SQL KONFERENZ 2025

DATABASE DEVELOPMENT IN A 24/7 ENVIRONMENT



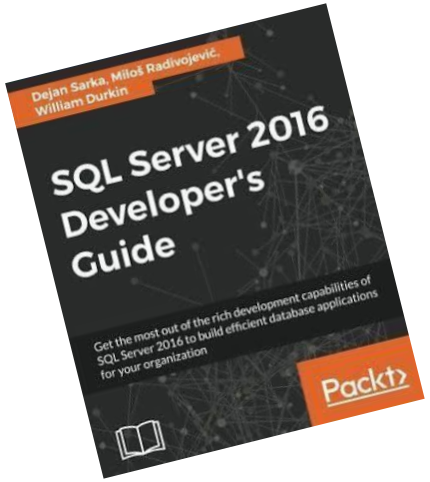
Miloš Radivojević

Entain, Vienna, Austria

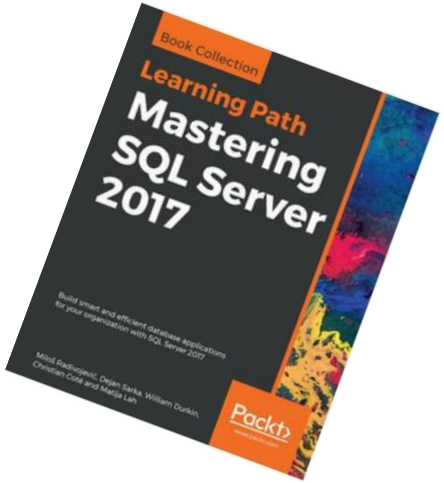
MILOŠ RADIVOJEVIĆ



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







SESSION FEEDBACK



WHAT IS A 24/7 ENVIRONMENT?



- All services and apps must be continuously **available**, **reliable**, and **well-performing**
  - All the time
  - Solutions should be scalable!
- Zero or minimal downtime for maintenance & deployments
- In my case
  - If a core database is not available for >30sec, I have to report to CTO


AVAILABILITY
RELIABILITY
PERFORMANCE

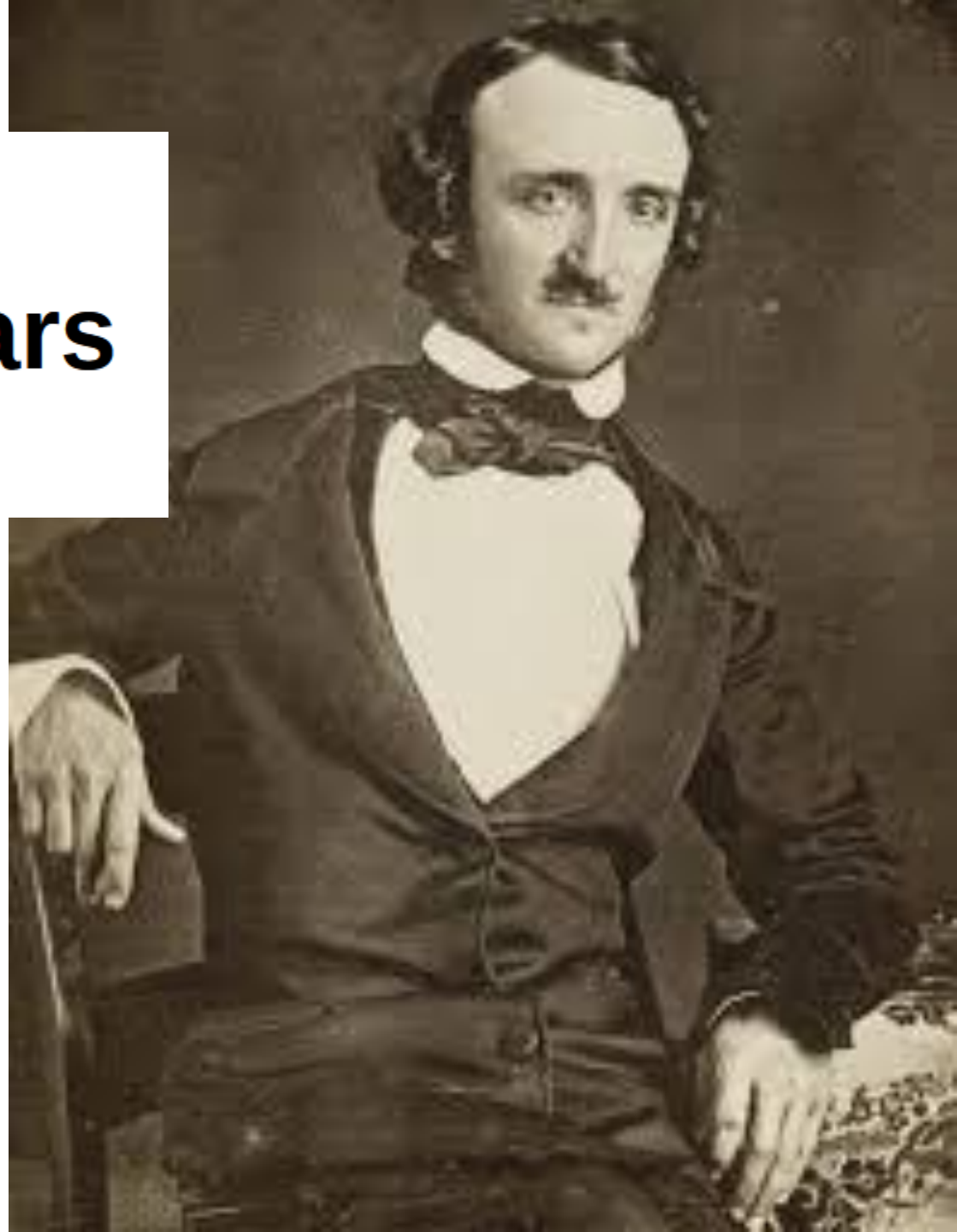
ONCE UPON A TIME

Missing Novell server discovered after four years

BOFH meets Edgar Alan Poe

In the campus of the University of North Carolina **they couldn't find the server**.

Attempts to follow network cabling to find the missing box led to the discovery that **maintenance workers had sealed the server behind a wall**.



24/7 AND A SINGLE SERVER





NO WAY!

AVAILABILITY: PROBLEMS WITH ONE SERVER

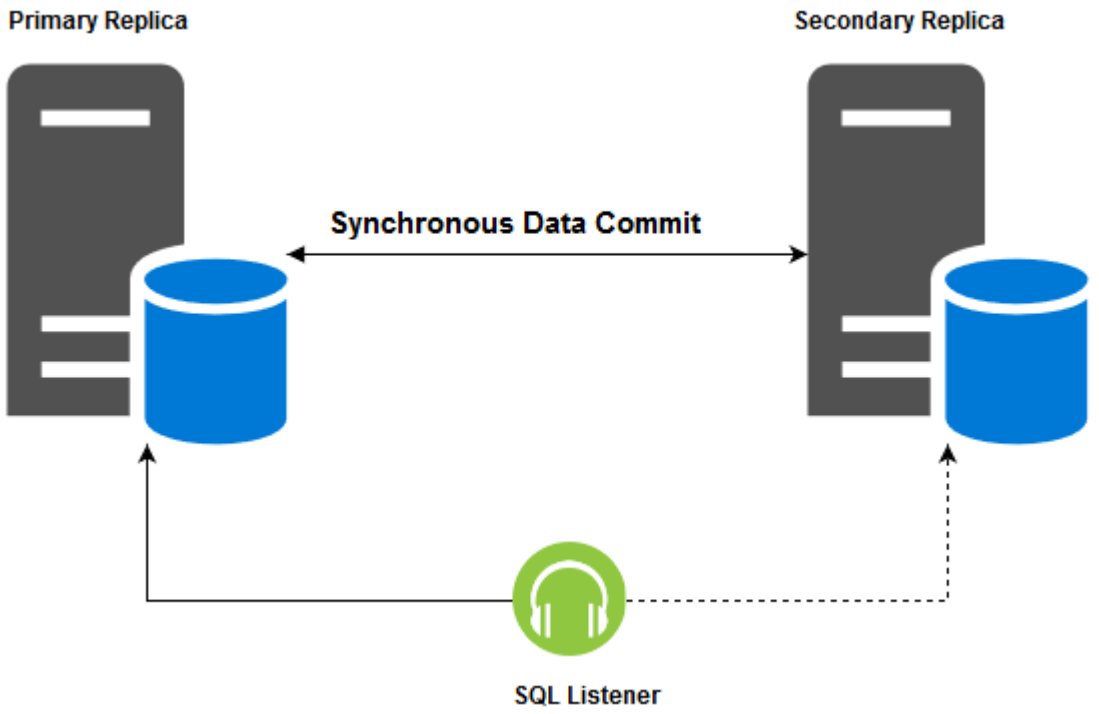
- Hardware failure
  - Disk and memory failures, power outages, network issues
- Software bugs and operating system issues
  - Bugs in SQL Server, Windows (OS) and other installed services
- Security threats
  - DDoS attacks, unauthorized access, SQL injection…

