIMIZER_COMPATIBILITY_LEVEL_140'))
```



SQL Server Execution Times:
CPU time = 219 ms, **elapsed time = 210 ms.**

Query 2: Query cost (relative to the batch): 47%

```
SELECT COUNT(*), MAX(UnitsIn) FROM dbo.FactProductInventory
```



SQL Server Execution Times:
CPU time = 46 ms, **elapsed time = 50 ms.**

Physical Operation	Hash Match
Logical Operation	Aggregate
Actual Execution Mode	Batch
Estimated Execution Mode	Batch

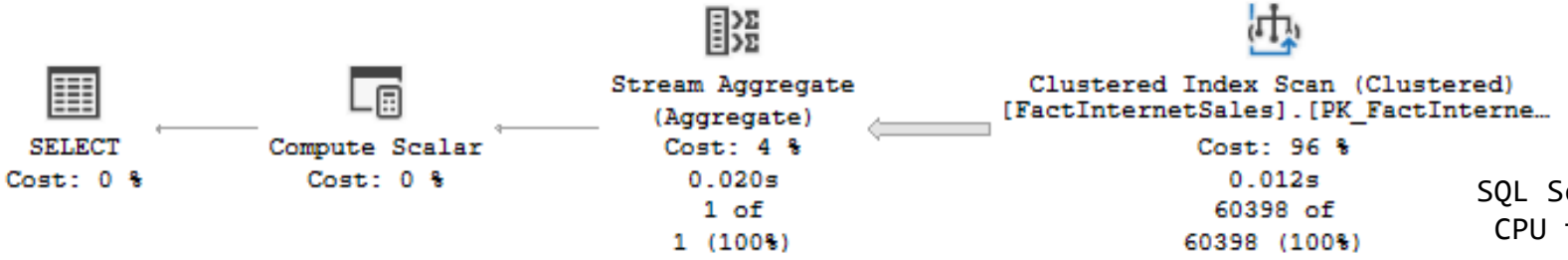
Physical Operation	Compute Scalar
Logical Operation	Compute Scalar
Actual Execution Mode	Batch
Estimated Execution Mode	Batch

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Batch
Estimated Execution Mode	Batch

4x

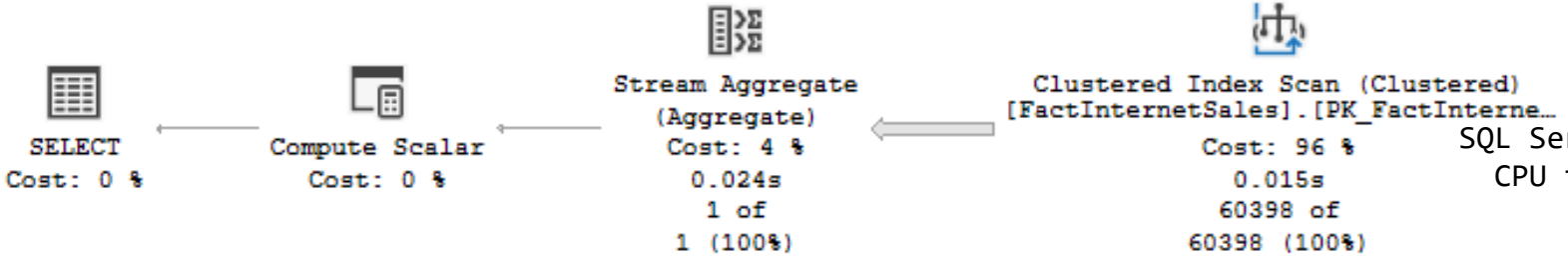
```
SELECT COUNT(*), MAX(UnitPrice) FROM dbo.FactInternetSales OPTION (USE HINT ('QUERY_OPTIMIZER_COMPATIBILITY_LEVEL_140'));
GO
SELECT COUNT(*), MAX(UnitPrice) FROM dbo.FactInternetSales;
```

Query 1: Query cost (relative to the batch): 50%
SELECT COUNT(*), MAX(UnitPrice) FROM dbo.FactInternetSales OPTION (USE HINT ('QUERY_OPTIMIZER_COMPATIBILITY_LEVEL_140'))



SQL Server Execution Times:
CPU time = 16 ms, elapsed time = **19** ms.

Query 2: Query cost (relative to the batch): 50%
SELECT COUNT(*), MAX(UnitPrice) FROM dbo.FactInternetSales



SQL Server Execution Times:
CPU time = 16 ms, elapsed time = **18** ms.

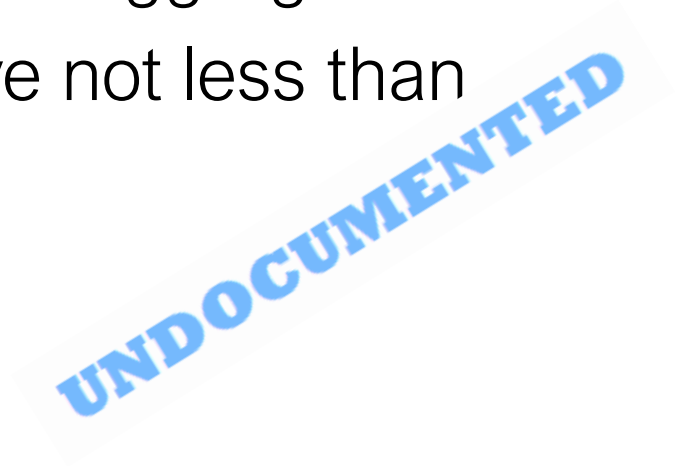
No Batch mode!

0%

Batch Mode on Rowstore

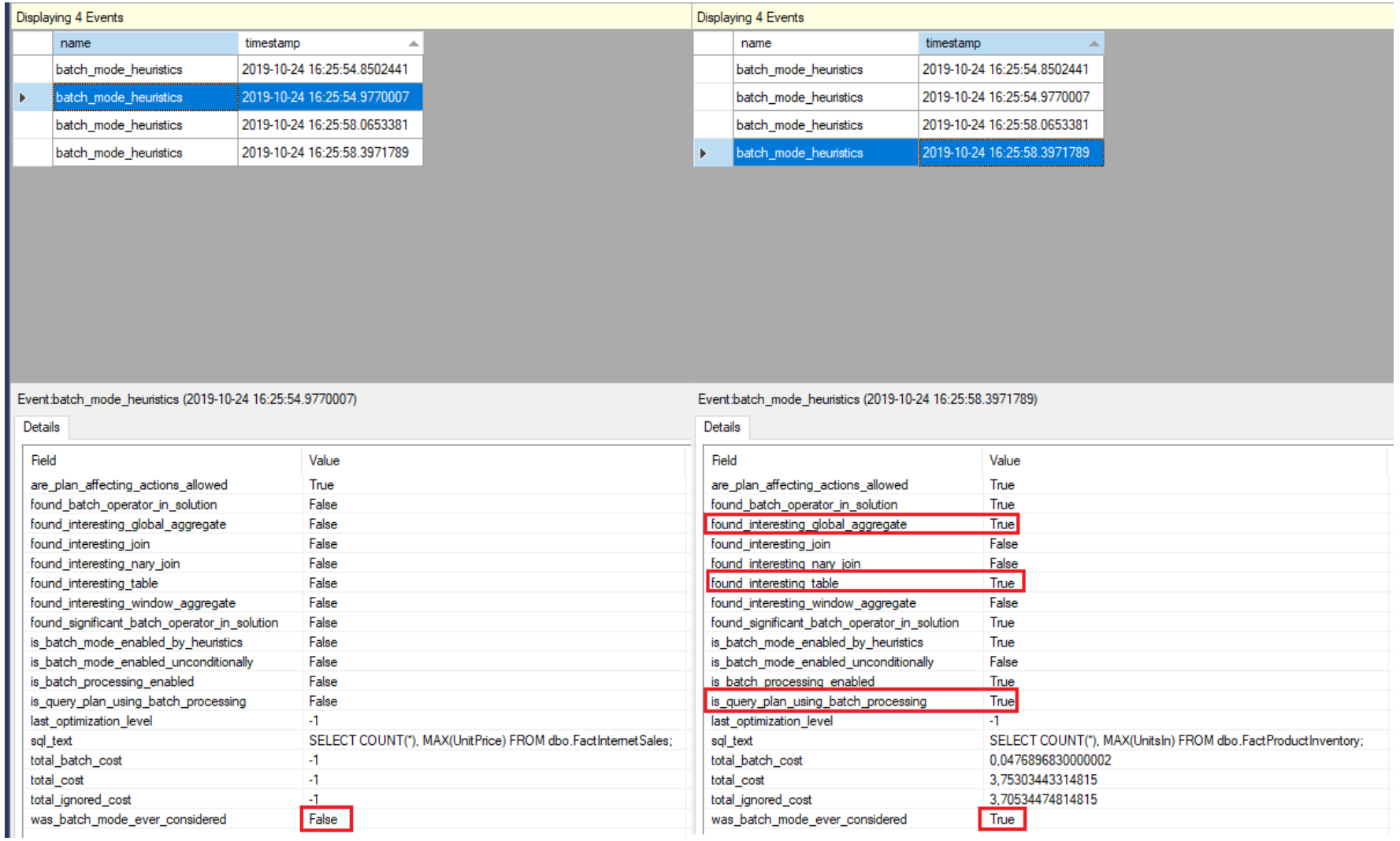
- Native support
  - No tricks with fake columnstore indexes or other optimizer delusions

- Initial heuristics considers potential benefits of batch mode for operators
- Interesting table (at least 131.072 rows)
- Interesting batch operators: join, aggregate or window aggregate
- At least one of the batch operator's input should have not less than 131.072 rows



Batch Mode on Rowstore

sqlserver.batch_mode_heuristics Extended Event



Displaying 4 Events

name	timestamp
batch_mode_heuristics	2019-10-24 16:25:54.8502441
batch_mode_heuristics	2019-10-24 16:25:54.9770007
batch_mode_heuristics	2019-10-24 16:25:58.0653381
batch_mode_heuristics	2019-10-24 16:25:58.3971789

Event:batch_mode_heuristics (2019-10-24 16:25:54.9770007)

Details

Field	Value
are_plan_affecting_actions_allowed	True
found_batch_operator_in_solution	False
found_interesting_global_aggregate	False
found_interesting_join	False
found_interesting_nary_join	False
found_interesting_table	False
found_interesting_window_aggregate	False
found_significant_batch_operator_in_solution	False
is_batch_mode_enabled_by_heuristics	False
is_batch_mode_enabled_unconditionally	False
is_batch_processing_enabled	False
is_query_plan_using_batch_processing	False
last_optimization_level	-1
sql_text	SELECT COUNT(*), MAX(UnitPrice) FROM dbo.FactInternetSales;
total_batch_cost	-1
total_cost	-1
total_ignored_cost	-1
was_batch_mode_ever_considered	False

Displaying 4 Events

name	timestamp
batch_mode_heuristics	2019-10-24 16:25:54.8502441
batch_mode_heuristics	2019-10-24 16:25:54.9770007
batch_mode_heuristics	2019-10-24 16:25:58.0653381
batch_mode_heuristics	2019-10-24 16:25:58.3971789

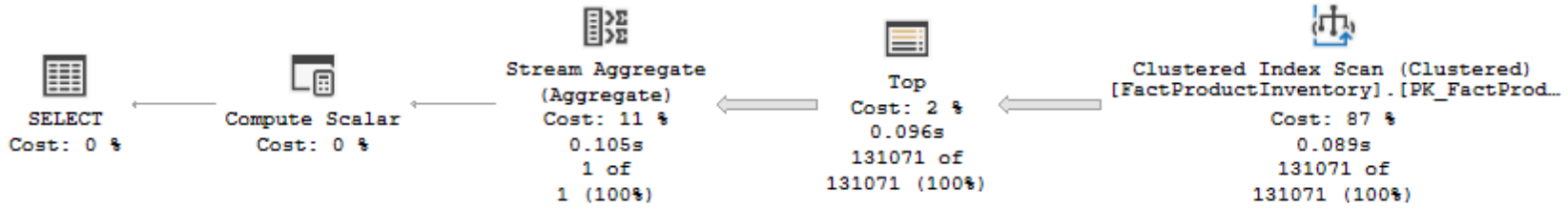
Event:batch_mode_heuristics (2019-10-24 16:25:58.3971789)

Details

Field	Value
are_plan_affecting_actions_allowed	True
found_batch_operator_in_solution	True
found_interesting_global_aggregate	True
found_interesting_join	False
found_interesting_nary_join	False
found_interesting_table	True
found_interesting_window_aggregate	False
found_significant_batch_operator_in_solution	True
is_batch_mode_enabled_by_heuristics	True
is_batch_mode_enabled_unconditionally	False
is_batch_processing_enabled	True
is_query_plan_using_batch_processing	True
last_optimization_level	-1
sql_text	SELECT COUNT(*), MAX(UnitsIn) FROM dbo.FactProductInventory;
total_batch_cost	0,0476896830000002
total_cost	3,75303443314815
total_ignored_cost	3,70534474814815
was_batch_mode_ever_considered	True

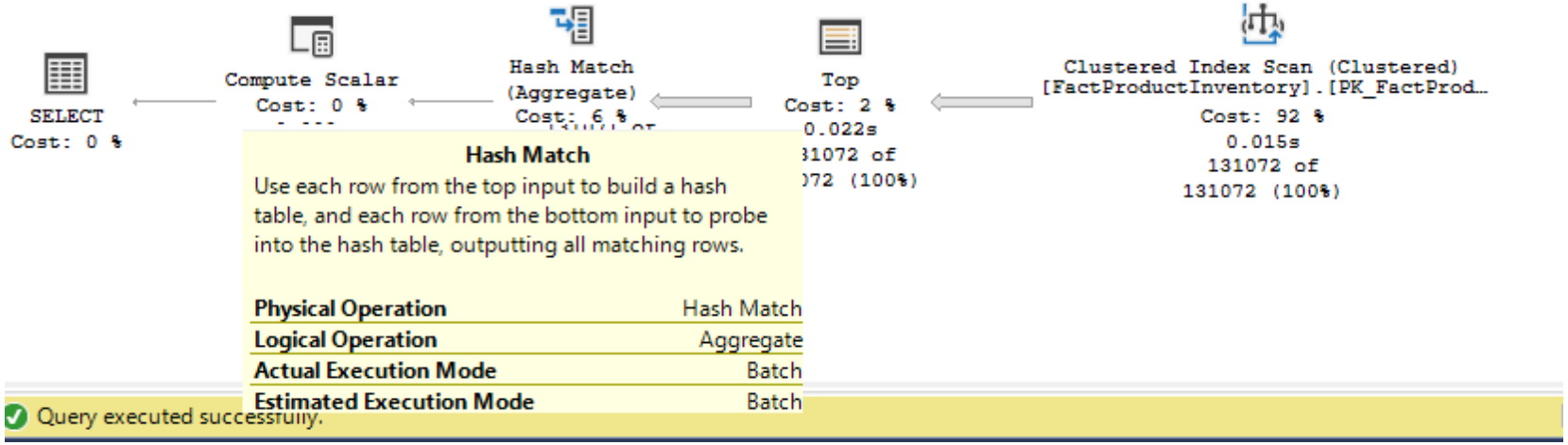
```sql
USE AdventureWorksDW2019;
GO
--row mode
SELECT COUNT(*), MAX(UnitsIn) FROM (SELECT TOP (131071) * FROM dbo.FactProductInventory) xxx;
GO
--batch mode
SELECT COUNT(*), MAX(UnitsIn) FROM (SELECT TOP (131072) * FROM dbo.FactProductInventory) xxx;
GO;
```

Query 1: Query cost (relative to the batch): 51%
SELECT COUNT(*), MAX(UnitsIn) FROM (SELECT TOP (131071) * FROM dbo.FactProductInventory) xxx



row mode

Query 2: Query cost (relative to the batch): 49%
SELECT COUNT(*), MAX(UnitsIn) FROM (SELECT TOP (131072) * FROM dbo.FactProductInventory) xxx

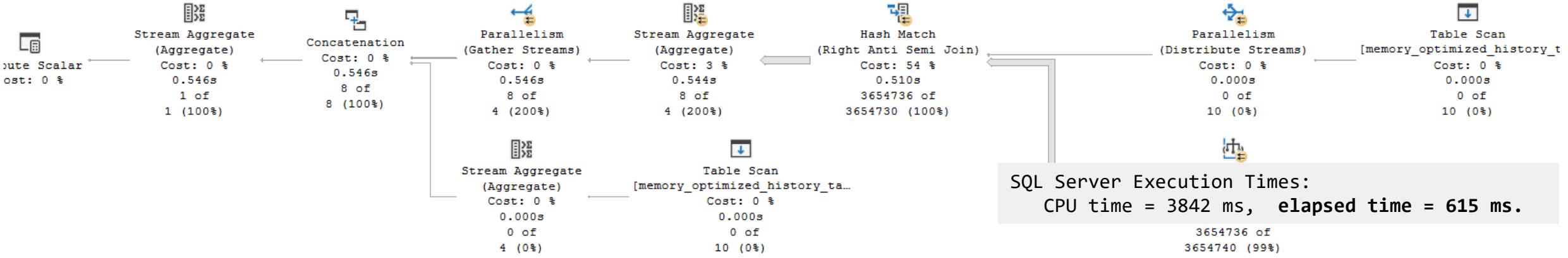


batch mode

```
USE WideWorldImporters;
GO
SELECT COUNT(*) FROM  Warehouse.ColdRoomTemperatures_Archive OPTION(USE HINT('QUERY_OPTIMIZER_COMPATIBILITY_LEVEL_140'));
GO
SELECT COUNT(*) FROM  Warehouse.ColdRoomTemperatures_Archive;
```

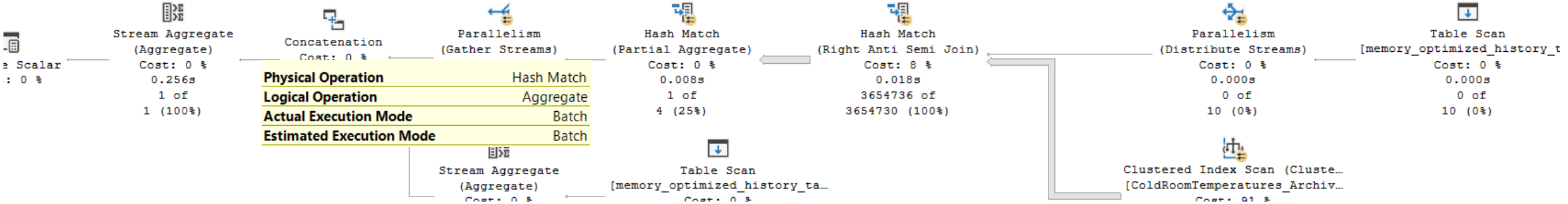
2x

Query 1: Query cost (relative to the batch): 68%
SELECT COUNT(*) FROM Warehouse.ColdRoomTemperatures_Archive OPTION (USE HINT ('QUERY_OPTIMIZER_COMPATIBILITY_LEVEL_140'))



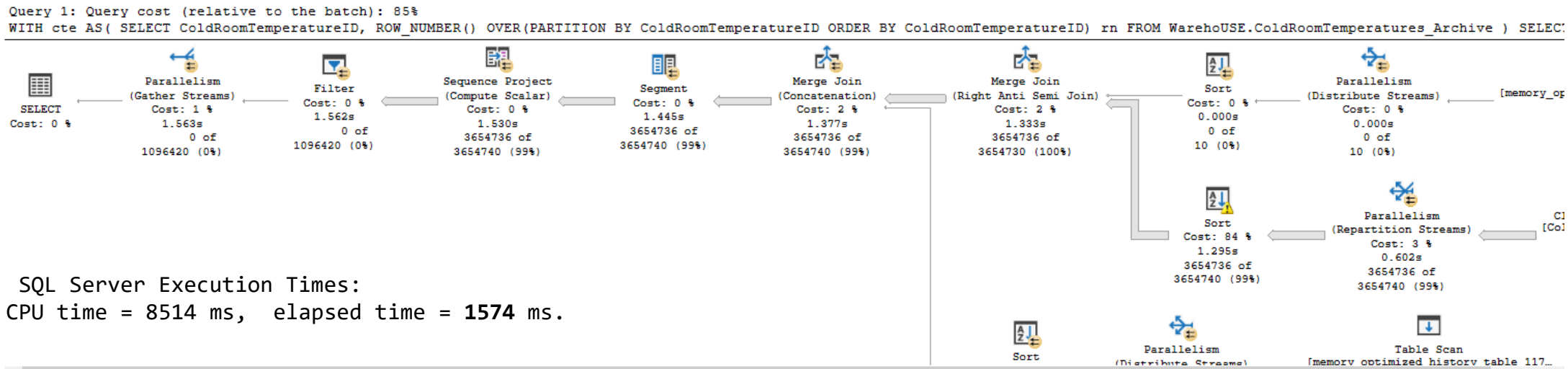
SQL Server Execution Times:
CPU time = 3842 ms, **elapsed time = 615 ms.**

Query 2: Query cost (relative to the batch): 32%
SELECT COUNT(*) FROM Warehouse.ColdRoomTemperatures_Archive

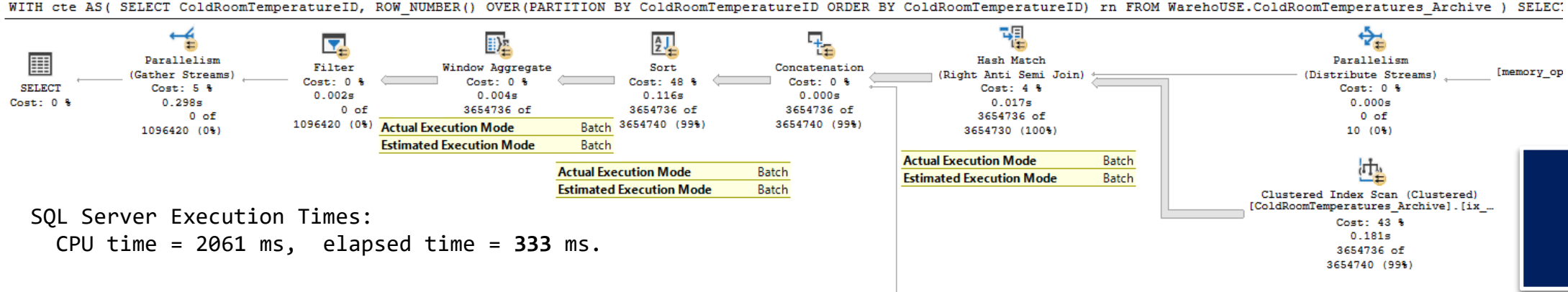


SQL Server Execution Times:
CPU time = 1642 ms, **elapsed time = 306 ms.**

