

Microsoft SQL Server

Optimizing Your Query Plans with the SQL Server 2014 Cardinality Estimator

SQL Server Technical Article

Summary: SQL Server 2014 introduces the first major redesign of the SQL Server Query Optimizer cardinality estimation process since version 7.0. The goal for the redesign was to improve accuracy, consistency and supportability of key areas within the cardinality estimation process, ultimately affecting average query execution plan quality and associated workload performance. This paper provides an overview of the primary changes made to the cardinality estimator functionality by the Microsoft query processor team, covering how to enable and disable the new cardinality estimator behavior, and showing how to troubleshoot plan-quality regressions if and when they occur.

Writer: Joseph Sack (SQLskills.com)

Contributors: Yi Fang (Microsoft), Vassilis Papadimos (Microsoft)

Technical Reviewer: Barbara Kess (Microsoft), Jack Li (Microsoft), Jimmy May (Microsoft), Sanjay Mishra (Microsoft), Shep Sheppard (Microsoft), Mike Weiner (Microsoft), Paul White (SQL Kiwi Limited)

Published: April 2014

Applies to: SQL Server 2014

Copyright

This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2014 Microsoft. All rights reserved.

Contents

Introduction	5
The Importance of Accurate Cardinality Estimation	5
Model Assumptions	7
Enabling the New Cardinality Estimator.....	8
Changing the Database Compatibility Level.....	8
Validating a Query's Cardinality Estimator Version.....	9
Using Query Trace Flags	11
Enabling the New Cardinality Estimator with Trace Flag 2312.....	12
Reverting to the Legacy Cardinality Estimator Using Trace Flag 9481	13
Testing Workloads before Migrating to the New Cardinality Estimator	14
Validating Cardinality Estimates.....	15
Problematic Skews	17
Which Changes Require Action?	17
What Actions can You Take if You See a Plan Regression?	18
What Changed in SQL Server 2014?.....	19
Increased Correlation Assumption for Multiple Predicates	19
Modified Ascending Key and Out-Of-Range Value Estimation.....	23
Join Estimate Algorithm Changes.....	27
Simple Join Conditions	27
Multiple Join Conditions.....	29
Joins with Equality and Inequality Predicates	31
Join Containment Assumption Changes.....	32
Distinct Value Count Estimation Changes	34
Advanced Diagnostic Output.....	35
New CE Troubleshooting Methods	38
Changing the Database Compatibility Level.....	38
Using Trace Flags.....	38
Fundamental Troubleshooting Methods.....	39
Missing Statistics	39
Stale Statistics	39

Statistic Object Sampling Issues	39
Filtered Statistics	40
Multi-column Statistics.....	40
Parameter Sensitivity	40
Table Variables	40
Multi-Statement User-Defined Functions	41
XML Reader Table-Valued Function Operations	41
Data Type Conversions	41
Intra-column Comparison	41
Query Hints	41
Distributed Queries	41
Recursive Common Table Expressions	42
Predicate Complexity	42
Query Complexity.....	42
Summary	42
References.....	42
For more information:.....	42

Introduction

The SQL Server Query Optimizer's purpose is to find an efficient physical execution plan that fulfills a query request. It attempts this by assigning estimated costs to various query execution plan alternatives and then choosing the plan alternative with the lowest estimated cost. One key factor for determining operator cost is the estimation of rows that will be processed for each operator within a query execution plan. This row estimation process is commonly referred to as *cardinality estimation*. SQL Server 2014 marks the first, significant redesign of the SQL Server Query Optimizer cardinality estimation component since version SQL Server 7.0.

The SQL Server query optimization process seeks the most efficient processing strategy for executing queries across a wide variety of workloads. Achieving predictable query performance across online transaction processing (OLTP), relational data warehousing, and hybrid database schemas is inherently difficult. While many workloads will benefit from the new cardinality estimator changes, in some cases, workload performance may degrade without a specific tuning effort.

In this paper, we will discuss the fundamentals of the SQL Server 2014 cardinality estimator changes. We will provide details on activating and deactivating the new cardinality estimator. We will also provide troubleshooting guidance for scenarios where query performance degrades as a direct result of cardinality estimate issues.

The Importance of Accurate Cardinality Estimation

At a basic level, cardinality estimates are row count estimates calculated for each operator within a query execution plan. In addition to row count estimation, the cardinality estimator component is also responsible for providing information on:

- The distribution of values.
- Distinct value counts.
- Duplicate counts as input for parent operator estimation calculations.

Estimates are calculated using input from statistics associated with objects referenced in the query. *Statistics objects* used for estimation can be associated with an index or they can exist independently. You can create statistics objects manually or the query optimization process can generate them automatically. A statistics object has three main areas of information associated with it: the header, density vector, and histogram.

- The header information includes information such as the last time statistics were updated and the number of sampled rows.
- The density vector information measures the uniqueness of a column or set of columns, with lower density values indicating a higher uniqueness.
- Histogram data represents a column's data distribution and frequency of occurrence for distinct values. Histograms are limited to 200 contiguous steps, with steps representing noteworthy boundary values.

For additional details on header, density and histogram data, see the Books Online topic, DBCC SHOW_STATISTICS. [http://technet.microsoft.com/en-us/library/ms174384\(v=sql.120\).aspx](http://technet.microsoft.com/en-us/library/ms174384(v=sql.120).aspx)

The cardinality estimator component (CE) can make use of density vector and histogram information for calculating estimates when this information exists. In the absence of supporting statistics or constraints, the cardinality estimation process will provide the estimates using heuristics based on the provided filter and join predicates. The results from using heuristics are much less accurate.

CE calculations attempt to answer questions like the following:

- How many rows will satisfy a single filter predicate? Multiple filter predicates?
- How many rows will satisfy a join predicate between two tables?
- How many distinct values do we expect from a specific column? A set of columns?

Note: A predicate is an expression that evaluates to TRUE, FALSE or UNKNOWN. It comes in two varieties for SQL Server: filter predicates and join predicates. Filter predicates are used in a search condition, for example, in the WHERE and HAVING clauses. Join predicates are typically designated in JOIN conditions of FROM clauses.

Think of the CE as the component that attempts to answer *selectivity* questions posed by the WHERE, JOIN, and HAVING clauses of a query. CE also attempts to answer distinct value questions asked using the DISTINCT keyword or GROUP BY clause.

Note: Selectivity is a measure of how selective a predicate is. We calculate selectivity by dividing the number of rows that satisfy a predicate by the total number of input rows.

We calculate cardinality estimates from the leaf level of a query execution plan all the way up to the plan root. The descendant operators provide estimates to their parents. Figure 1 shows row estimates for a Clustered Index Scan of the Product table of 38 rows and Clustered Index Scan of the SalesOrderDetail table of 54 rows. Non-leaf level query execution plan operators then make use of descendent operator row estimates and apply additional estimation activities as required (such as filtering).

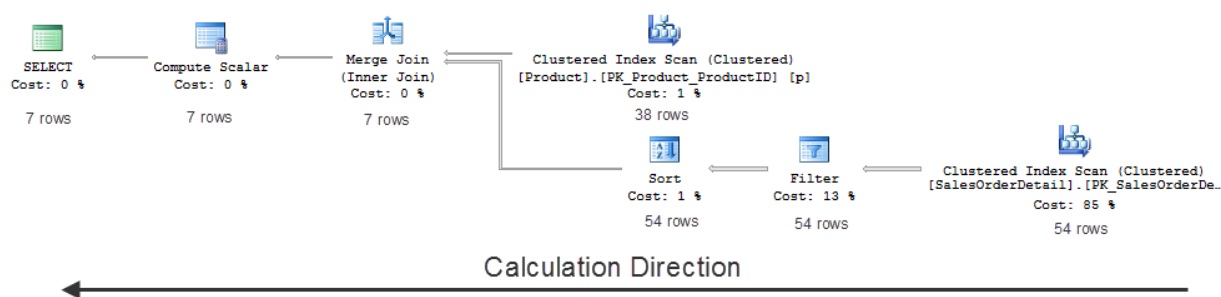


Figure 1

The individual operator cost models receive the estimates as input. The estimates are a major factor in deciding which physical operator algorithms and plan shapes (such as join orders) are chosen. They also

determine the final query plan that executes. Given these critical plan choices, *when the cardinality estimation process contains a significantly skewed assumption, this can lead to an inefficient plan choice. This can, in turn, result in degraded performance.*

Under estimating rows can lead to memory spills to disk, for example, where not enough memory was requested for sort or hash operations. Under estimating rows can also result in:

- The selection of serial plan when parallelism would have been more optimal.
- Inappropriate join strategies.
- Inefficient index selection and navigation strategies.