- Nature disasters, ISP & data center issues

- Therefore, you cannot seal the server ☹

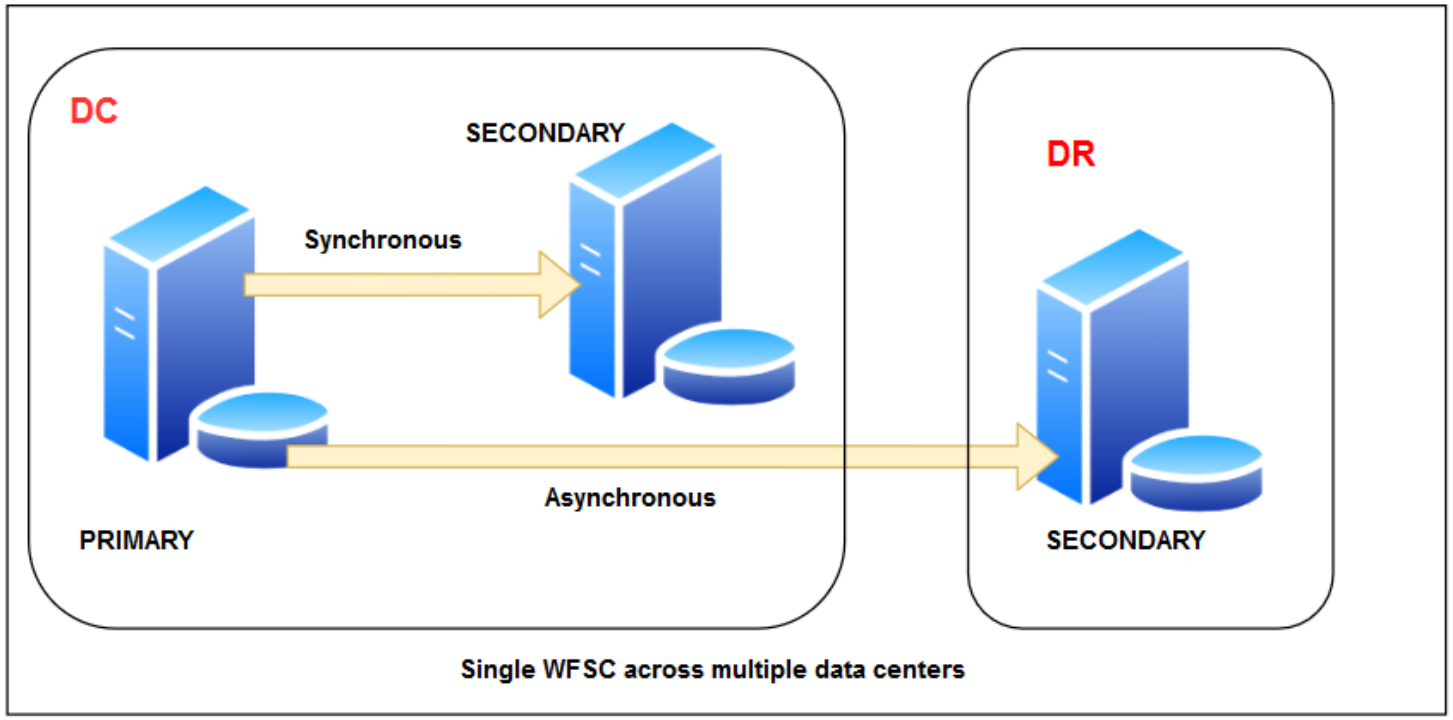
MSSQL HIGH AVAILABILITY – ALWAYS ON

- Two servers in the same or different data centers
- At least one synchronous replica
  - Bonus: additonal async replicas
- Apps and services connect to the listener=> no changes in the app in cause of a failover



* Image credit: https://www.sqlshack.com

MSSQL HA & DR SOLUTION



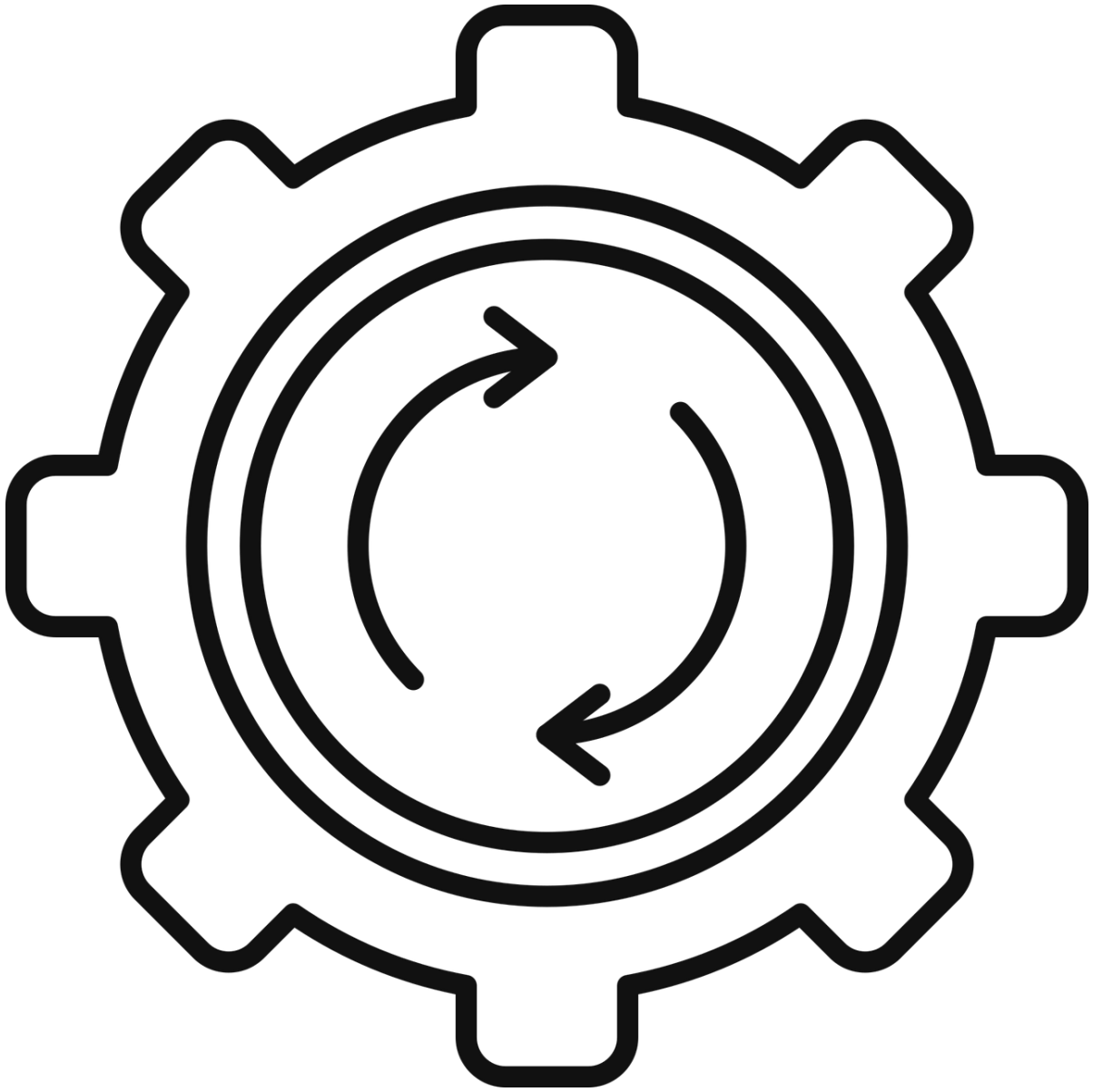
- To protect from nature disasters, ISP & data center issues
- Additional asynchronous replicas in a remote data center

* Image credit: https://www.sqlshack.com

CHANGES



- Patching
  - Bug fixes, SQL Server CUs, security patches
- SQL Server or OS version upgrade
- Configuration changes
- Hardware upgrade



PATCHING

✅ Get an OK for patching (align times with business people, remember, you are 24/7)

✅ Online Patching (Rolling Upgrades)
- Update secondary nodes first, then failover and update the primary

✅ Monitor Server Performance

✅ Automate patching (in large environments)

✅ Validation
- Run health checks, validate system performance, and confirm data integrity

PATCHING – HOW FREQUENTLY?

- At least quarterly

- Patching is a test for a failover!
- Patching means your tempdb will be recreated from scratch!

- Ideally, you would have three servers, keeping two in sync while patching the third.
- However, a more realistic approach is to patch the secondary server and temporarily operate as a single server.

SQL SERVER UPGRADE

- Why are we upgrading SQL Server to a newer version?

Support Dates

Expand table

Listing	Start Date	Mainstream End Date	Extended End Date
SQL Server 2019	Nov 4, 2019	Feb 28, 2025	Jan 8, 2030