```sql
WITH cte AS(
SELECT ColdRoomTemperatureID, ROW_NUMBER() OVER(PARTITION BY ColdRoomTemperatureID ORDER BY ColdRoomTemperatureID) rn
FROM WarehoUSE.ColdRoomTemperatures_Archive
)
SELECT * FROM cte WHERE rn > 1 OPTION (USE HINT ('QUERY_OPTIMIZER_COMPATIBILITY_LEVEL_140')) ;
GO
 WITH cte AS(
SELECT ColdRoomTemperatureID, ROW_NUMBER() OVER(PARTITION BY ColdRoomTemperatureID ORDER BY ColdRoomTemperatureID) rn
FROM WarehoUSE.ColdRoomTemperatures_Archive
)
SELECT * FROM cte WHERE rn > 1;
```



SQL Server Execution Times:
CPU time = 8514 ms, elapsed time = **1574** ms.

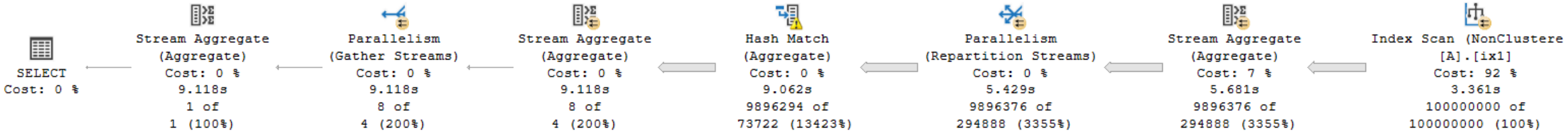


SQL Server Execution Times:
CPU time = 2061 ms, elapsed time = **333** ms.

5x

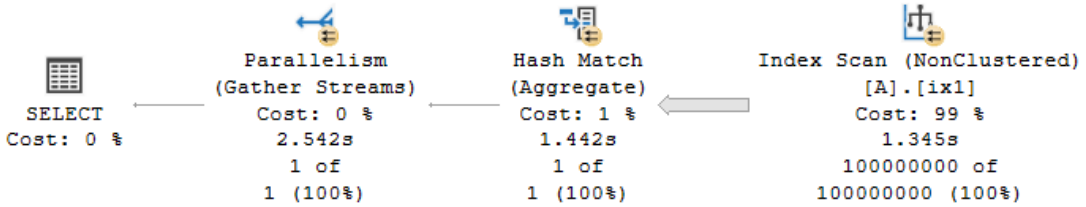
```
USE Statistik;
GO
ALTER DATABASE Statistik SET COMPATIBILITY_LEVEL = 150;
GO
SELECT COUNT_BIG(DISTINCT pid) from dbo.A OPTION (USE HINT ('QUERY_OPTIMIZER_COMPATIBILITY_LEVEL_140'));
GO
SELECT COUNT_BIG(DISTINCT pid) from dbo.A;
```

Query 1: Query cost (relative to the batch): 52%
SELECT COUNT_BIG(DISTINCT pid) from dbo.A OPTION (USE HINT ('QUERY_OPTIMIZER_COMPATIBILITY_LEVEL_140'))



SQL Server Execution Times:
CPU time = 43405 ms, elapsed time = **9136** ms.

Query 2: Query cost (relative to the batch): 48%
SELECT COUNT_BIG(DISTINCT pid) from dbo.A



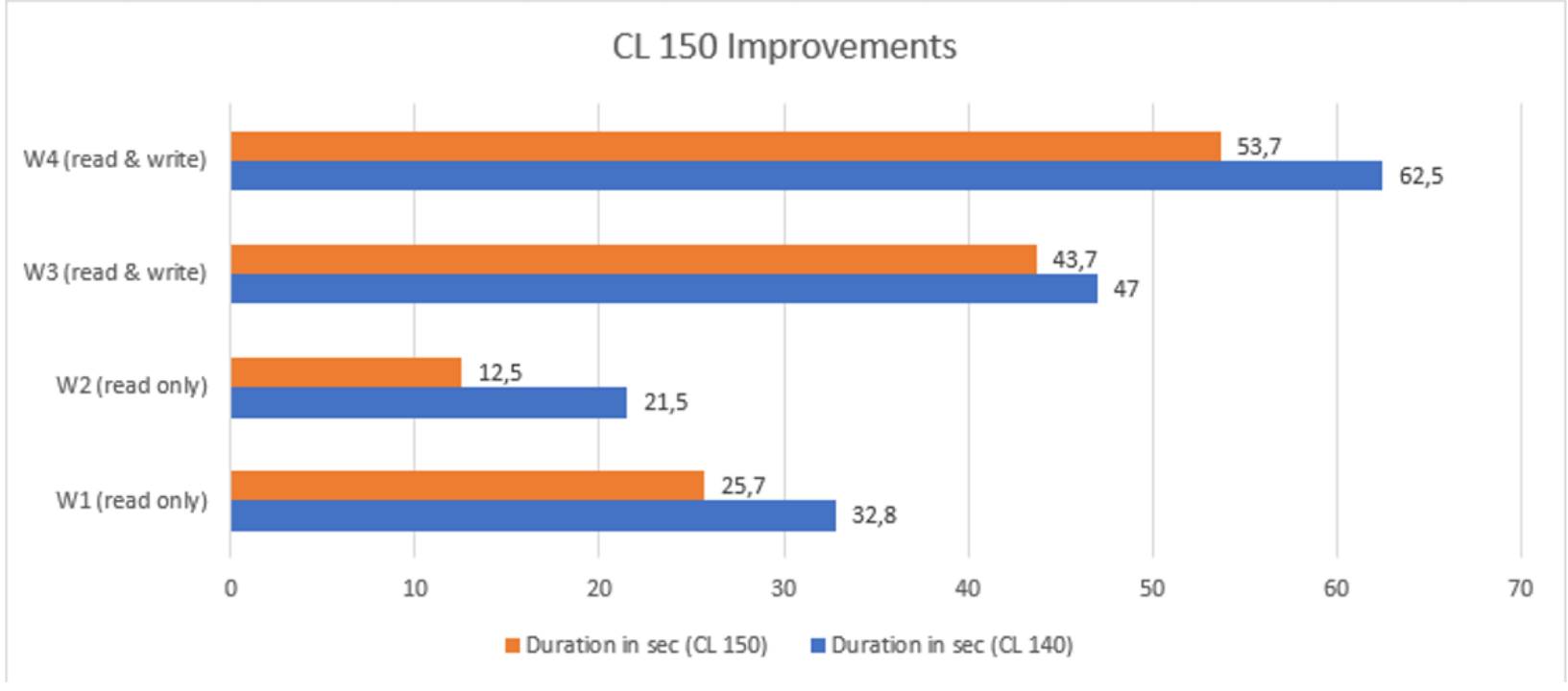
SQL Server Execution Times:
CPU time = 17171 ms, elapsed time = **2619** ms.

3,5x

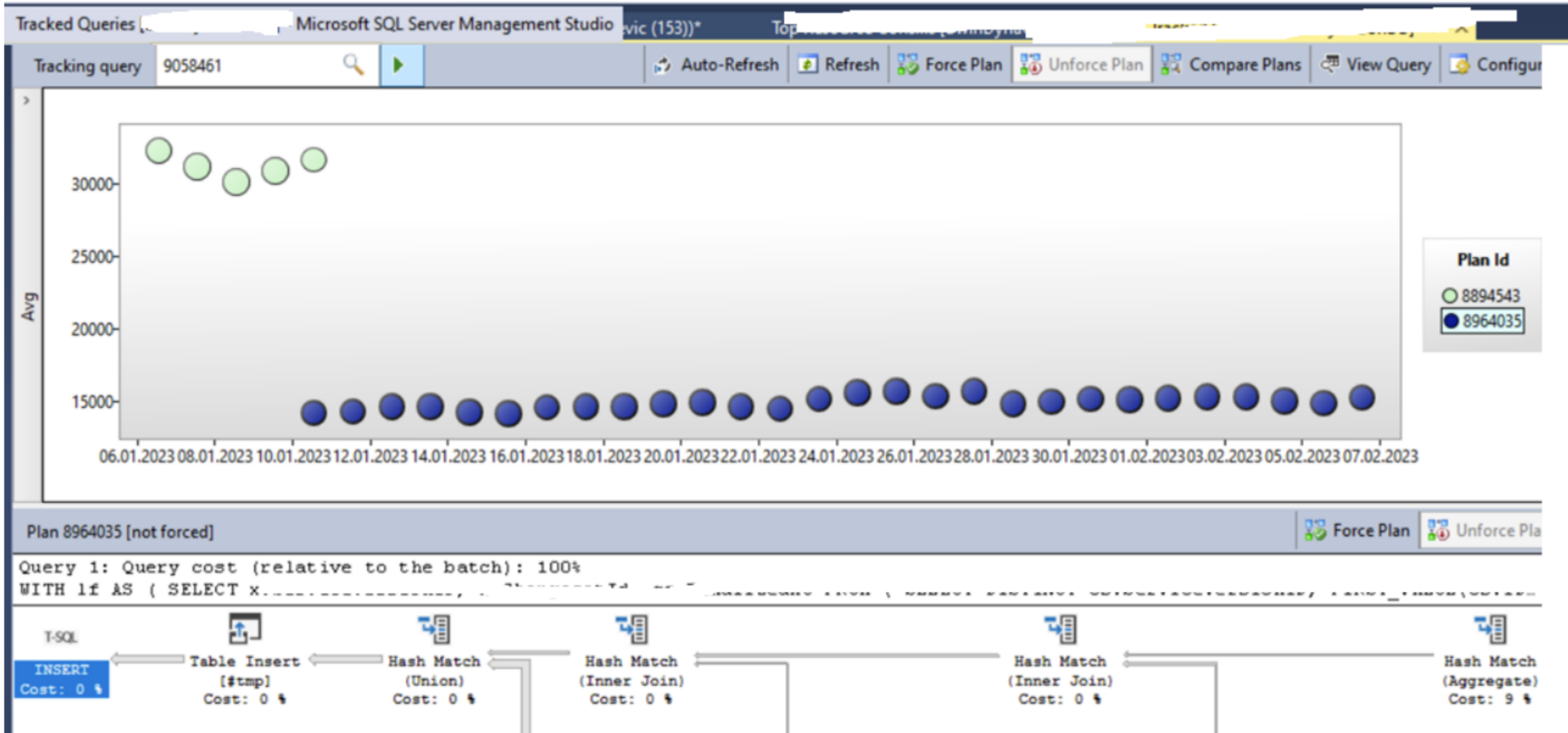
Some test results - a pure OLTP workload

Workload	Duration in sec (CL 140)	Duration in sec (CL 150)	Improvement
W1 (read only)	32,8	25,7	21,6 %
W2 (read only)	21,5	12,5	41, 8 %
W3 (read & write)	47,0	43,7	7 %
W4 (read & write)	62,5	53,7	14,1 %

an overall improvement – 17%



An example from our production system

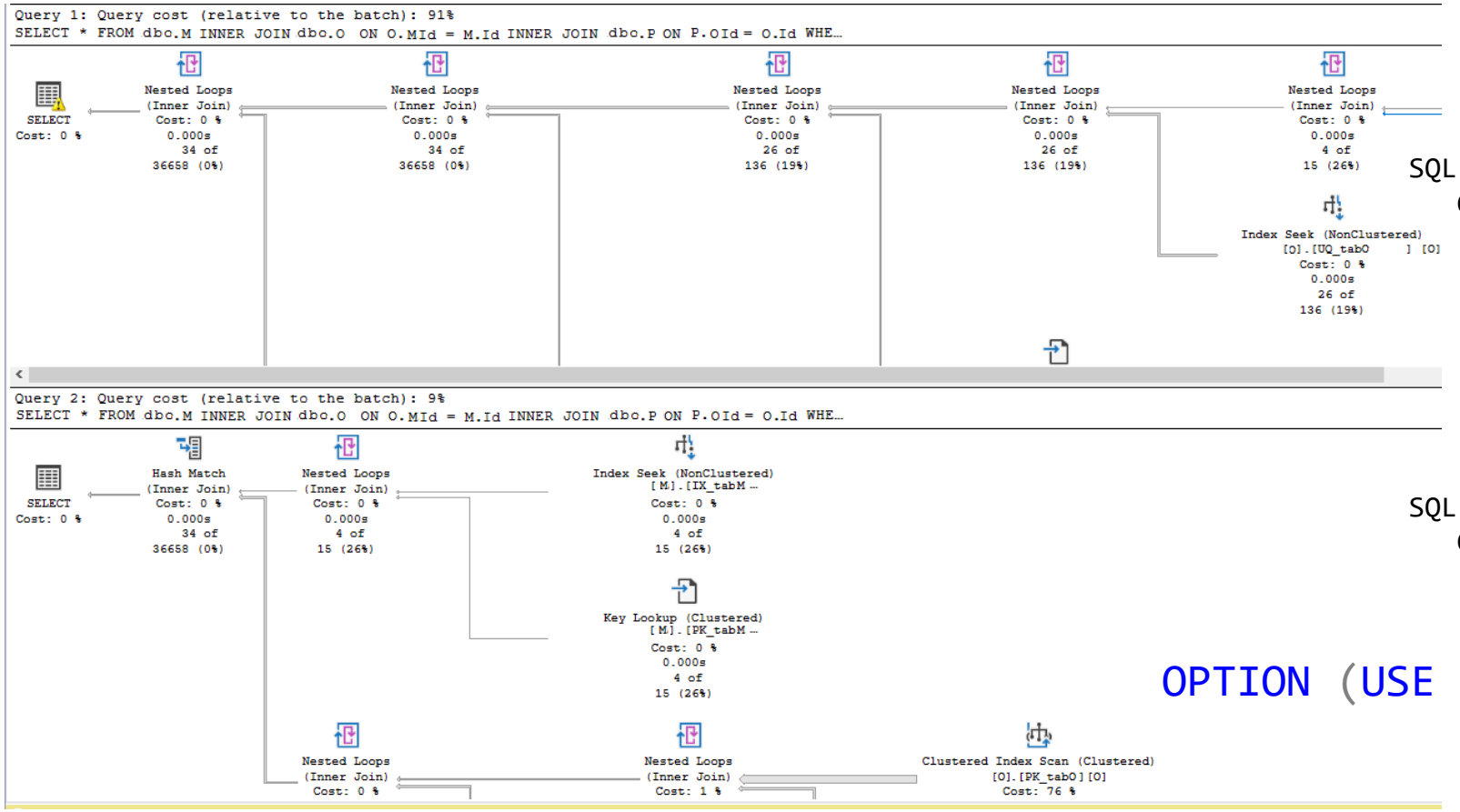


Regressions?

25x

```
SELECT * FROM dbo.M
INNER JOIN dbo.O ON O.Mid = M.Id
INNER JOIN dbo.P ON P.Oid = O.Id
WHERE M.c1 = 2462782;
```

```
SELECT * FROM dbo.M
INNER JOIN dbo.O ON O.Mid = M.Id
INNER JOIN dbo.P ON P.Oid = O.Id
WHERE M.c1 = 2462782 OPTION (USE HINT ('QUERY_OPTIMIZER_COMPATIBILITY_LEVEL_150'));
```




```
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 2 ms.
```

```
SQL Server Execution Times:
   CPU time = 47 ms,  elapsed time = 50 ms.
```

```
OPTION (USE HINT('DISALLOW_BATCH_MODE'));
```

Enabling/Disabling Batch Mode

- Enable Batch mode on rowstore

```
ALTER DATABASE current SET COMPATIBILITY_LEVEL = 150;
ALTER DATABASE SCOPED CONFIGURATION SET BATCH_MODE_ON_ROWSTORE = ON;
OPTION (USE HINT('ALLOW_BATCH_MODE'));
```

- Disable Batch mode on rowstore

```
ALTER DATABASE SCOPED CONFIGURATION SET BATCH_MODE_ON_ROWSTORE = OFF;

OPTION (USE HINT('DISALLOW_BATCH_MODE'));
```

Batch Mode on Rowstore - Limitations

- Reading from memory-optimized tables will be always done in row mode
- Queries use table has (B)LOB, XML or sparse columns in the SELECT or WHERE clause
- Queries using full-text or cursors

Batch Mode on Rowstore - Conclusion

- Improvements within the database engine, with no efforts
- **It could be a reason for upgrade for some companies!!!**

- It will not optimize all queries, where you would expect the optimization, but generally it is beneficial for most workloads

- Possible regressions, but you can enable/disable feature at two levels

- It brings benefits for queries with large tables and datasets

5

Scalar UDF Inlining

Scalar UDFs in SQL Server

Why do SQL Server Scalar-valued functions get slower?

Refactor SQL Server scalar UDF to inline TVF to improve performance

Why SQL Server scalar functions are bad?

T-SQL Best Practices - Don't Use Scalar Value Functions in Column .

Are SQL Server Functions Dragging Your Query Down?

SQL functions rarely perform well.

Scalar UDFs in SQL Server

- They are very slow
  - Iterative invocation
  - Overhead for invoking function – once per row
- No cross-statement optimization
- Only serial execution plans possible

Scalar UDF Inlining

- Goal – improve queries with scalar UDFs
- Scalar UDF Inlining feature (Froid framework):
  - Transforms scalar UDF into relational expressions or subqueries (IF => CASE WHEN)
  - Embeds them in the calling query by using APPLY operator
  - Optimizes expressions or subqueries
- Result:
  - More efficient plan (better choice of operators)
  - Execution plan can be parallel

Scalar UDF Inlining

Froid: Optimization of Imperative Programs in a Relational Database

Karthik Ramachandra
Microsoft Gray Systems Lab
karam@microsoft.com

Kwanghyun Park
Microsoft Gray Systems Lab
kwpark@microsoft.com

K. Venkatesh Emani*
IIT Bombay
venkateshek@cse.iitb.ac.in

Alan Halverson
Microsoft Gray Systems Lab
alanhal@microsoft.com

César Galindo-Legaria
Microsoft
cesarg@microsoft.com

Conor Cunningham
Microsoft
conorc@microsoft.com

ABSTRACT

For decades, RDBMSs have supported declarative SQL as well as imperative functions and procedures as ways for users to express data processing tasks. While the evaluation of declarative SQL has received a lot of attention resulting in highly sophisticated techniques, the evaluation of imperative programs has remained naïve and highly inefficient. Imperative programs offer several benefits over SQL and hence are

expressing intent has on one hand provided high-level abstractions for data processing, while on the other hand, has enabled the growth of sophisticated query evaluation techniques and highly efficient ways to process data.

Despite the expressive power of declarative SQL, almost all RDBMSs support procedural extensions that allow users to write programs in various languages (such as Transact-SQL, C#, Java and R) using imperative constructs such as variable assignments, conditional branching, and loops

http://www.vldb.org/pvldb/vol11/p432-ramachandra.pdf

Froid Transformation

```
DECLARE @val VARCHAR(10);
DECLARE @a INT = 2000;

    IF @a > 1000
        SET @val = 'HIGH';
    ELSE IF @a > 500
        SET @val = 'MEDIUM'
    ELSE
        SET @val = 'LOW'

