INTELLIGENT QUERY PROCESSING IN SQL SERVER 2019

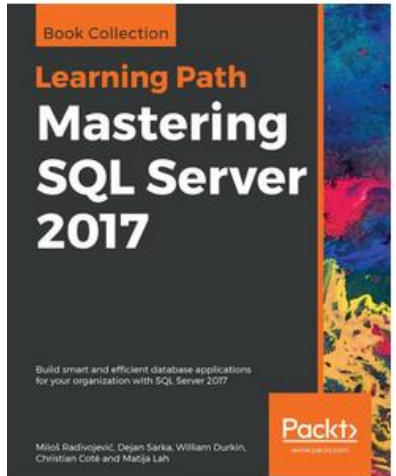
MILOŠ RADIVOJEVIĆ

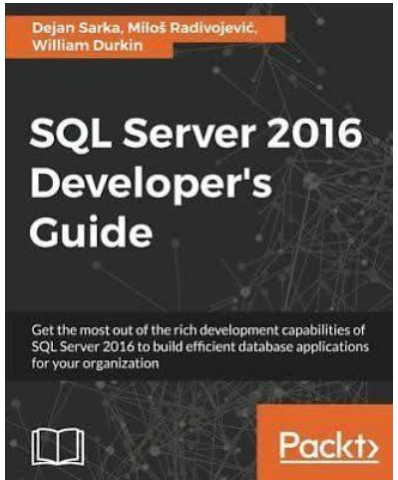
DATA PLATFORM MVP, BWIN GVC, AUSTRIA

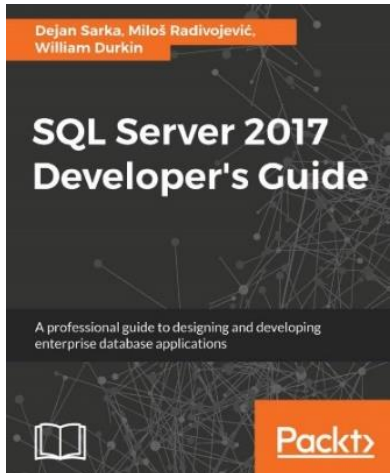
Miloš Radivojević

- Data Platform MVP
- Principal Database Consultant at bwin, Vienna, Austria
- Co-Founder: SQL Pass Austria
- Conference Speaker, Book Author

- Contact: *https://milossql.wordpress.com*

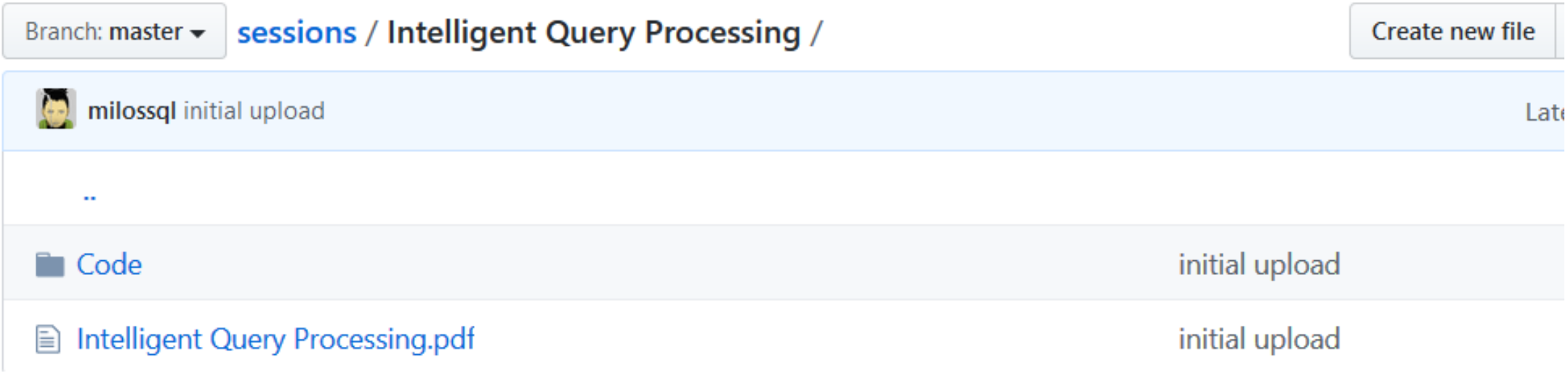






Slides and Code

- https://bit.ly/2yUglSb



Inefficient Execution Plans in SQL Server

- Execution plans are sometimes suboptimal

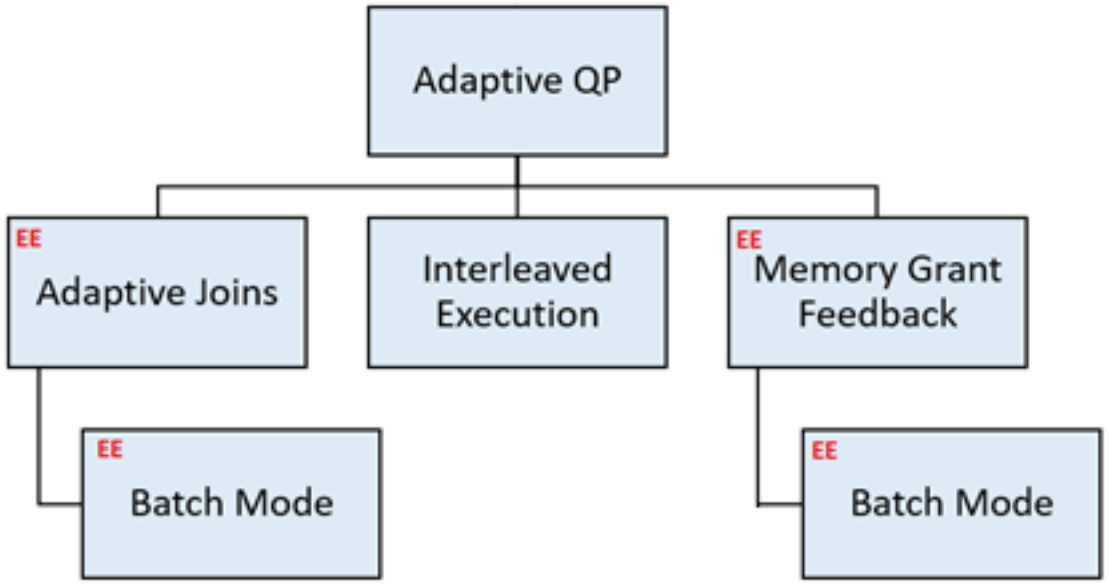
- Affected Queries:
  - Queries using table variables
  - Queries with scalar user-defined functions
  - Queries referencing multi-statement table valued functions
  - Complex queries
  - Queries with tables with skew data distribution

- Issues:
  - Inappropriate operator choice (Nested Loops vs. Hash Match Join)
  - Memory Grant under- or overestimation

Inefficient Execution Plans in SQL Server

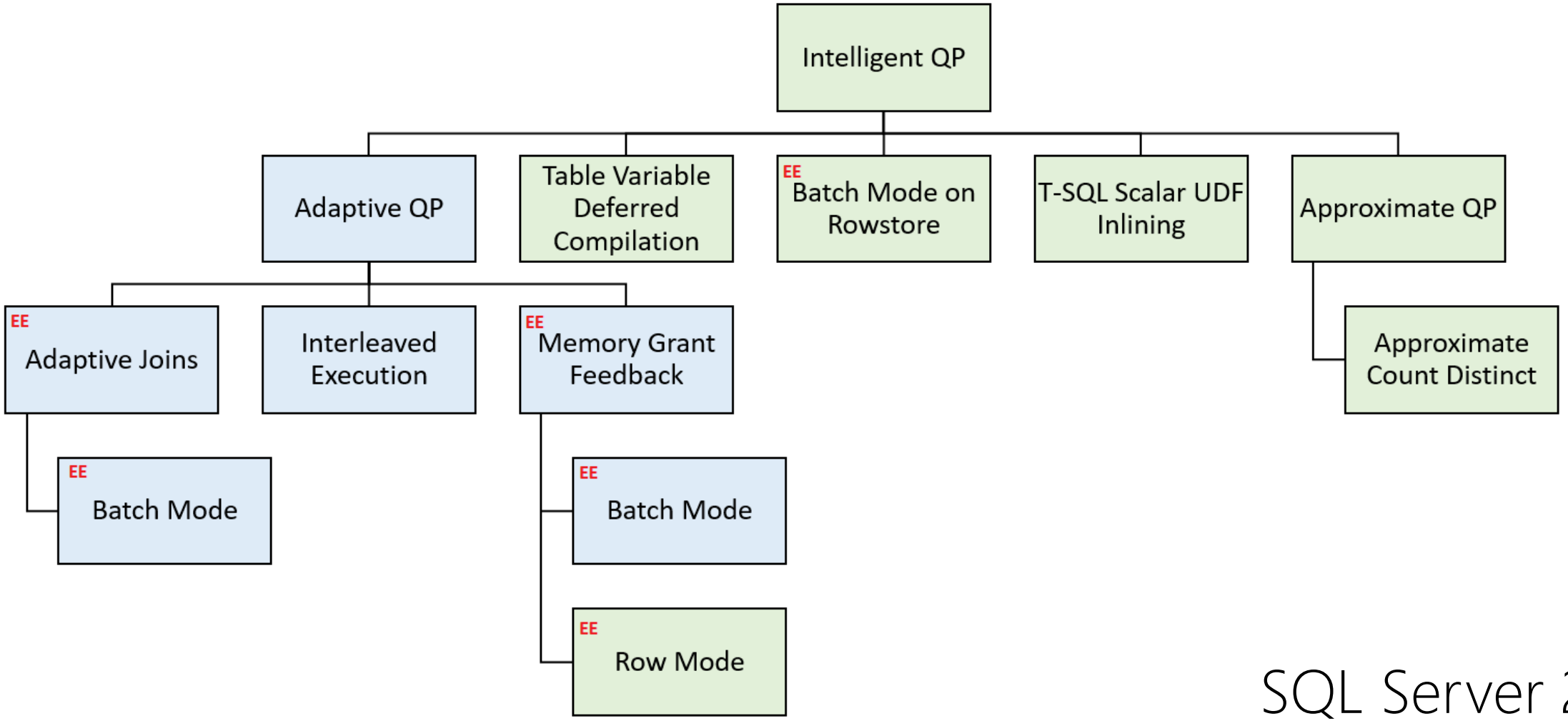
- SQL Server 2016 and prior
  - All plan decisions at the compile time (operators, memory)
  - Used “blindly” for consecutive query executions
  - No changes in cached plan (without recompiling)
- SQL Server 2017 Adaptive Query Processing
  - Creating a better plan for queries using MSTVF with the interleaved execution
  - Postponing decision about the join operator to the runtime (adaptive join)
  - Updating a part of the cached plan (memory grant)
- SQL Server 2019 Intelligent Query Processing
  - Additional adaptive improvements, but also some overall; therefore new name => Intelligent QP

Adaptive Query Processing



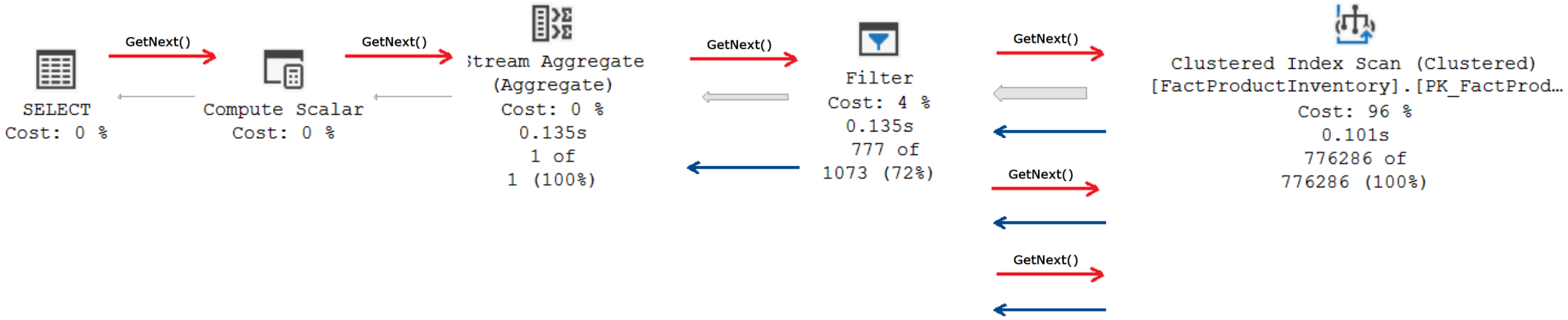
SQL Server 2017

Intelligent Query Processing



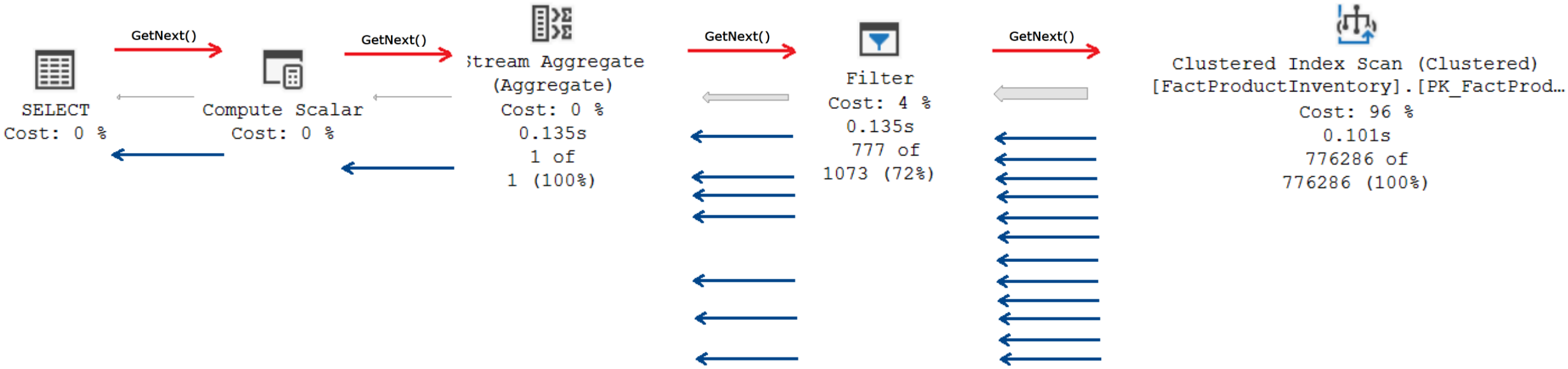
SQL Server 2019

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

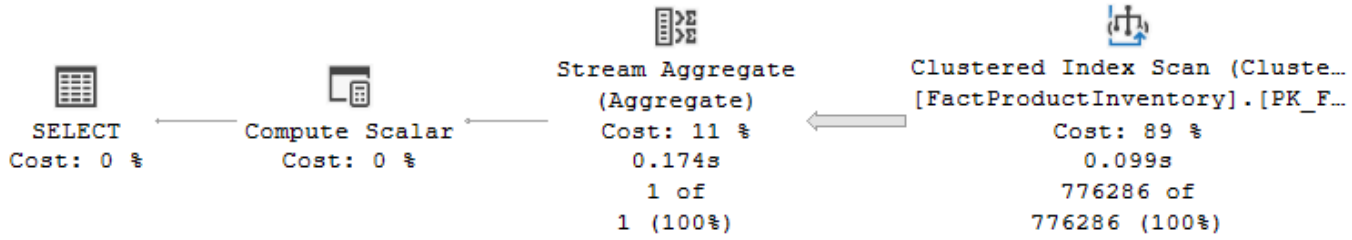
- Batch Mode on columstore introduced in SQL Server 2012
  - Improvements – up to 20x faster queries!