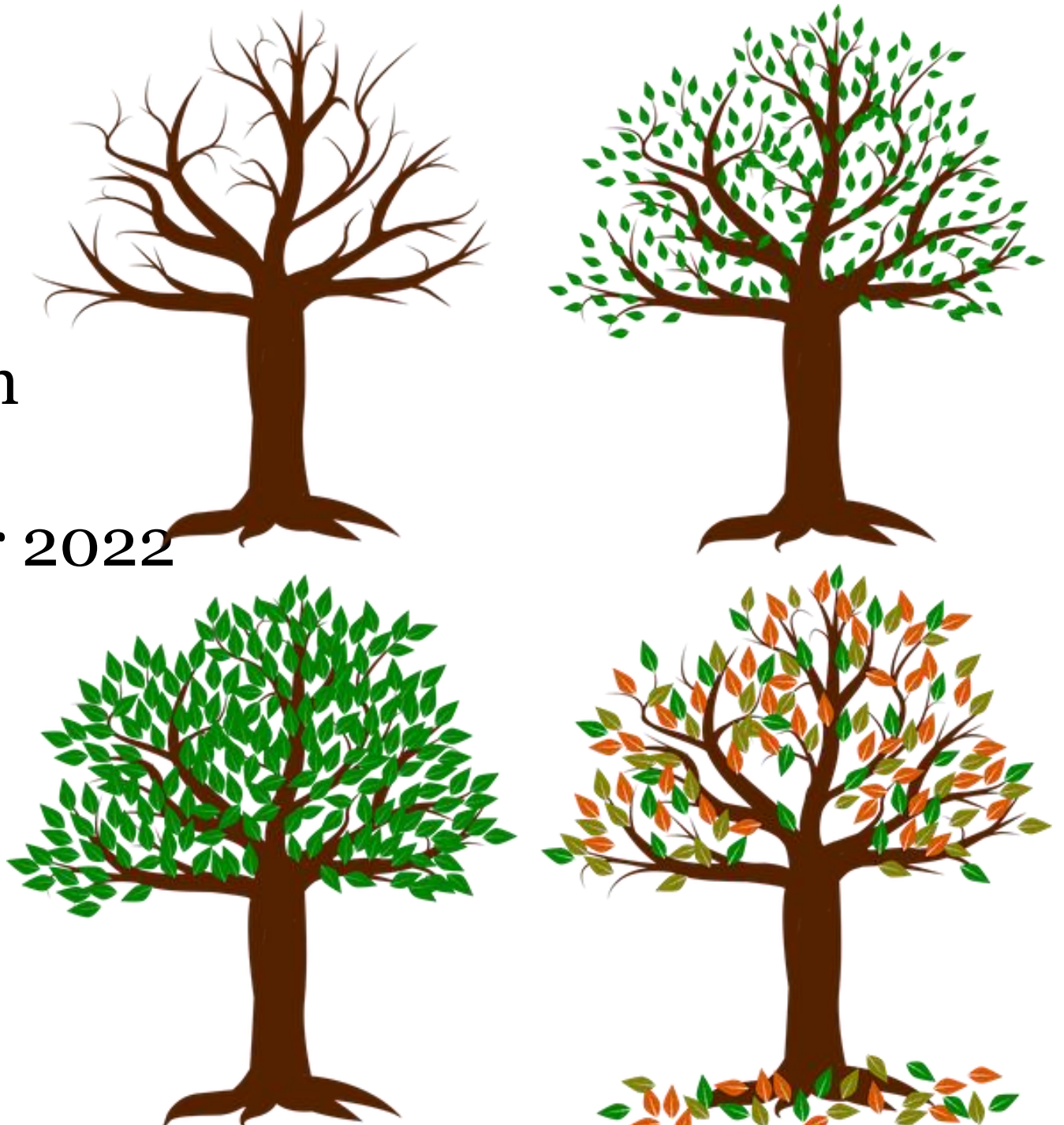
SQL SERVER UPGRADE

- In a 24/7 a **risky, resource, people- and time-intensive** activity
- It usually takes **several months** (dev/test/stress/prod), aligning dates with business etc.
- Usually **no expectations** from the new version
- **Do not change Compatibility Level** of databases during the upgrade, change it later!



Execution plans and execution statistics for one query

SQL Server 2012 - 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
```

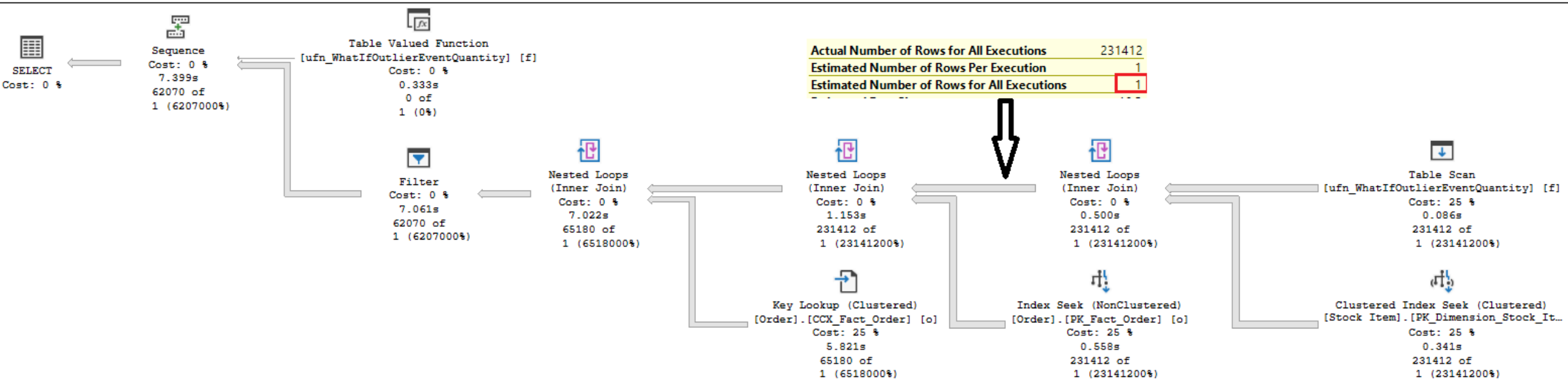
SQL Server 2012 / CL 110

```
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
```



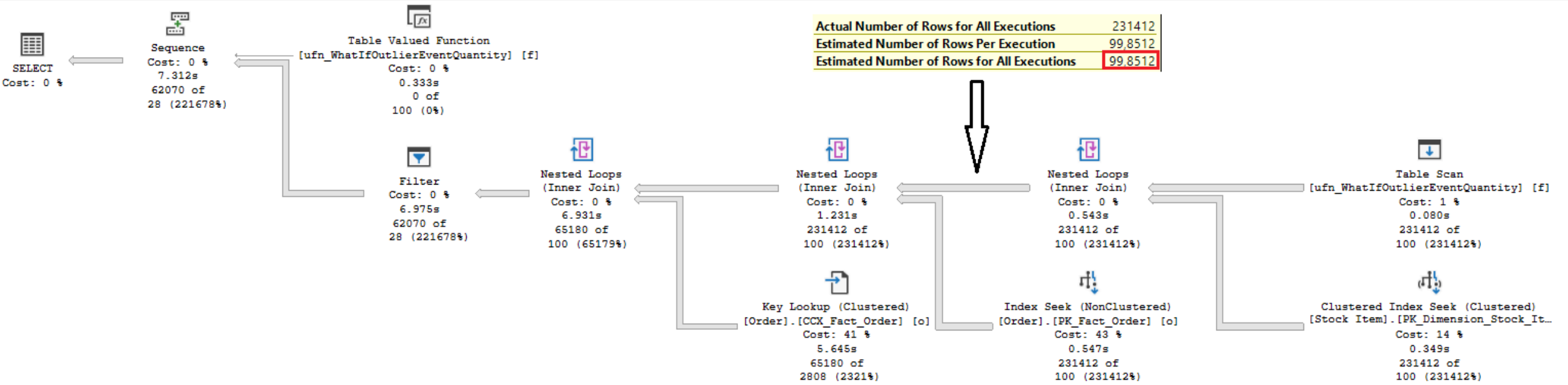
SQL Server 2014 / CL 120

```
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 %

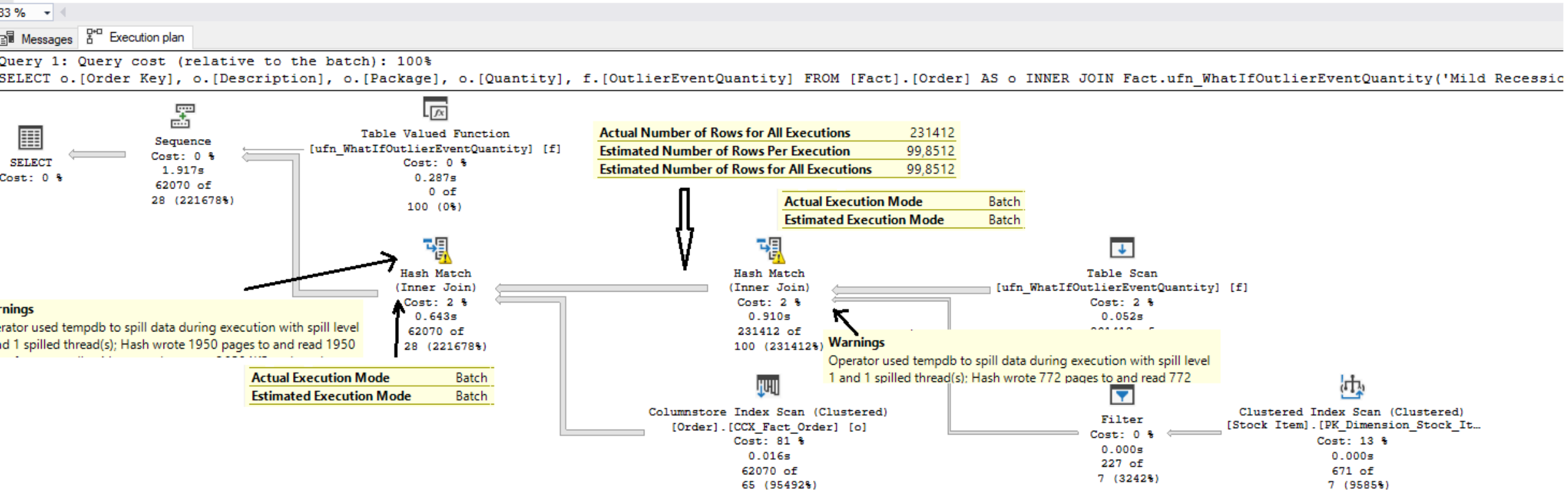
Messages | Execution plan

```
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
```