    SELECT @val;
```

```
SELECT q5.v
FROM
(
        (SELECT 2000 AS a) AS q1
        OUTER APPLY
                (SELECT CASE WHEN q1.a > 1000 THEN 'HIGH' END AS val)
AS q2
        OUTER APPLY
        (SELECT CASE WHEN q1.a > 500 THEN 'HIGH' END AS val) AS q3
        OUTER APPLY
                (SELECT 'LOW' AS val) AS q4
        OUTER APPLY
                (SELECT CASE WHEN q2.val IS NOT NULL
           THEN q2.val
           WHEN q3.val IS NOT NULL
           THEN q3.val
           ELSE q4.val
        END v) AS q5
);
```

Froid Transformation

```
DECLARE @val VARCHAR(10);
DECLARE @a INT = 2000;

    IF @a > 1000
        SET @val = 'HIGH';
    ELSE IF @a > 500
        SET @val = 'MEDIUM'
    ELSE
        SET @val = 'LOW'

    SELECT @val;
```

```
SELECT q5.v
FROM
(
        (SELECT 2000 AS a) AS q1
        OUTER APPLY
                (SELECT CASE WHEN q1.a > 1000 THEN 'HIGH' END AS val)
AS q2
        OUTER APPLY
        (SELECT CASE WHEN q1.a > 500 THEN 'HIGH' END AS val) AS q3
        OUTER APPLY
                (SELECT 'LOW' AS val) AS q4
        OUTER APPLY
                (SELECT CASE WHEN q2.val IS NOT NULL
           THEN q2.val
           WHEN q3.val IS NOT NULL
           THEN q3.val
           ELSE q4.val
        END v) AS q5
);
```

Froid Transformation

```
DECLARE @val VARCHAR(10);
DECLARE @a INT = 2000;

    IF @a > 1000
        SET @val = 'HIGH';
    ELSE IF @a > 500
        SET @val = 'MEDIUM'
    ELSE
        SET @val = 'LOW'

    SELECT @val;
```

```
SELECT q5.v
FROM
(
        (SELECT 2000 AS a) AS q1
        OUTER APPLY
                (SELECT CASE WHEN q1.a > 1000 THEN 'HIGH' END AS val)
AS q2
        OUTER APPLY
        (SELECT CASE WHEN q1.a > 500 THEN 'HIGH' END AS val) AS q3
        OUTER APPLY
                (SELECT 'LOW' AS val) AS q4
        OUTER APPLY
                (SELECT CASE WHEN q2.val IS NOT NULL
           THEN q2.val
           WHEN q3.val IS NOT NULL
           THEN q3.val
           ELSE q4.val
        END v) AS q5
);
```

Froid Transformation

```
DECLARE @val VARCHAR(10);
DECLARE @a INT = 2000;

    IF @a > 1000
        SET @val = 'HIGH';
    ELSE IF @a > 500
        SET @val = 'MEDIUM'
    ELSE
        SET @val = 'LOW'

    SELECT @val;
```

```
SELECT q5.v
FROM
(
        (SELECT 2000 AS a) AS q1
        OUTER APPLY
                (SELECT CASE WHEN q1.a > 1000 THEN 'HIGH' END AS val)
AS q2
        OUTER APPLY
        (SELECT CASE WHEN q1.a > 500 THEN 'HIGH' END AS val) AS q3
        OUTER APPLY
                (SELECT 'LOW' AS val) AS q4
        OUTER APPLY
                (SELECT CASE WHEN q2.val IS NOT NULL
           THEN q2.val
           WHEN q3.val IS NOT NULL
           THEN q3.val
           ELSE q4.val
        END v) AS q5
);
```

Froid Transformation

```
DECLARE @val VARCHAR(10);
DECLARE @a INT = 2000;

    IF @a > 1000
        SET @val = 'HIGH';
    ELSE IF @a > 500
        SET @val = 'MEDIUM'
    ELSE
        SET @val = 'LOW'

    SELECT @val;
```

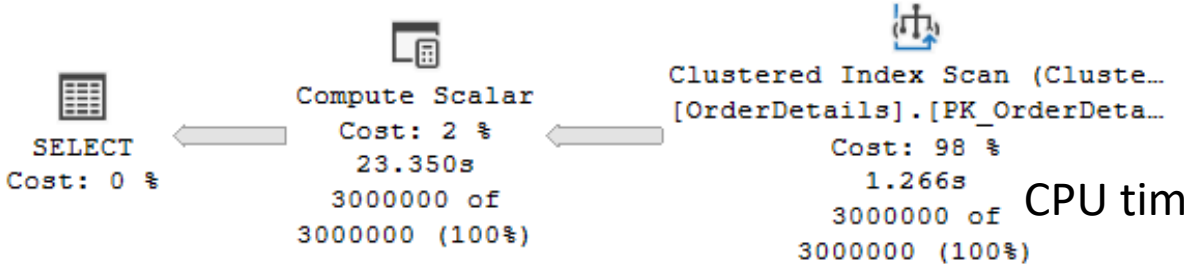
```
SELECT q5.v
FROM
(
        (SELECT 2000 AS a) AS q1
        OUTER APPLY
                (SELECT CASE WHEN q1.a > 1000 THEN 'HIGH' END AS val)
AS q2
        OUTER APPLY
        (SELECT CASE WHEN q1.a > 500 THEN 'HIGH' END AS val) AS q3
        OUTER APPLY
                (SELECT 'LOW' AS val) AS q4
        OUTER APPLY
                (SELECT CASE WHEN q2.val IS NOT NULL
           THEN q2.val
           WHEN q3.val IS NOT NULL
           THEN q3.val
           ELSE q4.val
        END v) AS q5
);
```

```sql
CREATE OR ALTER FUNCTION dbo.GetOrderItemStatus(
@Quantity INT, @UnitPrice DECIMAL(10,2))
RETURNS VARCHAR(20)
AS
BEGIN
    DECLARE @Ret VARCHAR(20) = '';
    DECLARE @Amount  DECIMAL(10,2) =
@Quantity * @UnitPrice;
    IF @Amount > 1000
        SET @Ret = 'TOP 1000'
    ELSE IF @Amount > 500
        SET @Ret = 'TOP 500'
    RETURN @Ret;
END
```

```sql
SET STATISTICS TIME ON;
GO
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 140;
GO
SELECT *, dbo.GetOrderItemStatus(Quantity,UnitPrice) ItemStatus FROM dbo.OrderDetails;
GO
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 150;
GO
SELECT *, dbo.GetOrderItemStatus(Quantity,UnitPrice) ItemStatus FROM dbo.OrderDetails;
```

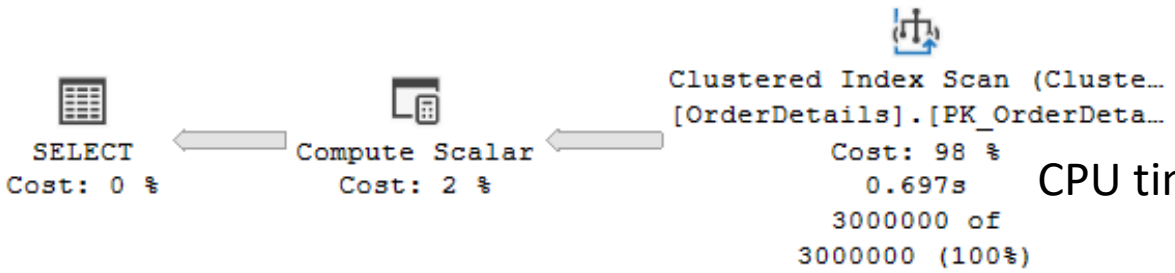
12x

Query 1: Query cost (relative to the batch): 50%
SELECT *, dbo.GetOrderItemStatus(Quantity,UnitPrice) ItemStatus FROM dbo.OrderDetails



CPU time = 18656 ms, elapsed time = **24306** ms.

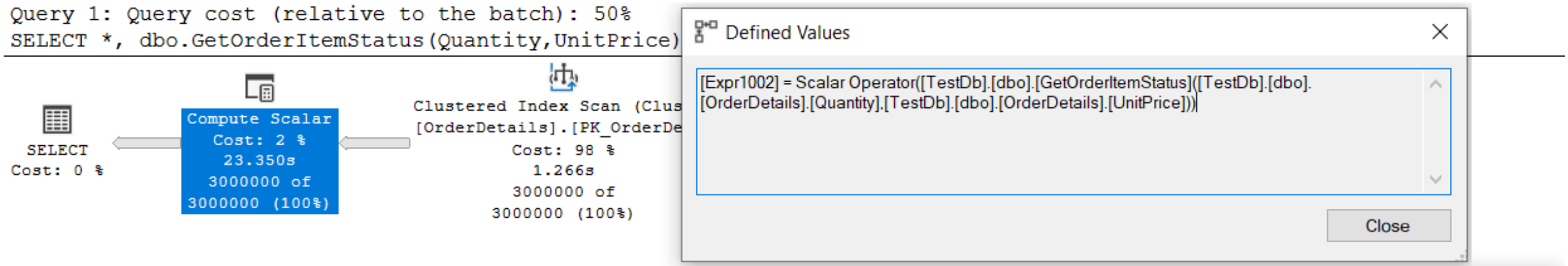
Query 2: Query cost (relative to the batch): 50%
SELECT *, dbo.GetOrderItemStatus(Quantity,UnitPrice) ItemStatus FROM dbo.OrderDetails

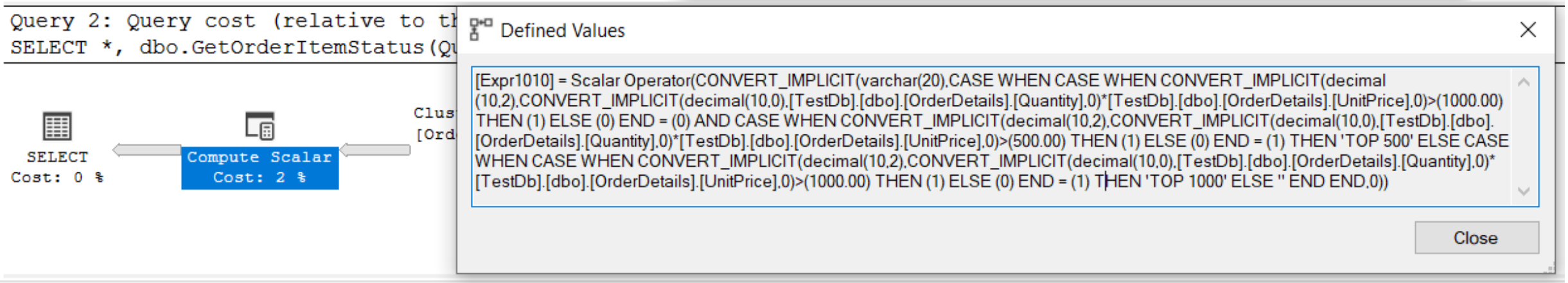


CPU time = 2047 ms, elapsed time = **2075** ms.

```sql
CREATE OR ALTER FUNCTION dbo.GetOrderItemStatus(
@Quantity INT, @UnitPrice DECIMAL(10,2))
RETURNS VARCHAR(20)
AS
BEGIN
    DECLARE @Ret VARCHAR(20) = '';
    DECLARE @Amount  DECIMAL(10,2) =
@Quantity * @UnitPrice;
    IF @Amount > 1000
        SET @Ret = 'TOP 1000'
    ELSE IF @Amount > 500
        SET @Ret = 'TOP 500'
    RETURN @Ret;
END
```

```sql
SET STATISTICS TIME ON;
GO
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 140;
GO
SELECT *, dbo.GetOrderItemStatus(Quantity,UnitPrice) ItemStatus FROM dbo.OrderDetails;
GO
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 150;
GO
SELECT *, dbo.GetOrderItemStatus(Quantity,UnitPrice) ItemStatus FROM dbo.OrderDetails;
```

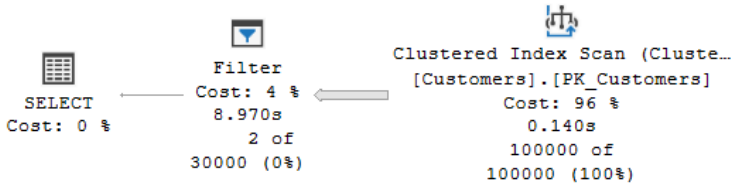




```sql
CREATE OR ALTER FUNCTION dbo.GetOrderCnt (@CustomerId INT)
RETURNS INT
AS
BEGIN
    DECLARE @Cnt INT;
    SELECT @Cnt = COUNT(*) FROM dbo.Orders WHERE CustomerId = @CustomerId;
    RETURN @Cnt;
END

ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 140;
GO
SELECT * FROM dbo.Customers WHERE dbo.GetOrderCnt(CustomerId) > 25;
GO
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 150;
GO
SELECT * FROM dbo.Customers WHERE dbo.GetOrderCnt(CustomerId) > 25;
```

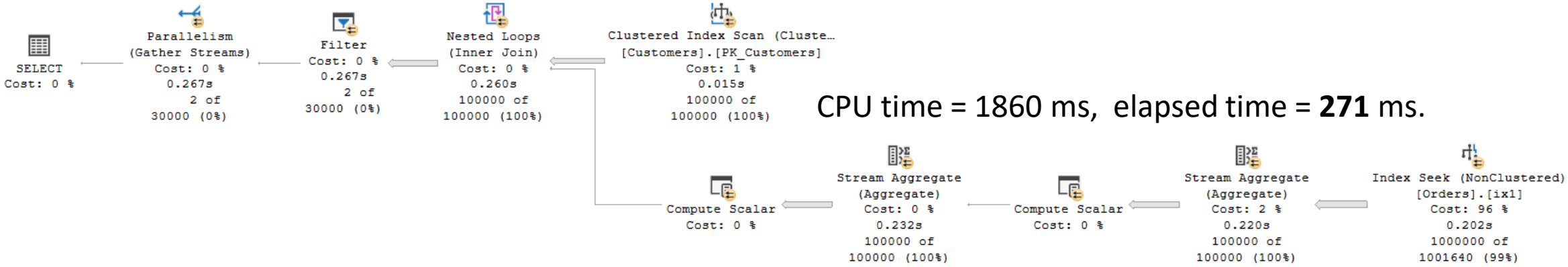
33x

Query 1: Query cost (relative to the batch): 1%
SELECT * FROM dbo.Customers WHERE dbo.GetOrderCnt(CustomerId)>25



CPU time = 8610 ms, elapsed time = **8971** ms.

Query 2: Query cost (relative to the batch): 99%
SELECT * FROM dbo.Customers WHERE dbo.GetOrderCnt(CustomerId)>25



CPU time = 1860 ms, elapsed time = **271** ms.

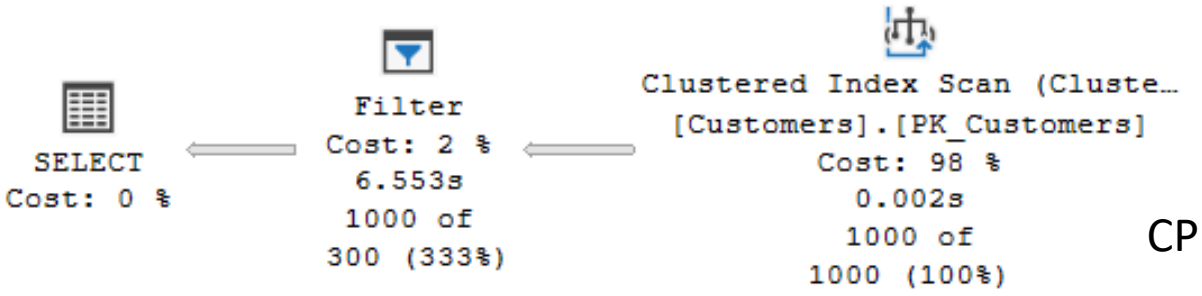
0%

```sql
CREATE OR ALTER FUNCTION dbo.GetDaysFromLastOrder(@CustomerId INT)
RETURNS INT
AS
BEGIN
    DECLARE @Days INT;
    DECLARE @LastOrder DATETIME;
    SET @LastOrder = (SELECT TOP (1) OrderDate FROM dbo.Orders WHERE CustomerId = @CustomerId ORDER BY OrderDate DESC);
    SELECT @Days = DATEDIFF(day, @LastOrder, GETDATE());
    RETURN @Days;
END
```

```sql
SELECT * FROM dbo.Customers WHERE dbo.GetDaysFromLastOrder(CustomerId) > 365;
```

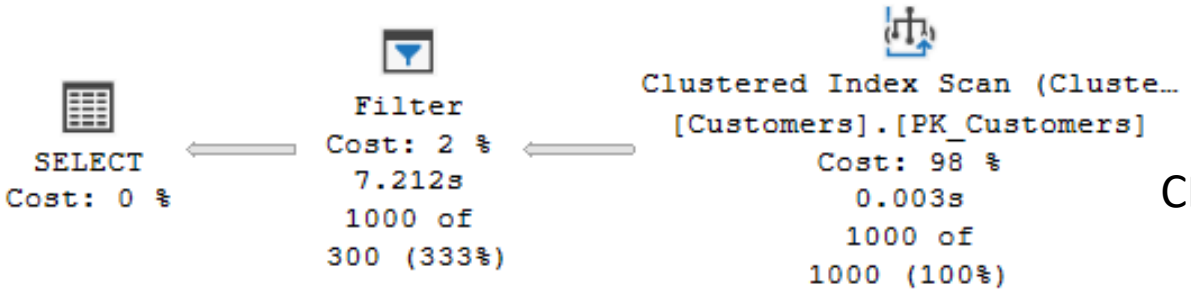
Scalar UDF Inline does not work with **GETDATE()** function

Query 1: Query cost (relative to the batch): 50%
SELECT * FROM dbo.Customers WHERE dbo.GetDaysFromLastOrder(CustomerId) > 365



CPU time = 12531 ms, elapsed time = **12648** ms.

Query 2: Query cost (relative to the batch): 50%
SELECT * FROM dbo.Customers WHERE dbo.GetDaysFromLastOrder(CustomerId) > 365



CPU time = 12485 ms, elapsed time = **13079** ms.

Scalar UDF Inlining

- Not all scalar UDFs can be inlined
  - Check whether a function can be inlined:

- **is_inlineable = 1** does not imply that it will always be inlined
  - Decision is made when the query referencing a scalar UDF is compiled

```
SELECT CONCAT(SCHEMA_NAME(o.schema_id),'.',o.name), is_inlineable
FROM sys.sql_modules m INNER JOIN sys.objects o ON o.object_id = m.object_id WHERE o.type = 'FN';
```

Scalar UDF Inlining - Limitations

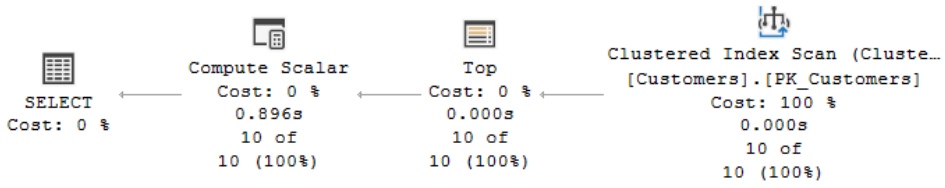
- UDF cannot invoke GETDATE() or NEWSEQUENTIALID
- The UDF does not reference table variables, table-valued parameters or user-defined types
- UDF is not natively compiled (interop is supported)
- UDF is not used in a computed column or a check constraint definition
- The UDF is not a partition function
- Full list of limitations: https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/scalar-udf-inlining?view=sql-server-ver15

Regressions?

5x

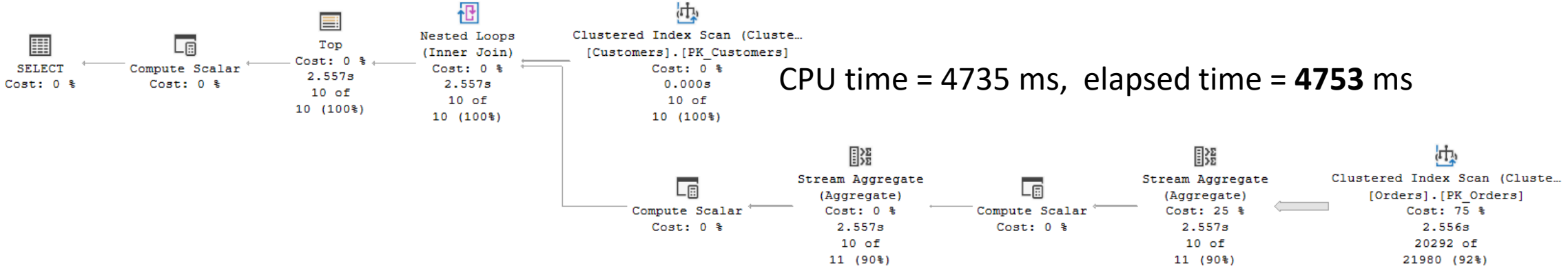
```
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 140;
GO
SELECT TOP (10) * FROM dbo.Customers WHERE dbo.GetOrderCnt(CustomerId) > 25;
GO
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 150;
GO
SELECT TOP (10) * FROM dbo.Customers WHERE dbo.GetOrderCnt(CustomerId) > 25;
```

```
Query 1: Query cost (relative to the batch): 0%
SELECT TOP (10) *, dbo.GetOrderCnt(CustomerId) FROM dbo.Customers
```



CPU time = 5922 ms, elapsed time = **873** ms.