Inversely, over estimating rows can lead to:

- Selection of a parallel plan when a serial plan might be more optimal.
- Inappropriate join strategy selection.
- Inefficient index navigation strategies (scan versus seek).
- Inflated memory grants.
- Wasted memory and unnecessarily throttled concurrency.

Improving the accuracy of row estimates can improve the quality of the query execution plan and, as a result, improve the performance of the query.

Model Assumptions

SQL Server's CE component makes certain assumptions based on typical customer database designs, data distributions, and query patterns. The core assumptions are:

- *Independence*: Data distributions on different columns are independent unless correlation information is available.
- *Uniformity*: Within each statistics object histogram step, distinct values are evenly spread and each value has the same frequency.
- *Containment*: If something is being searched for, it is assumed that it actually exists. For a join predicate involving an equijoin for two tables, it is assumed that distinct join column values from one side of the join *will* exist on the other side of the join. In addition, the smaller range of distinct values is assumed to be contained in the larger range.
- *Inclusion*: For filter predicates involving a *column-equal-constant* expression, the constant is assumed to actually exist for the associated column. If a corresponding histogram step is non-empty, one of the step's distinct values is assumed to match the value from the predicate.

Given the vast potential for variations in data distribution, volume and query patterns, there are circumstances where the model assumptions are not applicable. This paper will elaborate on adjustments to the core assumptions introduced in SQL Server 2014.

Enabling the New Cardinality Estimator

The database context of a SQL Server session determines the CE version. If you connect to a database that is set to the SQL Server 2014 database compatibility level, the query request will use the new CE. If the database compatibility level is for an earlier version, the legacy CE will be used. New databases created on a SQL Server 2014 instance will use the SQL Server 2014 compatibility level by default. This assumes that the database compatibility level of the model database has not been changed. The system retains the earlier compatibility level in the following scenarios:

- You have migrated a database to SQL Server 2014 using in-place upgrades.
- You have attached a database from a lower version of SQL Server.
- You have restored a database from a lower version of SQL Server.

In these scenarios, database session connections to the databases will continue to use the legacy CE.

Note: The database context of the session always determines the cardinality estimation version that is used.

Changing the Database Compatibility Level

You can verify the compatibility level of a database by querying sys.databases. The following query displays all databases and associated compatibility levels on a SQL Server instance.

```
SELECT [name],  
       [compatibility_level]  
FROM sys.[databases];
```

To move a database to the SQL Server 2014 database compatibility level, alter the database compatibility level to the latest version, which is “120”.

You can download the sample databases referenced in this paper from <http://msftdbprodsamples.codeplex.com/>.

Example:

```
USE [master];  
GO  
  
-- SQL Server 2014 compatibility level  
ALTER DATABASE [AdventureWorks2012] SET COMPATIBILITY_LEVEL = 120;  
GO
```

After testing database workloads, you may choose to revert to the legacy CE behavior for sessions connecting to this database. You can do so by changing the database compatibility level to a level lower than 120.

Example:

```
USE [master];  
GO  
  
-- SQL Server 2012 compatibility level
```



```
ALTER DATABASE [AdventureWorks2012] SET COMPATIBILITY_LEVEL = 110;  
GO
```

System databases, such as the master database, will always have the highest database compatibility level on the SQL Server instance. If a database session uses a system database session context, the new CE will be used. If trace flag 9481 (discussed later) is used to revert to the legacy CE behavior, the new CE will not be used.

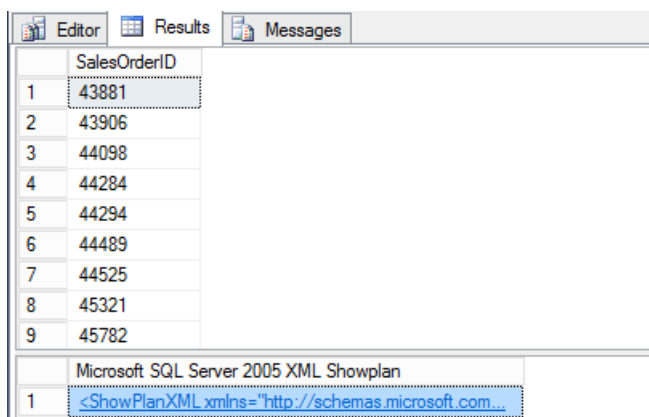
Validating a Query's Cardinality Estimator Version

In addition to checking the database's compatibility level, you can also verify which CE version was used for a specific query. To verify the CE version, inspect the query execution plan (both estimated and actual plans).

For example, see the following query execution in the context of a pre-SQL Server 2014 compatibility level database. It uses SET STATISTICS XML ON to display the actual execution plan in addition to the query execution results.

```
USE [AdventureWorks2012];  
GO  
  
SET STATISTICS XML ON;  
  
SELECT [SalesOrderID]  
FROM Sales.[SalesOrderDetail]  
WHERE [OrderQty] > 15;  
GO  
  
SET STATISTICS XML OFF;
```

It returns the result set and then provides a link to the XML Showplan output.



The screenshot shows the SQL Server Enterprise Manager interface with three tabs: Editor, Results, and Messages. The Results tab is active, displaying a table with 9 rows of SalesOrderID values. Below the table, the XML Showplan output is visible, showing a link to the XML Showplan.

	SalesOrderID
1	43881
2	43906
3	44098
4	44284
5	44294
6	44489
7	44525
8	45321
9	45782

Microsoft SQL Server 2005 XML Showplan	
1	<ShowPlanXML xmlns="http://schemas.microsoft.com...

Figure 2

Clicking the link shows the graphical rendering of the execution plan by-default. To see the cardinality estimation model version for this plan, click the root (left-most) operator in the query plan tree (in this example, the SELECT logical operator).

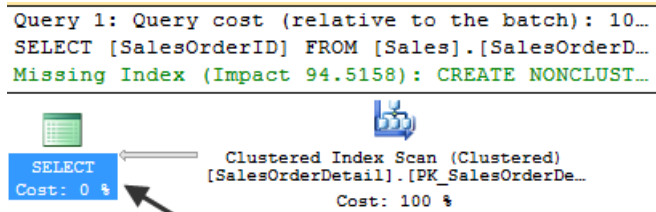


Figure 3

Select the root operator and click the F4 key to reveal the operator properties. (You can also click the **View** menu in SQL Server Management Studio and select **Properties Window**.) Select the root operator, and, in the properties window, look for the *CardinalityEstimationModelVersion* attribute value.

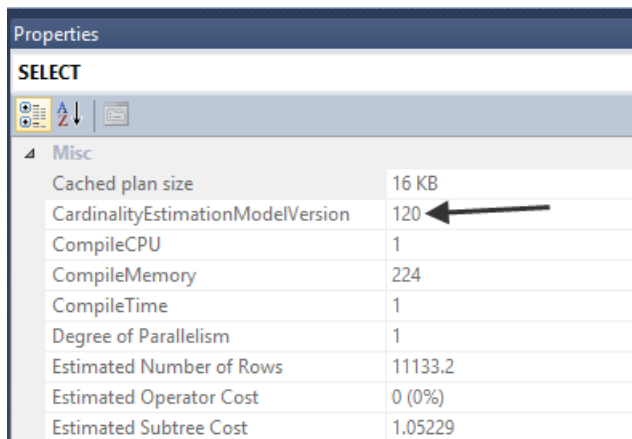


Figure 4

A value of 120 means that the new SQL Server 2014 CE functionality generated the plan. A value of 70, as seen below, means that the legacy CE functionality generated the plan.