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

```sql
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%

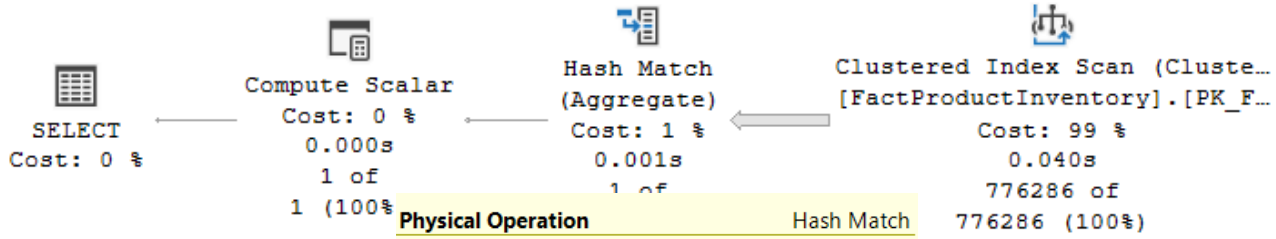
```sql
SELECT COUNT(*), MAX(UnitsIn) FROM dbo.FactProductInventory OPTION (USE HINT ('QUERY_OPTIMIZER_COMPATIBILITY_LEVEL_140'))
```



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

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

```sql
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

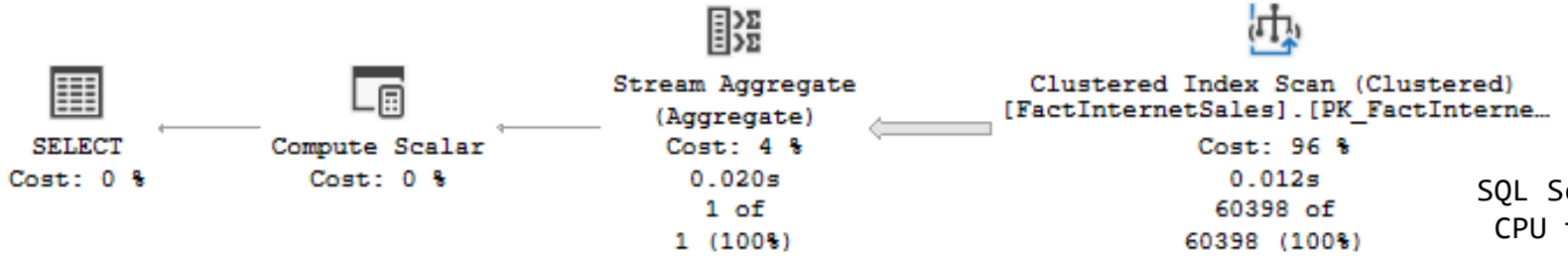
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Batch
Estimated Execution Mode	Batch

4x

```sql
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%

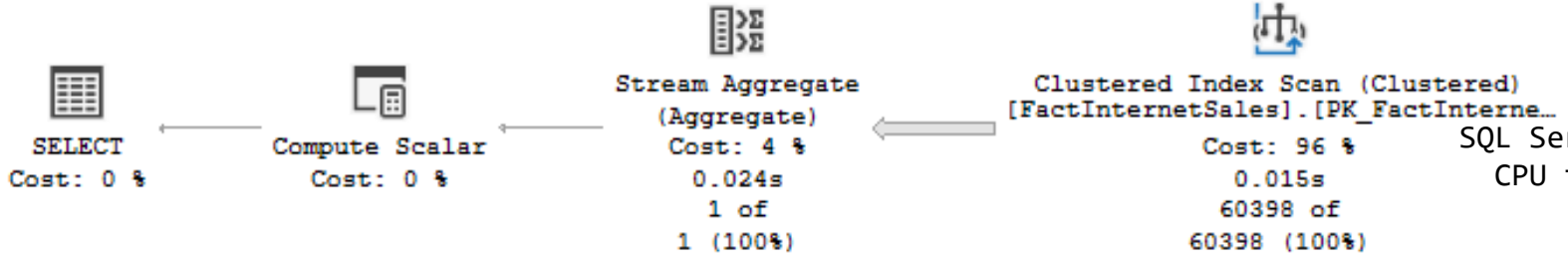
```sql
SELECT COUNT(*), MAX(UnitPrice) FROM dbo.FactInternetSales OPTION (USE HINT ('QUERY_OPTIMIZER_COMPATIBILITY_LEVEL_140'))
```



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

Query 2: Query cost (relative to the batch): 50%

```sql
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

UNDOCUMENTED

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

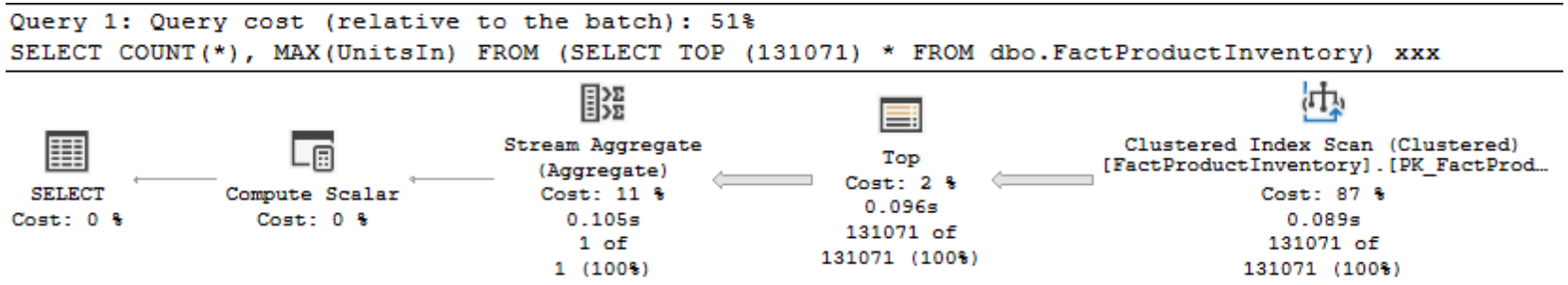
name	timestamp
batch_mode_heuristics	2019-10-24 16:25:54.8502441
batch_mode_heuristics	2019-10-24 16:25:54.9770007
batch_mode_heuristics	2019-10-24 16:25:58.0653381
batch_mode_heuristics	2019-10-24 16:25:58.3971789

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

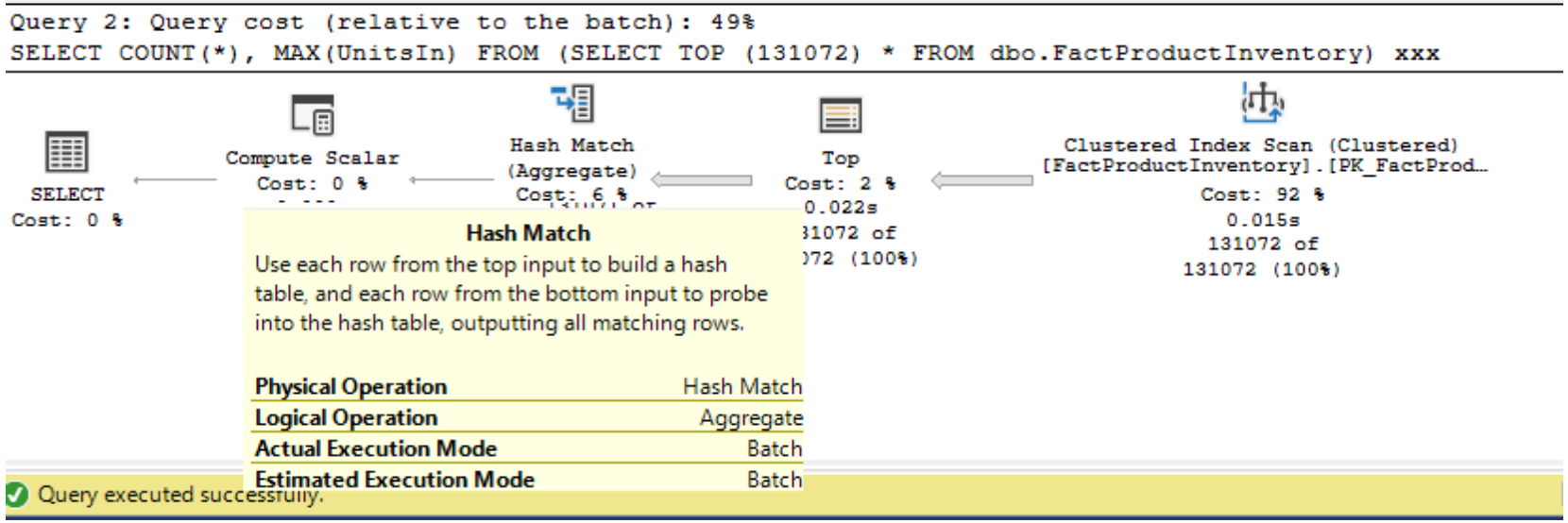
Details

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;
```



row mode

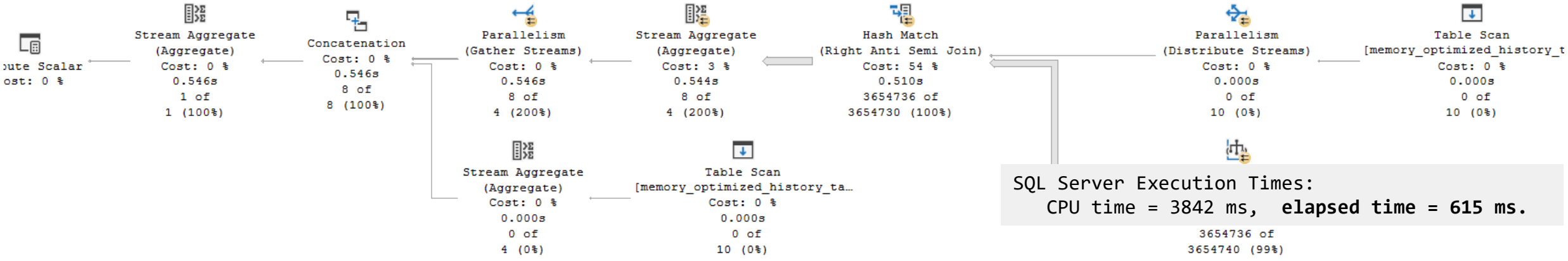


batch mode

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

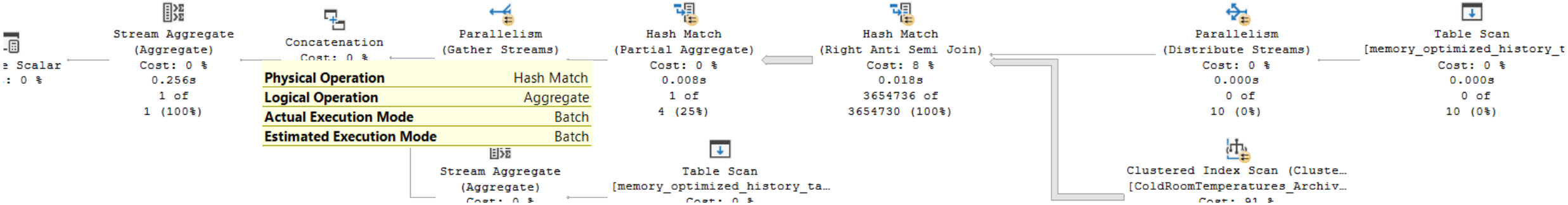
2x

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



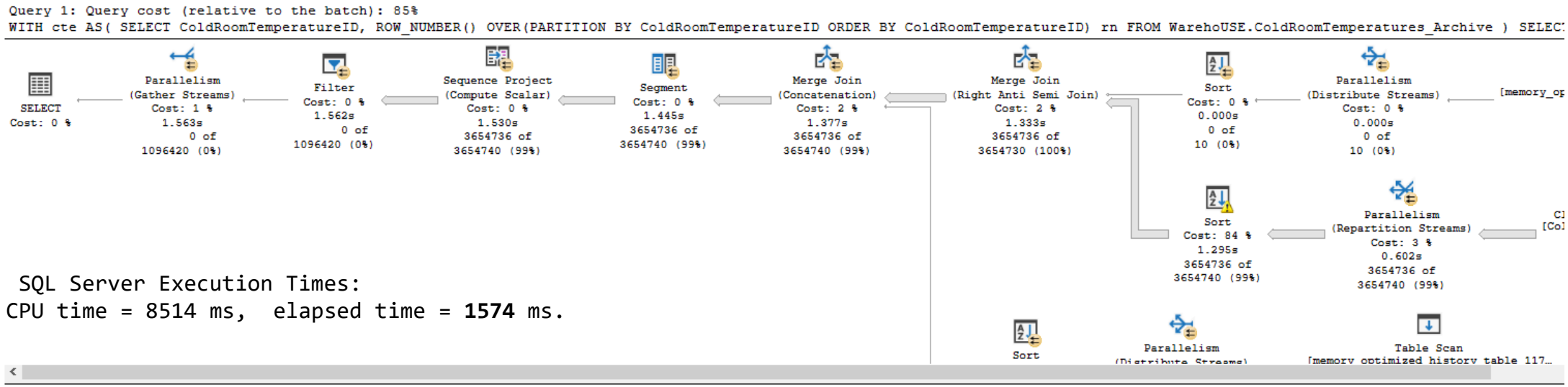
SQL Server Execution Times:
CPU time = 3842 ms, **elapsed time = 615 ms.**

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

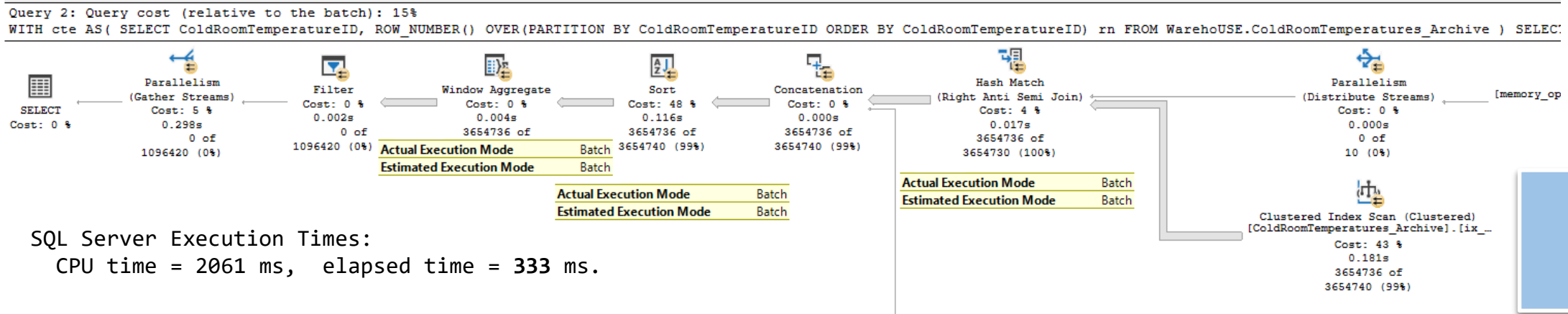


SQL Server Execution Times:
CPU time = 1642 ms, **elapsed time = 306 ms.**

```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.



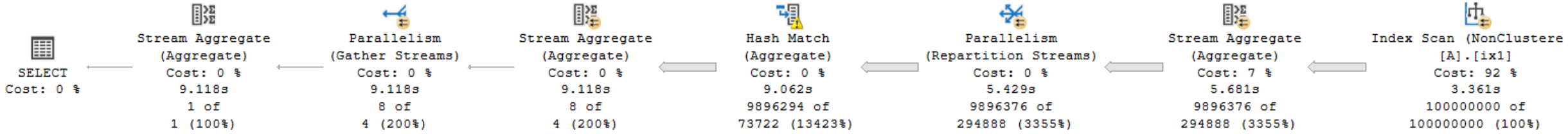
SQL Server Execution Times:
CPU time = 2061 ms, elapsed time = **333** ms.