```
Query 2: Query cost (relative to the batch): 100%
SELECT TOP (10) *, dbo.GetOrderCnt(CustomerId) FROM dbo.Customers
Missing Index (Impact 99.8008): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Orders] ([CustomerID])
```



CPU time = 4735 ms, elapsed time = **4753** ms

Regressions?

```
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 140;
GO
SELECT TOP (10) * FROM dbo.Customers WHERE dbo.GetOrderCnt(CustomerId) > 25;
GO
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 150;
GO
SELECT TOP (10) * FROM dbo.Customers WHERE dbo.GetOrderCnt(CustomerId) > 25;
```

Solution

```
SELECT TOP (10) * FROM dbo.Customers WHERE dbo.GetOrderCnt(CustomerId) > 25
OPTION (USE HINT('DISABLE_TSQL_SCALAR_UDF_INLINING'));
```

Scalar UDF Inlining - Settings

- Enable

```
ALTER DATABASE current SET COMPATIBILITY_LEVEL = 150;
ALTER DATABASE SCOPED CONFIGURATION SET
TSQL_SCALAR_UDF_INLINING = ON;
CREATE OR ALTER FUNCTION dbo.getMaxOrderDate(@CustID INT)
RETURNS DATETIME WITH INLINE = ON
```

- Disable

```
ALTER DATABASE SCOPED CONFIGURATION SET
TSQL_SCALAR_UDF_INLINING = OFF;
OPTION (USE HINT('DISABLE_TSQL_SCALAR_UDF_INLINING'));
CREATE OR ALTER FUNCTION dbo.getMaxOrderDate(@CustID INT)
RETURNS DATETIME WITH INLINE = OFF
```

Regressions?
Scalar UDF Inlining - Conclusion

- Very promising feature
  - Improvements with no efforts
  - Part of the Standard Edition
- Many limitations (GETDATE(), table variables…)
- Very useful for small and medium companies (not enough people to rewrite UDFs) 3rd party tools
- it’s a first version

Scalar UDF Inlining - Conclusion

- Improved after 20 years, a bit late for many workloads
- It is considerable for new code (code on the left is more intuitive)

```
DECLARE @val VARCHAR(10);
DECLARE @a INT = 2000;

    IF @a > 1000
        SET @val = 'HIGH';
    ELSE IF @a > 500
        SET @val = 'MEDIUM'
    ELSE
        SET @val = 'LOW'

    SELECT @val;
```

```
SELECT q5.v
FROM
(
        (SELECT 2000 AS a) AS q1
        OUTER APPLY
                (SELECT CASE WHEN q1.a > 1000 THEN 'HIGH' END AS val)
AS q2
        OUTER APPLY
        (SELECT CASE WHEN q1.a > 500 THEN 'HIGH' END AS val) AS q3
        OUTER APPLY
                (SELECT 'LOW' AS val) AS q4
        OUTER APPLY
                (SELECT CASE WHEN q2.val IS NOT NULL
           THEN q2.val
           WHEN q3.val IS NOT NULL
           THEN q3.val
           ELSE q4.val
        END v) AS q5
);
```

Scalar UDF Inlining - Conclusion

- Do not use it for complex functions, the list of bugs and issues is very impressive
- In almost each CU, there are many bugfixes (and sometimes new bugs) related to Scalar UDF Inlining
- As far I know, it is still OFF by default in SQL Azure

- My recommendation is still to
  - Avoid user-defined functions or
  - Use inline table-valued functions

6

Table Variable Deferred Compilation

Problems with Queries Using Table Variables

- Queries using table variables with large number of rows (> 10K) can have an inefficient plan

- Inappropriate operators in the execution plan
  - mostly Nested Loop Join instead of Hash Join and vice versa

- Insufficient memory grants
  - number of processing rows is usually underestimated => less Memory Grant reserved for the query => spills to tempdb

Table Variable Deferred Compilation

```
DECLARE @T AS TABLE (ProductID INT);
INSERT INTO @T SELECT ProductID
FROM Production.Product
WHERE ProductLine IS NOT NULL;
```

```
SELECT * FROM @T t
INNER JOIN Sales.SalesOrderDetail od
ON t.ProductID = od.ProductID
INNER JOIN Sales.SalesOrderHeader h
ON h.SalesOrderID = od.SalesOrderID
ORDER BY od.UnitPrice DESC;
```

t_0

t_1

t_2

SQL Server 2017 and all previous versions

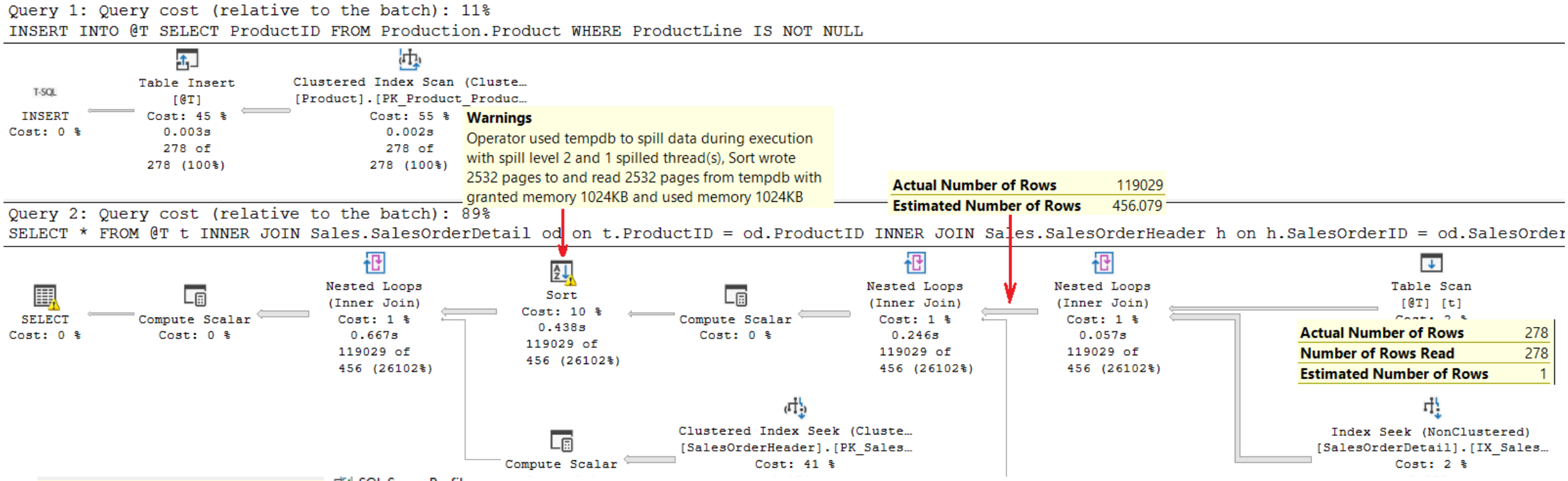
- Content of the table variable unknown at the compile time = cardinality 1

SQL Server 2019

- Content of the table variable known at the compile time => cardinality = actual number of rows

```sql
DECLARE @T AS TABLE (ProductID INT);
INSERT INTO @T SELECT ProductID FROM Production.Product WHERE ProductLine IS NOT NULL;

SELECT * FROM @T t
INNER JOIN Sales.SalesOrderDetail od on t.ProductID = od.ProductID
INNER JOIN Sales.SalesOrderHeader h on h.SalesOrderID = od.SalesOrderID
ORDER BY od.UnitPrice DESC;
```

SQL Server 2017



SELECT	
Cached plan size	96 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	1
Estimated Subtree Cost	0,181817
Memory Grant	1024
Estimated Number of Rows	456,079

SQL Server Profiler

File Edit View Replay Tools Window Help

execution time: **873 ms**

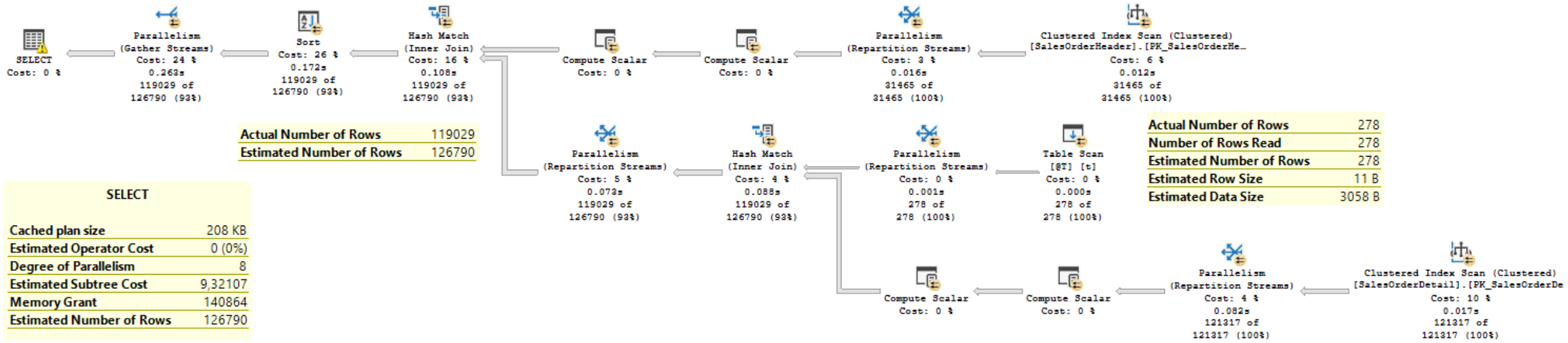
Untitled - 1 (\SQL2019)

EventClass	ApplicationName	ClientProcessID	DatabaseID	DatabaseName	EventSequence	EventSubClass
Sort Warnings	Microsoft SQ...	9928	9	AdventureWorks2019	2464	2 - Multiple pass

```sql
DECLARE @T AS TABLE (ProductID INT);
INSERT INTO @T SELECT ProductID FROM Production.Product WHERE ProductLine IS NOT NULL;

SELECT * FROM @T t
INNER JOIN Sales.SalesOrderDetail od on t.ProductID = od.ProductID
INNER JOIN Sales.SalesOrderHeader h on h.SalesOrderID = od.SalesOrderID
ORDER BY od.UnitPrice DESC;
```

SQL Server 2019

```
Query 2: Query cost (relative to the batch): 100%
SELECT * FROM @T t INNER JOIN Sales.SalesOrderDetail od on t.ProductID = od.ProductID INNER JOIN Sales.SalesOrderHeader h on h.SalesOrderID = od.SalesOrderID ORDER BY od.UnitPr…
Missing Index (Impact 18.7112): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales].[SalesOrderDetail] ([ProductID]) INCLUDE ([CarrierTrackingNumber],[Orde…
```



execution time: **484 ms**

2x

An example of improvement by TVDC from prod.

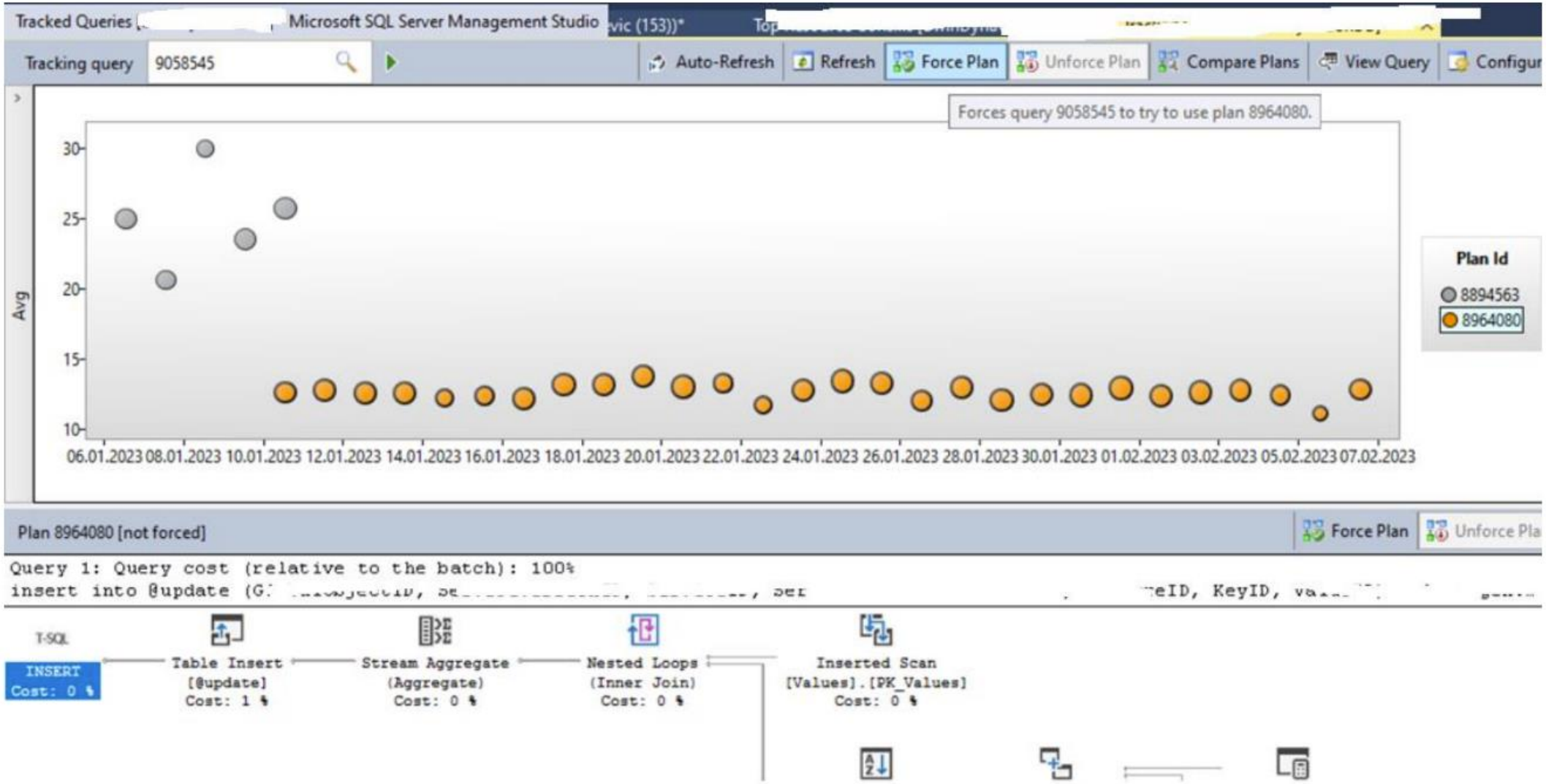


Table Variable Deferred Compilation

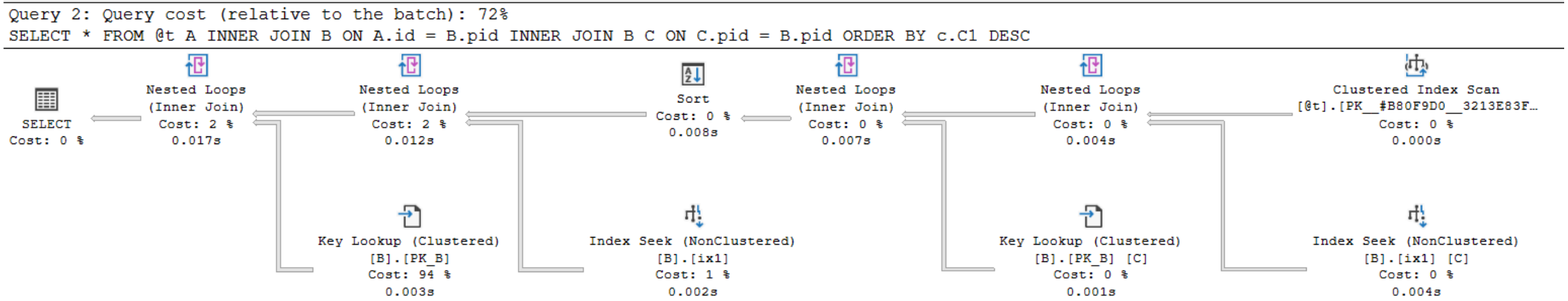
- It looks great!
  - Now, the estimation is ALWAYS CORRECT!!!
  - In all previous versions, it was almost ALWAYS WRONG and partially very underestimated!
  - This should be only good?!
- But…
  - We had to live 20 years with cardinality estimation of one row
  - And we a
  - And sometimes, we misused it!
  - Or SQL Server used or misused it!

Table Variable Deferred Compilation - Performance Regressions

```
DECLARE @t TABLE(id INT PRIMARY KEY, pid INT, c1 CHAR(100));
INSERT INTO @t SELECT * FROM A WHERE A.pid = 413032;
SELECT * FROM @t A
INNER JOIN B ON A.id = B.pid
INNER JOIN B C ON C.pid = B.pid
ORDER BY C.c1 DESC;
```

SQL Server 2017

Query returns 3 222 rows

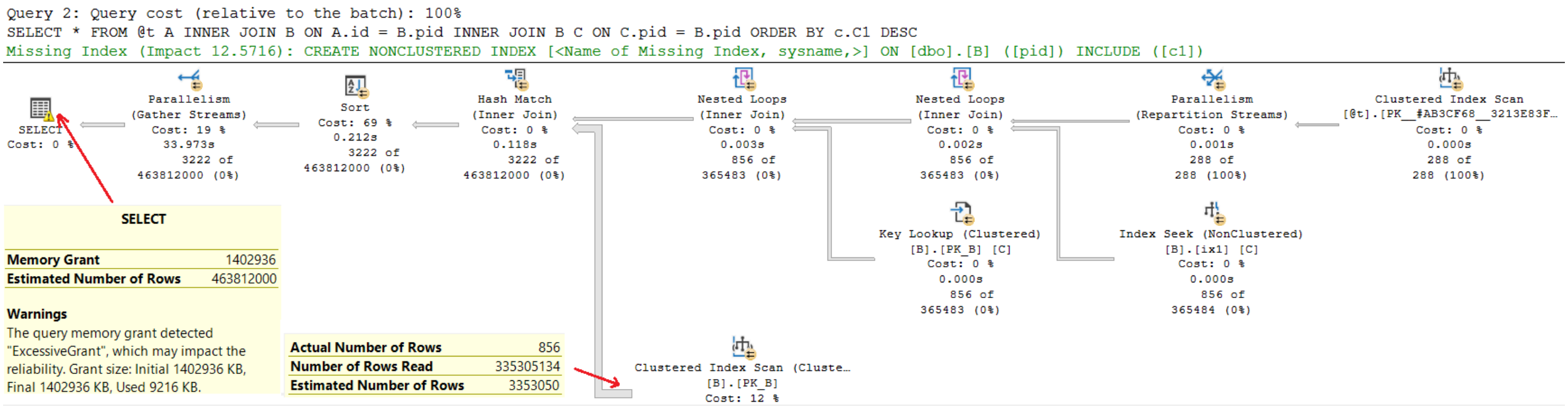


SQL Server Execution Times:

CPU time = 31 ms, elapsed time = **116** ms.