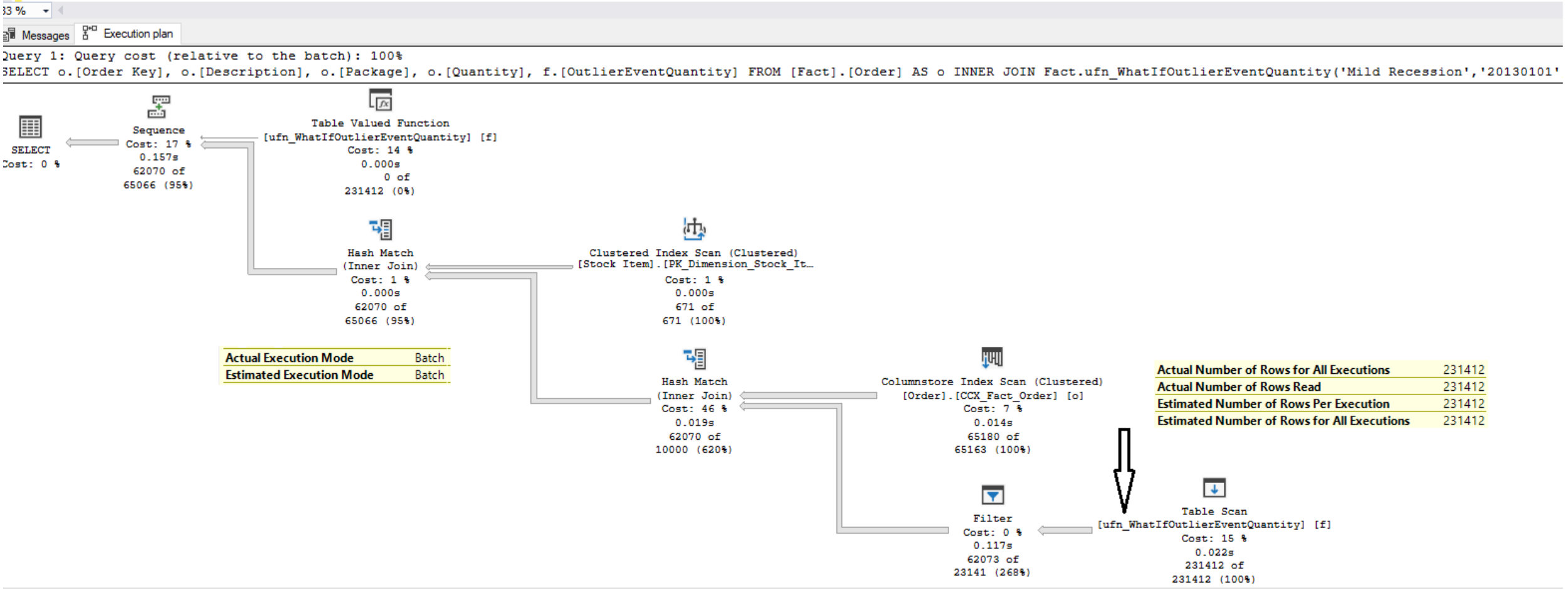
SQL Server 2016 / CL 130

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



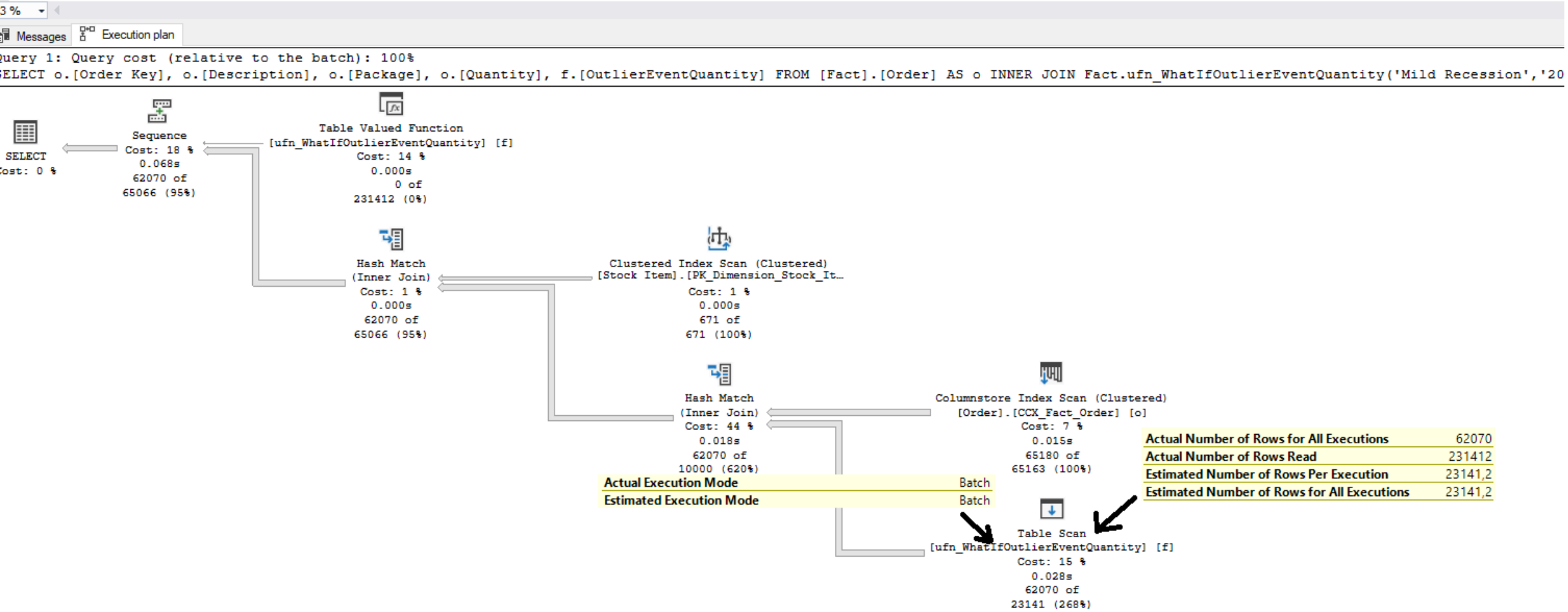
SQL Server 2017 / CL 140

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



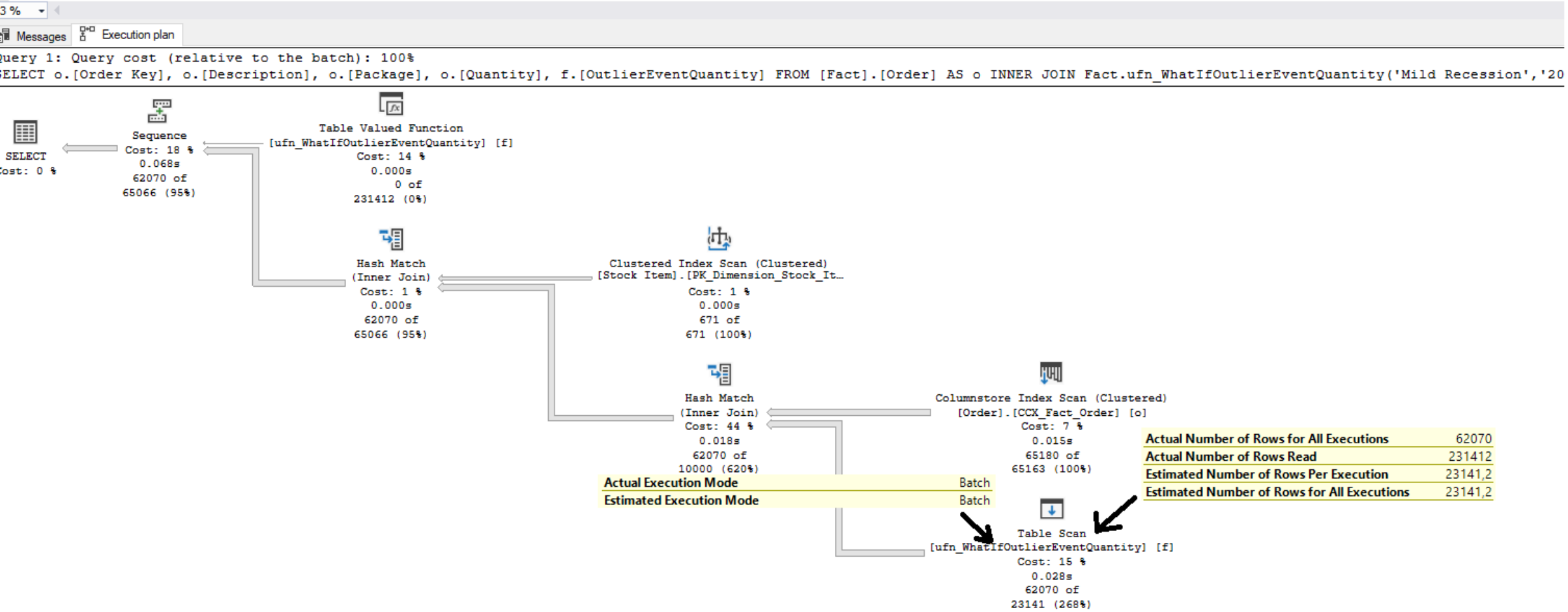
SQL Server 2019 / CL 150

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



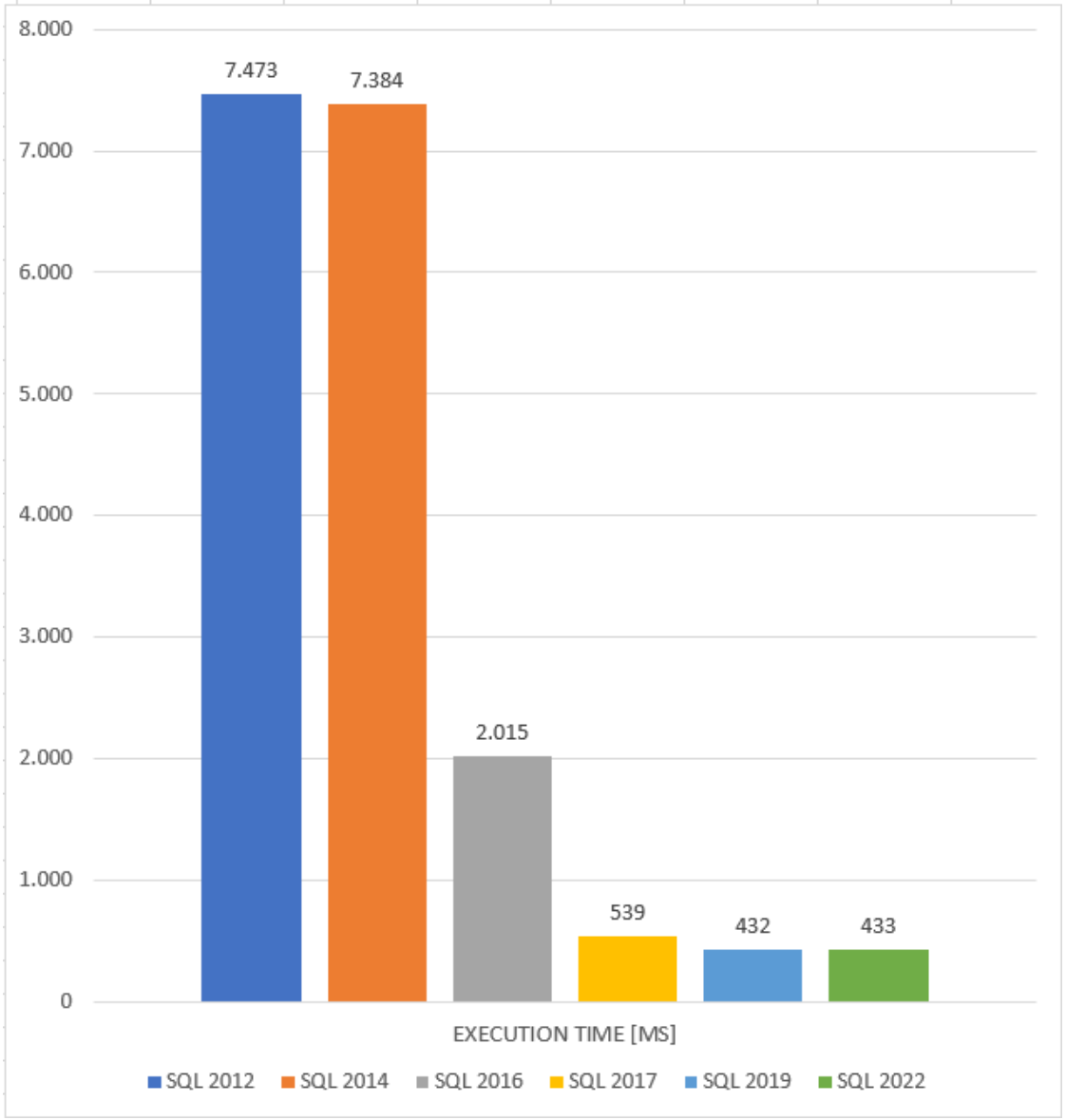
SQL Server 2022 / CL 160

```sql
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 have not fine-tuned their queries
  - those managing a large number of databases
- Some features could even be dangerous
  - For databases where performance is critical to business operations

SQL SERVER UPGRADE

- Intensive testing in non-prod environments
- Be careful with Intelligent Query Processing
- Use **Query Store** to identify and mitigate performance regressions
  - Upgrade the server version, leaving the old CL
  - Let Query Store collect execution plans and statistics
  - Upgrade the CL to the latest version and **use Query Store to mitigate** regressions
- **Involve** developers!
- **Teach them** about new features and capabilities!



AVAILABILITY

- ✅ HA Setup Completed
- ✅ Upgraded
- ✅ Patched

YES,



BUT



RESTRICT ACCESS

- Nobody should have access to prod databases except DBAs
- An inexperienced or malicious user with sufficient privileges can:
  - Manipulate data
  - Add, remove, or modify database objects
  - Run expensive queries
  - Export data
  - Install unauthorized software on the server



- This can compromise both **reliability** and **performance**

RESTRICT ACCESS: EXCEPTIONS

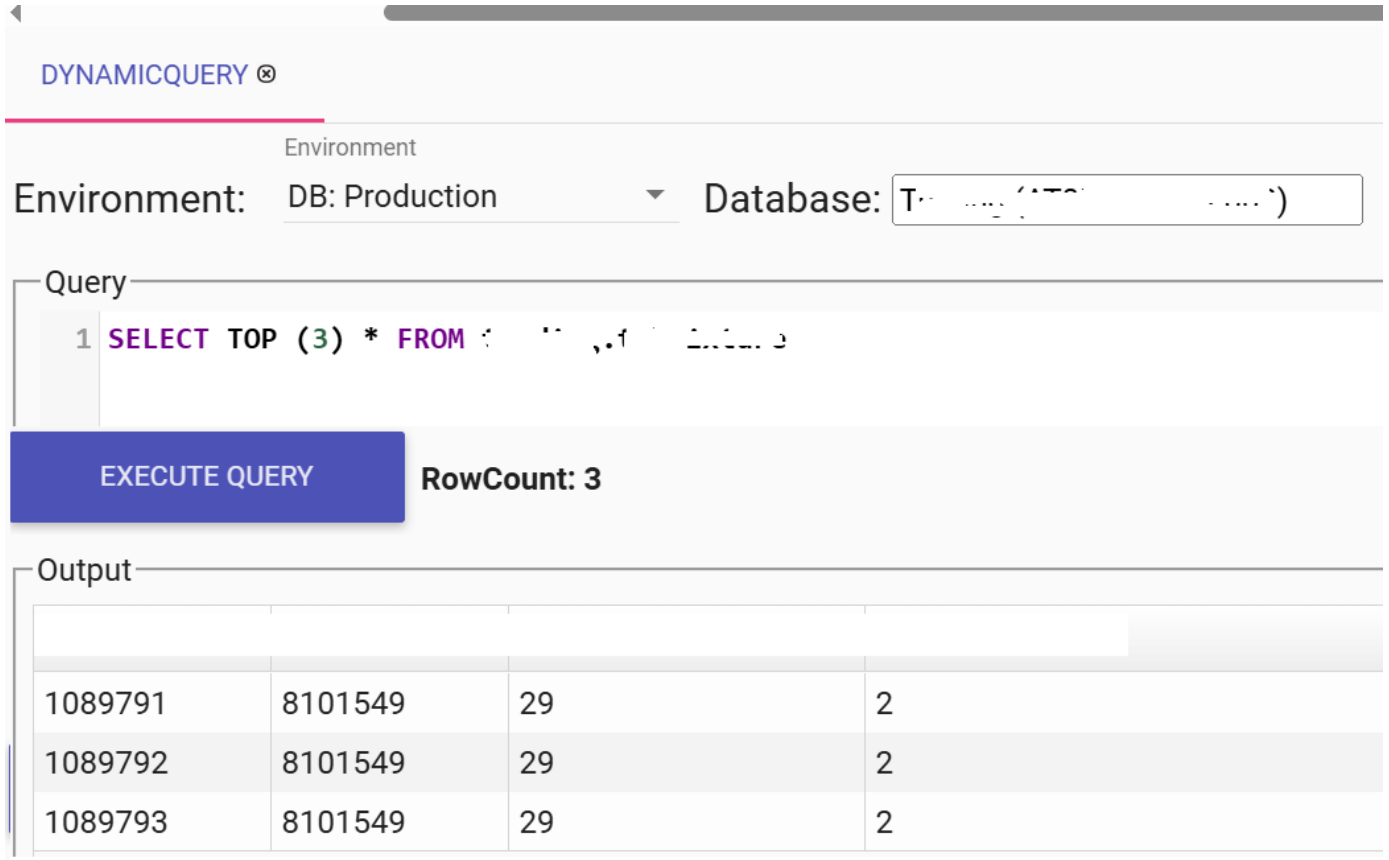
- Only for those involved in troubleshooting
- Limited to:
  - VIEW SERVER STATE
  - VIEW DATABASE STATE
  - Similar permissions
- No access to data or user database objects!



ACCESS TO PRODUCTION DATA

- Either a delayed copy or an almost real-time async replica
- Alternatively, create your own application and restrict access
- **Do not allow queries to run directly on the production database!**

OUR TOOL FOR QUERYING PROD DATA



- Read-only access
- Limited to 5K rows
- Estimated subtree cost threshold applied

- Users can access the application, but they have no direct permissions on the server or database