5x

```sql
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%

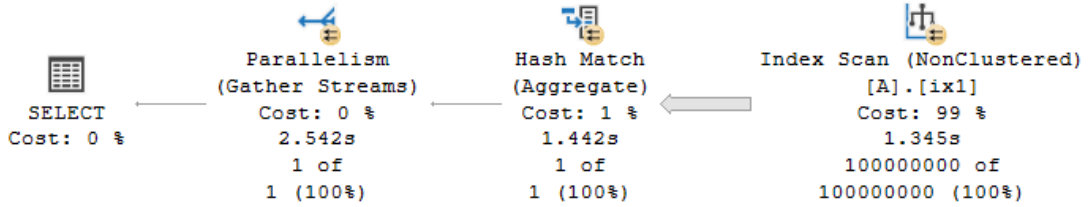
```sql
SELECT COUNT_BIG(DISTINCT pid) from dbo.A OPTION (USE HINT ('QUERY_OPTIMIZER_COMPATIBILITY_LEVEL_140'))
```



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

Query 2: Query cost (relative to the batch): 48%

```sql
SELECT COUNT_BIG(DISTINCT pid) from dbo.A
```



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

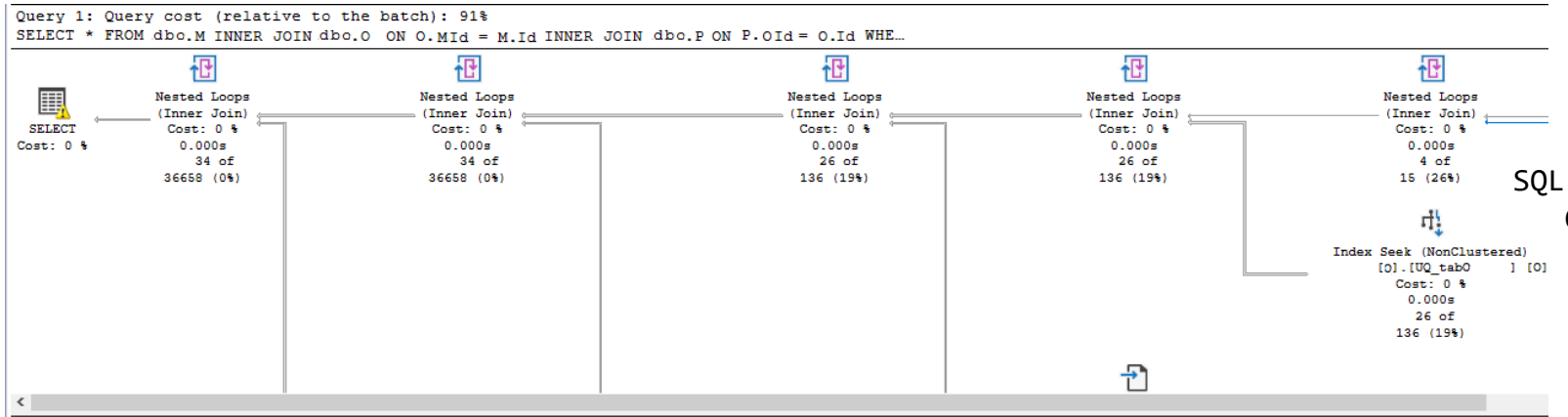
3,5x

Regressions?

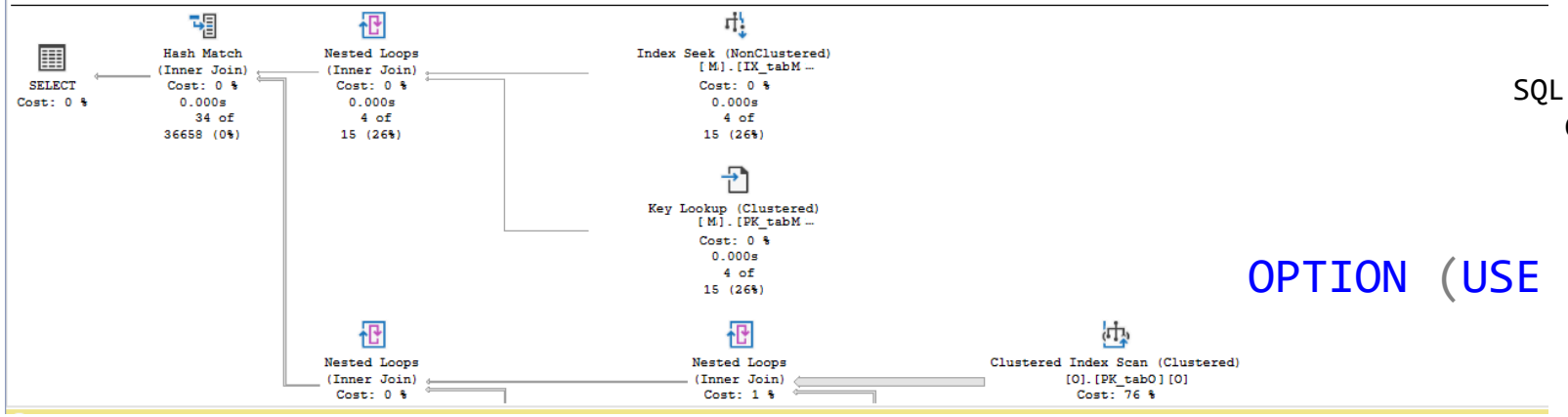
25x

```
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

Enabling

```
ALTER DATABASE current SET COMPATIBILITY_LEVEL = 150;

ALTER DATABASE SCOPED CONFIGURATION SET BATCH_MODE_ON_ROWSTORE = ON;

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

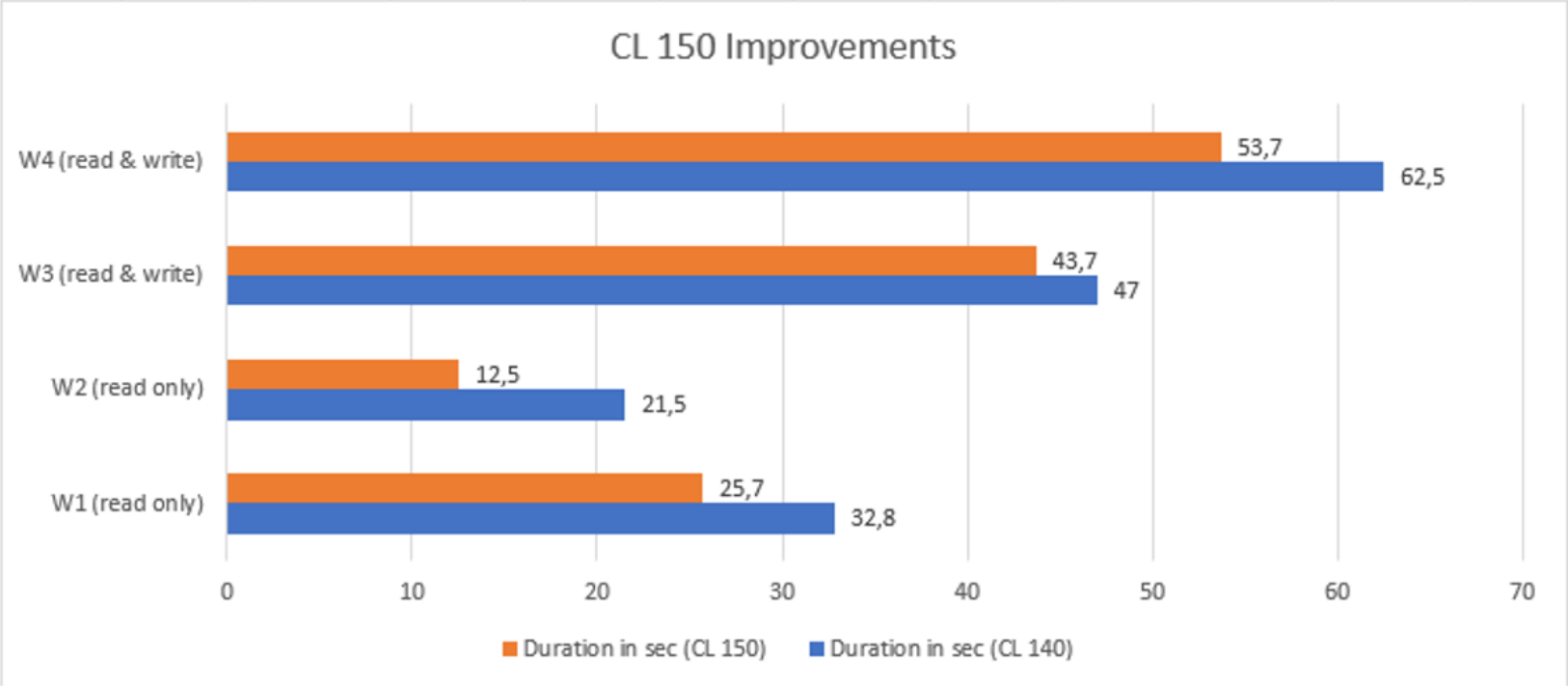
Disabling

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

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

Our tests – with 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%



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

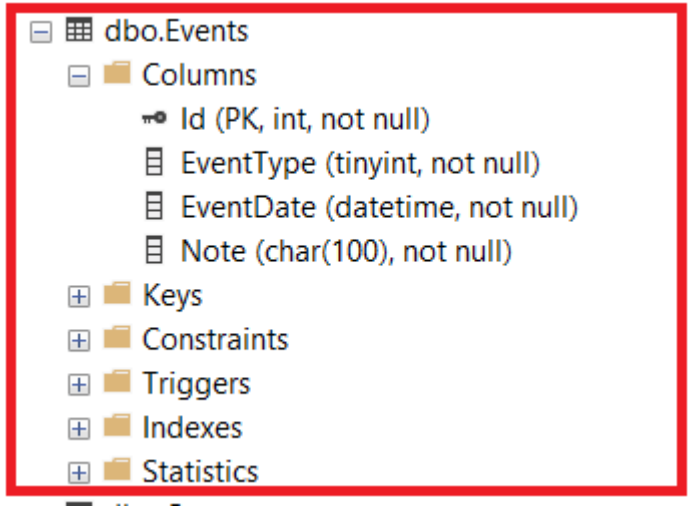
Batch Mode on Rowstore - Conclusion

- Very promising feature
  - Improvements with no efforts
  - It could be a reason for upgrade for some companies
- First version, probably will not optimize all queries, where you would expect the optimization
- Possible regressions, but you can enable/disable feature at two levels
- It brings benefits for queries with large tables and datasets

Memory Grant Feedback

- Adjusting memory grant parameter in the execution plan AFTER the plan is generated (after a few query executions)
- Memory is adjusted for a query when
  - It used less than 50% of granted memory
  - Is spilling out to tempdb
- In SQL Server 2017 requires a columnstore index on the affected table

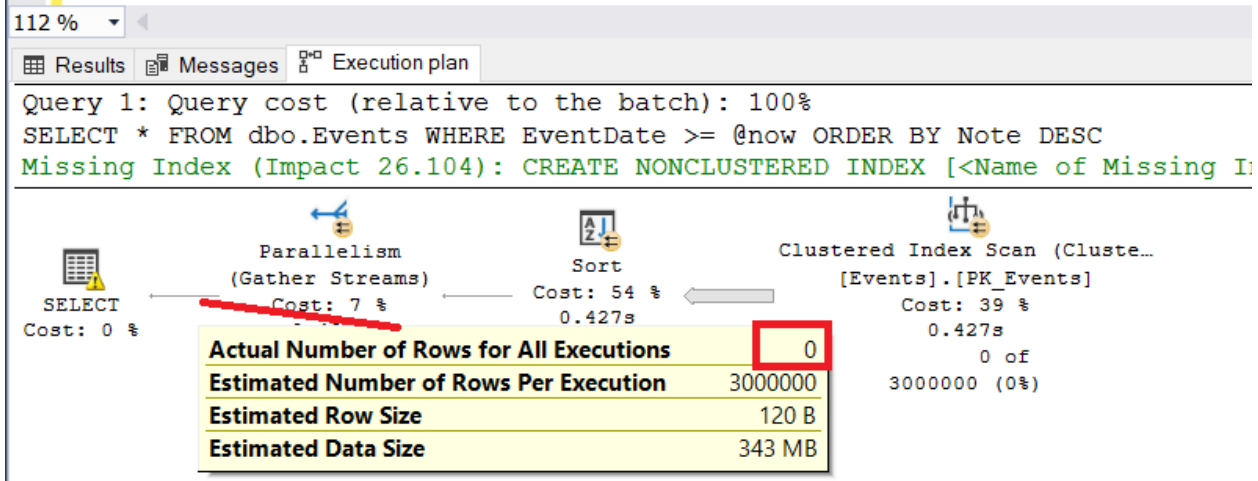
Memory Grant Feedback - Example



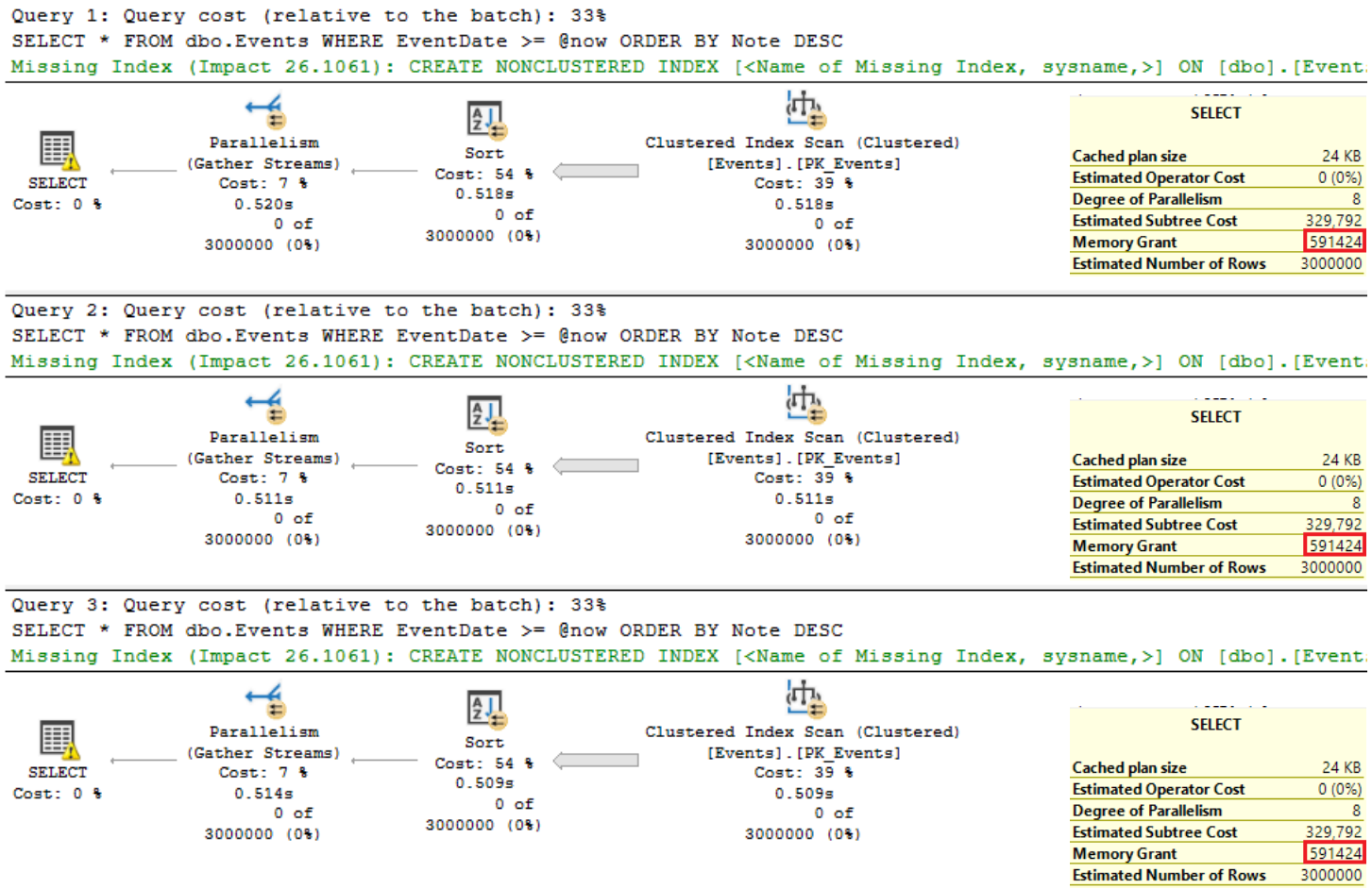
dbo.Events – 10M rows

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

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

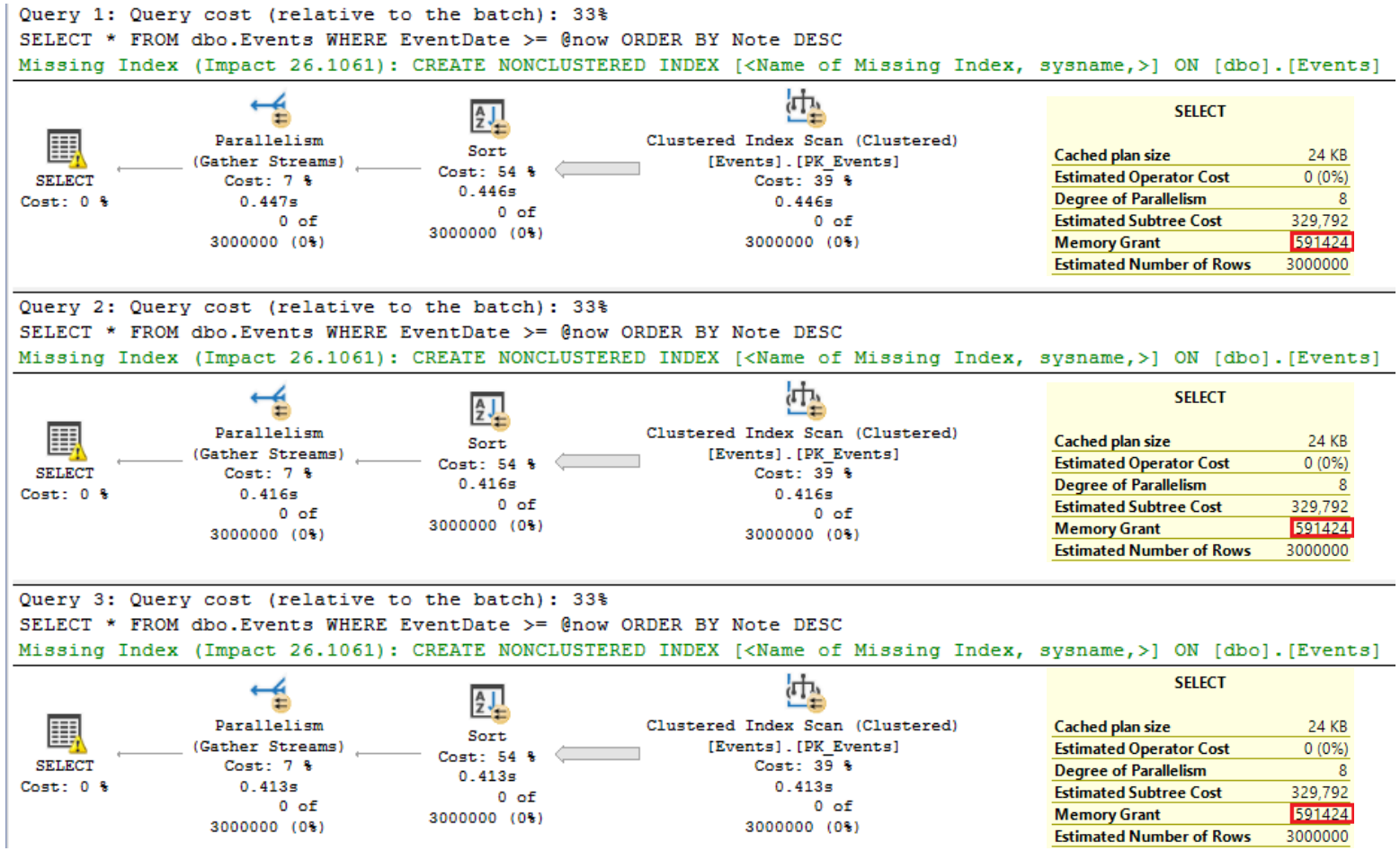