```sql
DECLARE @t TABLE(id INT PRIMARY KEY, pid INT, c1 CHAR(100));
INSERT INTO @t SELECT * FROM A WHERE A.pid = 413032;
SELECT * FROM @t A
INNER JOIN B ON A.id = B.pid
INNER JOIN B C ON C.pid = B.pid
ORDER BY C.c1 DESC;
```

SQL Server 2019



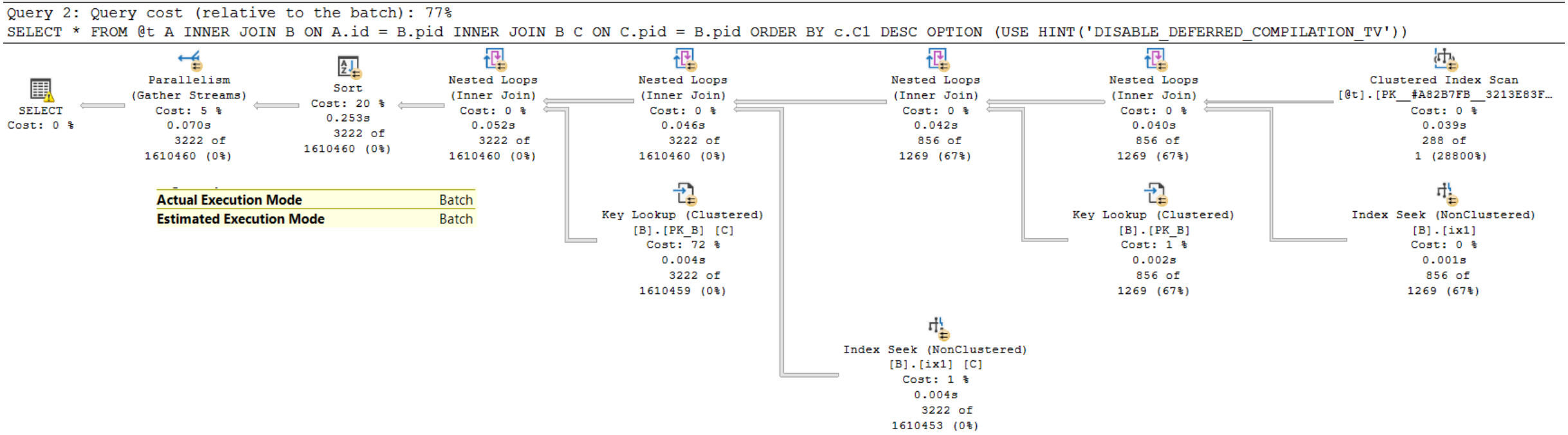
SQL Server Execution Times:

CPU time = 43999 ms, elapsed time = **34482** ms.

300x

```
DECLARE @t TABLE(id INT PRIMARY KEY, pid INT, c1 CHAR(100));
INSERT INTO @t SELECT * FROM A WHERE A.pid = 413032;
SELECT * FROM @t A
INNER JOIN B ON A.id = B.pid
INNER JOIN B C ON C.pid = B.pid
ORDER BY c.C1 DESC
OPTION (USE HINT('DISABLE_DEFERRED_COMPILATION_TV'));
```

SQL Server 2019



SQL Server Execution Times:

CPU time = 416 ms, elapsed time = **109** ms

Table Variable Deferred Compilation - Conclusion

- Designed to address cardinality issues caused by fixed estimation:
  - Nested Loop Joins where Hash Joins are more appropriate
  - Memory grant underestimation issues
- Better estimation for execution plans for new queries
- Excellent for new code (you have now 2 behaviors of TV)
- Prone to Parameter Sniffing
- Can break existing workarounds and dramatically degrade performance for some queries!!!

7

Approximate Processing

Approximate Query Processing

- No interaction with the existing code
  - all functions are new
- Aggregating across large datasets where responsiveness is more critical than absolute precision
- AQP in SQL Server is about reducing resources
  - it is more about reducing memory than about the speed of the query
- SQL Server 2019
  - a single function APPROX_DISTINCT_COUNT
- SQL Server 2022
  - two more functions APPROX_PERCENTILE_CONT and APPROX_PERCENTILE_DISC

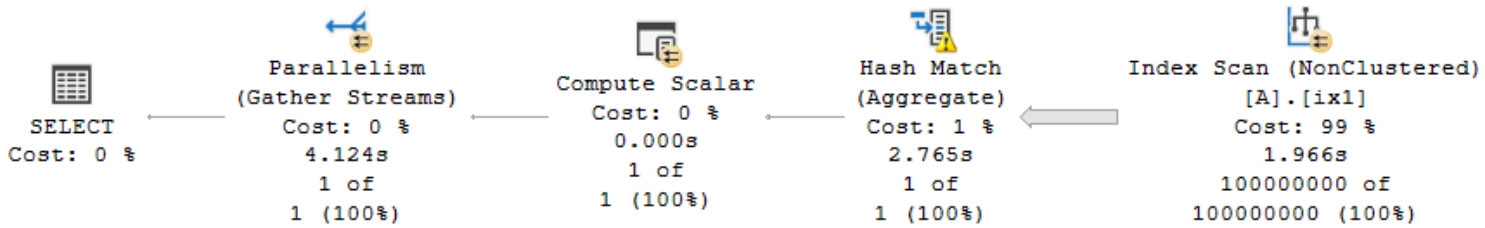
APPROX_DISTINCT_COUNT

- It uses significantly less memory resources
- Error from the precise COUNT DISTINCT equivalent within 2% for most workloads
- Great performance for data sets with a high number of distinct values
- Implemented HyperLogLog algorithm
- It’s more about used resources then about the speed
- Documentation for HyperLogLog
- http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf

```sql
--Table A 100M rows
SELECT COUNT(DISTINCT(pid)) FROM dbo.A;
SELECT APPROX_COUNT_DISTINCT(pid) FROM dbo.A;
```

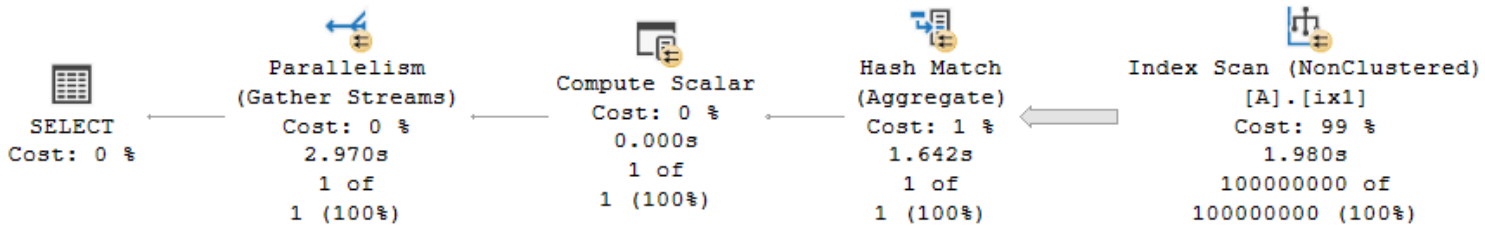
```
SQL Server Execution Times:
  CPU time = 15735 ms,  elapsed time = 2493 ms.

SQL Server Execution Times:
  CPU time = 12360 ms,  elapsed time = 1988 ms.
```

Query 1: Query cost (relative to the batch): 50%
SELECT COUNT(DISTINCT(pid)) FROM dbo.A



SELECT	
Cached plan size	32 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	8
Estimated Subtree Cost	156.691
Memory Grant	411864
Estimated Number of Rows	1

Query 2: Query cost (relative to the batch): 50%
SELECT APPROX_COUNT_DISTINCT(pid) FROM dbo.A



SELECT	
Cached plan size	40 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	8
Estimated Subtree Cost	156.691
Memory Grant	24776
Estimated Number of Rows	1

8

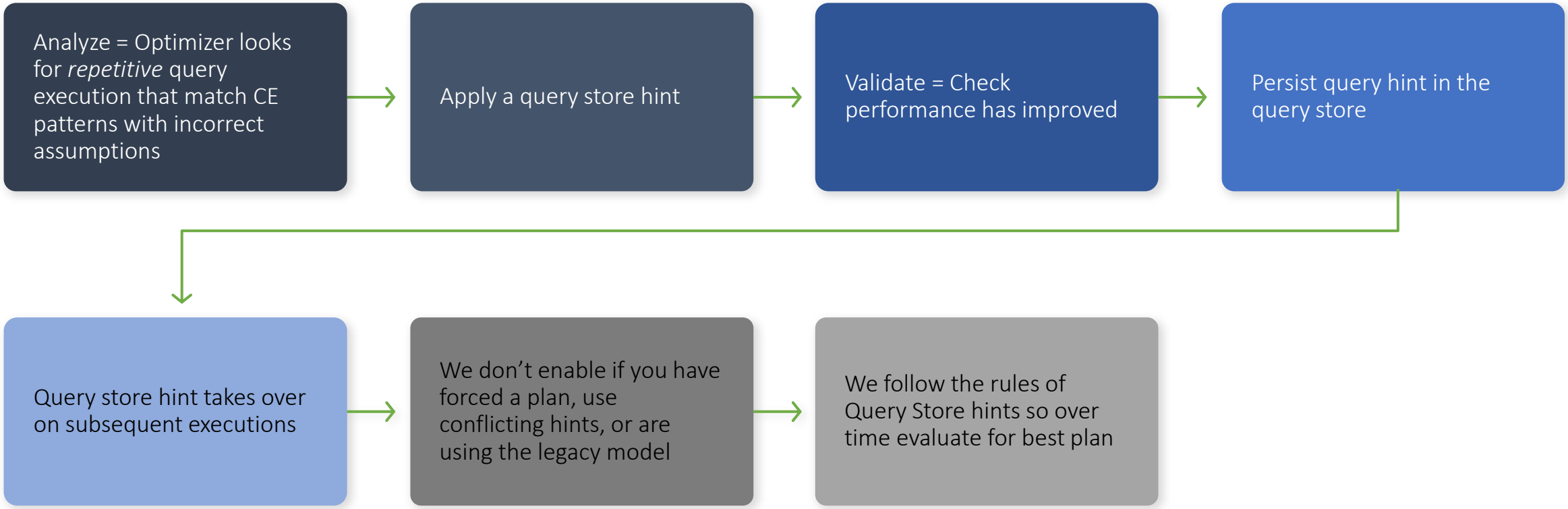
Cardinality Estimation Feedback

Cardinality Estimation Feedback

- For queries that execute frequently the optimizer tries to execute the query by using other assumptions of Cardinality Estimation models
  - Row Goals,
  - Predicate independence/correlation
  - Join containment (simple or base)

- Query Store must be enabled

Cardinality Estimation (CE) Feedback



*** Microsoft slide**

CE Model Related Assumptions

- Correlation
  - Fully independent (ASSUME_FULL_INDEPENDENCE_FOR_FILTER_ESTIMATES)
  - Partially correlated (ASSUME_PARTIAL_CORRELATION_FOR_FILTER_ESTIMATES)
  - Fully correlated (ASSUME_MIN_SELECTIVITY_FOR_FILTER_ESTIMATES)

- Join Containment
  - Simple containment (ASSUME_JOIN_PREDICATE_DEPENDS_ON_FILTERS)
  - Base containment /default with new CE)

- Optimizer row goal
  - DISABLE_OPTIMIZER_ROWGOAL

Corelation

- A = @par1 AND B = @par2
  - P(A = @par1) = 30%
  - P(AND B = @par2) = 30%

- ASSUME_FULL_INDEPENDENCE_FOR_FILTER_ESTIMATES
  - P resulted = 9% (0.3 * 0.3)
- ASSUME_PARTIAL_CORRELATION_FOR_FILTER_ESTIMATES
  - P resulted = 16% (SQRT(0.3) * 0.3)
- ASSUME_MIN_SELECTIVITY_FOR_FILTER_ESTIMATES
  - P resulted = 30% (MIN(0.3, 0.3))

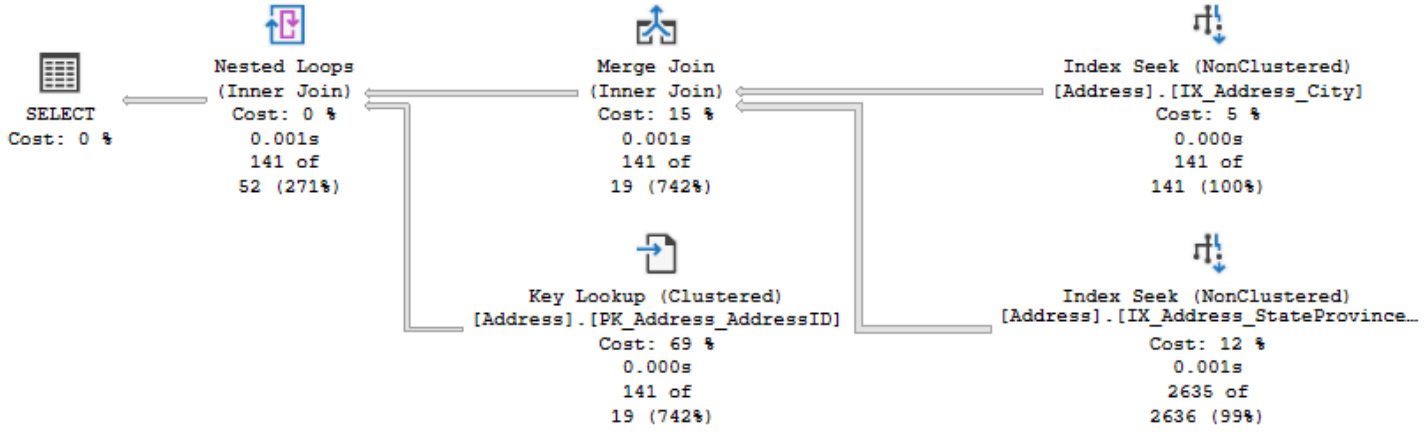
CE Feedback Example

```
SELECT AddressID, AddressLine1, AddressLine2,StateProvinceID
FROM Person.Address
WHERE StateProvinceID = 79 AND City = N'Seattle';
GO 15
```

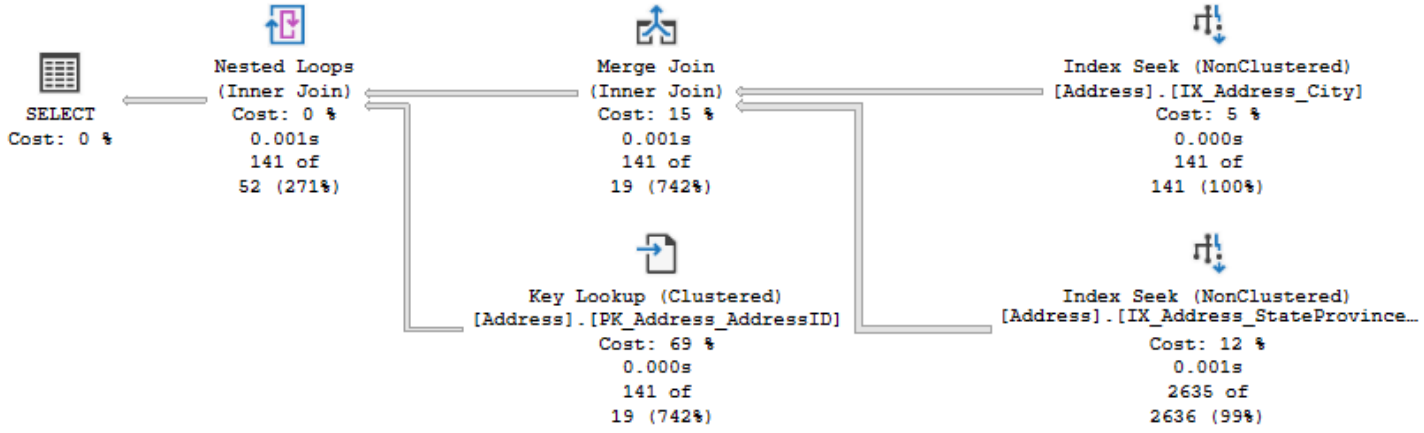
133 %

Results | Messages | Execution plan

Query 14: Query cost (relative to the batch): 7%
SELECT [AddressID],[AddressLine1],[AddressLine2],[StateProvinceID] FROM [Person].[Address] WHERE [StateProvinceID]=@1 AND [City]=@2



Query 15: Query cost (relative to the batch): 7%
SELECT [AddressID],[AddressLine1],[AddressLine2],[StateProvinceID] FROM [Person].[Address] WHERE [StateProvinceID]=@1 AND [City]=@2



CE Feedback Example

- After 16th execution SQL Server is considering CE Feedback

```
SELECT * FROM sys.query_store_plan_feedback;
```

133 %

Results | Messages

	plan_feedback_id	plan_id	feature_id	feature_desc	feedback_data	state	state_desc	create_time	last_updated_time
1	22	1	1	CE Feedback	{"Feedback hints":""}	2	PENDING_VALIDATION	2023-06-13 10:41:21.6700000 +00:00	2023-06-13 10:41:21.6700000 +00:00

CE Feedback Example

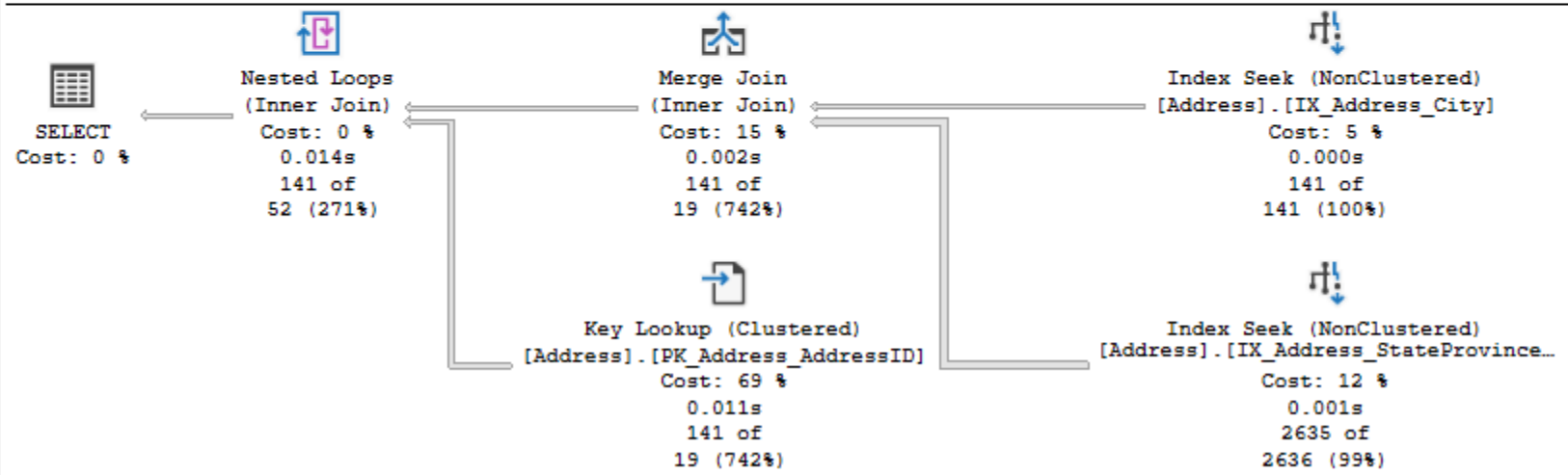
- 16th execution

```
SELECT AddressID, AddressLine1, AddressLine2,StateProvinceID
FROM Person.Address
WHERE StateProvinceID = 79 AND City = N'Seattle';
```

133 %

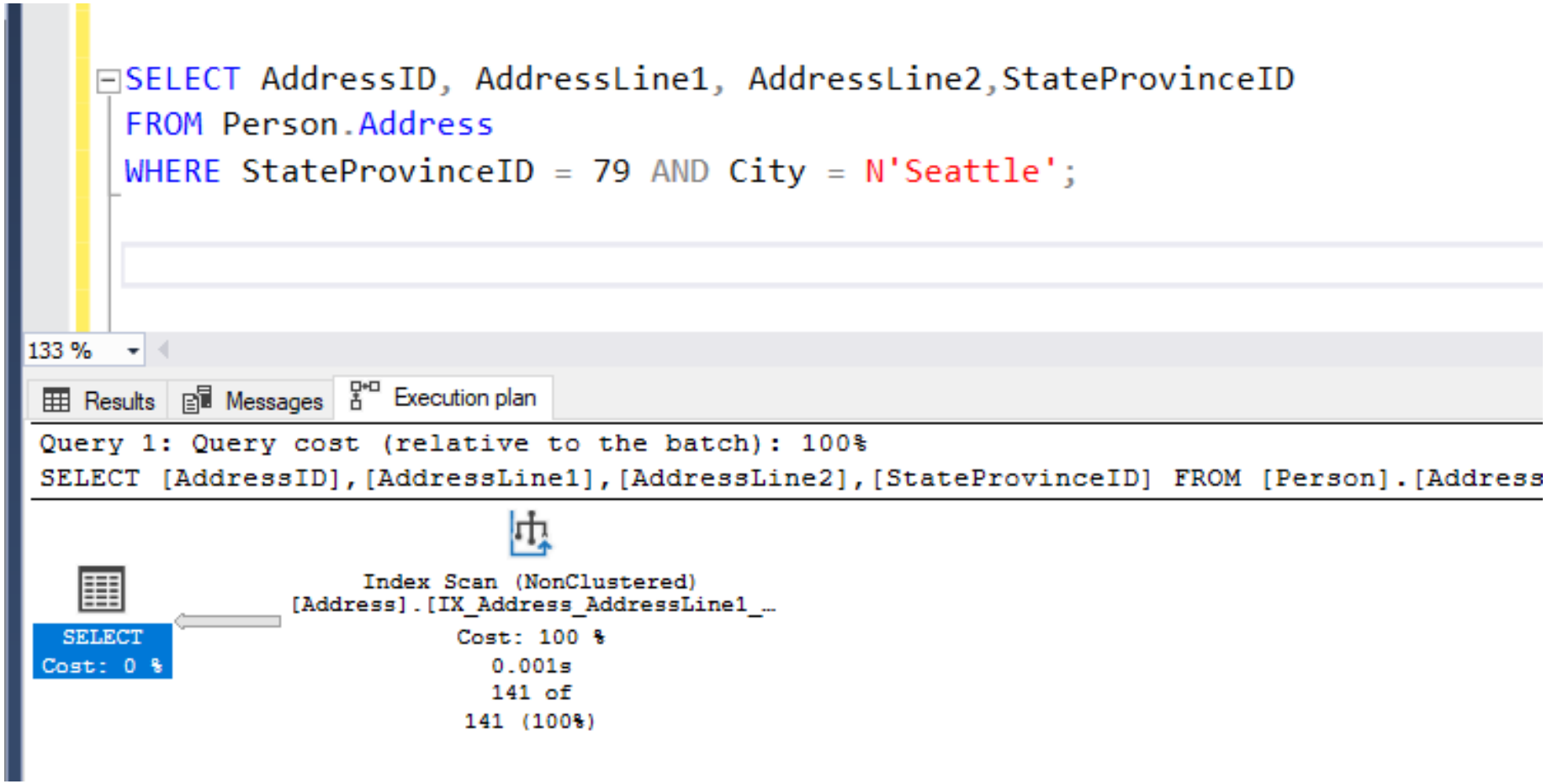
Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT [AddressID],[AddressLine1],[AddressLine2],[StateProvinceID] FROM [Person].[Address] WHERE [StateProvinceID]=@1 AND [City]=@2
```