YES,

- ✅ HA Setup Completed
- ✅ Upgraded
- ✅ Patched
- ✅ Access restricted

BUT



ROLLOUTS

APPLICATION AND SERVICE ROLLOUTS



- Who writes Transact-SQL code?
  - Application developers
  - Database developers
  - DBAs

WHAT CAN HAPPEN AFTER A ROLLOUT?

- New code can compromise r**eliability** and p**erformance**
  - Not only its own service but also neighboring databases

- How to avoid it?
  - Testing functionality
  - Testing performance (tricky, resource- and time-intensive, boring, and lacking knowledge)
  - Following SQL best practices
  - **Code review**

TRANSACT-SQL CODE REVIEW - TOOLS

- Redgate **SQL Prompt**
  - Good for formatting, clean code
- Monitoring tools
- **VS Code**
- **ChatGPT**
- Limited to basic and common advices and suggestions
- What a reviewer **cannot** answer is whether the code meets the requirements and whether it is performant, in best case, he/she/they can find **common mistakes**

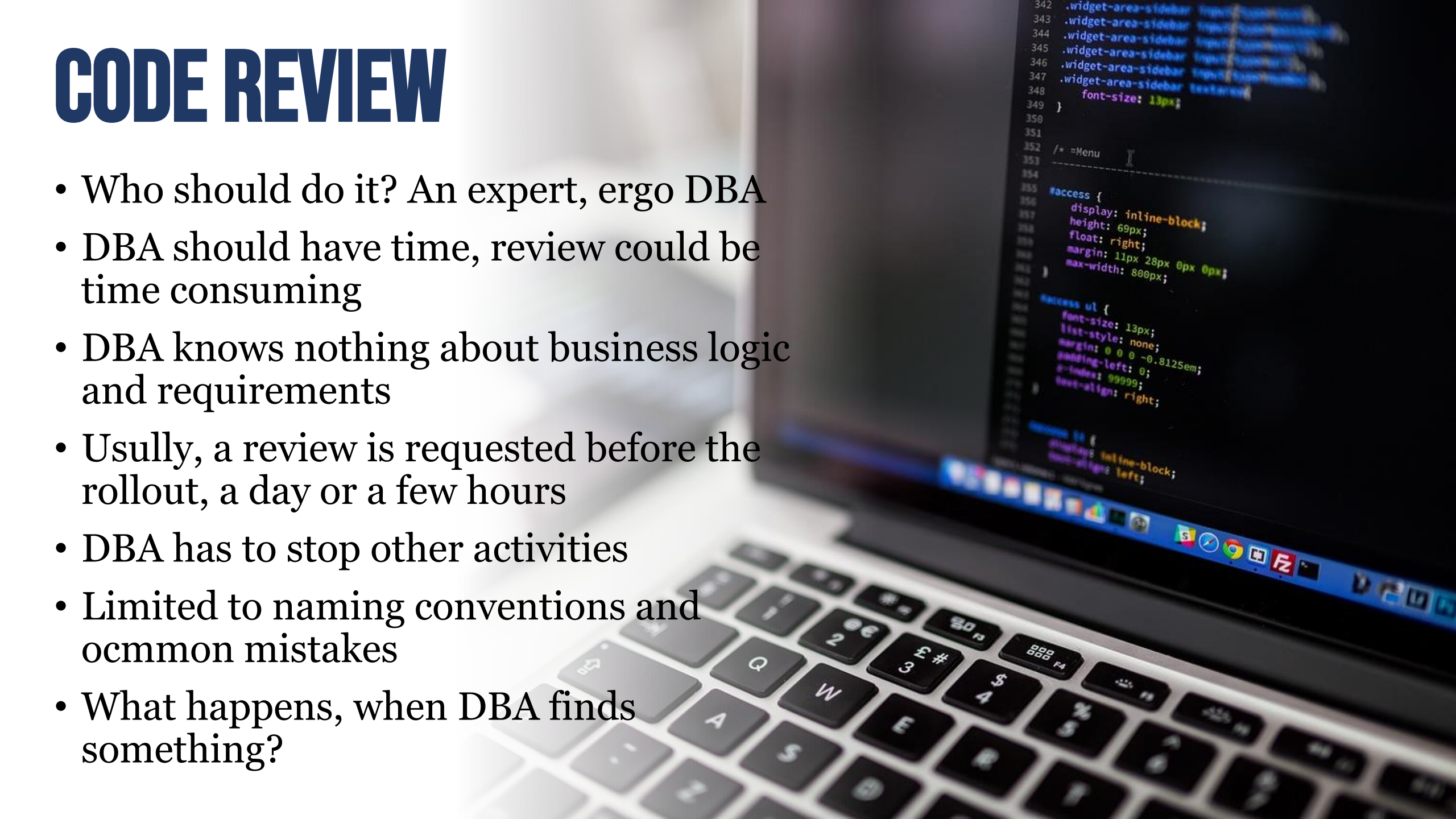






CODE REVIEW

- Who should do it? An expert, ergo DBA
- DBA should have time, review could be time consuming
- DBA knows nothing about business logic and requirements
- Usully, a review is requested before the rollout, a day or a few hours
- DBA has to stop other activities
- Limited to naming conventions and ocmmon mistakes
- What happens, when DBA finds something?



MY RECOMMENDATION

- **Proactive** code review
  - Consider support during development, not after
  - Continuous knowledge improvement
  - Involve DBAs or other experts
- **Follow up incidents** and issues!



- **Educate** your developers!
  - Ensure they follow best practices
  - Help them learn about new features and capabilities in SQL Server

DEPLOYMENT CHALLENGES

- Who should deploy the code?
  - **DBAs**
  - Tested, reliable and trusted tools
  - Developers should **never be allowed** to deploy the code
- Schema changes without downtime
  - Check for existence of objects before creating
  - Online operations (**ONLINE =ON**) for index rebuilding and altering columns
  - Rolling back changes without disruption
- Version Control for Transact-SQL code
  - Developer should know the exact version of all database objects in production by checking the code repository



AN EXAMPLE