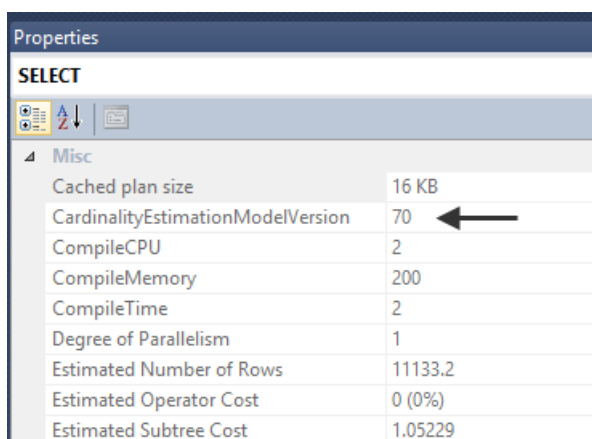


Figure 5

You can also validate the CE version in the query plan XML output by looking for the value of the “CardinalityEstimationModelVersion” attribute.

```
<StmtSimple StatementCompId="1" StatementEstRows="45.6074" StatementId="1"
StatementOptmLevel="FULL" StatementOptmEarlyAbortReason="GoodEnoughPlanFound"
CardinalityEstimationModelVersion="70" StatementSubTreeCost="0.180413"
StatementText="SELECT [AddressID],[AddressLine1],[AddressLine2] FROM [Person].[Address]
WHERE [StateProvinceID]=@1 AND [City]=@2" StatementType="SELECT"
QueryHash="0xC8ACCD7DC44F7942" QueryPlanHash="0xD5F23E0FCD5A723"
RetrievedFromCache="false">
```

The 70 value does **not** mean the database compatibility level is also set to the SQL Server 7.0 version. Rather, the 70 value represents the legacy CE functionality. This legacy CE functionality has not had major revisions to the “on by default” functionality since SQL Server 7.0.

Using Query Trace Flags

After testing critical application workloads, you may decide to use a more fine-grained method for invoking the new CE functionality. There may be several queries with improved execution performance because of the new CE functionality. Other queries may experience degradation in performance compared to the legacy CE behavior. You may want to compare legacy and new CE generated plans and validate estimates of an identical query or workload within the same batch. However, you may want to make these comparisons and perform these validations without changing database compatibility levels between statement executions.

To use new features that are tied to the SQL Server 2014 database compatibility level without using the new CE, enable a trace flag at the server level. Use *DBCC TRACEON* with the -1 argument to enable the trace flag globally. As an alternative, you can use the -T startup option to enable the trace flag during SQL Server startup.

Note: DBCC TRACEON execution requires membership in the sysadmin fixed server role.

To accommodate this finer-grained CE control, use the following fully supported query trace flags:

- *Trace Flag 9481* reverts query compilation and execution to the pre-SQL Server 2014 legacy CE behavior for a specific statement.
- *Trace Flag 2312* enables the new SQL Server 2014 CE for a specific query compilation and execution.

These trace flags are described in the Microsoft knowledge base article, “[Enable plan-affecting SQL Server query optimizer behavior that can be controlled by different trace flags on a specific-query level](http://support.microsoft.com/kb/2801413/en-us)” (http://support.microsoft.com/kb/2801413/en-us). This knowledge base article also describes the query-level option “QUERYTRACEON”, which lets you enable a plan-affecting trace flag that is applicable to a single-query compilation.

Note: QUERYTRACEON requires sysadmin permissions. You can also use QUERYTRACEON with Plan Guides. The Plan Guide author still requires sysadmin permissions, but the query executor does not.

QUERYTRACEON takes precedence over server and session-level enabled trace flags. Server and session-level trace flags take precedence over database compatibility level configuration and context.

Enabling the New Cardinality Estimator with Trace Flag 2312

In this first example, the database uses the SQL Server 2012 compatibility level 110. For the first query, the default legacy CE behavior is used. For the second query, the new SQL Server 2014 CE is used by designating the QUERYTRACEON query hint and trace flag 2312. Trace flag 2312 forces the Query Optimizer to use the new CE.

```
USE [master];
GO

ALTER DATABASE [AdventureWorks2012] SET COMPATIBILITY_LEVEL = 110;
GO

USE [AdventureWorks2012];
GO

SET STATISTICS XML ON;

-- Legacy
SELECT [AddressID],
       [AddressLine1],
       [AddressLine2]
FROM Person.[Address]
WHERE [StateProvinceID] = 9 AND
      [City] = 'Burbank';

-- New CE
SELECT [AddressID],
       [AddressLine1],
       [AddressLine2]
FROM Person.[Address]
WHERE [StateProvinceID] = 9 AND
      [City] = 'Burbank'
OPTION (QUERYTRACEON 2312);
GO

SET STATISTICS XML OFF;
```

Even though the database compatibility level is pre-SQL Server 2014, trace flag 2312 forces the second statement to use the new CE. The actual query execution plans in this example have identical estimated subtree costs, however, there are two applicable, distinguishing factors. First, the CE models are different as validated in the root operator's "CardinalityEstimationModelVersion" attribute. The second difference, while subtle in this example, is the row estimate for the Index Scan operator. The estimate is 45.6074 rows for the legacy CE plan and 94.5466 rows estimated for the new CE plan.

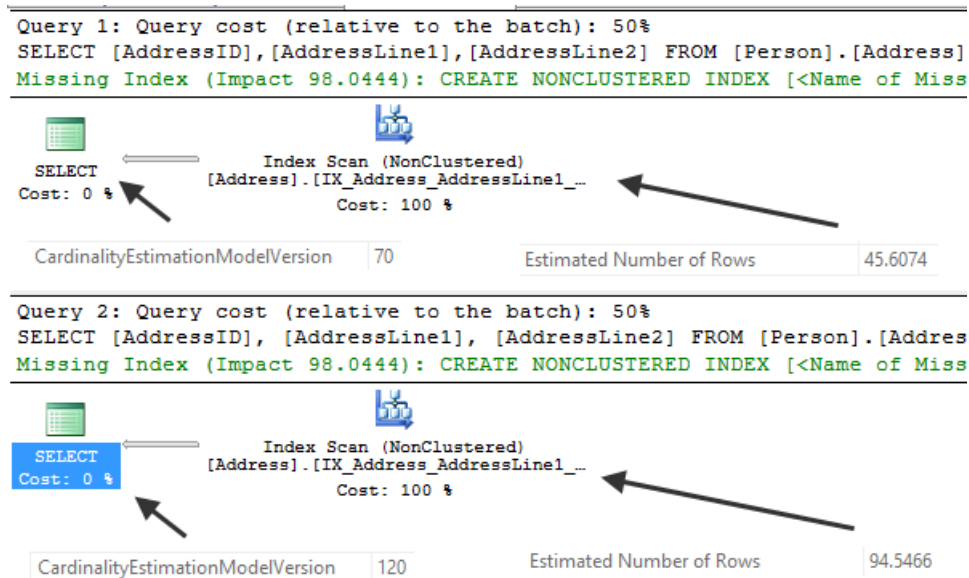


Figure 6

We will explain the reasons for this particular row estimate difference later in this paper.