```sql
--ensure that the database runs under CL 130 (SQL Server 2016)
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 130;
GO
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
EXEC dbo.GetEvents '20180101';
GO 3
```



The same memory grant

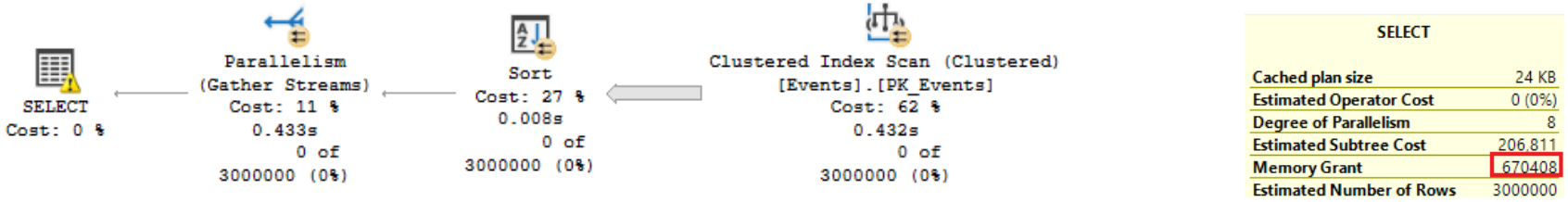
```sql
--ensure that the database runs under CL 140 (SQL Server 2017)
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 140;
GO
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
EXEC dbo.GetEvents '20180101';
GO 3
```



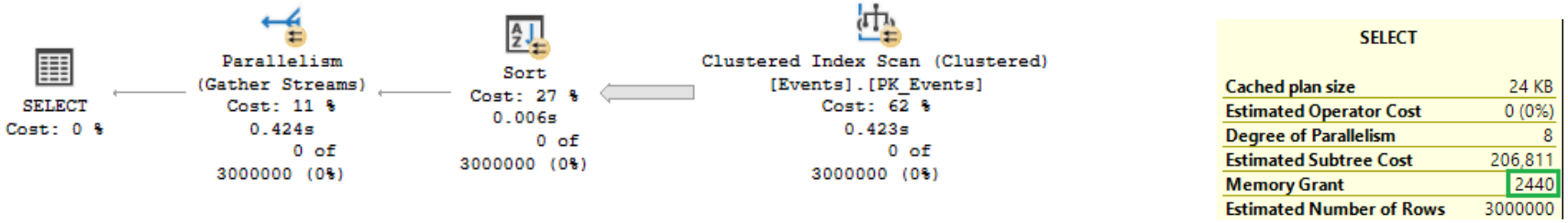
Batch mode Memory Grant in SQL Server 2017 requires a columnstore index

```sql
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
```

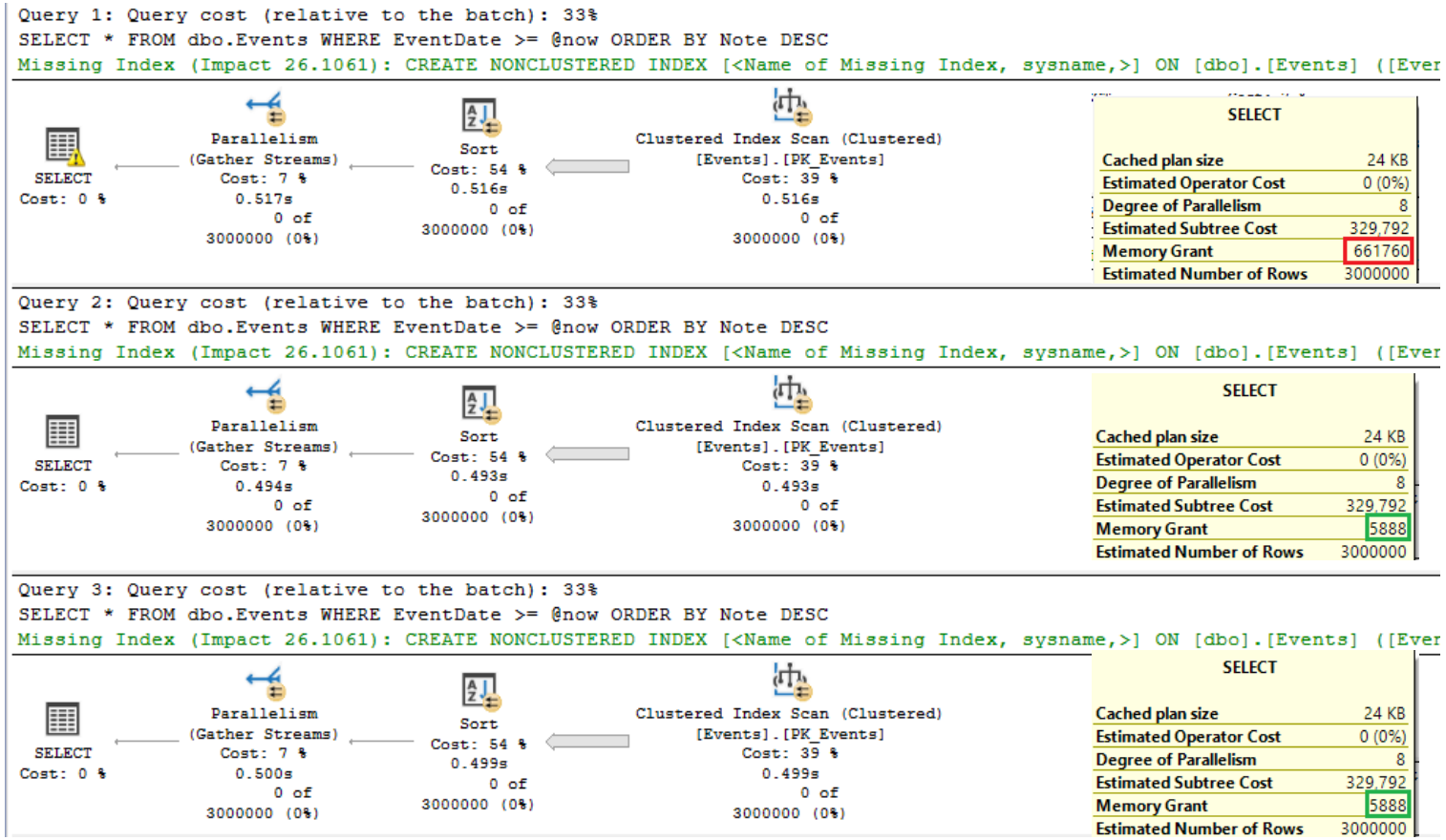



```
Query 2: Query cost (relative to the batch): 33%
SELECT * FROM dbo.Events WHERE EventDate >= @now ORDER BY Note DESC
Missing Index (Impact 41.6303): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[Events] ([Ev
```




```
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
```

```sql
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

Memory Grant Feedback

- New plan attributes in the XML plan
- It works 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
- In SQL Server 2019, it works in both Batch and Row Mode
- If memory grant memory values oscillate, the feature is disabled

Memory Grant Feedback

```sql
EXEC dbo.GetEvents '20180101';
EXEC dbo.GetEvents '20160101';
```

```sql
EXEC dbo.GetEvents '20160101';
```

112 %

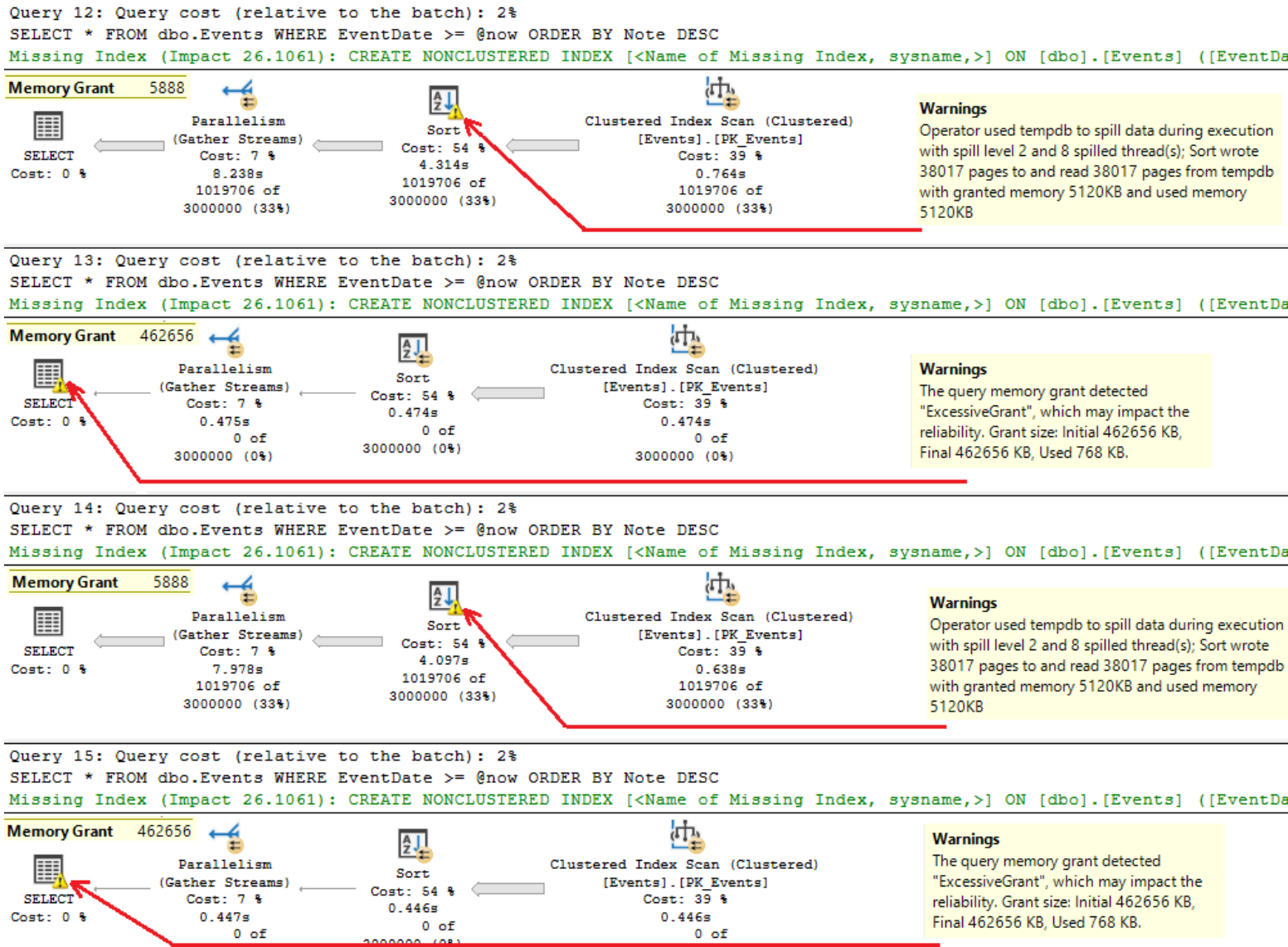
Results | Messages | Execution plan

	Id	EventType	EventDate	Note
1	21	3	2016-10-28 00:00:00.000	test
2	47628	2	2016-12-23 00:00:00.000	test
3	15875	4	2016-01-20 00:00:00.000	test
4	79363	2	2017-02-28 00:00:00.000	test
5	31747	2	2016-01-13 00:00:00.000	test
6	63503	4	2017-05-30 00:00:00.000	test
7	95242	1	2017-01-12 00:00:00.000	test
8	111100	3	2017-03-21 00:00:00.000	test

Query executed successfully. | TestDb | 00:00:08 | 1 018 603 rows

Memory Grant Feedback

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



Memory Grant Feedback

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

```
EXEC dbo.GetEvents '20190901';
EXEC dbo.GetEvents '20160901';
GO 30
```

Displaying 1 Events

name	timestamp
memory_grant_feedback_loop_disabled	2019-11-25 00:09:48.6804572

Event:memory_grant_feedback_loop_disabled (2019-11-25 00:09:48.6804572)

Details

Field	Value
sql_text	EXEC dbo.GetEvents '20190901'; EXEC dbo.GetEvents '20160901';
total_execution_count	33
total_update_count	31