CE Feedback Example

- 17th execution => changed plan



CE Feedback Example

- Verification of newly used assumption
- Using Query Store hints
- Used hint ASSUME_MIN_SELECTIVITY_FOR_FILTER_ESTIMATES

```
SELECT * FROM sys.query_store_plan_feedback;
SELECT * FROM sys.query_store_query_hints;
```

133 %

Results | Messages

	plan_feedback_id	plan_id	feature_id	feature_desc	feedback_data	state	state_desc	create_time	last_updated_time
1	28	1	1	CE Feedback	{"Feedback hints":"Min selectivity"}	5	VERIFICATION_PASSED	2023-06-13 10:47:50.5300000 +00:00	2023-06-13 10:47:54.4830000 +00:00

	query_hint_id	query_id	replica_group_id	query_hint_text	last_query_hint_failure_reason	last_query_hint_failure_reason_desc	query_hint_failure_count	source	source_desc	comment
1	4	1	1	OPTION(USE HINT('ASSUME_MIN_SELECTIVITY_FOR_FILTER_ESTIMATES'))	0	NONE	0	1	CE feedback	NULL

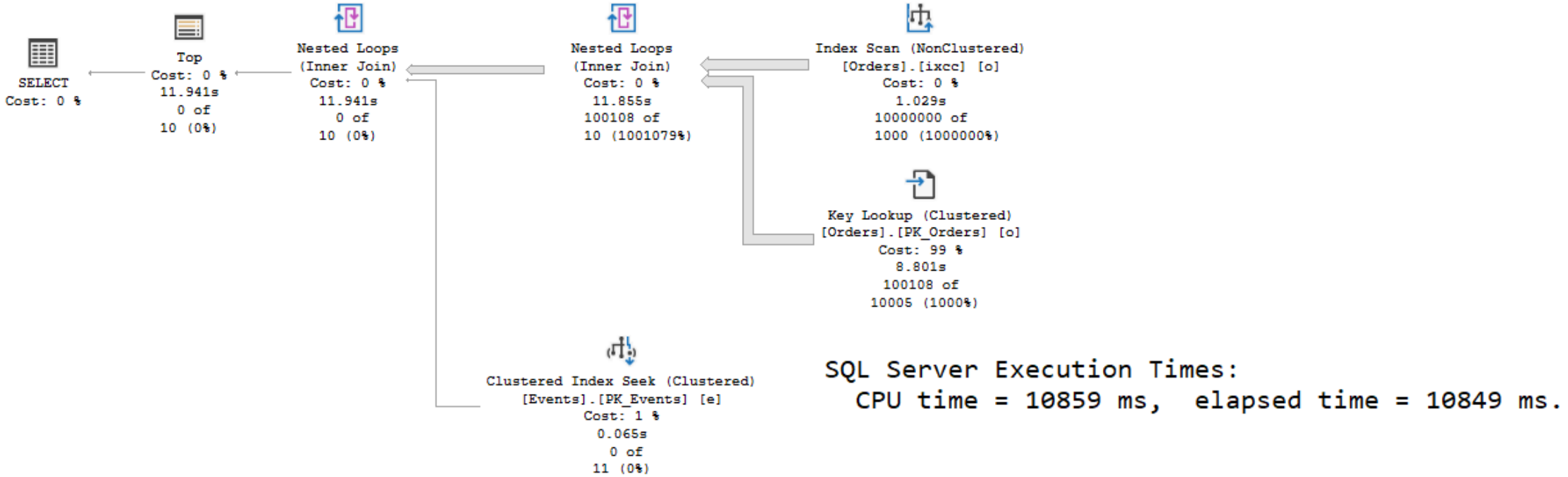
CE Feedback example that does not work

```
SELECT TOP (10) o.DeliveryDate
FROM
    Orders AS o
    INNER JOIN Events AS e ON o.ID = e.ID
WHERE
    o.OrderDate >= '20220502'
ORDER BY DeliveryDate
```

133 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT TOP (10) o.DeliveryDate FROM Orders AS o INNER JOIN Events AS e ON o.ID = e.ID WHERE o.OrderDate >= '20220502' ORDER BY DeliveryDate
```




```
SQL Server Execution Times:
  CPU time = 10859 ms,  elapsed time = 10849 ms.
```

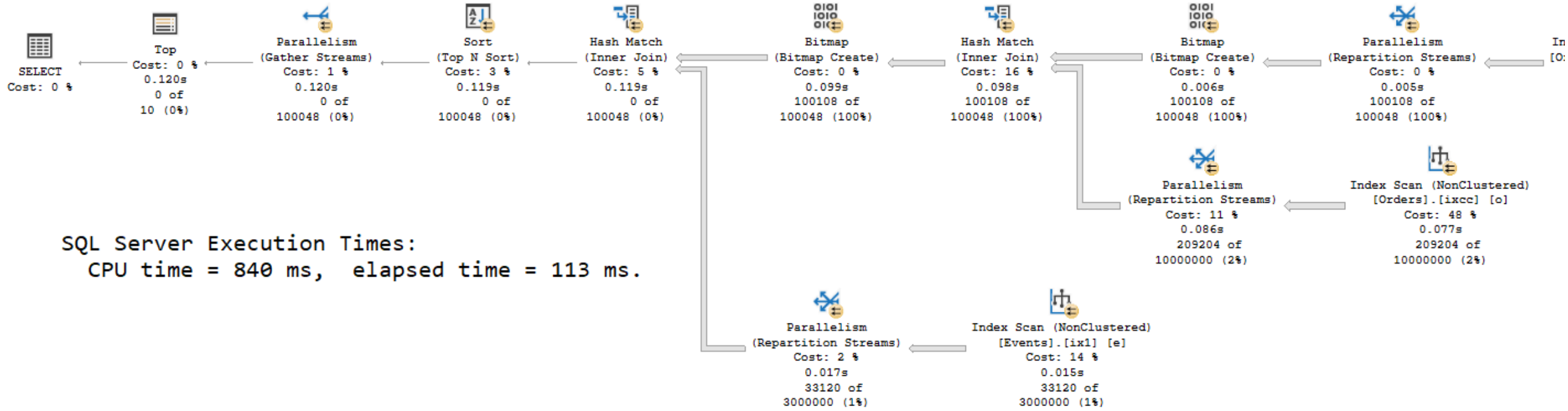
CE Feedback example that does not work

```sql
SELECT TOP (10) o.DeliveryDate
FROM
    Orders AS o
    INNER JOIN Events AS e ON o.ID = e.ID
WHERE
    o.OrderDate >= '20220502'
ORDER BY DeliveryDate
OPTION(USE HINT('DISABLE_OPTIMIZER_ROWGOAL'))
```

133 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT TOP (10) o.DeliveryDate FROM Orders AS o INNER JOIN Events AS e ON o.ID = e.ID WHERE o.OrderDate >= '20220502' ORDER BY DeliveryDate OPTION(USE HINT('DISABLE_OPTI…
Missing Index (Impact 49.4359): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Orders] ([OrderDate]) INCLUDE ([DeliveryDate])
```




```
SQL Server Execution Times:
  CPU time = 840 ms,  elapsed time = 113 ms.
```

What CE Feedback does not try (by design)

- It does not try the legacy CE hints
  - FORCE_LEGACY_CARDINALITY_ESTIMATION
  - QUERY_OPTIMIZER_COMPATIBILITY_LEVEL_110

- You still need to tune such queries!

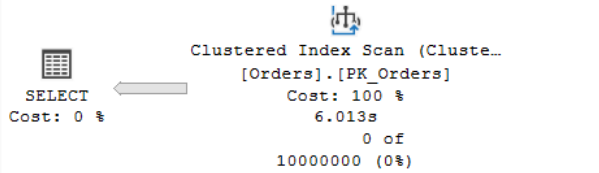
What CE Feedback does not try (by design)

```
SELECT * FROM Orders WHERE Status IN (0, 3);
GO
SELECT * FROM Orders WHERE Status IN (0,3)
OPTION(USE HINT('FORCE_LEGACY_CARDINALITY_ESTIMATION'))
```

176 %

Results | Messages | Execution plan

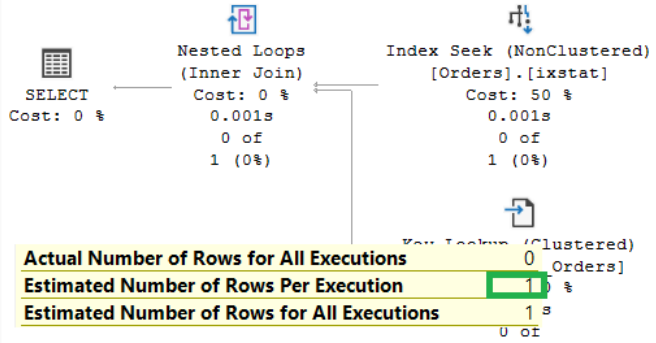
```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM Orders WHERE Status IN (0, 3)
Missing Index (Impact 64.0877): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Orders] ([Status]) INCLUDE ([CustomerId]
```



Actual Number of Rows for All Executions	0
Actual Number of Rows Read	10000000
Estimated Number of Rows Per Execution	10000000

```
Table 'Orders'. Scan count 1, logical reads 715444

 SQL Server Execution Times:
   CPU time = 2609 ms,  elapsed time = 6096 ms.
```

```
Query 2: Query cost (relative to the batch): 0%
SELECT * FROM Orders WHERE Status IN (0,3) OPTION(USE HINT('FORCE_LEGACY_CARDINALITY_ESTIMATION'))
```



Actual Number of Rows for All Executions	0
Estimated Number of Rows Per Execution	1
Estimated Number of Rows for All Executions	1

```
Table 'Orders'. Scan count 2, logical reads 6

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 22 ms.
```

CE Feedback - Conclusion

- Automatically improves performance by playing with CE model assumptions
- I did not (yet) find regressions (I have a limited number of servers in 2022)
- I found queries which are not improved (row goal optimization)
- Criteria for eligibility and measuring performance improvement not documented
- Requires CL 160
- Requires Query Store
- Enterprise feature

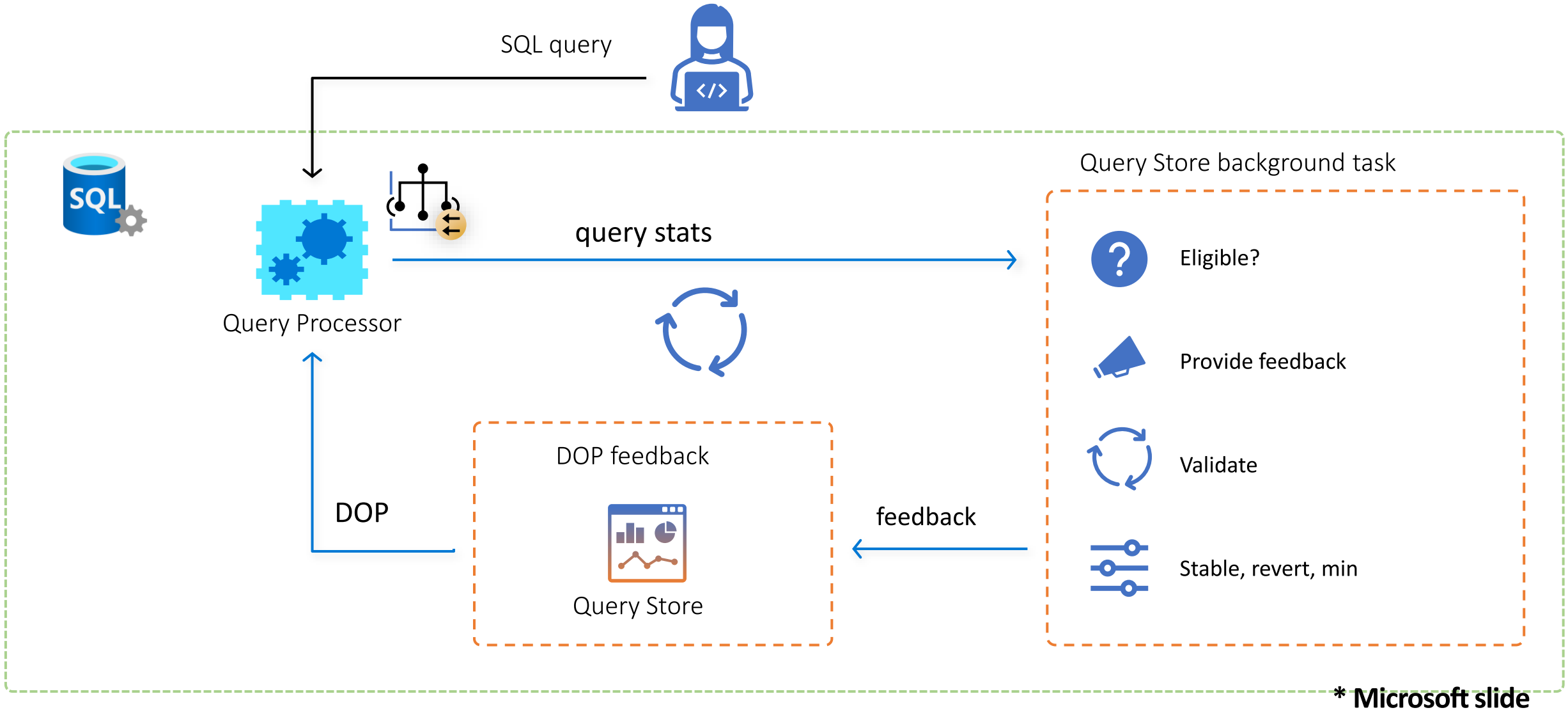
9

DOP Feedback

Degree of Parallelism Feedback

- improve query performance by identifying parallelism inefficiencies for repeating queries, based on elapsed time and waits
- By lowering DOP for the next execution (until MAXDOP = 2)
- self-adjusts DOP to avoid when excessive parallelism can cause performance issues
- Query Store must be enabled
- enable the DOP_FEEDBACK database scoped configuration in a database
- Compat Level 160 required
- Only verified feedback is persisted. If the adjusted DOP results in a performance regression, DOP feedback will go back to the last known good DOP.
- DISABLE_DOP_FEEDBACK query hint

Degree of Parallelism Feedback



*** Microsoft slide**

DOP Example by Microsoft

- https://github.com/microsoft/sqlworkshops-sql2022workshop/tree/main/sql2022workshop/03_BuiltinQueryIntelligence/dopfeedback

DOP Feedback in SQL Server 2022

This demo will show you how to see how to get consistent performance with less CPU resources for queries that require parallel operators

Pre-requisites

- SQL Server 2022 Evaluation or Developer Edition
- VM or computer with 8 CPUs and at least 24Gb RAM.

> **Note:** This exercise is very CPU sensitive. You should execute this exercise on a VM or computer that is "dedicated" to SQL Server and has fast I/O storage. Slow I/O or slow CPU speeds may affect the ability to see the results of this exercise.

- SQL Server Management Studio (SSMS). The latest 18.x build or 19.x build will work.
- Download **ostress.exe** from https://aka.ms/ostress. Install using the RMLSetup.msi file that is downloaded. Use all defaults.

> **Note:** If you are using a named instance you will need to edit **workload_index_scan_users.cmd** to include a -S.`<instance name>`

> **IMPORTANT:** The workload requires queries to have a duration of *at least 10 seconds per execution* to be eligible for DOP feedback. You can use the XEProfiler in SSMS to see how fast your queries are executing. If you run the workload and don't see queries are eligible you can try to edit workload_index_scan_users.cmd to add more users using the -n parameter by providing a value > 1.

DoP Feedback - Conclusion

- Not enough experience from my side, I hope it works as described, if so, it is a useful feature
  - Limited number of queries where it can be beneficial
  - Large servers and large tables and inefficient written queries
- Less worrying about the MaxDop setting
- Not documented
- Requires 160
- Requires Query Store
- Enterprise Edition feature

10

Parameter Sensitive Plan Optimization

Parameter Sniffing Example

- 1 and 2 are VIP customers

```
SELECT COUNT(*) FROM Orders WHERE CustomerId = 1; --1102154
SELECT COUNT(*) FROM Orders WHERE CustomerId = 2; --901868
SELECT COUNT(*) FROM Orders WHERE CustomerId = 4796; --2
GO

--sample stored procedure
CREATE OR ALTER PROCEDURE dbo.GetOrdersTotal
@CustomerId INT
AS
BEGIN
    SELECT SUM(Amount*1.0) AS SumAmount, COUNT(*) AS CntOrders
    FROM dbo.Orders
    WHERE CustomerId = @CustomerId
END
GO
```

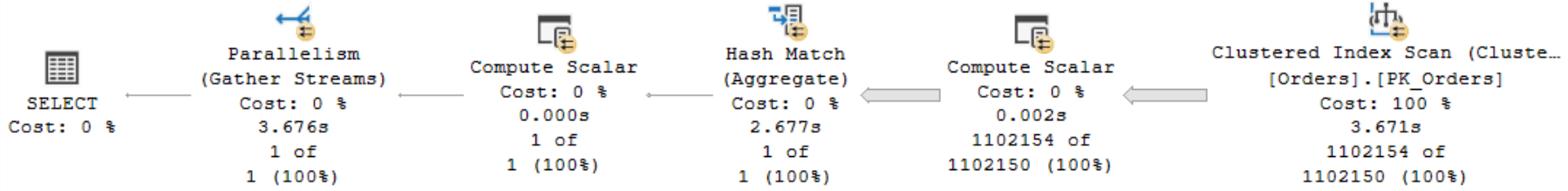
SQL Server 2019 – the first call with 1

```
ALTER DATABASE db2 SET COMPATIBILITY_LEVEL = 150;
GO
EXEC GetOrdersTotal 1;
EXEC GetOrdersTotal 4796;
GO
```

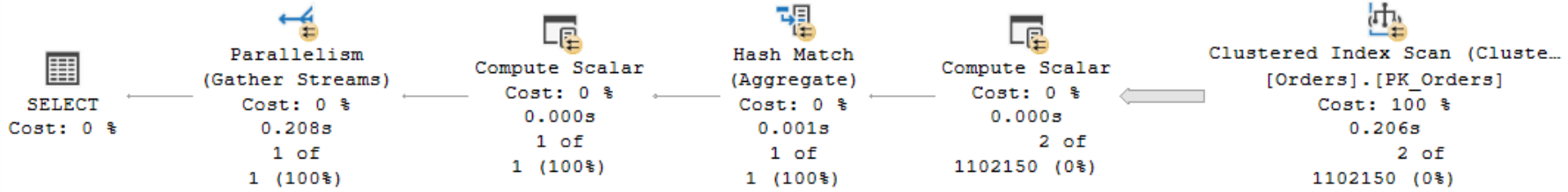
176 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 50%
SELECT SUM(Amount*1.0) AS SumAmount, COUNT(*) AS CntOrders FROM dbo.Orders WHERE CustomerId = @CustomerId
Missing Index (Impact 99.9906): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Orders] ([CustomerId]) INCLUDE ([Amount])
```