Reverting to the Legacy Cardinality Estimator Using Trace Flag 9481

In this next example, the database will use the SQL Server 2014 compatibility level 120. The first query uses the default new CE behavior. The second query uses the legacy, pre-SQL Server 2014 CE by designating trace flag 9481.

```
USE [master];
GO
```

```
ALTER DATABASE [AdventureWorks2012] SET COMPATIBILITY_LEVEL = 120;
GO
```

```
USE [AdventureWorks2012];
GO
```

```
SET STATISTICS XML ON;
```

```
-- New CE
SELECT [AddressID],
       [AddressLine1],
       [AddressLine2]
FROM Person.[Address]
WHERE [StateProvinceID] = 9 AND
      [City] = 'Burbank';
```

```
-- Legacy
SELECT [AddressID],
       [AddressLine1],
       [AddressLine2]
FROM Person.[Address]
WHERE [StateProvinceID] = 9 AND
      [City] = 'Burbank'
OPTION (QUERYTRACEON 9481);
GO

SET STATISTICS XML OFF;
```

Even though the database compatibility level is set to SQL Server 2014, trace flag 9481 forces the second statement to use the legacy CE. The first statement in this example used the new CE (CardinalityEstimationModelVersion “120”). The second statement used the legacy CE (CardinalityEstimationModelVersion “70”). Index Scan estimates were also different between the two plans, and, as with the previous example, this will be explained later in this paper.

Note: If both trace flag 9481 and 2312 are enabled for a single statement compilation, they will be ignored and the XEvent query_optimizer_force_both_cardinality_estimation_behaviors will be raised.

Testing Workloads before Migrating to the New Cardinality Estimator

Microsoft’s query processor team made changes to the new CE in order to improve plan quality, consistency, supportability, and performance. While your workloads may show overall improvements, you may encounter performance degradation for some queries after moving to the new CE.

Given the risk of performance regression, *it is very important that existing applications be thoroughly tested on the new cardinality model before migrating to production*. The CE migration recommendations are as follows:

- Before changing to the new CE in production, test *all* critical workloads using representative data in a production-like environment.
- If you cannot change or fully test the existing application, you can still migrate to SQL Server 2014. However, the database compatibility level should remain below 120 until you can perform thorough testing.
- To leverage new SQL Server 2014 features without activating the new CE, change to the latest database compatibility level and enable trace flag 9481 at the server level. Use DBCC TRACEON with the -1 argument to enable the trace flag globally. As an alternative, you can use the -T startup option to enable the trace flag during SQL Server startup.
- If creating a new database for a new application, use database compatibility level 120 by default.

Note: We do not discourage an upgrade to SQL Server 2014 if you cannot perform thorough testing of your application query workloads. However, we strongly recommend a pre-120 database compatibility level until performance characteristics can be properly validated.

Validating Cardinality Estimates

You can validate cardinality estimates against actual row counts by looking at an actual *query execution plan*. The *estimated query execution plan* does not include run-time statistics. Therefore, using techniques like pulling the estimated plan from cache using `sys.dm_exec_query_plan` will only show estimated rows per operator. You cannot use the estimated plan to directly diagnose large skews between estimated and actual rows.

Unlike an estimated plan, the actual query execution plan parses and executes the query being analyzed. The actual plan can be captured using the following methods:

- SET STATISTICS XML.
- SET STATISTICS PROFILE (deprecated, but not removed from SQL Server).
- Graphical Showplan (using the “Include Actual Execution Plan” option).
- The `query_post_execution_showplan` XEvent (*use with caution as it has significant performance overhead*).
- The “Showplan Statistics Profile” and “Showplan XML Statistics Profile” SQL Trace events (*again, use with caution as these events have significant performance overhead*).

SQL Server 2014 also introduces the `sys.dm_exec_query_profiles` dynamic management view. This view allows for real-time query execution progress monitoring. The query returns the estimated rows per operator and the actual rows returned by the operator at the time the DMV was queried. Information in the DMV is available while the target query is executing. Once the target query’s execution is complete, the information is no longer available.

To return information for a specific query, you must enable the query’s actual execution plan capture for the query session. You can use `SET STATISTICS PROFILE ON`, `SET STATISTICS XML ON`, Graphical Showplan, the `query_post_execution_showplan` XEvent, or the corresponding SQL Trace event.

To demonstrate a query with a longer execution time, the following INSERT to `FactInternetSales` table is first executed in order to increase the overall table row count:

```
USE [AdventureWorksDW2012];
GO

INSERT INTO dbo.[FactInternetSales]
SELECT [ProductKey], [OrderDateKey], [DueDateKey], [ShipDateKey],
       [CustomerKey], [PromotionKey], [CurrencyKey], [SalesTerritoryKey],
       LEFT(CAST(NEWID() AS NVARCHAR(36)), 15), [SalesOrderLineNumber], [RevisionNumber],
       [OrderQuantity], [UnitPrice], [ExtendedAmount], [UnitPriceDiscountPct],
       [DiscountAmount], [ProductStandardCost], [TotalProductCost],
       [SalesAmount], [TaxAmt], [Freight], [CarrierTrackingNumber],
       [CustomerPONumber], [OrderDate], [DueDate], [ShipDate]
FROM dbo.[FactInternetSales];
GO 5 -- Executes 5 times and may take a few minutes depending on your test system
```

After increasing the `FactInternetSales` row count, consider a scenario where you want to check the run-time operator statistics for the following query that is executing:

```

SET STATISTICS XML ON;

SELECT p.[ProductLine],
       SUM(f.[SalesAmount]) AS [TotalSalesAmount],
       AVG(f.[DiscountAmount]) AS [AverageDiscountAmount]
FROM   dbo.[FactInternetSales] AS [f]
INNER JOIN dbo.[DimProduct] AS [p]
       ON f.[ProductKey] = p.[ProductKey]
GROUP BY p.[ProductLine]
ORDER BY p.[ProductLine];
GO

SET STATISTICS XML OFF;

```

This particular query's session is also capturing the actual execution plan. Therefore, the execution status for each operator will also be visible in a separate query session using `sys.dm_exec_query_profiles`.

```

SELECT [session_id],
       [node_id],
       [physical_operator_name],
       [estimate_row_count],
       [row_count]
FROM   sys.[dm_exec_query_profiles]
ORDER BY [node_id];

```

This returns the following result set, which shows the estimated and actual row counts at a point in time for each plan operator:

session_id	node_id	physical_operator_name	estimate_row_count	row_count
55	2	Stream Aggregate	4	4
55	3	Sort	158	158
55	4	Merge Join	158	158
55	5	Sort	158	158
55	6	Hash Match	158	158
55	7	Clustered Index Scan	483184	483184
55	15	Clustered Index Scan	606	606

Figure 7

In this example, the output shows perfect estimates. You can assume that this specific query execution is nearly complete at the time that `sys.dm_exec_query_profiles` was queried. In contrast, the following output shows the query in an in-progress, early-execution state:

session_id	node_id	physical_operator_name	estimate_row_count	row_count
55	2	Stream Aggregate	4	0
55	3	Sort	158	0
55	4	Merge Join	158	0
55	5	Sort	158	0
55	6	Hash Match	158	0
55	7	Clustered Index Scan	483184	166905
55	15	Clustered Index Scan	606	0

Figure 8

You can use the sys.dm_exec_query_profiles DMV to check on query execution progress and identify where major estimate skews are occurring at query runtime. It is important not to confuse an in-progress execution estimate skew with a true, final cardinality estimate skew.

For more on this subject, see the SQL Server Books Online topic “[sys.dm_exec_query_profiles \(Transact-SQL\)](http://msdn.microsoft.com/en-us/library/dn223301(v=sql.120).aspx)” ([http://msdn.microsoft.com/en-us/library/dn223301\(v=sql.120\).aspx](http://msdn.microsoft.com/en-us/library/dn223301(v=sql.120).aspx)).

In general, CE skews can be determined by looking at the actual execution plan that reflects final row counts versus estimates.

Note: An exception to this is for the inner-side of Nested Loop Join operation. You must multiply the estimated executions by the per-iteration estimate to get the total estimated rows for the inner-side operator.