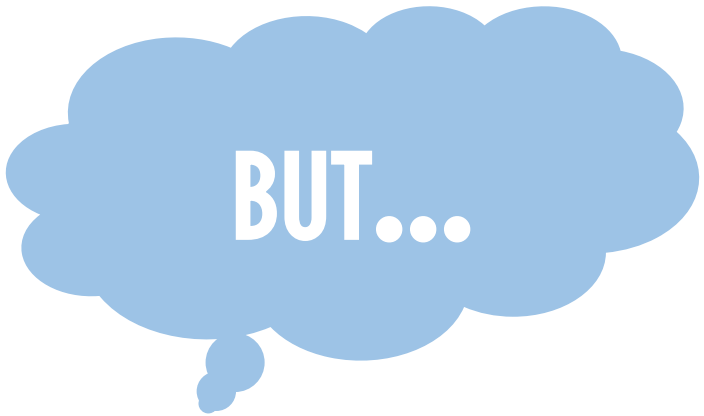
Memory Grant Feedback - Conclusion

- Very useful feature
- In SQL Server 2019 works in both modes
- Will affect ALL queries having an operator that requires memory
-
- Almost no measurable overhead (*)

- (*) in all my tests

Scalar UDFs in SQL Server

- Code reuse, encapsulation and modularity
- Complex business rules or computations
- Single place change
- Written once, invoke from many modules
- Reduce network traffic



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

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

Scalar UDF Inlining

- Goal – improve queries where scalar UDFs are problem

- 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
  - Optimize expressions or subqueries

- Result:
  - Performance improved (more efficient plan)
  - Execution plan could be parallel

Ménage à FROID

```sql
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;
```

```sql
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
);
```

Ménage à FROID

```sql
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;
```

```sql
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
);
```

Ménage à FROID

```sql
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;
```

```sql
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
);
```

Ménage à FROID

```sql
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;
```

```sql
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
);
```

Ménage à FROID

```sql
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;
```

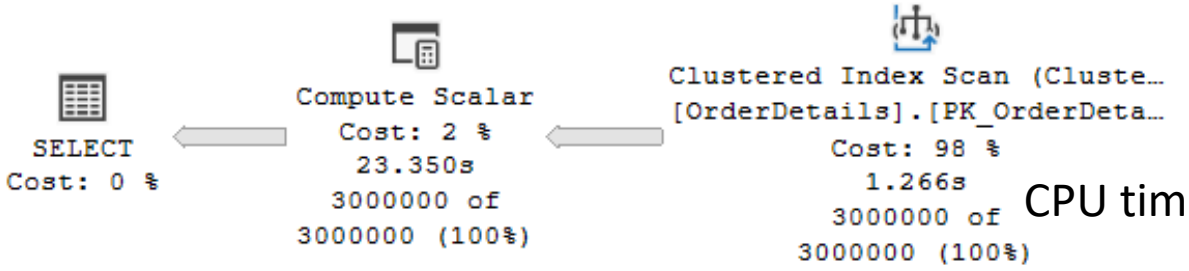
```sql
SELECT q5.v
FROM
(
        (SELECT 2000 AS a) AS q1
        OUTER APPLY
                (SELECT CASE WHEN q1.a > 1000 THEN 'HIGH' END AS val)
AS q2
        OUTER APPLY
        (SELECT CASE WHEN q1.a > 500 THEN 'HIGH' END AS val) AS q3
        OUTER APPLY
                (SELECT 'LOW' AS val) AS 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;
```

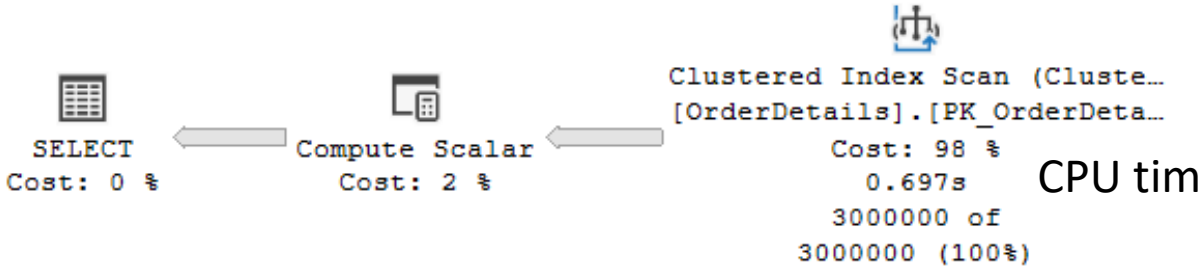
12x

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.

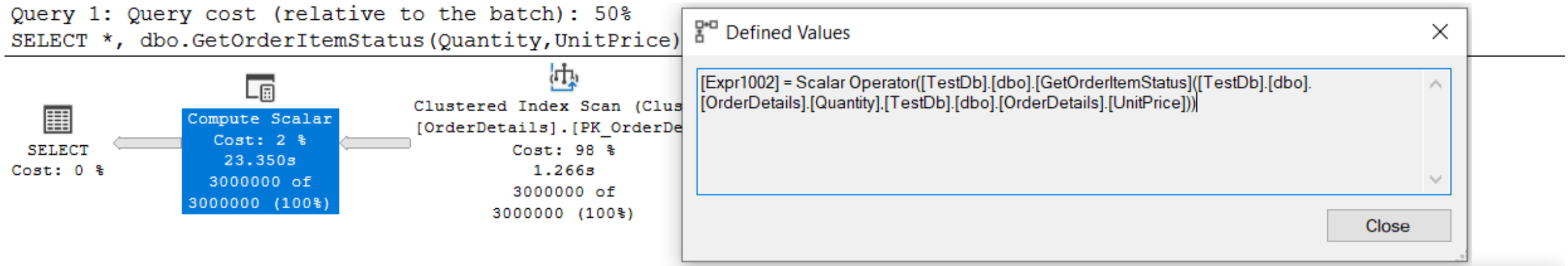
Query 2: Query cost (relative to the batch): 50%
SELECT *, dbo.GetOrderItemStatus(Quantity,UnitPrice) ItemStatus FROM dbo.OrderDetails

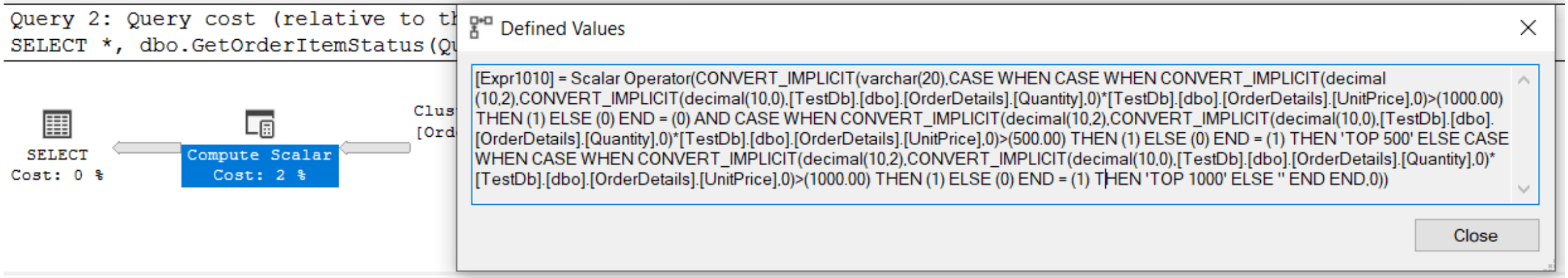


CPU time = 2047 ms, elapsed time = **2075** ms.

```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;
```

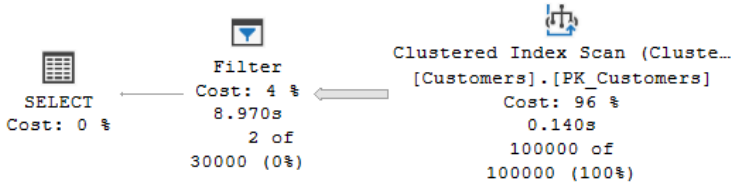




33x

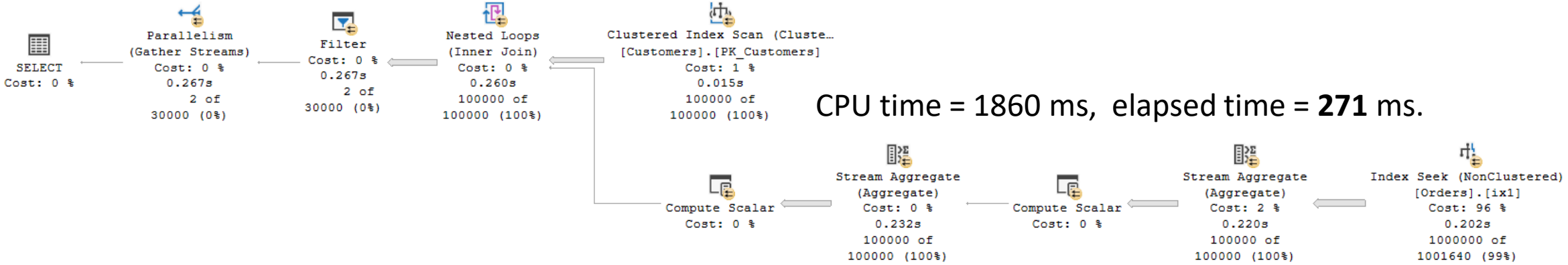
```sql
CREATE OR ALTER FUNCTION dbo.GetOrderCnt (@CustomerId INT)
RETURNS INT
AS
BEGIN
    DECLARE @Cnt INT;
    SELECT @Cnt = COUNT(*) FROM dbo.Orders WHERE CustomerId = @CustomerId;
    RETURN @Cnt;
END

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;
```

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.

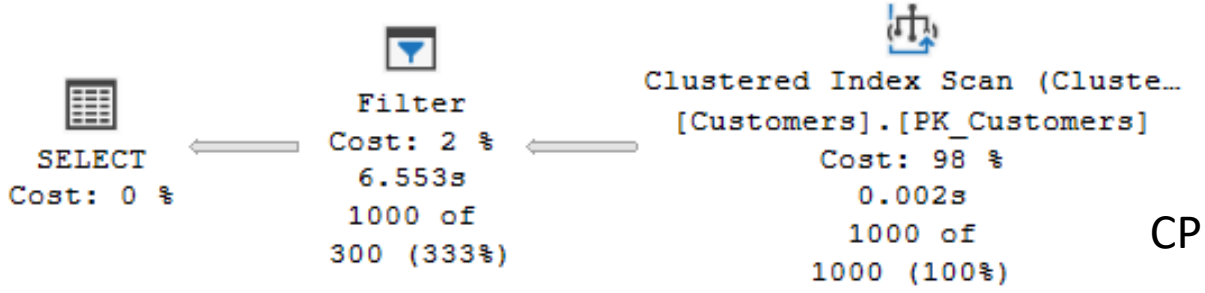
```sql
CREATE OR ALTER FUNCTION dbo.GetDaysFromLastOrder(@CustomerId INT)
RETURNS INT
AS
BEGIN
    DECLARE @Days INT;
    DECLARE @LastOrder DATETIME;
    SET @LastOrder = (SELECT TOP (1) OrderDate FROM dbo.Orders WHERE CustomerId = @CustomerId ORDER BY OrderDate DESC);
    SELECT @Days = DATEDIFF(day,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

0%

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

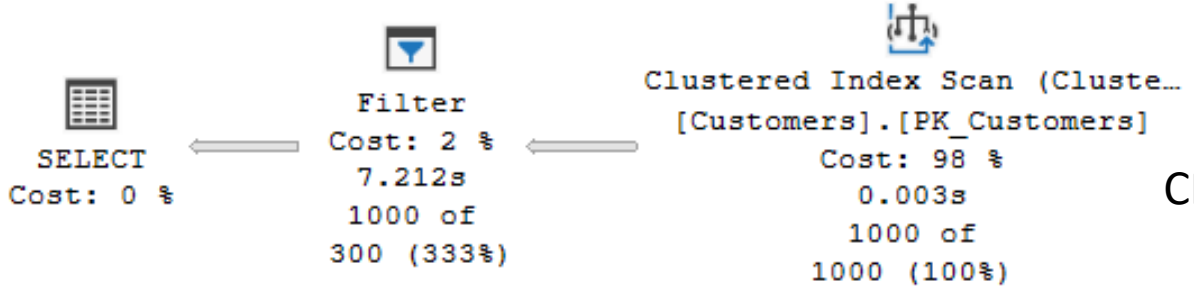
Scalar UDF Inline does not work with **GETDATE()** function

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:

```
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';
```

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

Scalar UDF Inlining - Limitations

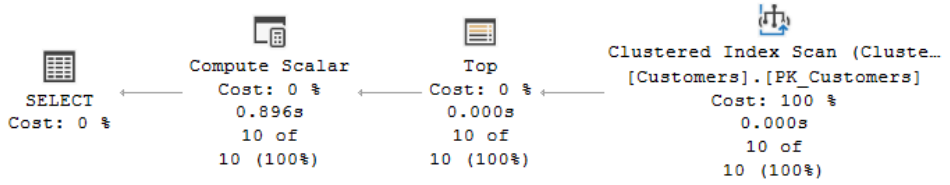
- UDF does not invoke any intrinsic function that is either time-dependent or has side effects such as GETDATE() or NEWSEQUENTIALID
- The UDF does not reference table variables, table-valued parameters or user-defined types
- UDF is not natively compiled (interop is supported)
- UDF is not used in a 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

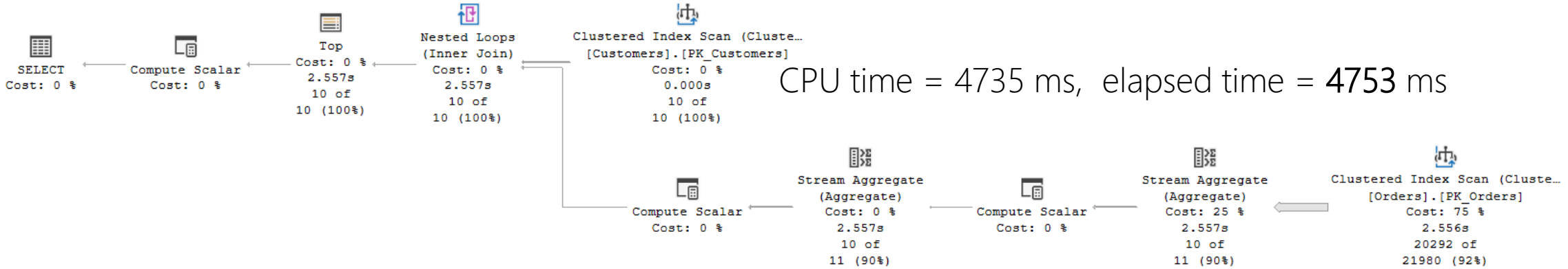
```
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 140;
GO
SELECT TOP (10) * FROM dbo.Customers WHERE dbo.GetOrderCnt(CustomerId) > 25;
GO
ALTER DATABASE TestDb SET COMPATIBILITY_LEVEL = 150;
GO
SELECT TOP (10) * FROM dbo.Customers WHERE dbo.GetOrderCnt(CustomerId) > 25;
```

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

```sql
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

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

Scalar UDF Inlining - Configuration

- 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
```

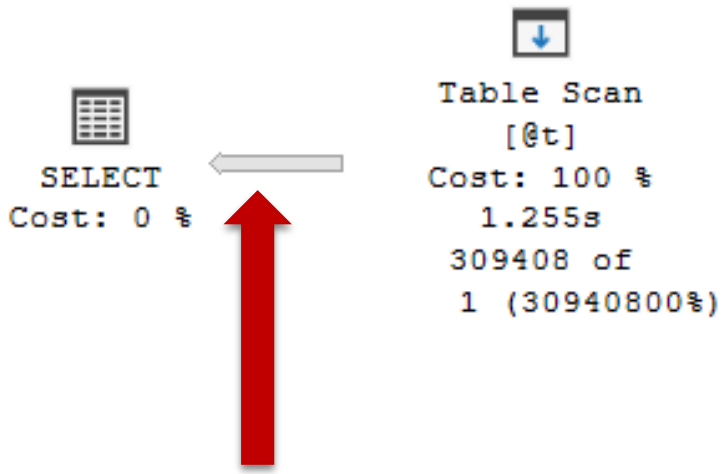
Scalar UDF Inlining - Conclusion

- 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
- As far I know, it is still not available in Azure

Problems with Table Variables