```
IF OBJECT_ID('dbo.tabProdTable','U') IS NULL
CREATE TABLE dbo.tabProdTable(
    Id BIGINT NOT NULL,
    C1 BIGINT NOT NULL,
    C2 SMALLINT NOT NULL,
    C3 DATETIME2(7) NOT NULL CONSTRAINT DF_tabProdTable_C3 DEFAULT (SYSUTCDATETIME()),
    CONSTRAINT PK_tabProdTable PRIMARY KEY CLUSTERED
    (
        Id ASC
    )
)
ELSE
BEGIN
    IF NOT EXISTS(SELECT 1 FROM sys.columns WHERE name = N'C2' AND object_id=OBJECT_ID('dbo.tabProdTable'))
        ALTER TABLE dbo.tabProdTable ADD C2 SMALLINT NULL;
END
GO
-----------------------------
--Indexes
-----------------------------
IF NOT EXISTS(SELECT 1 FROM sys.indexes WHERE name = 'IX_tabProdTable_C1' AND object_id = OBJECT_ID('dbo.tabProdTable'
BEGIN
    CREATE INDEX IX_tabProdTable_C1
    ON dbo.tabProdTable(C1) WITH (ONLINE = ON, MAXDOP = 8);
END
GO
```

DEVELOPMENT AND DEPLOYMENT RECOMMENDATIONS

- **Use stored procedures!**
  - For performance troubleshooting, security and visibility
- Move reporting queries to prod copies or datawarehouse
- Update/delete in chunks
- The principle of least privilege
  - Services and apps are getting minimum privileges needed to carry out their job
- **Knowledge transfer** experts ⇔ developers

YES,

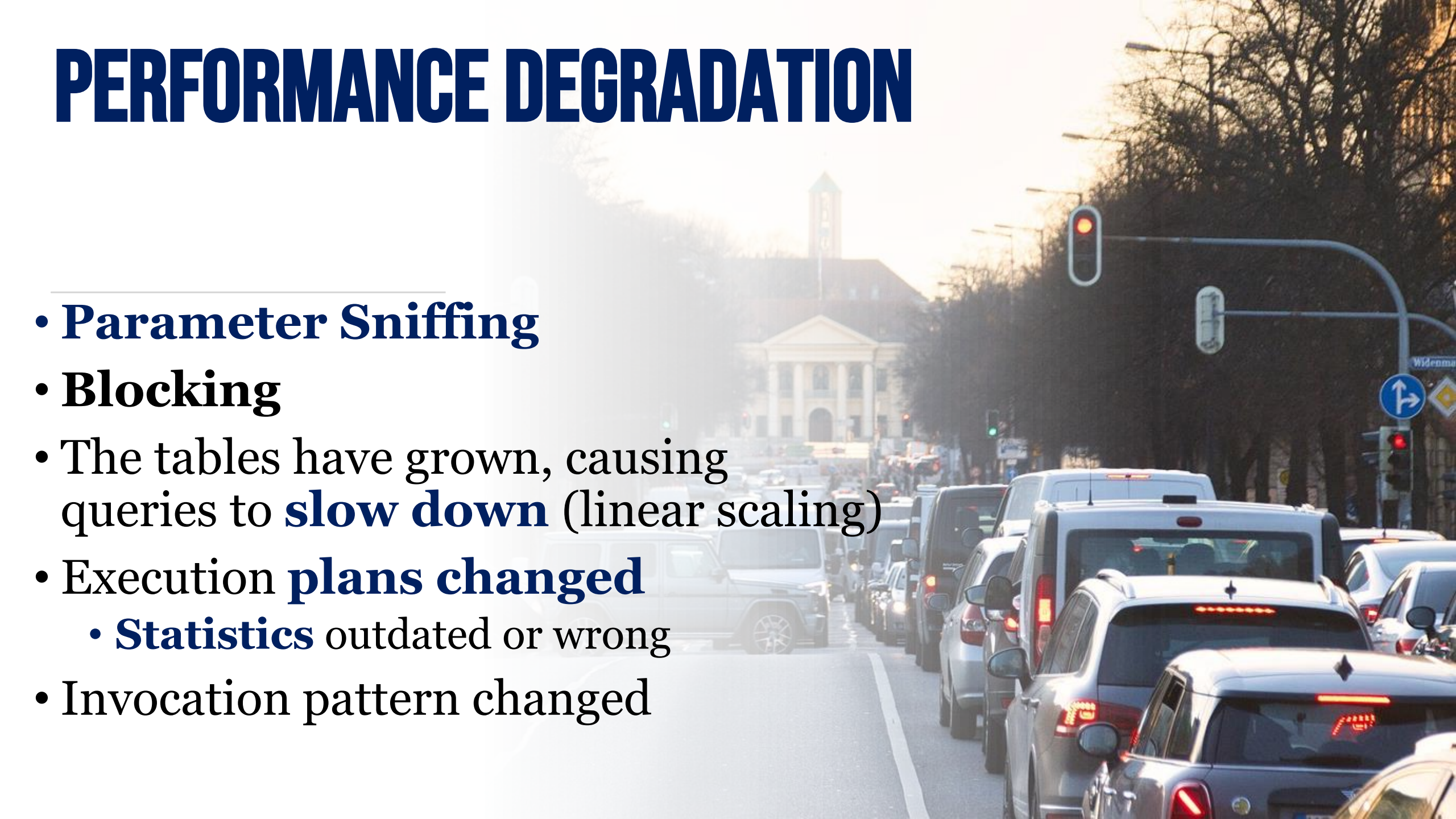
- ✅ HA Setup Completed
- ✅ Upgraded
- ✅ Patched
- ✅ Access restricted
- ✅ Rollouts controlled
- ✅ Developers educated

BUT

PERFORMANCE DEGRADATION



PERFORMANCE DEGRADATION



- **Parameter Sniffing**
- **Blocking**
- The tables have grown, causing queries to **slow down** (linear scaling)
- Execution **plans changed**
  - **Statistics** outdated or wrong
- Invocation pattern changed



PARAMETER SNIFFING

- Performance suddenly becomes slow
- It can lead to many **serious issues**
- Almost **all apps** are prone to it
- **There is no single solution!**
  - Especially no PSP optimization in SQL Server 2022.
- **Teach developers** how to handle it!

BLOCKING AND CONCURRENCY ISSUES

- Teach developers about **isolation levels** and concurrency
- Everyone should understand what **NOLOCK** means

- Consider **SNAPSHOT** isolation levels



INEFFICIENT QUERIES AND LINEAR SCALING

- Inefficient and suboptimal queries are not visible with small and moderate tables
- Common developer mistakes
  - Using functions and arithmetic operations in filters
  - Data type conversion
  - Using scalar UDFs
  - Not understanding local variables
  - Using on-SARG-able predicates
- **Teach developers!**

FUNCTIONS IN THE WHERE CLAUSE

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

133 %

Results | Messages

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

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

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

 SQL Server Execution Times:
```