Problematic Skews

Cardinality estimates for query plan operators may sometimes be 100% accurate. However, for higher complexity workloads and varying data distributions, you can expect *some* amount of variation between estimated and actual rows.

One common question is: How much variation between estimated and actual row counts is too much? There is no hard-coded variance that is guaranteed to indicate an actionable cardinality estimate problem. Instead, there are several overarching factors to consider beyond just differences between estimated and actual row counts:

- *Does the row estimate skew result in excessive resource consumption?* For example, spills to disk because of underestimates of rows or wasteful reservation of memory caused by row overestimates.
- *Does the row estimate skew coincide with specific query performance problems (e.g., longer execution time than expected)?*

If the answer is “yes” to either of the aforementioned questions, then further investigation of the cardinality estimate skew may be warranted.

Which Changes Require Action?

When moving to the new CE, you can expect that some query execution plans will remain the same and some will change. Neither condition inherently suggests an issue. The following table will help categorize which query execution plan changes might justify further actions related to the new CE:

	No change in performance	Improved Performance	Degraded Performance
No changes to estimates or the query execution plan	<i>No action necessary</i>	<i>No action necessary</i>	Not related to the new CE, but general performance tuning may be required
No changes to estimates but the query execution plan changed	<i>No action necessary</i>	<i>No action necessary</i>	Unlikely to be related to the new CE, but general performance tuning may be required
Changes to the estimates but not the query execution plan shape	<i>No action necessary</i>	<i>No action necessary</i>	Action related to the new CE may be necessary if the degradation exceeds workload performance service level agreements
Changes to the estimates and the query execution plan shape	<i>No action necessary</i>	<i>No action necessary</i>	Action related to the new CE may be necessary if the degradation exceeds workload performance service level agreements

What Actions can You Take if You See a Plan Regression?

Consider the following actions if you have encountered performance degradation directly caused by the new CE:

- *Retain the new CE setting if specific queries still benefit, and “design around” performance issues using alternative methods.* For example, for a relational data warehouse query with significant fact-table cardinality estimate skews, consider moving to a columnstore index solution. In this scenario, you retain the cardinality estimate skew. However, the compensating performance improvement of a columnstore index could remove the performance degradation from the legacy version.
- *Retain the new CE, and use trace flag 9481 for those queries that had performance degradations directly caused by the new CE.* This may be appropriate for tests where only a few queries in a workload had performance degradation caused by the new CE.
- *Revert to an older database compatibility level, and use trace flag 2312 for queries that had performance improvements using the new CE.* This may be appropriate for tests where only a few queries in the workload had improved performance.
- *Use fundamental cardinality estimate skew troubleshooting methods.* This option does not address the original root cause. However, using fundamental cardinality estimation troubleshooting methods might address an overall estimate skew problem and improve query performance. For information about troubleshooting, read the Fundamental Troubleshooting Methods section.
- *Revert to the legacy CE entirely.* This might be appropriate if multiple, critical queries encounter performance regressions and you do not have time to test and redesign around the issues.

What Changed in SQL Server 2014?

In this section, we discuss the primary changes made to the CE component for SQL Server 2014. The legacy CE's default behavior has remained relatively static since SQL Server 7.0. The SQL Server 2014 changes address performance issues for key scenarios based on benchmark and customer workload testing. The new CE design also lays the groundwork for future improvements.

While changes to the CE can improve overall workload performance, *on average*, over a range of applications, some queries may still regress. If this occurs, see the troubleshooting method section of this paper for potential solutions.

Increased Correlation Assumption for Multiple Predicates

In absence of existing multi-column statistics, the legacy SQL Server Query Optimizer views the distribution of data contained across different columns as uncorrelated with one another. This assumption of independence often does not reflect the reality of a typical SQL Server database schema, where implied correlations do actually exist.

Take the following query as an example:

```
USE [AdventureWorks2012];
GO

SELECT [AddressID],
       [AddressLine1],
       [AddressLine2]
FROM Person.[Address]
WHERE [StateProvinceID] = 9 AND
      [City] = N'Burbank' AND
      [PostalCode] = N'91502'
OPTION (QUERYTRACEON 9481); -- CardinalityEstimationModelVersion 70
GO
```

This query has three filter predicates referencing StateProvinceID, City, and PostalCode. Given that this is location-based data, you can assume *some* amount of correlation between the columns. The actual query execution plan shows both estimated rows and actual rows for the query plan operators. The following shows the graphical plan and the associated index scan details in the property window.

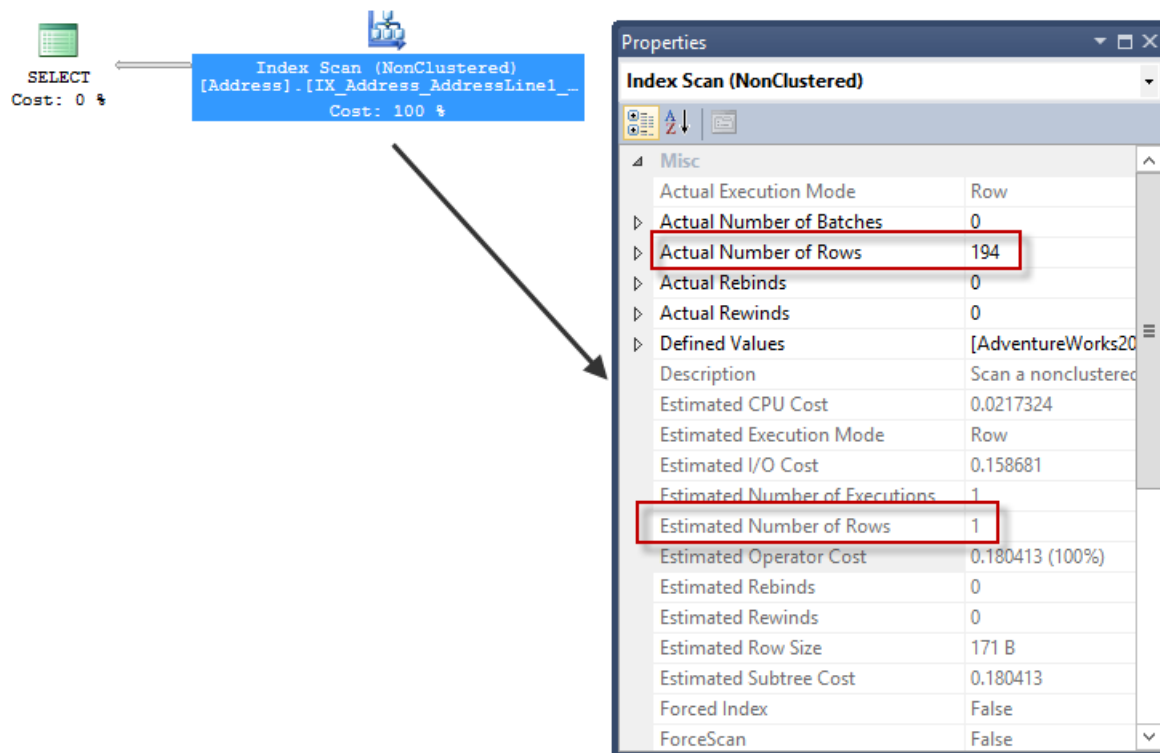


Figure 9

The estimated number of rows for the Index Scan was 1 and the actual number of rows was 194. The Query Optimizer assumed that the combination of filter predicates would result in a much higher selectivity. This assumption implied that there were fewer rows in the query result than what was actually the case.

The cardinality estimation process depends heavily on statistics when calculating row estimates. In this example, there are no multi-column statistics for the Query Optimizer to leverage. The Query Optimizer does not automatically create multi-column statistics objects. However, if you enable auto creation of statistics for the database, the Query Optimizer can create single-column statistics.