```sql
DECLARE @t TABLE(id INT)
INSERT INTO @t SELECT message_id FROM sys.messages;
SELECT * FROM @t;
```

```
Query 2: Query cost (relative to the batch): 2%
SELECT * FROM @t
```



Actual Number of Rows	309408
Number of Rows Read	309408
Estimated Number of Rows	1
Estimated Row Size	11 B
Estimated Data Size	11 B

- Inappropriate operators in the execution plan
- Insufficient memory grants

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

Table Variable Deferred Compilation

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

t_0

t_1

```sql
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_2

SQL Server 2017 and all previous versions

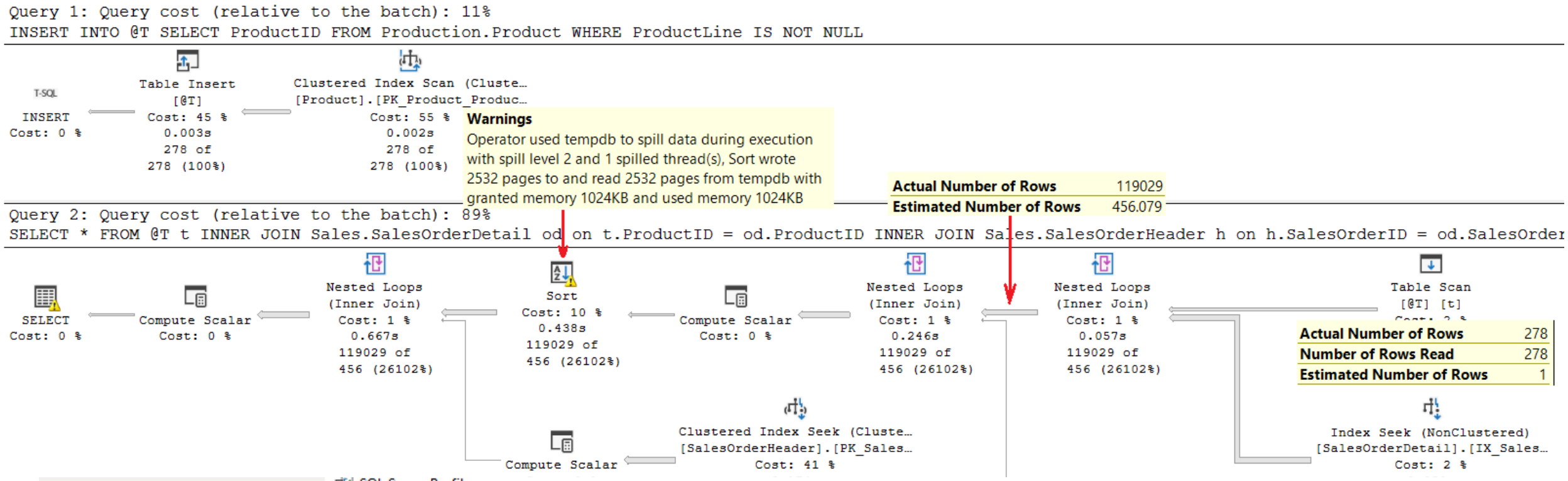
- Content of the table variable unknown at the compile time => cardinality 1

SQL Server 2019

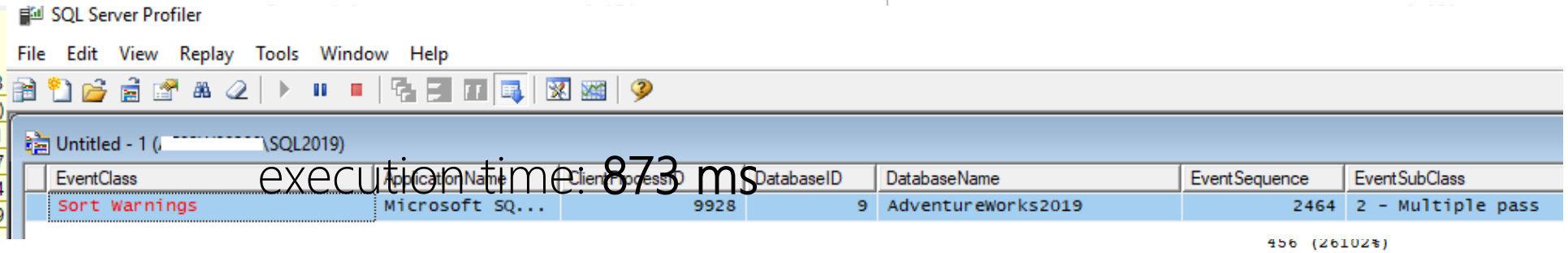
- Content of the table variable known at the compile time => cardinality = actual number of rows

```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

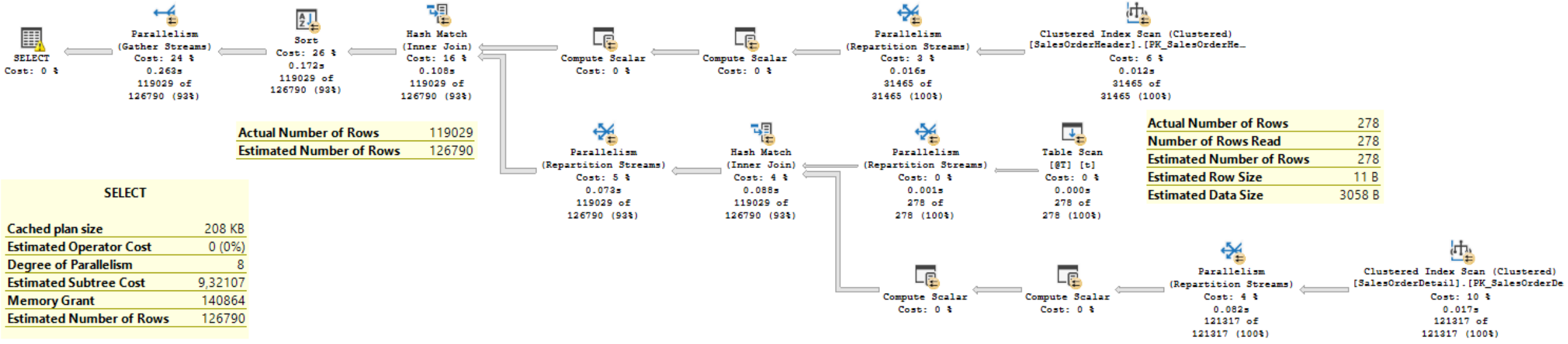


execution time: 873 ms

```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

Table Variable Deferred Compilation

- Improves plan quality and overall performance for queries referencing table variables
- Cardinality estimates are based on actual table variable row counts
- This accurate row count information will be used for optimizing downstream plan operations

Table Variable Deferred Compilation

100M (13 GB) **335 M (42 GB)**

A *

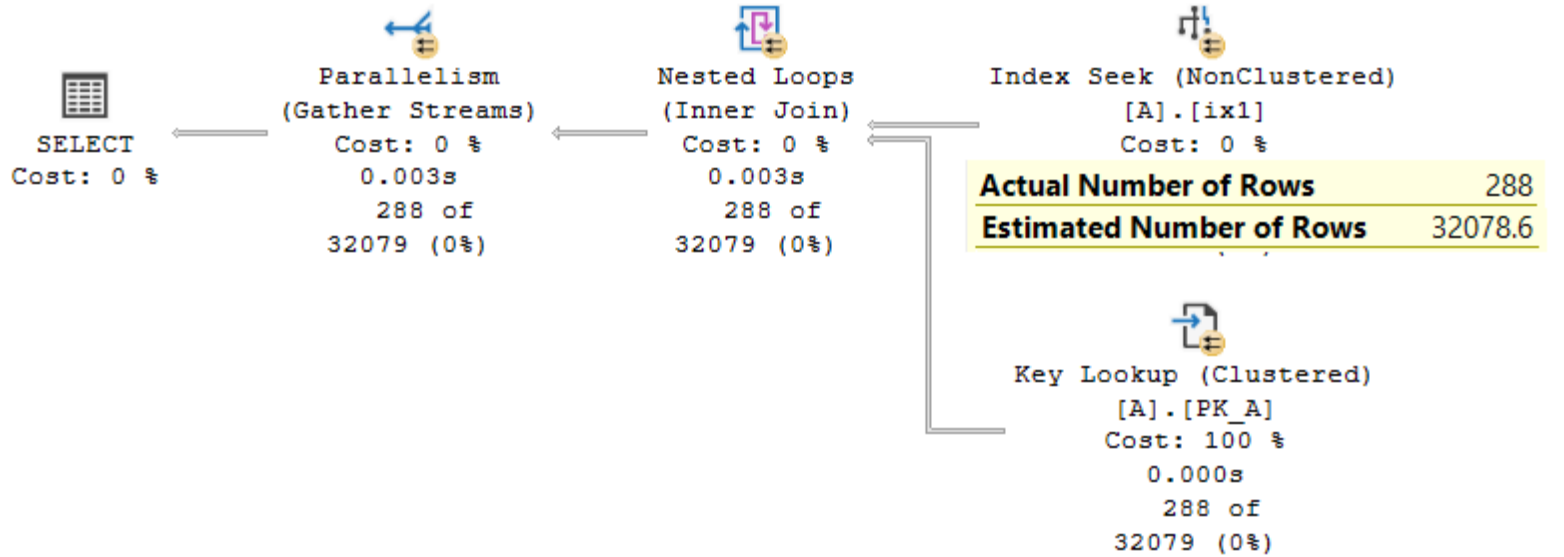
	Column Name	Data Type	Allow Nulls
🔑	id	int	☐
	pid	int	☐
	c1	char(100)	☐
			☐

B *

	Column Name	Data Type	Allow Nulls
🔑	id	int	☐
	pid	int	☐
	c1	char(100)	☐
			☐

```sql
SELECT * FROM A WHERE A.pid = 413032;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [A] WHERE [A].[pid]=@1
Missing Index (Impact 99.8765): CREATE NONCLUSTERED INDEX [<Name of Missing
```



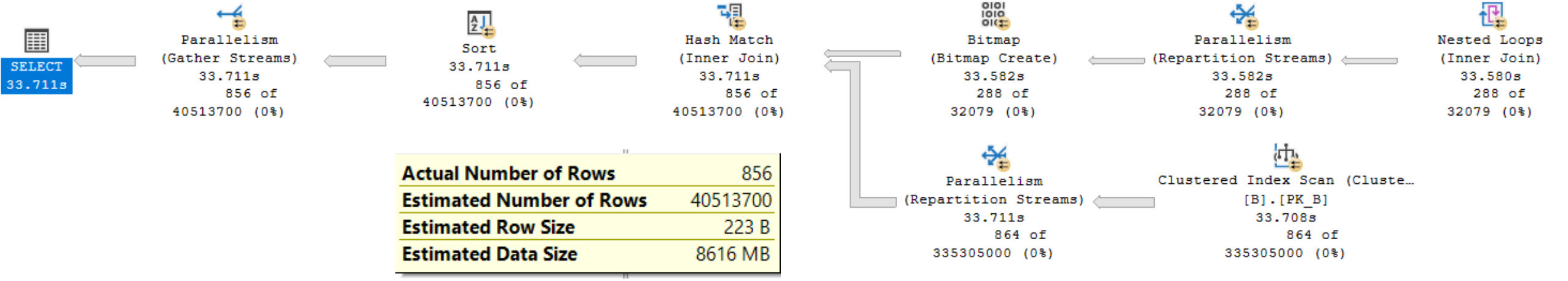
```sql
SELECT * FROM A
INNER JOIN B ON A.id = B.pid
WHERE A.pid = 413032 ORDER BY B.c1;
```

Query returns **856** rows

Results | Messages | Live Query Statistics | Execution plan

Estimated query progress:100%

```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM A INNER JOIN B ON A.id = B.pid WHERE A.pid = 413032 ORDER BY B.c1
Missing Index (Impact 52.4251): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[B] ([pid]) INCLUDE ([c
```

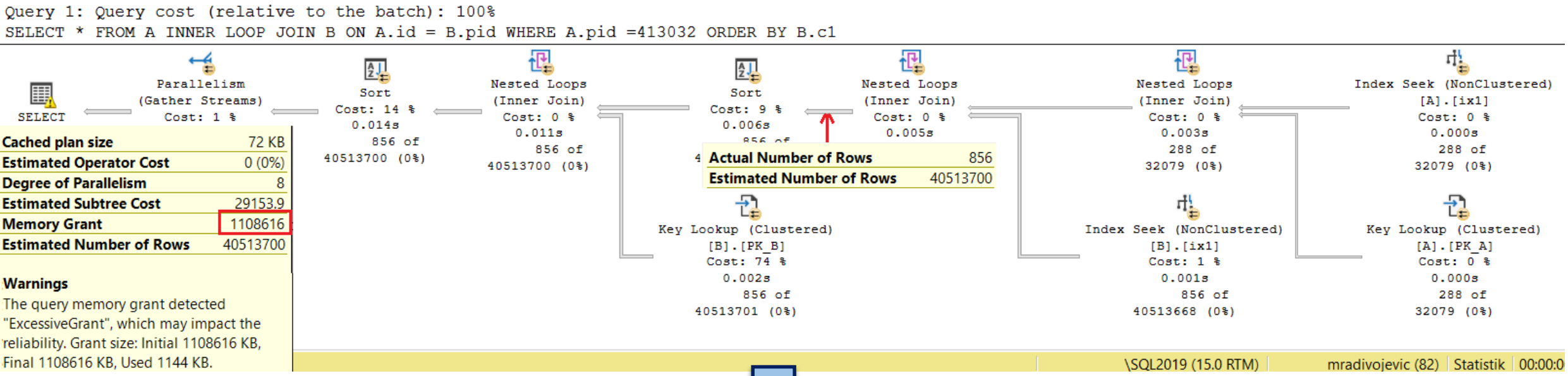


Actual Number of Rows	856
Estimated Number of Rows	40513700
Estimated Row Size	223 B
Estimated Data Size	8616 MB

Executing query... 100% | \SQL2019 (15.0 RTM) | mradivojevic (71) | Statistik | 00:00:34 | 0 rows

```sql
SELECT * FROM A
INNER LOOP JOIN B ON A.id = B.pid
WHERE A.pid = 413032 ORDER BY B.c1;
```

Query returns **856** rows

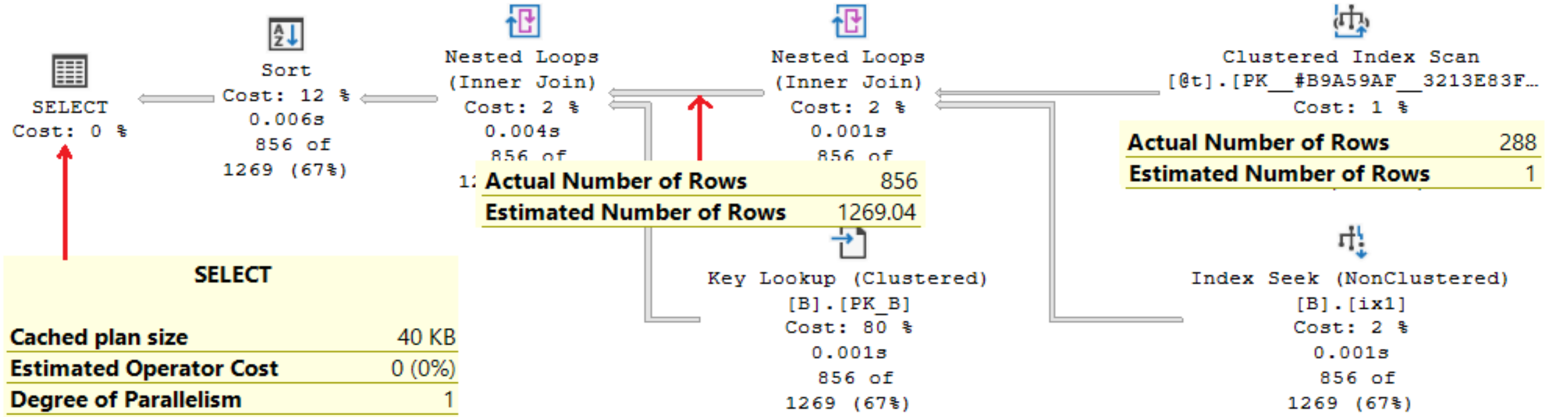



```sql
DECLARE @t TABLE(id INT PRIMARY KEY, pid INT, c1 CHAR(100));
INSERT INTO @t SELECT * FROM A WHERE A.pid = 413032;
SELECT * FROM @t A
INNER JOIN B ON A.id = B.pid;;
```

```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;
```