```
Query 2: Query cost (relative to the batch): 50%
SELECT SUM(Amount*1.0) AS SumAmount, COUNT(*) AS CntOrders FROM dbo.Orders WHERE CustomerId = @CustomerId
Missing Index (Impact 99.9906): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Orders] ([CustomerId]) INCLUDE ([Amount])
```



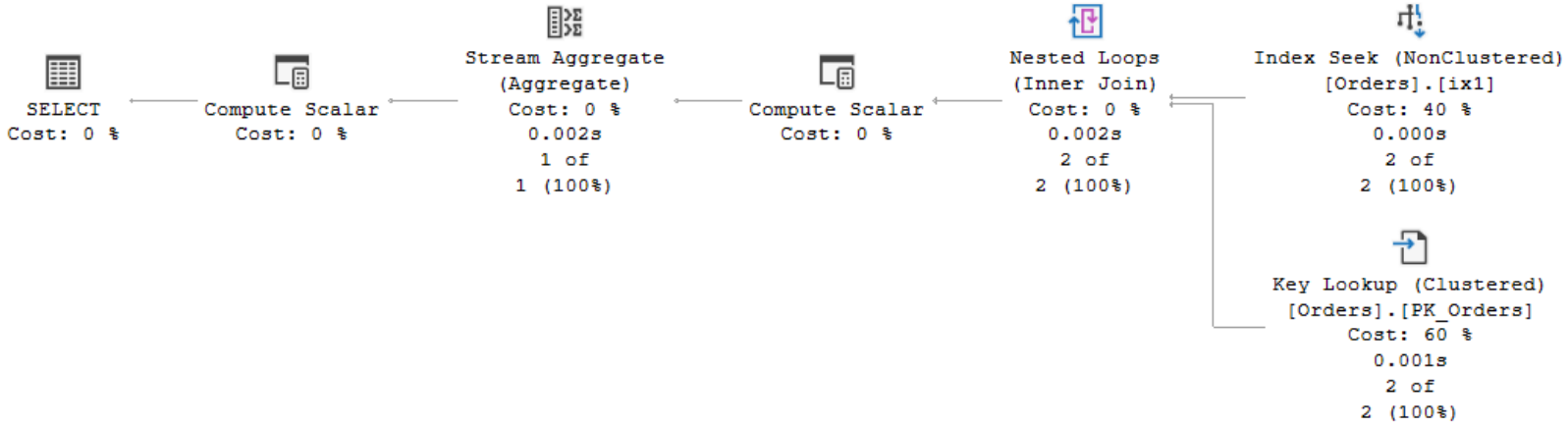
SQL Server 2019 – the first call with 4796

```
ALTER DATABASE db2 SET COMPATIBILITY_LEVEL = 150;
GO
EXEC GetOrdersTotal 4796;
EXEC GetOrdersTotal 1;
GO
```

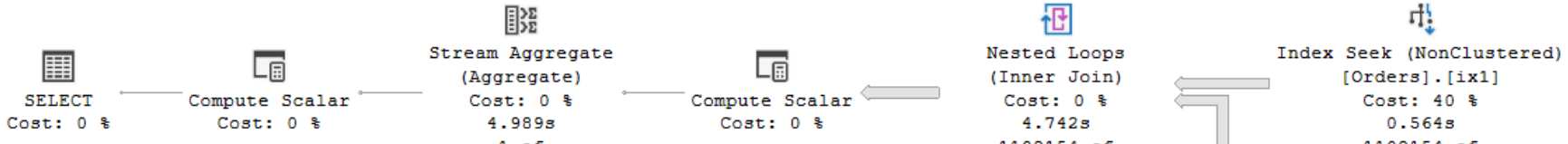
176 %

Results | Messages | Execution plan

Query 1: Query cost (relative to the batch): 50%
SELECT SUM(Amount*1.0) AS SumAmount, COUNT(*) AS CntOrders FROM dbo.Orders WHERE CustomerId = @CustomerId



Query 2: Query cost (relative to the batch): 50%
SELECT SUM(Amount*1.0) AS SumAmount, COUNT(*) AS CntOrders FROM dbo.Orders WHERE CustomerId = @CustomerId



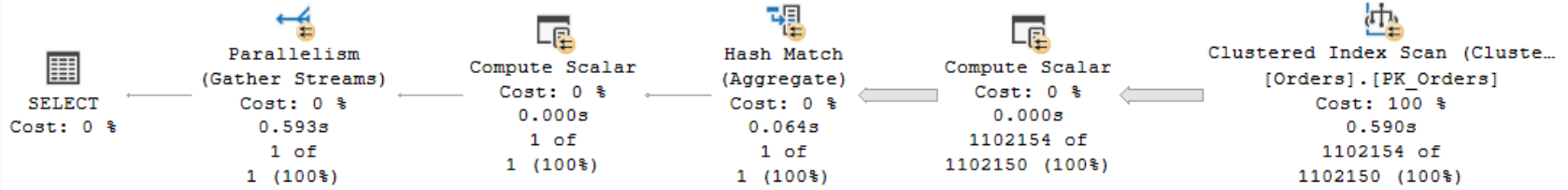
SQL Server 2022 – two different plans!

```
ALTER DATABASE db2 SET COMPATIBILITY_LEVEL = 160;
GO
EXEC GetOrdersTotal 1;
EXEC GetOrdersTotal 4796;
GO
```

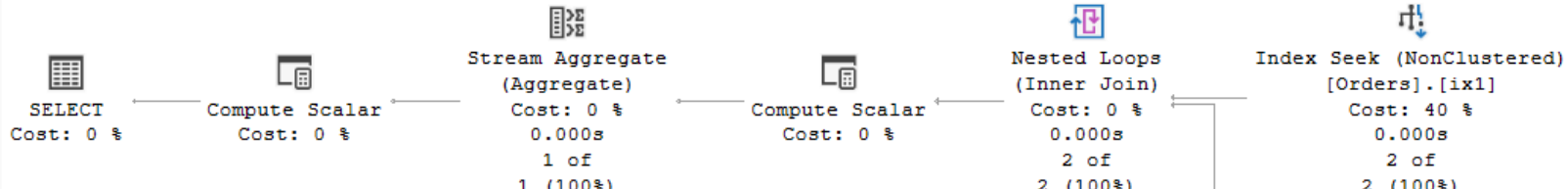
176 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT SUM(Amount*1.0) AS SumAmount, COUNT(*) AS CntOrders FROM dbo.Orders WHERE CustomerId = @CustomerId option (PLAN PER VALUE(ObjectID = 1269579561, Query
Missing Index (Impact 99.9906): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Orders] ([CustomerId]) INCLUDE ([Amount])
```




```
Query 2: Query cost (relative to the batch): 0%
SELECT SUM(Amount*1.0) AS SumAmount, COUNT(*) AS CntOrders FROM dbo.Orders WHERE CustomerId = @CustomerId option (PLAN PER VALUE(ObjectID = 1269579561, Query
```



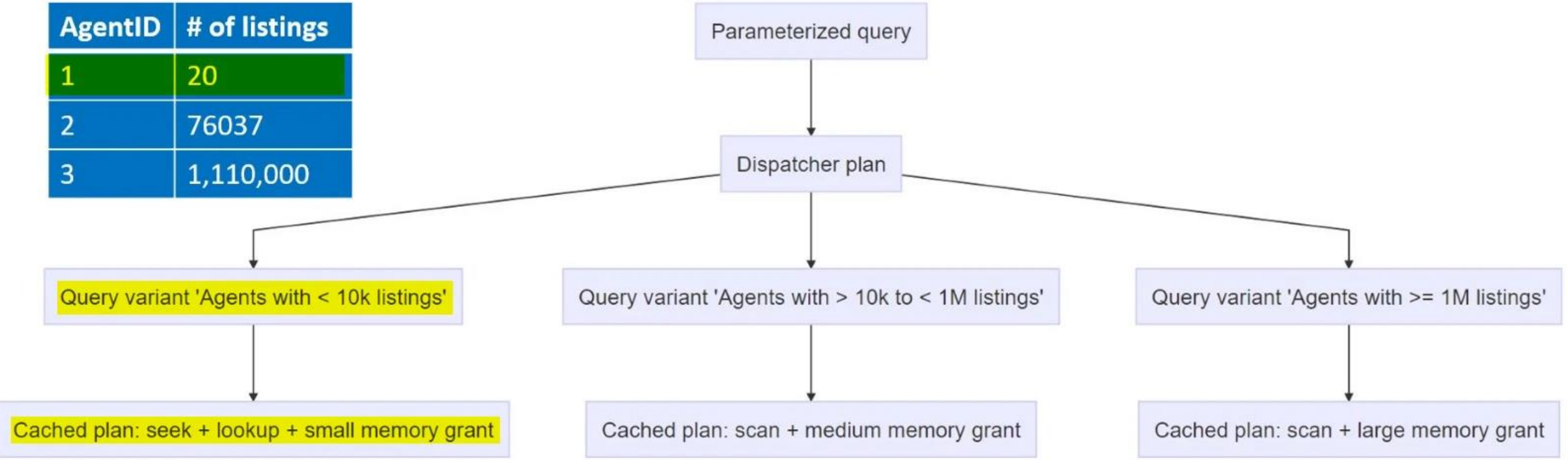
PSP Optimization

- Allows **multiple** active execution plans to be cached for a stored procedure or parameterized query
- SQL Server decides how many plans are stored and when PSP is activated
- It works only simple predicates **Col = @Par** and only for the equality operator
- Requires CL 160

PSP Optimization

- Three different plans depending on the value of the input parameter

AgentID	# of listings
1	20
2	76037
3	1,110,000



* Microsoft slide

Parameter Sniffing in SQL Server 2022

- Optional parameters **PSP Opt.** 

```
@CustomerId INT = NULL, @OrderDate DATETIME = NULL
```

- Equal operator and uneven distribution (outliers)

```
CustomerId = @CustomerId
```

PSP Opt.  **PARTIALLY!**

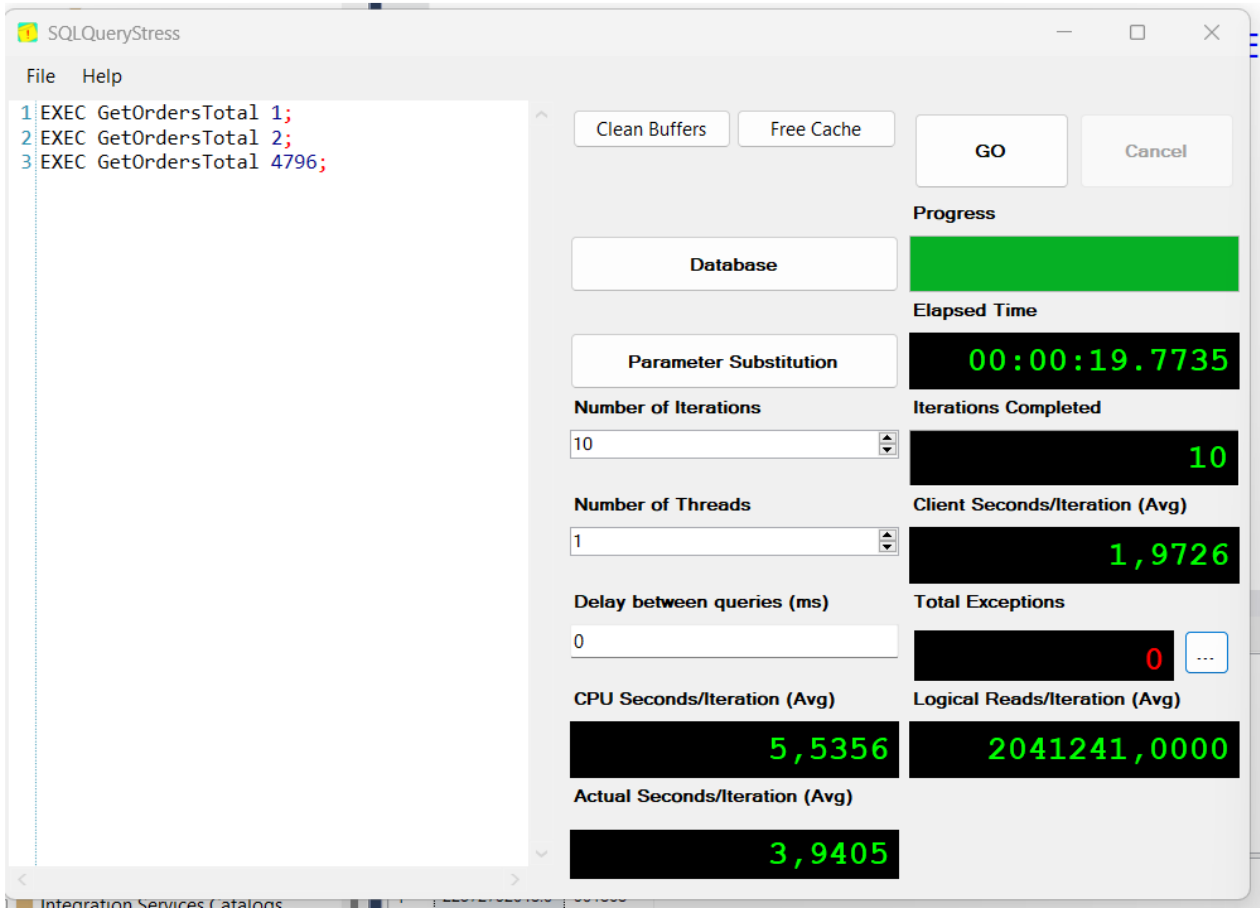
- Non-equal operators

```
OrderDate >= @Date
```

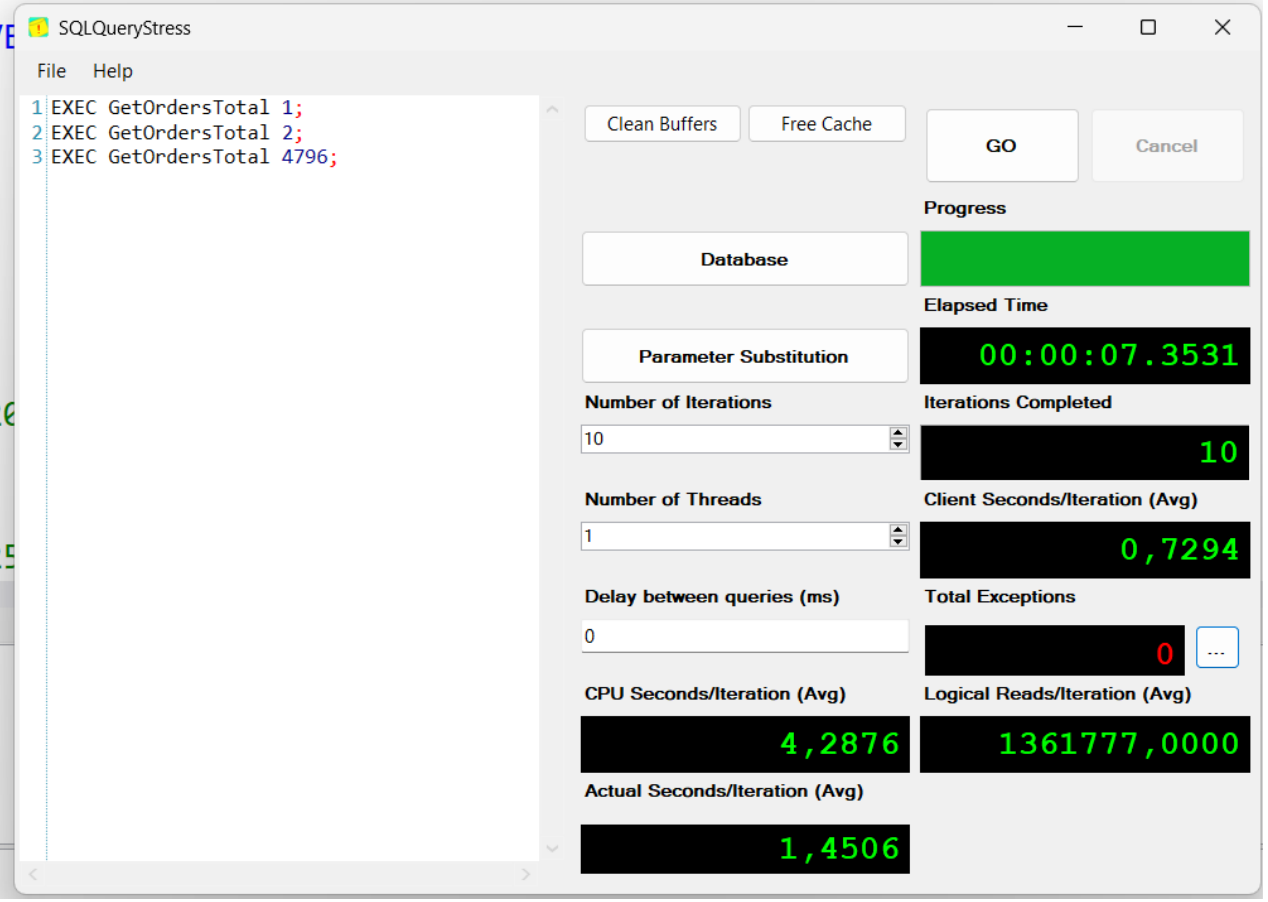
PSP Opt. 

Performance Improvement with PSP Opt.

- CL 150

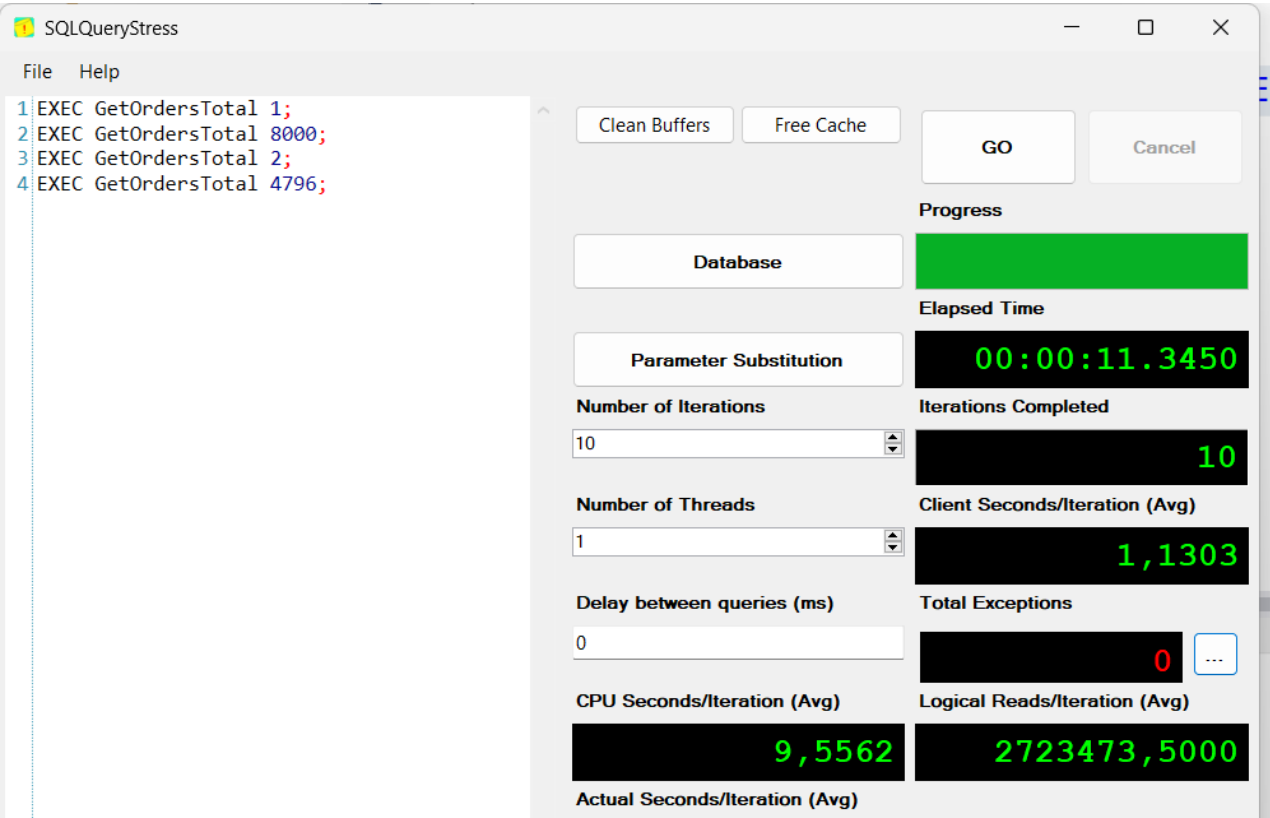


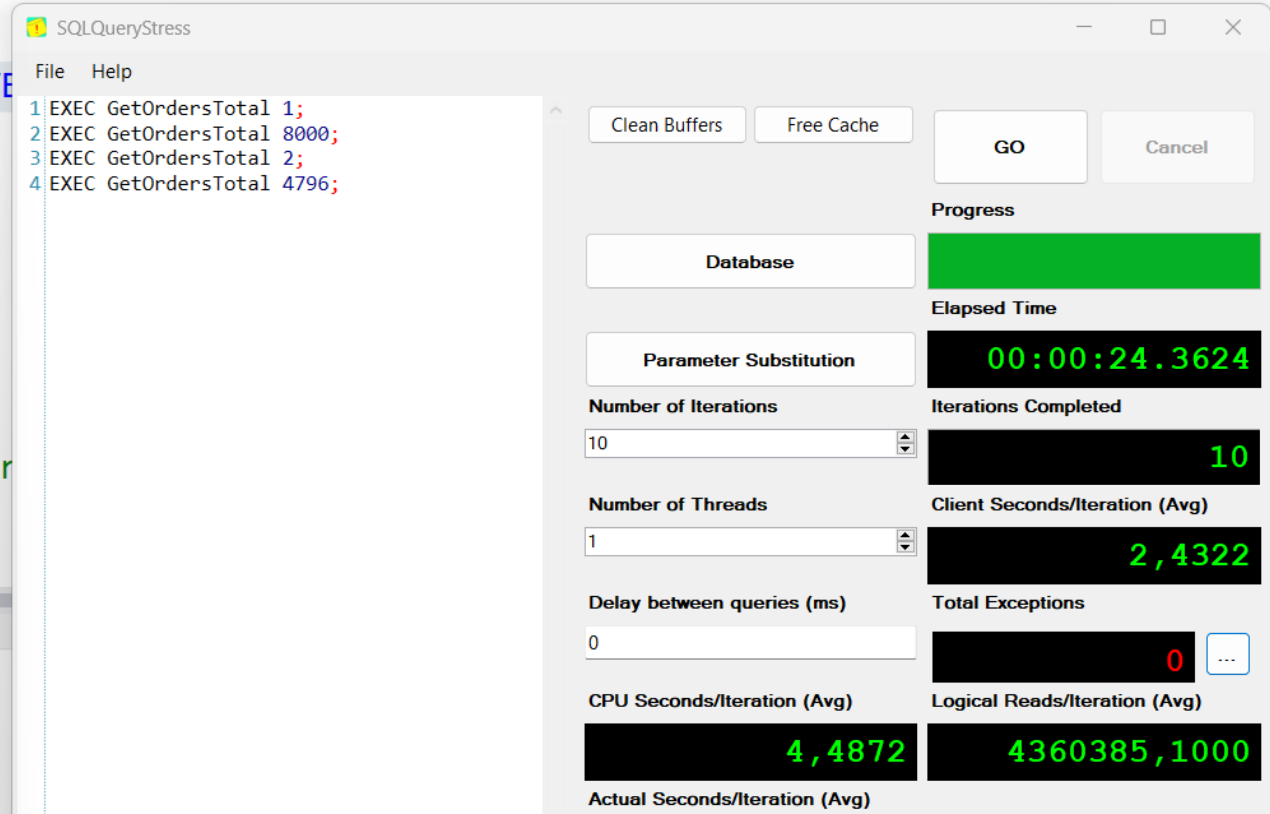
- CL 160



Performance Regression with PSP Opt.