To view both standalone and index-based statistics of the Person.Address, query the sys.stats and sys.stats_columns catalog views as follows:

```
SELECT [s].[object_id],
       [s].[name],
       [s].[auto_created],
       COL_NAME([s].[object_id], [sc].[column_id]) AS [col_name]
FROM   sys.[stats] AS s
INNER JOIN sys.[stats_columns] AS [sc]
        ON [s].[stats_id] = [sc].[stats_id] AND
           [s].[object_id] = [sc].[object_id]
WHERE  [s].[object_id] = OBJECT_ID('Person.Address');
```

This returns the following, with the applicable columns highlighted:

object_id	name	auto_created	col_name
373576369	PK_Address_AddressID	0	AddressID
373576369	AK_Address_rowguid	0	rowguid
373576369	IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode	0	AddressLine1
373576369	IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode	0	AddressLine2
373576369	IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode	0	City
373576369	IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode	0	StateProvinceID
373576369	IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode	0	PostalCode
373576369	IX_Address_StateProvinceID	0	StateProvinceID
373576369	_WA_Sys_00000004_164452B1	1	City
373576369	_WA_Sys_00000006_164452B1	1	PostalCode

Figure 10

The statistics objects prefixed with “_WA_” were automatically generated when the previous query that contained filter predicates referencing the City and PostalCode columns was executing. The StateProvinceID filter predicate already had existing statistics to leverage using the IX_Address_StateProvinceID index.

As part of cardinality estimation, you can derive each predicate’s selectivity using the associated column statistics objects. You can use the DBCC SHOW_STATISTICS command to show header, density vector, and histogram data as follows:

```
DBCC SHOW_STATISTICS ('Person.Address', _WA_Sys_00000004_164452B1); -- City
DBCC SHOW_STATISTICS ('Person.Address', IX_Address_StateProvinceID); -- StateProvinceID
DBCC SHOW_STATISTICS ('Person.Address', _WA_Sys_00000006_164452B1); -- PostalCode
```

You can derive selectivities for each predicate from the associated statistics objects by using the histogram or density vector information.

The PostalCode, City, and StateProvinceID histograms each had representation as a RANGE_HI_KEY step. A PostalCode with a value of 91502 has the following histogram step:

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
91502	31	194	1	31

Figure 11

RANGE_HI_KEY 91502 has 194 EQ_ROWS (estimated number of rows with a value equal to the RANGE_HI_KEY step). With 19,614 rows at execution time in the Person.Address table, dividing 194 by 19,614 results in a selectivity of ~ 0.009891.

City with a value of “Burbank” has the following histogram step:

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
Burbank	2	196	1	2

Figure 12

Dividing 196 by 19,614 gives us a selectivity of ~ 0.009993.

StateProvinceID with a value of 9 has the following histogram step:

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
9	0	4564	0	1

Figure 13

Dividing 4,564 by 19,614 gives us a selectivity of ~ 0.232691.

The PostalCode is more selective than City, and City is more selective than StateProvinceID. Even though these individual columns all describe a shared location and are correlated, the Query Optimizer before SQL Server 2014 assumes independence across the columns.

By assuming independence, the system computes the selectivity of conjunctive predicates by multiplying individual selectivities. The legacy row estimate is 1, and the system multiplies the selectivity of each predicate to derive the estimate.

```
SELECT      0.009891 * -- PostalCode predicate selectivity
            0.009993 * -- City predicate selectivity
            0.232691 * -- StateProvinceID predicate selectivity
            19614; -- Table cardinality
```

The value is so selective that it falls below 1 row. However, the Query Optimizer uses a minimum estimated number of rows, which is 1.

In an effort to assume *some* correlation, the new CE in SQL Server 2014 lessens the independence assumption slightly for conjunctions of predicates. The process used to model this correlation is called “exponential back-off”.

Note: “Exponential back-off” is used with disjunctions as well. The system calculates this value by first transforming disjunctions to a negation of conjunctions.

The formula below represents the new CE computation of selectivity for a conjunction of predicates. “P0” represents the most selective predicate and is followed by the three most selective predicates:

$$p_0 \cdot p_1^{1/2} \cdot p_2^{1/4} \cdot p_3^{1/8}$$

The new CE sorts predicates by selectivity and keeps the four most selective predicates for use in the calculation. The CE then “moderates” each successive predicate by taking larger square roots.

Re-executing the query with the new CE model shows the following increased row estimate (13.4692 rows):

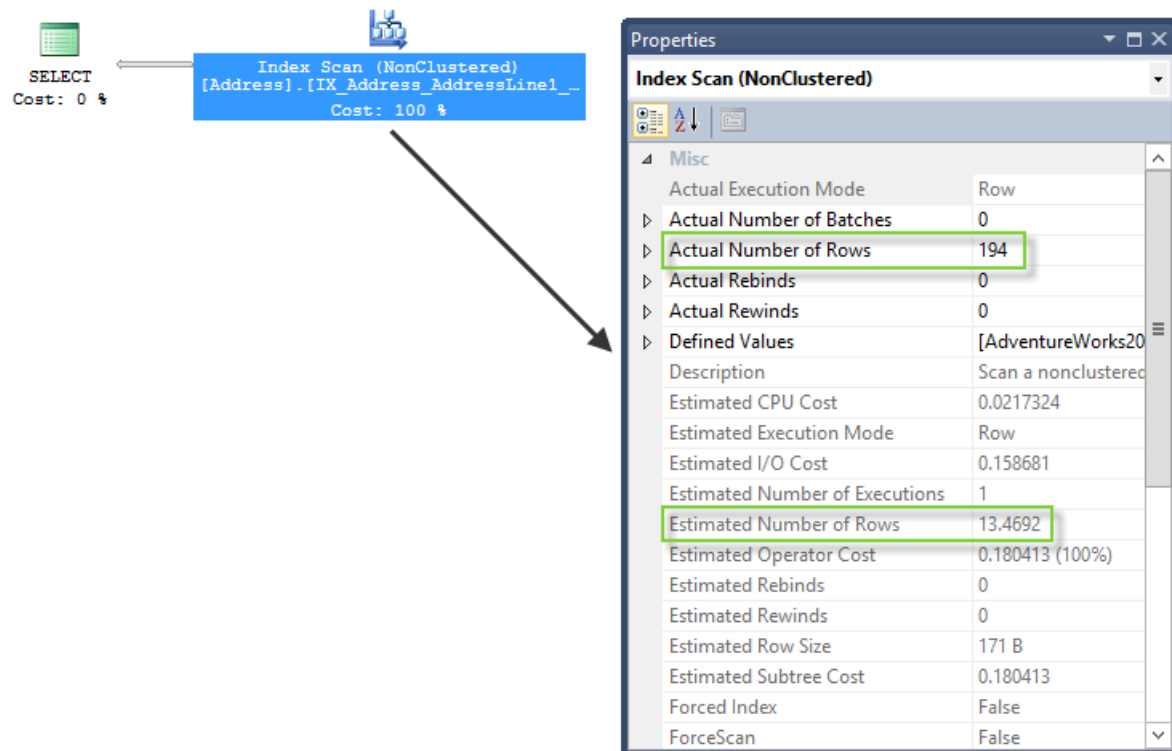


Figure 14

The following query shows how we derived the higher row estimate:

```
SELECT 0.009891 * -- PostalCode predicate selectivity
        SQRT(0.009993) * -- City predicate selectivity
        SQRT(SQRT(0.232691)) * -- StateProvinceID predicate selectivity
        19614; -- Table cardinality
```

The estimated number of rows moves from 1 to ~ 13. The gap between estimated and actual rows is somewhat reduced. For larger data sets and higher complexity queries, this reduction may result in a modified query plan shape.

Modified Ascending Key and Out-Of-Range Value Estimation

The “ascending key problem” arises when query predicates reference newly inserted data that fall out of the range of a statistic object histogram. For example, consider a sales order table that has a sales datetime data type column that is ever-increasing. Newly inserted rows have a greater value than the last sampled histogram step. Additionally, it is common for OLTP-centric queries to use predicates against more recently inserted data. For example, a query may search for all sales order table rows that were inserted in the last minute.