SQL Server 2017

Query 2: Query cost (relative to the batch): 0%
SELECT * FROM @t A INNER JOIN B ON A.id = B.pid ORDER BY B.c1

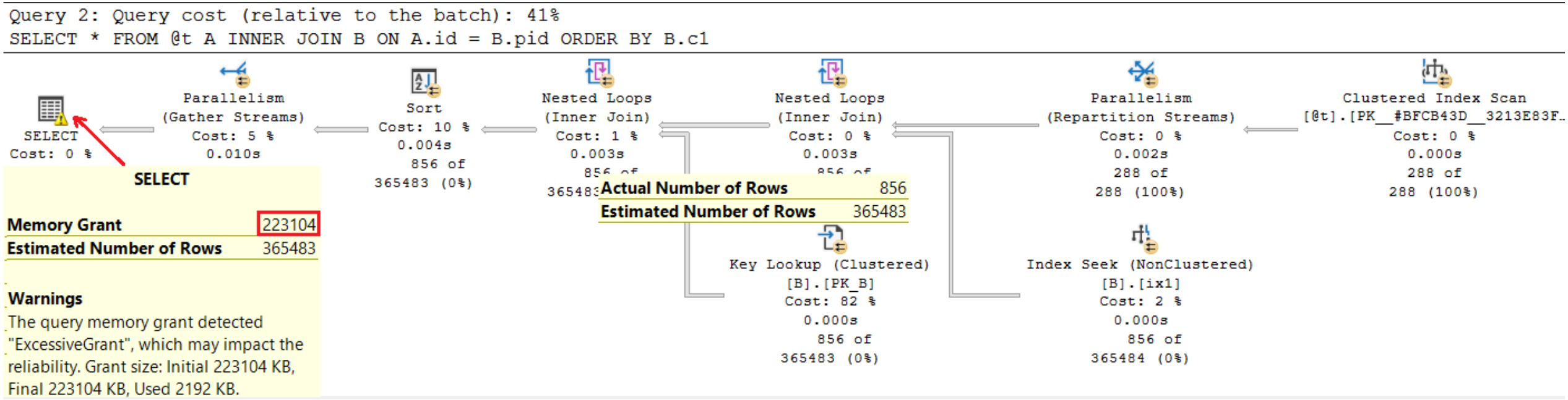


SELECT

Cached plan size	40 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	1
Estimated Subtree Cost	0.255583
Memory Grant	2128
Estimated Number of Rows	1269.04

```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;
```

SQL Server 2019

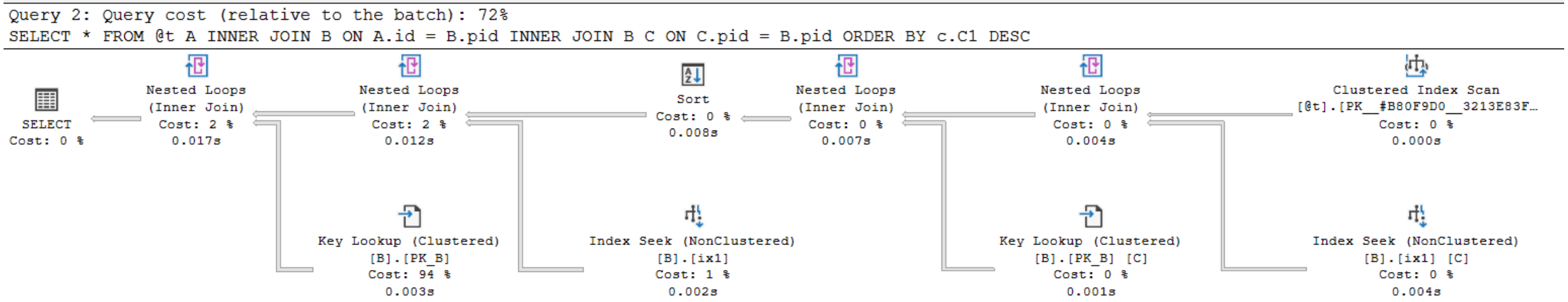


Parallel plan, higher estimations, higher memory grant

```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 2017

Query returns 3 222 rows

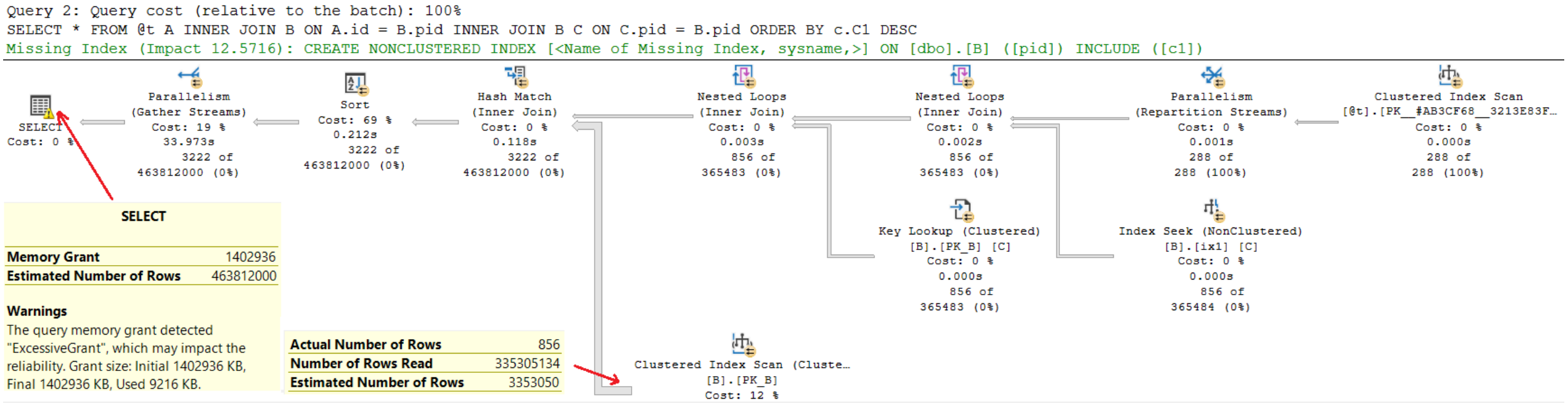


SQL Server Execution Times:

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



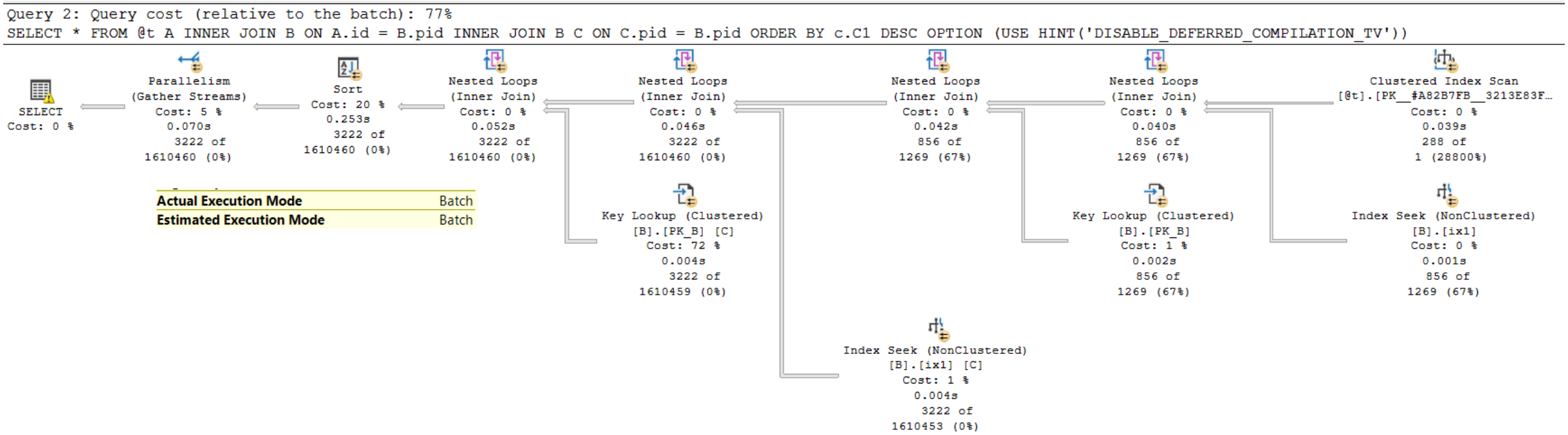
SQL Server Execution Times:

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

300x

```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
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

- 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

🙄 Prone to parameter sniffing

🤮 Can break existing workarounds

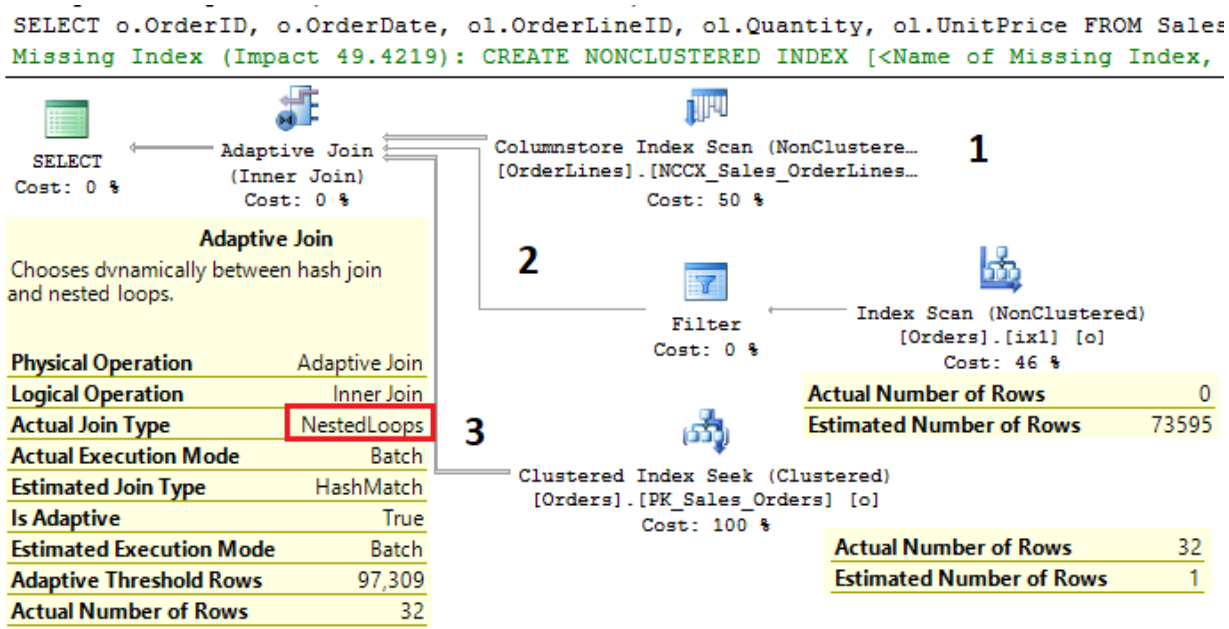
Batch Mode Adaptive Join

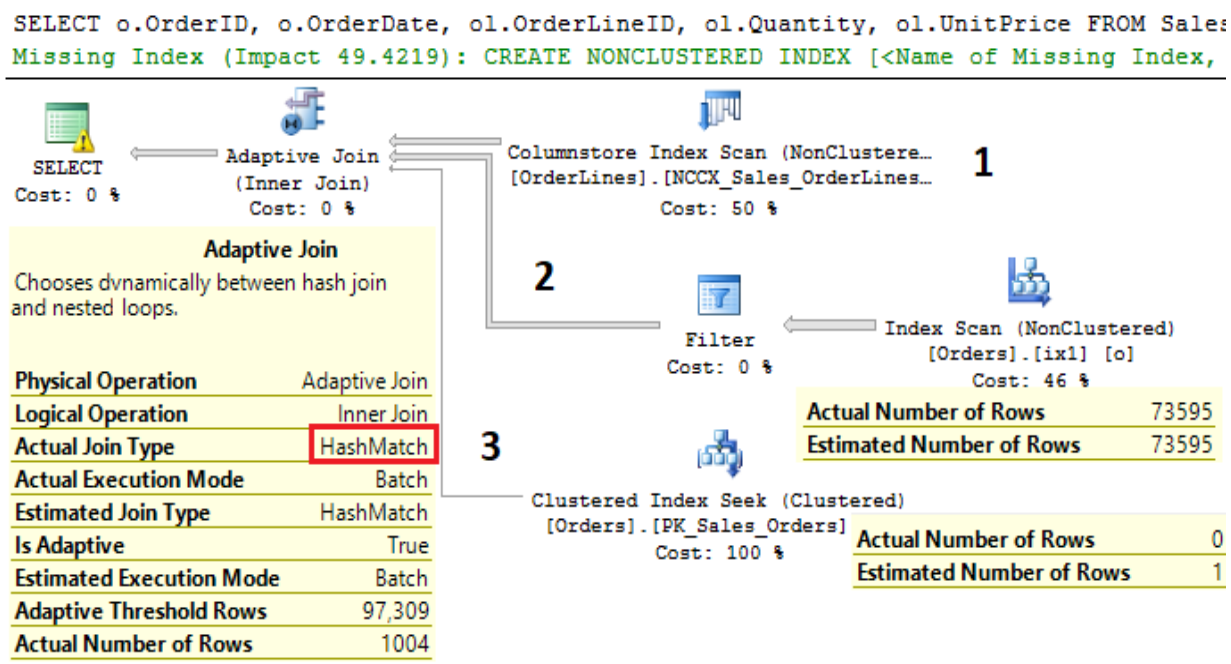
- New operator - Adaptive Join
- Allows to choose between Hash Join and Nested Loop Join at runtime
- It starts as Hash Join and if after input scanning
  - Estimated number of rows < threshold => switches to Nested Loop Join
  - Estimated number of rows >= threshold => continues as Hash Join
- It will better handle queries with variety of parameters, but it won't solve all issues caused by wrongly chosen Join operator
- It works only in Batch Mode