FUNCTIONS IN THE WHERE CLAUSE

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

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

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

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

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

DATA TYPE CONVERSION

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

33 %

Results | Messages | Execution plan

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

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

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

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

DATA TYPE CONVERSION

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

33 %

Results | Messages | Execution plan

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

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

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

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

LOCAL VARIABLES

```
USE AdventureWorks2019;
GO
DECLARE @x DATETIME = GETDATE()
SELECT * FROM Sales.SalesOrderHeader WHERE OrderDate >= @x;
GO
SELECT * FROM Sales.SalesOrderHeader WHERE OrderDate >= GETDATE();
```

Results | Messages | Execution plan

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

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

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
Table 'SalesOrderHeader'. Scan count 1, logical reads 689, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physica

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 29 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0
Table 'SalesOrderHeader'. Scan count 1, logical reads 2, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 25 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

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

LOCAL VARIABLES

```sql
USE TransactSQLTips;
GO
DECLARE @x DATETIME = GETDATE()
SELECT * FROM dbo.Orders WHERE OrderDate >= @x;
GO
SELECT * FROM dbo.Orders WHERE OrderDate >= GETDATE();
```

10 %

Results | Messages

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

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

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

(0 rows affected)
Table 'Orders'. Scan count 1, logical reads 715443, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads

 SQL Server Execution Times:
   CPU time = 1219 ms,  elapsed time = 1237 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

(0 rows affected)
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0,
Table 'Orders'. Scan count 1, logical reads 3, physical reads 0, page server reads 0, read-ahead reads 0, page server read-ahead reads 0, lob logical reads 0, lob physical reads 0, lo

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

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

SCALAR UDFS

```sql
USE AdventureWorks2019;
GO
SELECT dbo.GetFiscalYear(OrderDate), COUNT(1)
FROM Sales.SalesOrderHeader o
GROUP BY dbo.GetFiscalYear(OrderDate) ORDER BY 1
GO
SELECT x.Fisc, COUNT(1)
FROM Sales.SalesOrderHeader o
CROSS APPLY dbo.GetFiscalYearInline(o.OrderDate) x
GROUP BY x.Fisc ORDER BY 1
```

Results | Messages

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

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

 SQL Server Execution Times:
   CPU time = 125 ms,  elapsed time = 117 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

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

Completion time: 2024-11-13T15:31:43.9141720+01:00
```

SCALAR UDFS

```sql
USE TransactSQLTips;
GO
SELECT dbo.GetFiscalYear(OrderDate), COUNT(1)
FROM dbo.Orders
GROUP BY dbo.GetFiscalYear(OrderDate) ORDER BY 1
GO
SELECT x.Fisc, COUNT(1)
FROM dbo.Orders o
CROSS APPLY dbo.GetFiscalYearInline(o.OrderDate) x
GROUP BY x.Fisc ORDER BY 1
```

10 %

Results | Messages | Execution plan

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

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

 SQL Server Execution Times:
   CPU time = 29609 ms,  elapsed time = 33684 ms.
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 1812 ms,  elapsed time = 265 ms.
   CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2024-11-13T15:33:21.7012441+01:00
```

SERIAL PLAN

PARALLEL PLAN

DBA ACTIVITIES

- Statistics
  - **Asynchronuos stats update** => True
  - For large tables, **update stats manually, with a larger sample**

- Index maintenance
  - Check index usage statistics
  - Consolidate indexes
  - Consider filtered and columnstore indices, indexed views
  - Perform defragmentation only if you're sure it's a problem

AUTO UPDATE STATISTICS

- When are statistics automatically updated?

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

- Until SQL Server 2012 => 20% of rows
- From SQL Server 2014 => $\sqrt{(1000 * Table\ Card)}$

AUTO UPDATE STATISTICS

- Old vs. New Threshold

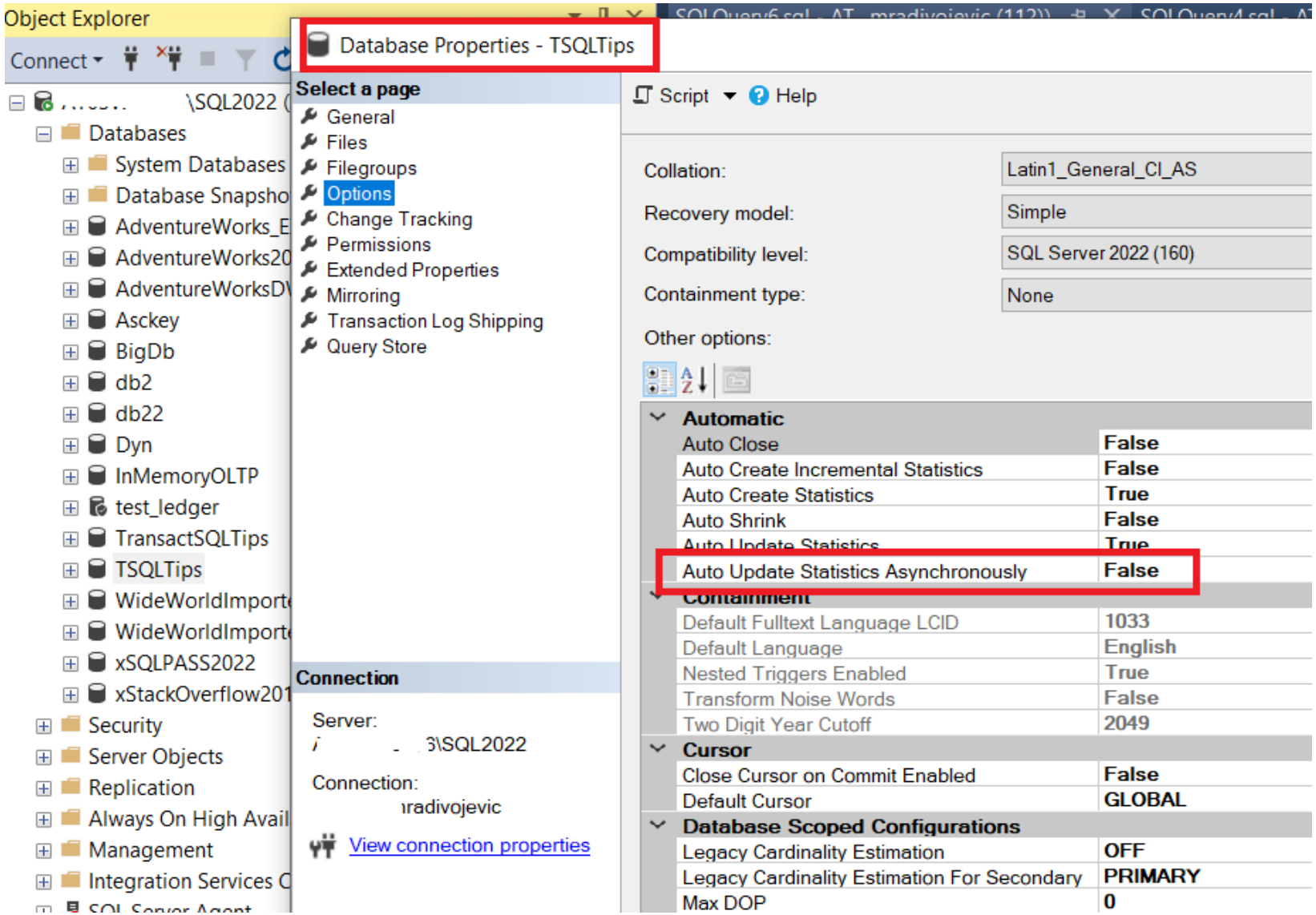
ROWS	OLD THRESHOLD	NEW THRESHOLD
100.000	20.000	10.000
1.000.000	200.000	31.622
10.000.000	2.000.000	100.000
100.000.000	20.000.000	316.227
1.000.000.000	200.000.000	1.000.000