Before SQL Server 2014, some one-off solutions were available to address this issue. One such solution was using trace flags 2389 and 2390 to enable automatic generation of statistics for ascending keys. However, the behavior was not on by default.

The following scenario illustrates this one-off solution. To generate automatic statistics for the OrderDate column in Sales.SalesOrderHeader, the following query references the OrderDate table for an older date in the filter predicate. This triggers automatic creation of statistics if auto creation of statistics is enabled.

```
USE [AdventureWorks2012];
GO

SELECT [SalesOrderID], [OrderDate]
FROM Sales.[SalesOrderHeader]
WHERE [OrderDate] = '2005-07-01 00:00:00.000';
```

Querying sys.stats and sys.stats_columns shows that there are now automatic stats generated for the OrderDate column.

```
SELECT [s].[object_id],
       [s].[name],
       [s].[auto_created]
FROM sys.[stats] AS s
INNER JOIN sys.[stats_columns] AS [sc]
ON [s].[stats_id] = [sc].[stats_id] AND
   [s].[object_id] = [sc].[object_id]
WHERE [s].[object_id] = OBJECT_ID('Sales.SalesOrderHeader') AND
      COL_NAME([s].[object_id], [sc].[column_id]) = 'OrderDate';
```

object_id	name	auto_created	col_name
1266103551	_WA_Sys_00000003_4B7734FF	1	OrderDate

Figure 15

Checking the histogram for this statistics object shows that the highest step RANGE_HI_KEY value is 2008-07-31 00:00:00.000.

```
DBCC SHOW_STATISTICS('Sales.SalesOrderHeader', _WA_Sys_00000003_4B7734FF);
```


	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
185	2008-06-14 00:00:00.000	123	102	2	61.5
186	2008-06-16 00:00:00.000	81	65	1	81
187	2008-06-18 00:00:00.000	86	72	1	86
188	2008-06-20 00:00:00.000	82	75	1	82
189	2008-06-22 00:00:00.000	91	76	1	91
190	2008-06-24 00:00:00.000	62	74	1	62
191	2008-06-26 00:00:00.000	72	79	1	72
192	2008-06-28 00:00:00.000	73	89	1	73
193	2008-07-01 00:00:00.000	119	37	2	59.5
194	2008-07-08 00:00:00.000	180	51	6	30
195	2008-07-13 00:00:00.000	117	19	4	29.25
196	2008-07-17 00:00:00.000	99	39	3	33
197	2008-07-24 00:00:00.000	183	40	6	30.5
198	2008-07-30 00:00:00.000	148	23	5	29.6
199	2008-07-31 00:00:00.000	0	40	0	1

Figure 16

The following statement inserts 50 new rows into the Sales.SalesOrderHeader table using a more recent value for the OrderDate column value:

```

INSERT INTO Sales.[SalesOrderHeader] ( [RevisionNumber], [OrderDate],
                                         [DueDate], [ShipDate], [Status],
                                         [OnlineOrderFlag],
                                         [PurchaseOrderNumber],
                                         [AccountNumber], [CustomerID],
                                         [SalesPersonID], [TerritoryID],
                                         [BillToAddressID], [ShipToAddressID],
                                         [ShipMethodID], [CreditCardID],
                                         [CreditCardApprovalCode],
                                         [CurrencyRateID], [SubTotal],
                                         [TaxAmt], [Freight], [Comment] )
VALUES ( 3, '2014-02-02 00:00:00.000', '5/1/2014', '4/1/2014', 5, 0, 'S043659',
        'P0522145787', 29825, 279, 5, 985, 985, 5, 21, 'Vi84182', NULL, 250.00,
        25.00, 10.00, '' );
GO 50 -- INSERT 50 rows, representing very recent data, with a current OrderDate value

```

50 rows inserted in this example were not enough to cross the change threshold value for automatic statistics updates. Therefore, the existing statistics object histogram is unchanged when the following query executes against more recent data:

```

SELECT [SalesOrderID], [OrderDate]
FROM Sales.[SalesOrderHeader]
WHERE [OrderDate] = '2014-02-02 00:00:00.000'
OPTION (QUERYTRACEON 9481); -- CardinalityEstimationModelVersion 70

```

The query execution plan for the legacy CE shows an estimate of 1 row versus the actual 50 returned rows.

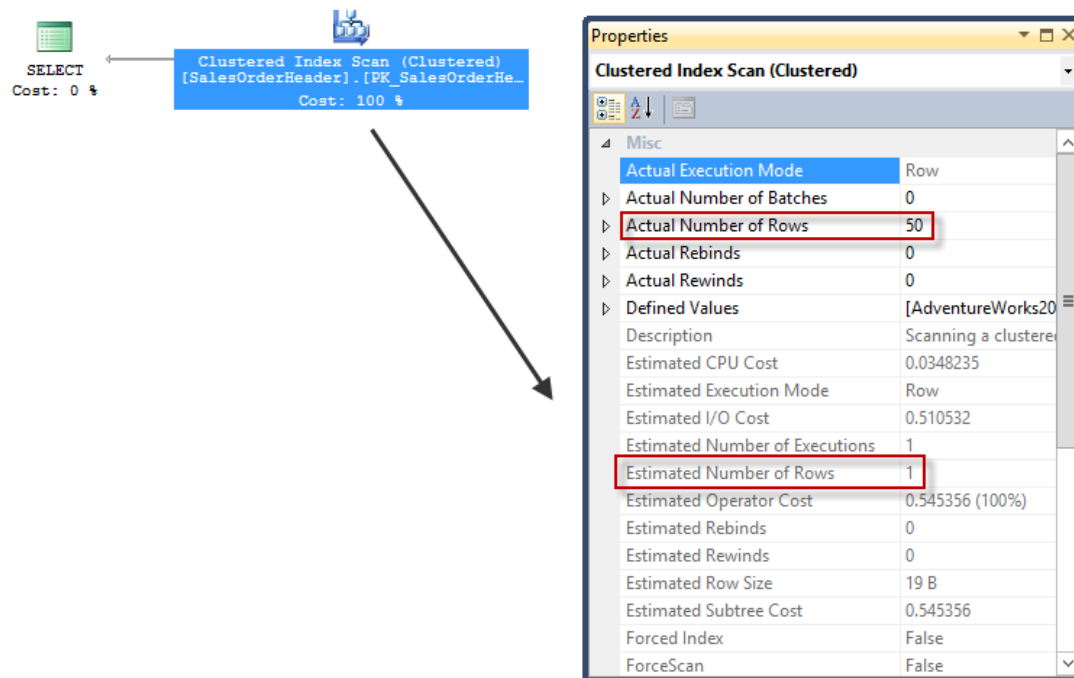


Figure 17

The reference to 2014-02-02 00:00:00.000 falls out of the range of the histogram.

The new CE assumes that the queried values *do* exist in the dataset even if the value falls out of the range of the histogram. The new CE in this example uses an average frequency that is calculated by multiplying the table cardinality by the density.

Note: If the statistics used for the calculation were computed by scanning all rows in the table or indexed view, for ranges outside of the histogram the new CE calculation caps cardinality using the estimated number of rows that were inserted since the last statistics object rebuild operation.

The revised estimate for the new CE appears in Figure 18.