Batch Mode Adaptive Join

```
EXEC dbo.GetOrderDetails 1;
```

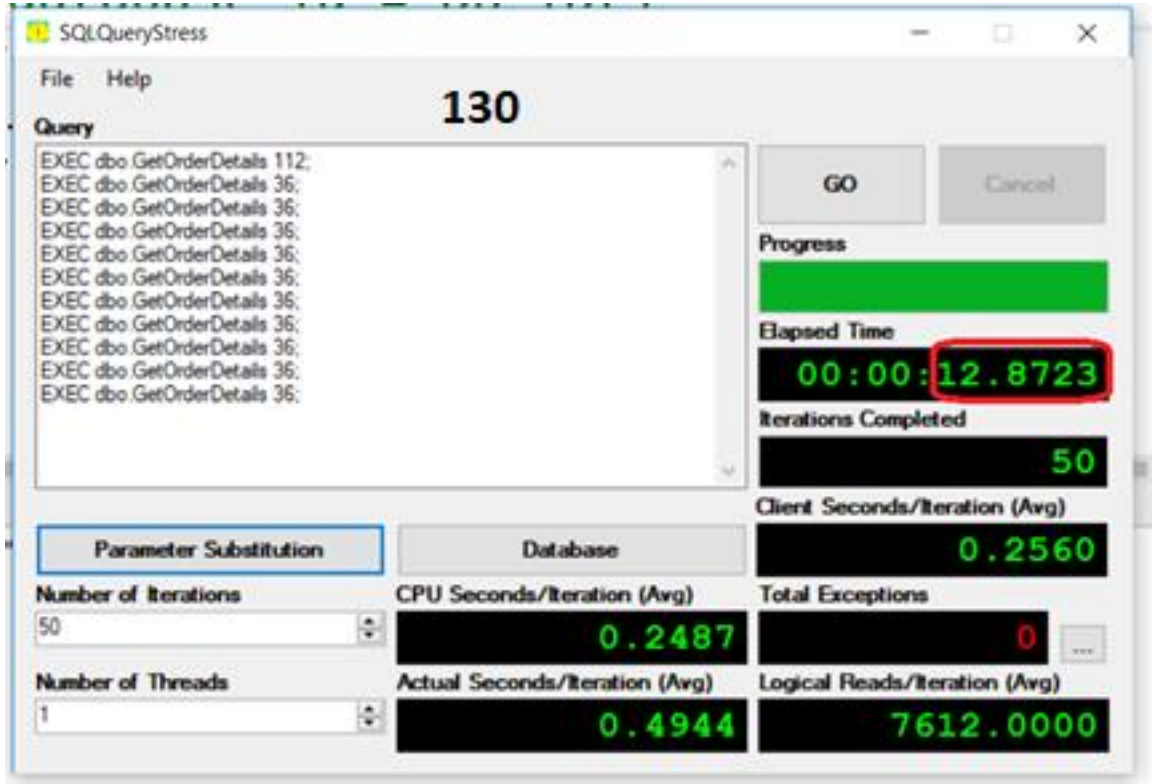
```
EXEC dbo.GetOrderDetails 112;
```

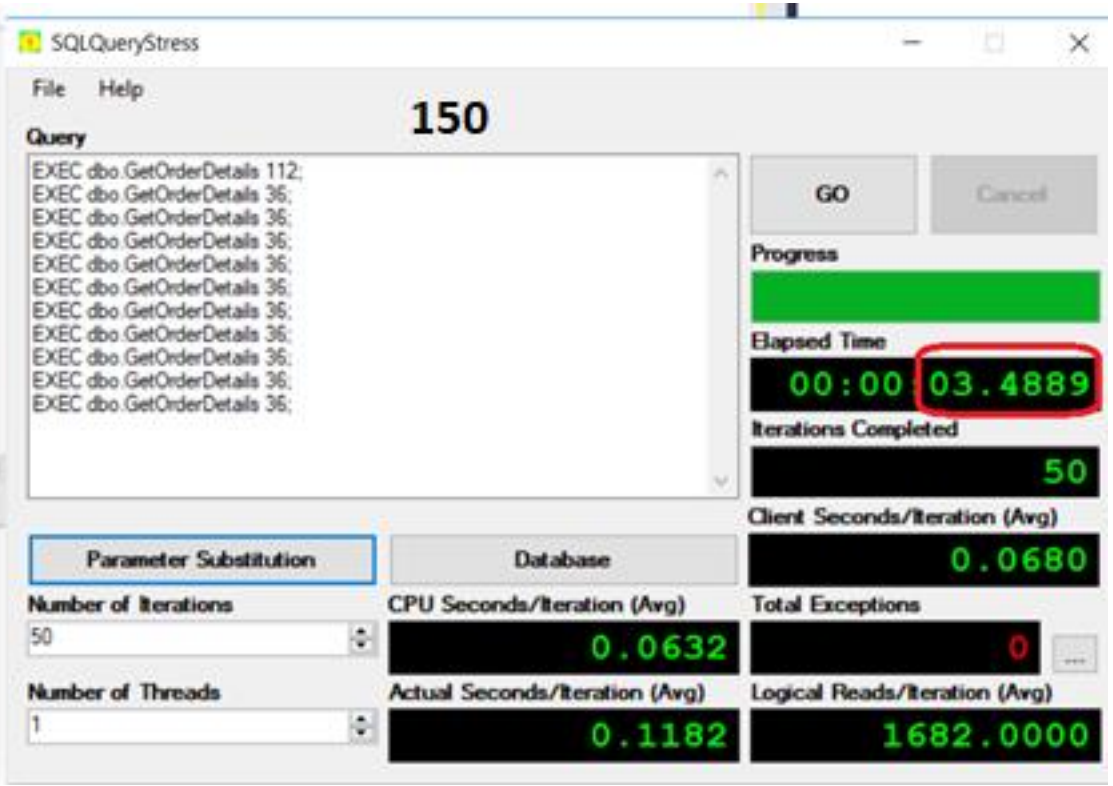




Batch Mode Adaptive Join

Q: Is it better with new Adaptive Join operator?



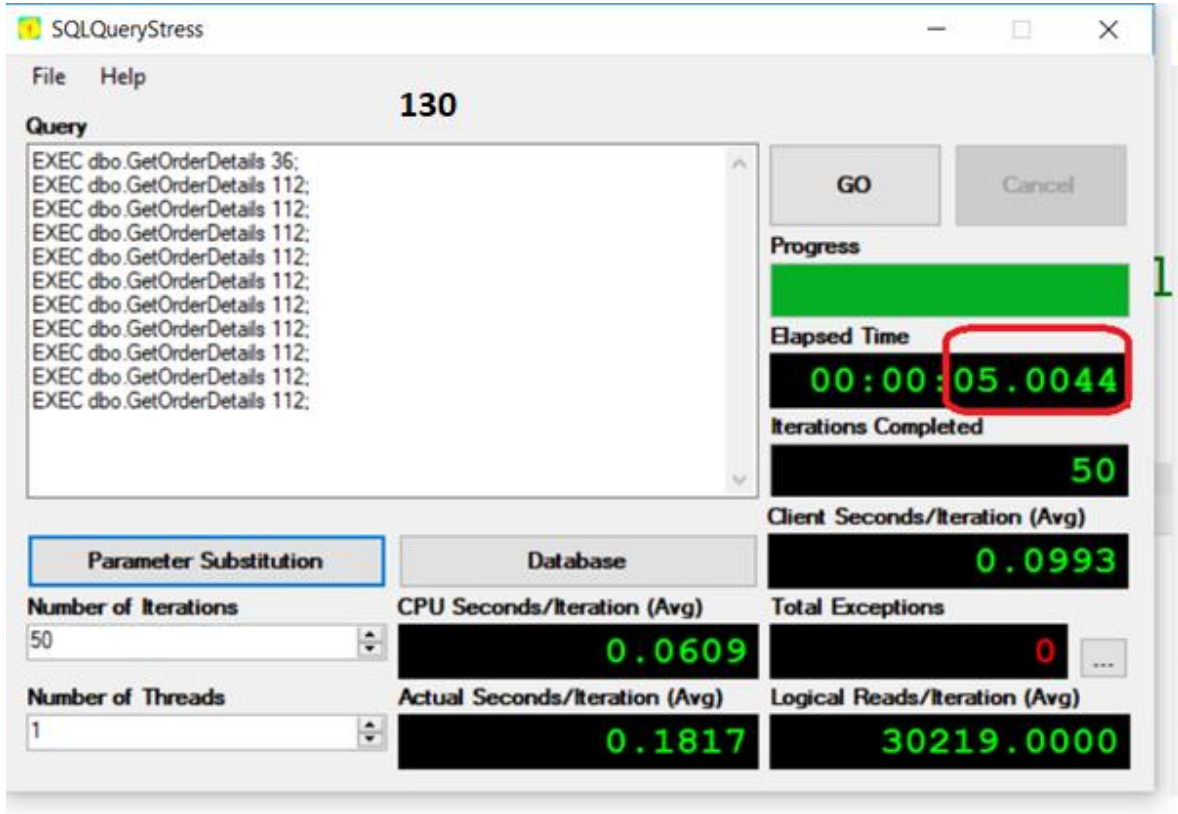


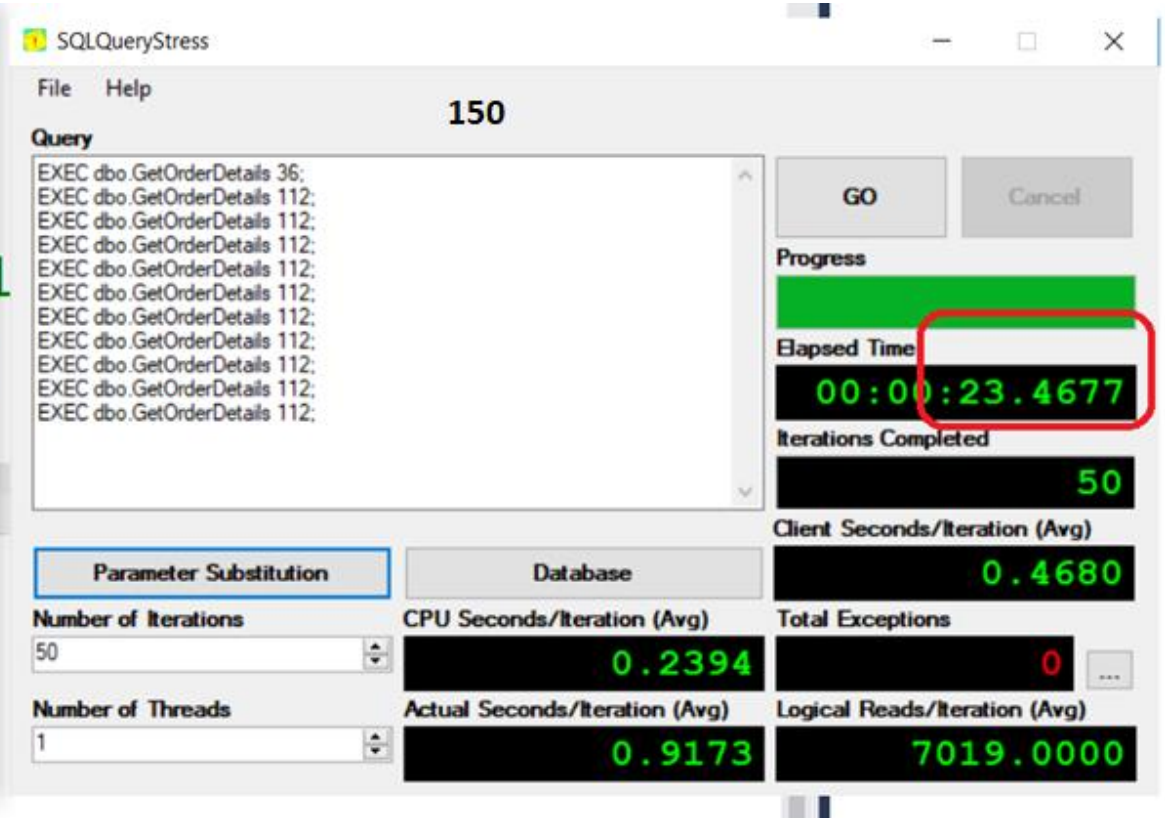
A: Yes!!! Under the CL 150, the query runs 4x faster!

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!

Batch Mode Adaptive Join - Conclusion

- It can bring benefits for some queries
- It can also bring regressions
- You should test, test and test it with your workload!
- Can handle basic parameter sniffing cases
- It requires operators in batch mode

Interleaved Execution

- Related to queries with multi statement table valued functions (MTVF)
  - Breaks the optimization process
  - Executes a part of the query with function call and get actual cardinality
- Epilogue: More appropriate plan (correct cardinality instead of cardinality 100)
- Costs: Increased CPU compile time
- Limits: It works with fixed parameters only

Interleaved Execution

```
DECLARE @d DATETIME = SYSDATETIME();
SELECT ol.OrderID,  ol.UnitPrice, ol.StockItemID FROM Sales.Orderlines olINNER JOIN dbo.SignificantOrders() f1
ON f1.Id = ol.OrderID WHERE PackageTypeID = 7;
PRINT CONCAT('Execution time: ', DATEDIFF(millisecond, @d, SYSDATETIME()), ' ms');
GO 5
```

`dbo.SignificantOrders()` cardinality: 74K rows

Compatibility Level 130

```
Beginning execution loop
Execution time: 777 ms
Execution time: 763 ms
Execution time: 781 ms
Execution time: 765 ms
Execution time: 772 ms
Batch execution completed 5 times.
```

Plan: Nested Loop Join

Compatibility Level 140

```
Beginning execution loop
Execution time: 385 ms
Execution time: 363 ms
Execution time: 368 ms
Execution time: 361 ms
Execution time: 364 ms
Batch execution completed 5 times.
```

Plan: Hash Join

2x

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 the HyperLogLog algorithm
- It's more about used resources then about the speed

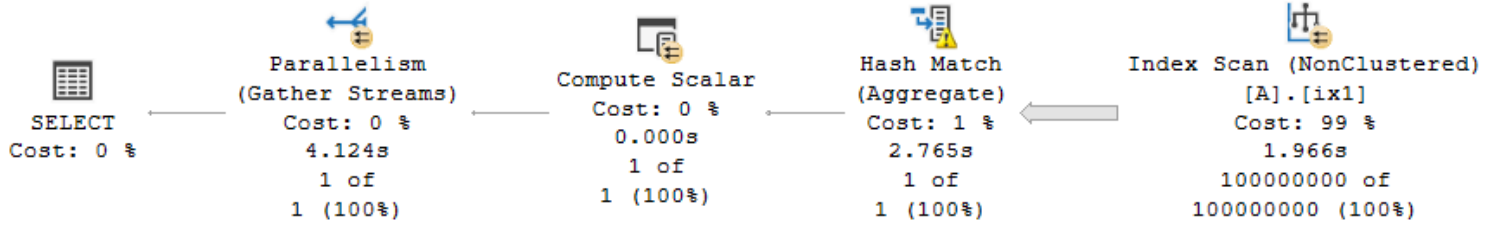
HyperLogLog Algorithm

- It hashes each element to make the data distribution more uniform
- After hashing all the elements, it looks for the binary representation of each hashed element
- HLL looks number of leading zero bits in the hash value of each element and finds maximum number of leading zero bits
- $Number\ of\ distinct\ elements$=2^($k$+1)
- where k is maximum number of leading zeros in data set
- Documentation: 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;
```

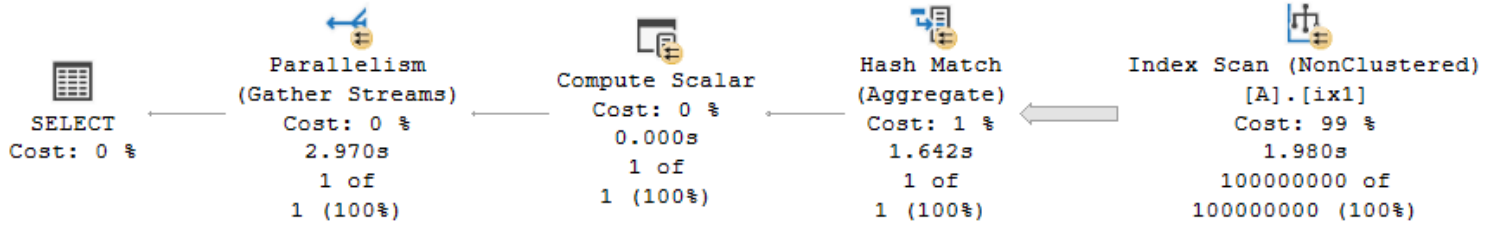
```
SQL Server Execution Times:
  CPU time = 15735 ms,  elapsed time = 2493 ms.

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

Intelligent Query Processing - Conclusion

- It will affect OLTP workload, significantly more than SQL Server 2017
- Many issues will be solved, but for specific query patterns
- Benefits can be even huge, depending on workload
- Regressions are possible, but you can easily mitigate them by disabling features at different levels
- Expectations for improvement in existing OLTP workloads
  - HIGH: Batch mode on rowstore and Memory Grant, Scalar UDF Inlining
  - LOW: Table Variable Deferred Compilation and Adaptive Joins