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

AUTO UPDATE STATISTICS

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

AUTO UPDATE STATS ASYNC

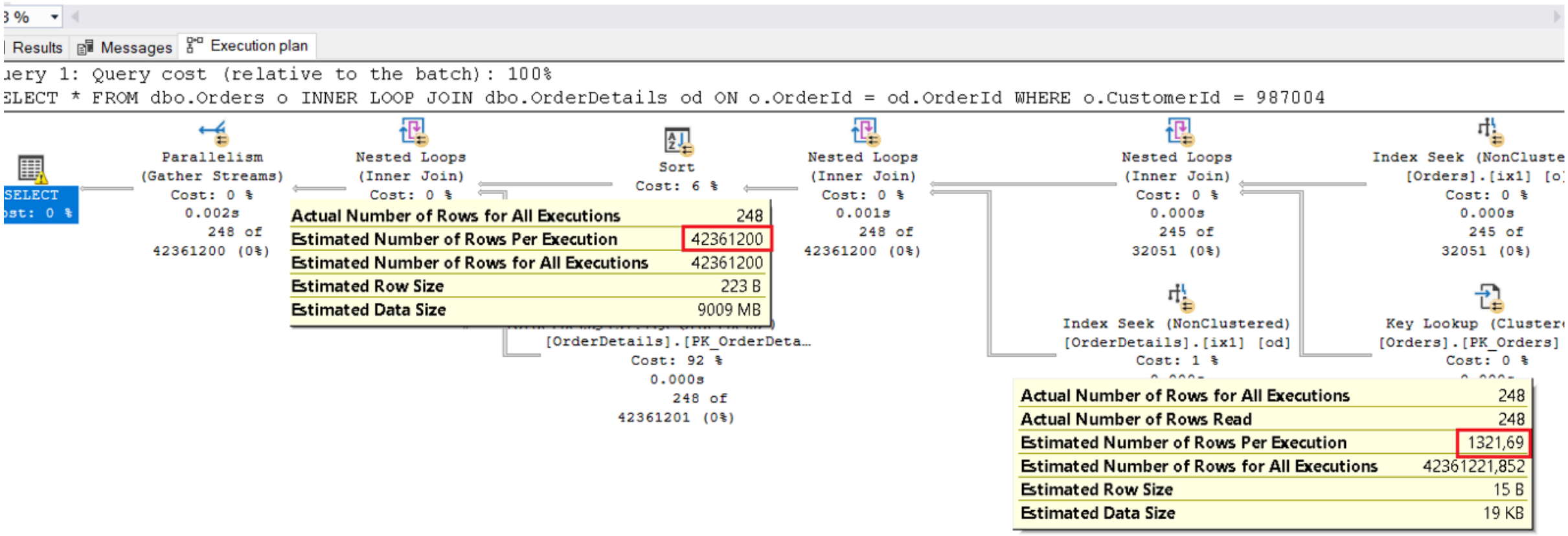


- Default is **False**
- It should be set to **True**

CARDINALITY ESTIMATION ISSUES

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

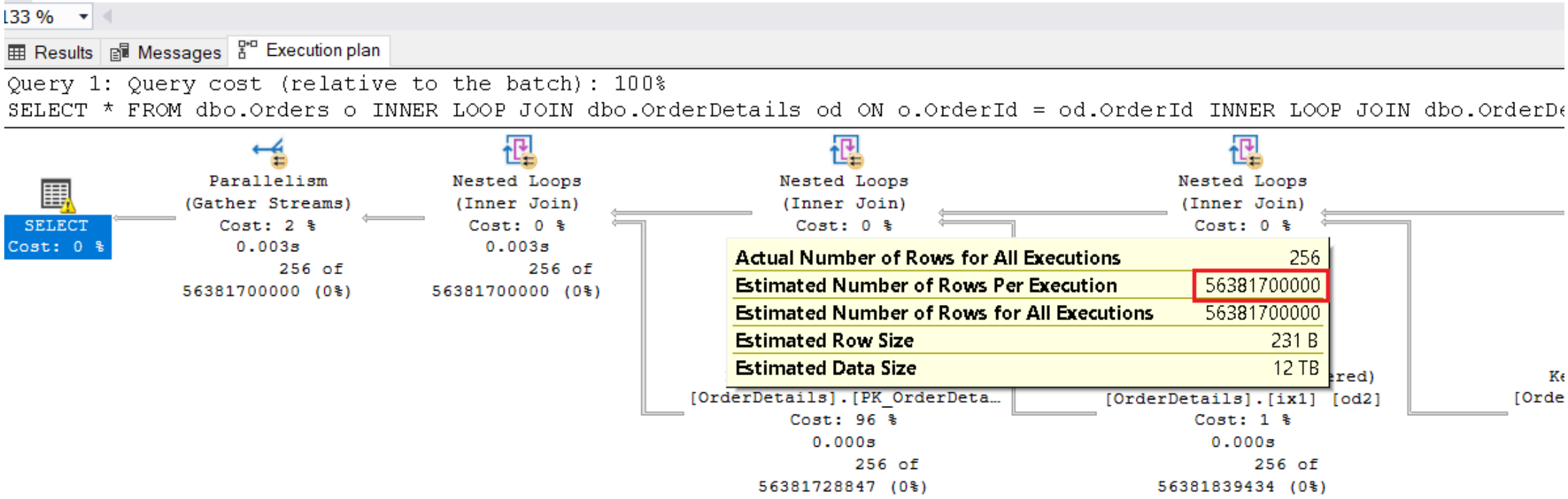
Overestimation – two tables



CARDINALITY ESTIMATION ISSUES

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

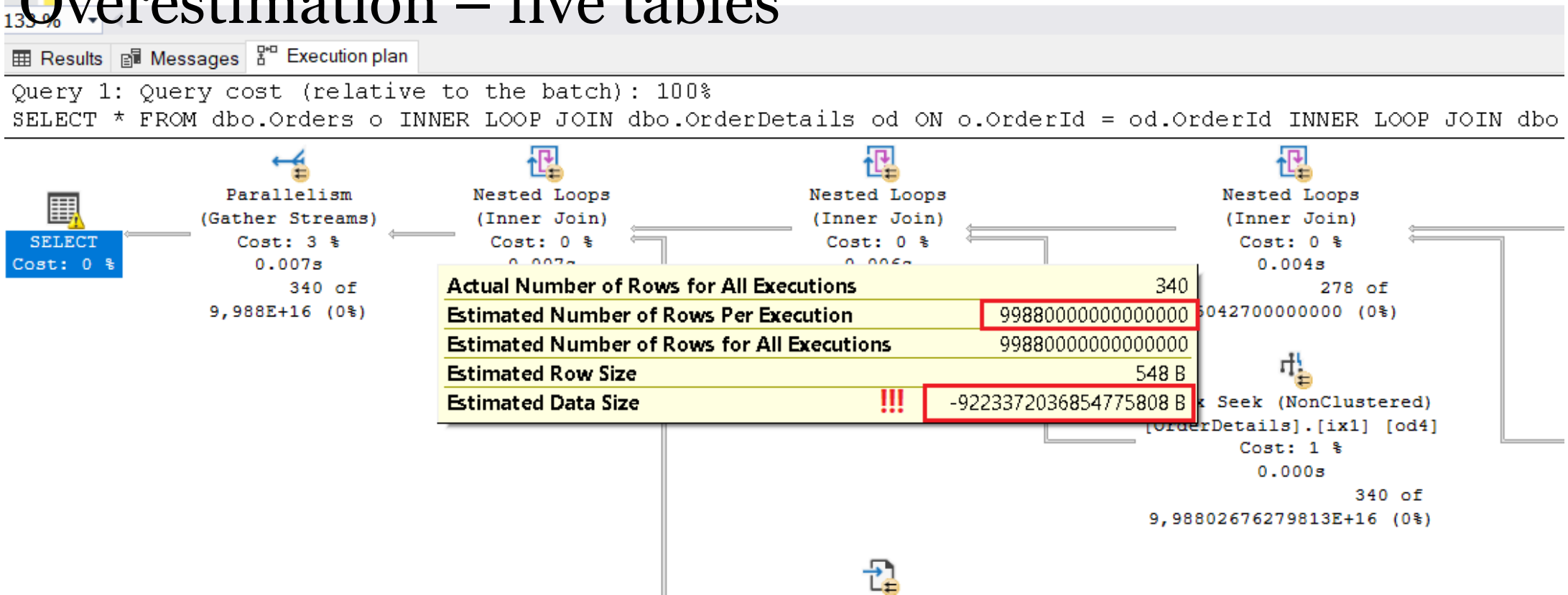
Overestimation – three tables



CARDINALITY ESTIMATION ISSUES

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

Overestimation – five tables



WHY IS CARDINALITY ESTIMATION SO IMPORTANT?

- Cardinality Estimation Errors – Reason #1 for Poor Execution Plans
- Discrepancies in estimations can lead to:
  - Inappropriate join strategies
  - The selection of a serial plan when parallelism would have been more optimal
  - Inappropriate index/table access strategies (Scan vs. Seek)
  - Memory spills to disk or inflated memory grants

AUTO UPDATE STATISTICS

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

AUTO UPDATE STATISTICS

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

SOLUTION

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

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

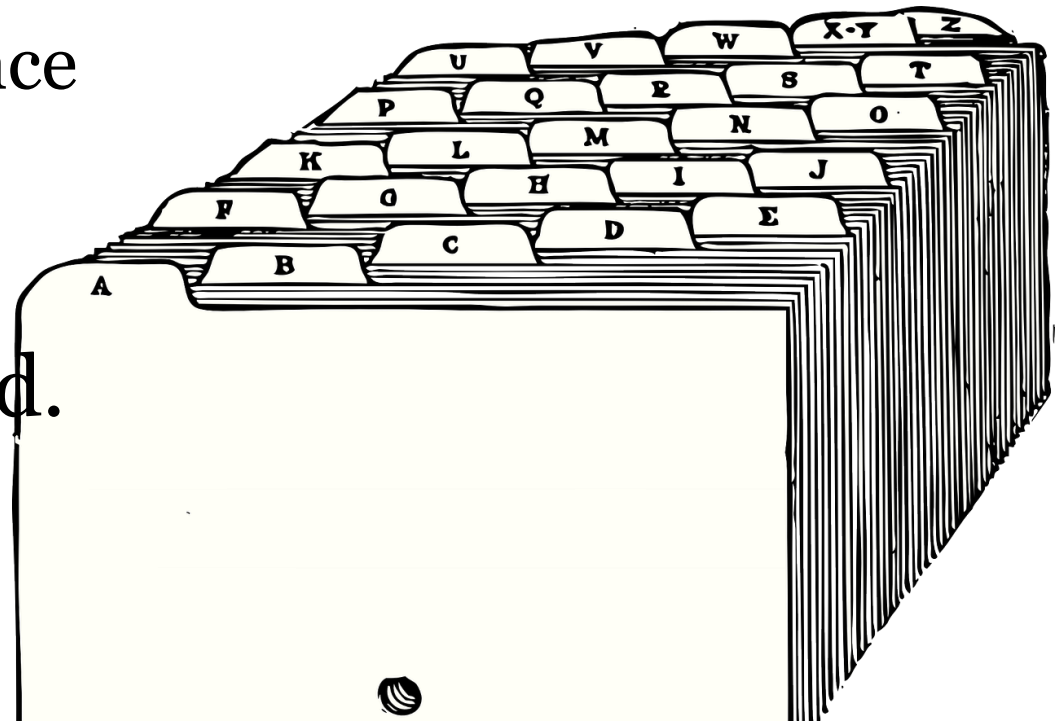
WARNING

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

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

INDEX MAINTENANCE

- Optimizing index usage
- **Deduplication** and **consolidation**
  - more beneficial than rebuilds and reorgs
  - saves storage and reduces backup/restore time
  - improves insert and update performance
- **Index evaluation** is crucial
- Consider special indexes: filtered, columnstore, and memory-optimized.



SESSION FEEDBACK