- Secondary Parameter Sniffing Possible!
- Customers with 101 order and 999.999 orders have the same plan!
  - CL 150
  - CL 160





Get execution plan from cache in SQL Server 2019

```
DECLARE @sp_name NVARCHAR(255) = N'GetOrdersTotal';
SELECT
    ep.usecounts,
    ep.cacheobjtype,
    ep.objtype,
    q.text AS query_text,
    pl.query_plan,
    ep.size_in_bytes
FROM
    sys.dm_exec_cached_plans ep
    CROSS APPLY sys.dm_exec_sql_text(ep.plan_handle) q
    CROSS APPLY sys.dm_exec_query_plan(ep.plan_handle) pl
WHERE
    ep.objtype = 'Proc' AND q.text LIKE '%' + @sp_name + '%'
GO
-----------------------------------------------
```

133 %

Results | Messages

	usecounts	cacheobjtype	objtype	query_text	query_plan	size_in_bytes
1	2	Compiled Plan	Proc	CREATE PROCEDURE dbo.GetOrdersTotal @Customer...	<ShowPlanXML xmlns="http://schemas.microsoft.com...	65536

```
Query 1: Query cost (relative to the batch): 100%
CREATE PROCEDURE dbo.GetOrdersTotal @CustomerId INT AS BEGIN SELECT SUM(Amount*1.0) AS SumAmount, COUNT(*) AS CntOrders FROM dbo.Orders WHERE CustomerId = @CustomerId
Missing Index (Impact 99.9906): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Orders] ([CustomerId]) INCLUDE ([Amount])
```

Get execution plan from cache in SQL Server 2019

```sql
DECLARE @sp_name NVARCHAR(255) = N'GetOrdersTotal';
SELECT
    ep.usecounts,
    ep.cacheobjtype,
    ep.objtype,
    q.text AS query_text,
    pl.query_plan,
    ep.size_in_bytes
FROM
    sys.dm_exec_cached_plans ep
    CROSS APPLY sys.dm_exec_sql_text(ep.plan_handle) q
    CROSS APPLY sys.dm_exec_query_plan(ep.plan_handle) pl
WHERE
    ep.objtype = 'Proc' AND q.text LIKE '%' + @sp_name + '%'
GO
```

133 %

Results | Messages

	usecounts	cacheobjtype	objtype	query_text	query_plan	size_in_bytes
1	3	Compiled Plan	Proc	CREATE PROCEDURE dbo.GetOrdersTotal @Customer...	<ShowPlanXML xmlns="http://schemas.microsoft.com...	114688

```
Query 1: Query cost (relative to the batch): 100%
CREATE PROCEDURE dbo.GetOrdersTotal @CustomerId INT AS BEGIN SELECT SUM(Amount*1.0) AS SumAmount, COUNT(*) AS CntOrders FROM dbo.Orders WHERE CustomerId = @CustomerId
```

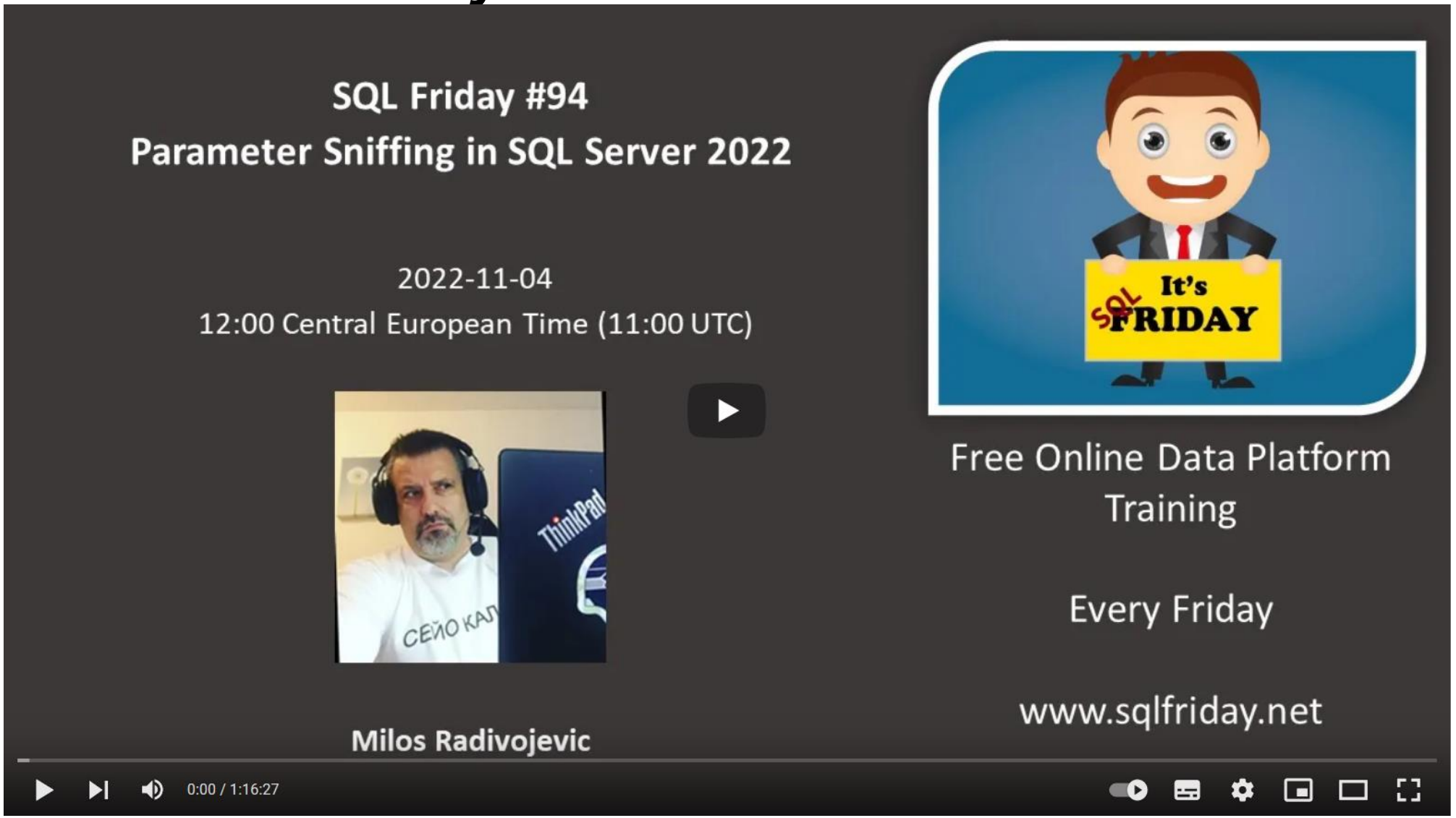
T-SQL

MULTIPLE PLAN
Cost: 0 %

Get execution plan from cache in SQL Server 2019

```
<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan" Version="1.564" Build="16.0.1000.6">
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementText="CREATE   PROCEDURE dbo.GetOrdersTotal&#xD;&#xA;@CustomerId INT&#xD;&#xA;AS&#xD;&#xA;BEGIN&#xD;&#xA;  SELECT SUM(Amount*1.0)
          <Dispatcher>
            <ParameterSensitivePredicate LowBoundary="100" HighBoundary="1e+06">
              <StatisticsInfo LastUpdate="2022-10-31T13:45:04.30" ModificationCount="0" SamplingPercent="100" Statistics="[ix1]" Table="[Orders]" Schema="[dbo
              <Predicate>
                <ScalarOperator ScalarString="[db2].[dbo].[Orders].[CustomerId]=[@CustomerId]">
                  <Compare CompareOp="EQ">
                    <ScalarOperator>
                      <Identifier>
                        <ColumnReference Database="[db2]" Schema="[dbo]" Table="[Orders]" Column="CustomerId" />
                      </Identifier>
                    </ScalarOperator>
                    <ScalarOperator>
                      <Identifier>
                        <ColumnReference Column="@CustomerId" />
                      </Identifier>
                    </ScalarOperator>
                  </Compare>
                </ScalarOperator>
              </Predicate>
            </ParameterSensitivePredicate>
          </Dispatcher>
        </StmtSimple>
      </Statements>
    </Batch>
  </BatchSequence>
</ShowPlanXML>
```

More info – my session about PSP at SQL Friday



- https://www.youtube.com/watch?v=Dy5HHnRrICY

PSP Optimization - Conclusion

- Parameter Sniffing issues will be reduced, but not eliminated
  - It does work only for predicates with Equality operator (A = @par1)
  - It does not work for optional parameters
  - It does not work for non-equality operators (A > @p1 or A BETWEEN @p1 AND @p2)
  - Secondary Parameter Sniffing is possible
- Beneficial for those who do not have time, people or know-how to deal with Parameter Sniffing issues
- It might break troubleshooting scripts and cause errors in monitoring tools
- Requires CL 160, Query Store is used, but not required

11

Optimized Plan Forcing

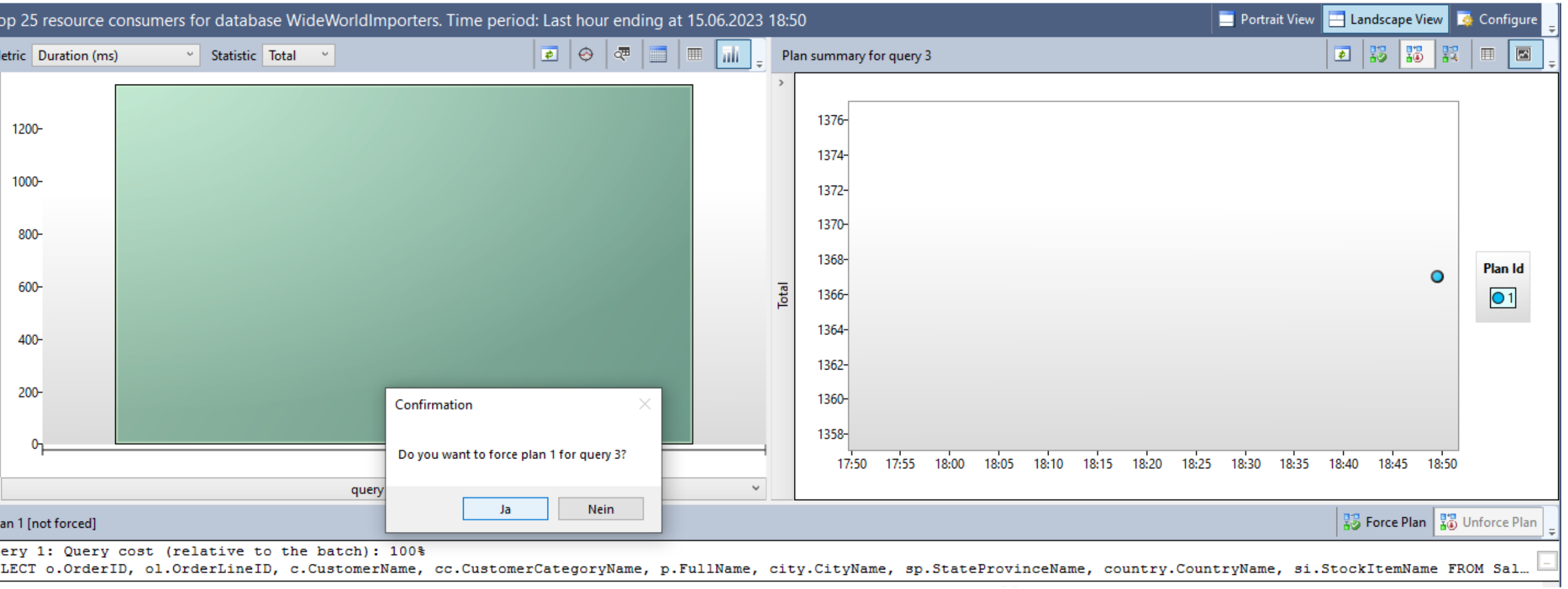
Optimized Plan Forcing

- Applicable only **for queries which plan is forced** in Query Store
  - When SQL Server creates a plan again for a query whose plan was forced, it does not go through all the optimization phases, but uses the so-called optimization reply script
- The epilogue is that it uses less CPU for compilation
- Only complex queries whose plan is forced will have a visible benefit
- Performance regressions are also possible
- Does not require CL 160

Improvement

Messages

```
SQL Server parse and compile time:
   CPU time = 78 ms, elapsed time = 88 ms.
```



Messages

```
SQL Server parse and compile time:
   CPU time = 43 ms, elapsed time = 43 ms.
```

Regression

```
/*                                                                              */
/********************************************************************************/
CREATE OR ALTER   PROCEDURE dbo.SPx1
@Quantity INT
AS
BEGIN
    SELECT *  FROM Sales.OrderLines WHERE Quantity = @Quantity
    AND OrderId IN (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,
         1809,1810,1811,1812,1813,1814,1815,1816,1817,1818,1819,1820,1821,1822,1823,1824,1825,1826,1827,1828,1829,1830,1831,1832,1833,1834,1835,1836,1837,1838,18
     3395,3396,3397,3398,3399,3400,3401,3402,3403,3404,3405,3406,3407,3408,3409,3410,3411,3412,3413,3414,3415,3416,3417,3418,3419,3420,3421,3422,3423,3424,3425,3
     4981,4982,4983,4984,4985,4986,4987,4988,4989,4990,4991,4992,4993,4994,4995,4996,4997,4998,4999,5000,5001,5002,5003,5004,5005,5006,5007,5008,5009,5010,5011,5
     6567,6568,6569,6570,6571,6572,6573,6574,6575,6576,6577,6578,6579,6580,6581,6582,6583,6584,6585,6586,6587,6588,6589,6590,6591,6592,6593,6594,6595,6596,6597,6
     8153,8154,8155,8156,8157,8158,8159,8160,8161,8162,8163,8164,8165,8166,8167,8168,8169,8170,8171,8172,8173,8174,8175,8176,8177,8178,8179,8180,8181,8182,8183,8
     9739,9740,9741,9742,9743,9744,9745,9746,9747,9748,9749,9750,9751,9752,9753,9754,9755,9756,9757,9758,9759,9760,9761,9762,9763,9764,9765,9766,9767,9768,9769,9
     )
END
GO
CREATE OR ALTER   PROCEDURE dbo.SPx2
@Quantity INT
AS
BEGIN
    SELECT *  FROM Sales.OrderLines WHERE Quantity = @Quantity
    AND OrderId IN (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,
         1809,1810,1811,1812,1813,1814,1815,1816,1817,1818,1819,1820,1821,1822,1823,1824,1825,1826,1827,1828,1829,1830,1831,1832,1833,1834,1835,1836,1837,1838,18
     3395,3396,3397,3398,3399,3400,3401,3402,3403,3404,3405,3406,3407,3408,3409,3410,3411,3412,3413,3414,3415,3416,3417,3418,3419,3420,3421,3422,3423,3424,3425,3
     4981,4982,4983,4984,4985,4986,4987,4988,4989,4990,4991,4992,4993,4994,4995,4996,4997,4998,4999,5000,5001,5002,5003,5004,5005,5006,5007,5008,5009,5010,5011,5
     6567,6568,6569,6570,6571,6572,6573,6574,6575,6576,6577,6578,6579,6580,6581,6582,6583,6584,6585,6586,6587,6588,6589,6590,6591,6592,6593,6594,6595,6596,6597,6
     8153,8154,8155,8156,8157,8158,8159,8160,8161,8162,8163,8164,8165,8166,8167,8168,8169,8170,8171,8172,8173,8174,8175,8176,8177,8178,8179,8180,8181,8182,8183,8
     9739,9740,9741,9742,9743,9744,9745,9746,9747,9748,9749,9750,9751,9752,9753,9754,9755,9756,9757,9758,9759,9760,9761,9762,9763,9764,9765,9766,9767,9768,9769,9
     )
    OPTION(USE HINT('DISABLE_OPTIMIZED_PLAN_FORCING') )
END
```

Regression

- Overhead of the optimization reply script *for some queries* outperforms benefits

```
        )

            OPTION(USE HINT('DISABLE_OPTIMIZED_PLAN_FORCING') )
END
GO

ALTER DATABASE SCOPED CONFIGURATION SET OPTIMIZED_PLAN_FORCING = ON;
GO
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
EXEC dbo.SPx1 200
GO
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
EXEC dbo.SPx2 200
GO
```

133 %

Messages

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 1828 ms, elapsed time = 1828 ms.

(216 rows affected)

 SQL Server Execution Times:
   CPU time = 16 ms,  elapsed time = 17 ms.

 SQL Server Execution Times:
   CPU time = 1844 ms,  elapsed time = 1846 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 750 ms, elapsed time = 758 ms.

(216 rows affected)

 SQL Server Execution Times:
   CPU time = 15 ms,  elapsed time = 17 ms.
```

Enterprise vs. Standard

Intelligent Database: approximate count distinct	✓	✓	✓	✓	✓
Intelligent Database: approximate percentile	✓	✓	✓	✓	✓
Intelligent Database: automatic tuning	✓	⊗	⊗	⊗	⊗
Intelligent Database: batch mode for row store [1]	✓	⊗	⊗	⊗	⊗
Intelligent Database: cardinality estimate feedback	✓	⊗	⊗	⊗	⊗
Intelligent Database: degree of parallelism feedback	✓	⊗	⊗	⊗	⊗
Intelligent Database: memory grant feedback persistence and percentile	✓	⊗	⊗	⊗	⊗
Intelligent Database: optimized plan forcing	✓	✓	✓	✓	✓
Intelligent Database: parameter sensitive plan optimization	✓	✓	✓	✓	✓
Intelligent Database: row mode memory grant feedback	✓	⊗	⊗	⊗	⊗
Intelligent Database: table variable deferred compilation	✓	✓	✓	✓	✓
Intelligent Database: scalar UDF inlining	✓	✓	✓	✓	✓
Intelligent Database: Batch mode adaptive joins	✓	⊗	⊗	⊗	⊗
Intelligent Database: Batch mode memory grant feedback	✓	⊗	⊗	⊗	⊗
Intelligent Database: Interleaved execution for multi-statement table valued functions	✓	✓	✓	✓	✓

Enterprise Edition Features

- Batch Mode Adaptive Join
- Memory Grant Feedback
- Batch Mode on RowStore
- Cardinality Estimation Feedback
- DOP Feedback