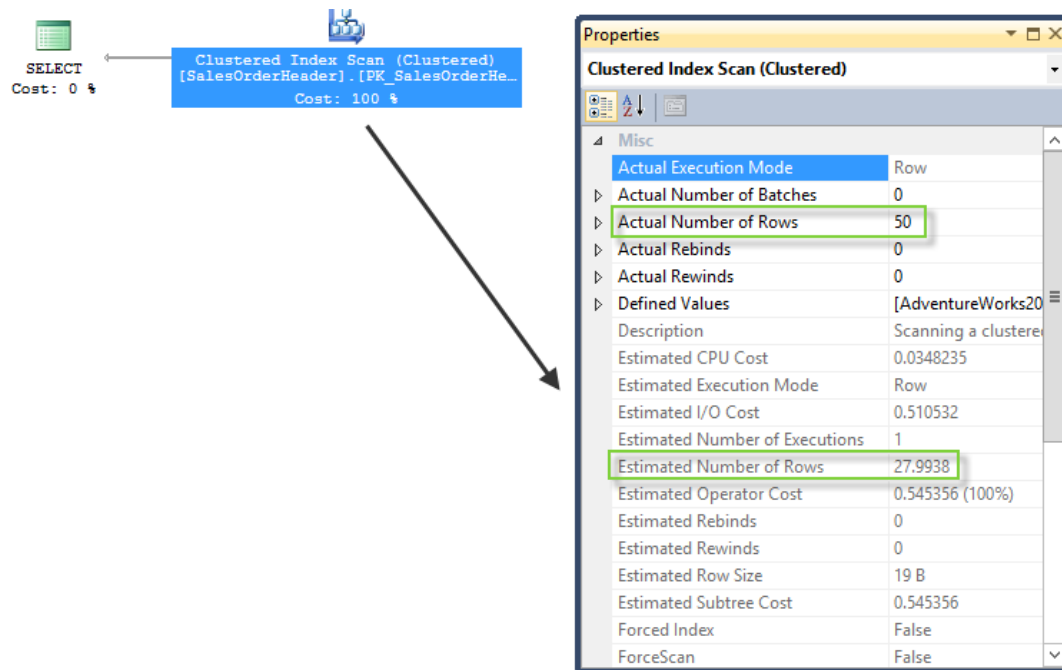


Figure 18

With the new CE, the row estimate is now 27.9938 instead of 1. You can derive the origin of this example's estimate by looking at the statistics header and density vector of the OrderDate statistics object in Figure 19.

Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	_WA_Sys_00000003_4B7734FF	Feb 2 2014 1:01PM	31465	199	0.05197797	8	NO	NULL	31465

All density	Average Length	Columns
1	0.0008896797	8
		OrderDate

Figure 19

Based on a recent statistics update, the table row count is now 31,465. Multiplying 31,465 by the "All density" value of 0.0008896797 results in our value of 27.9938 (rounded up).

Join Estimate Algorithm Changes

The new CE introduces changes to how join estimates are calculated.

Simple Join Conditions

For joins with a single equality or inequality predicate, the legacy CE joins the histograms on the join columns by aligning the two histograms step-by-step using linear interpolation. This method could result in inconsistent cardinality estimates. Therefore, the new CE now uses a simpler join estimate algorithm that aligns histograms using only minimum and maximum histogram boundaries.

In the query example below, the following join estimates are nearly identical in the new and legacy CE:

Query	New CE Estimated Rows – Nested Loop	Legacy CE Estimated Rows – Nested Loop	Actual Rows
<pre> USE [AdventureWorksDW2012]; GO SELECT [e1].[EmployeeKey], [e1].[ParentEmployeeKey], [e2].[EmployeeKey], [e2].[ParentEmployeeKey], [e2].[StartDate], [e2].[EndDate] FROM dbo.[DimEmployee] AS [e1] INNER JOIN dbo.[DimEmployee] AS [e2] ON [e1].[StartDate] < [e2].[StartDate]; </pre>	43,807 rows	43,481 rows	43,450 rows

While the estimates vary slightly using the new CE, the Query Optimizer selects the identical plan shape in both cases.

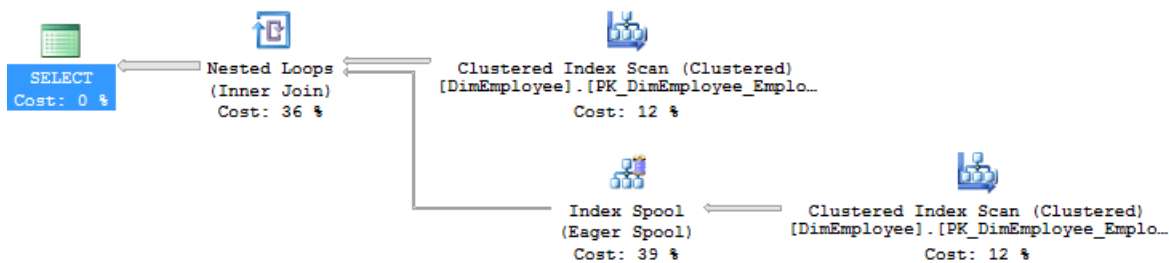


Figure 20

Although potentially less consistent, the legacy CE may produce slightly better simple-join condition estimates because of the step-by-step histogram alignment. The new CE uses a coarse alignment. However, the difference in estimates may be small enough that it will be less likely to cause a plan quality issue.

The following is an example where the join estimate changes result in a different query execution plan shape. (This example assumes that parallelism is available and enabled for the SQL Server instance). The following query uses a simple-join condition in the context of the new CE.

```

USE [AdventureWorksDW2012];
GO

-- New CE
-- Estimated rows = 58,949,200
-- Actual rows = 70,470,090
SELECT [fs].[ProductKey], [fs].[OrderDateKey], [fs].[DueDateKey],
       [fs].[ShipDateKey], [fc].[DateKey],
       [fc].[AverageRate], [fc].[EndOfDayRate], [fc].[Date]
FROM   dbo.[FactResellerSales] AS [fs]
INNER JOIN dbo.[FactCurrencyRate] AS [fc]
        ON [fs].[CurrencyKey] = [fc].[CurrencyKey]
OPTION (QUERYTRACEON 2312);

```

Using the new CE, the Query Optimizer selects a serial query execution plan:

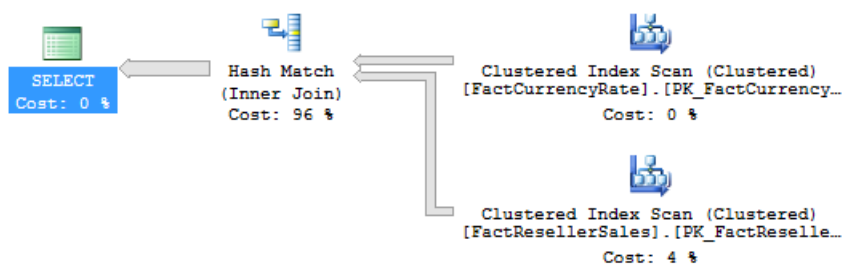


Figure 21

In contrast, the legacy cardinality estimate for the join is 70,470,100, which is very close to the actual row count of 70,470,090. The query plan shape also differs from the new CE; it uses a Merge Join physical algorithm and parallelism operators.

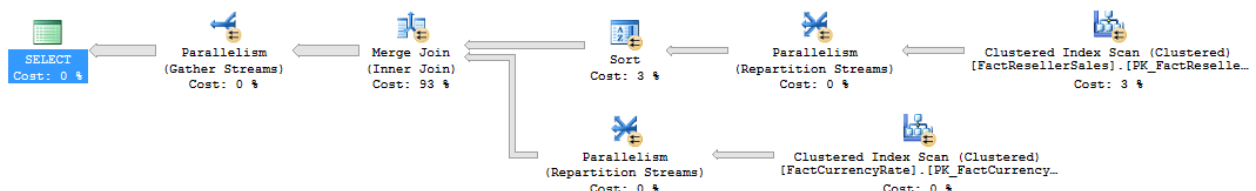


Figure 22

The estimate is worse for the new CE because the FactCurrencyRate table's CurrencyKey column has a jagged data distribution. For example, the key with the value of 100 has a higher occurrence than other values. With the new CE, jagged distributions are evened out by coarse alignment of the joined table histograms. Because of the coarse alignment of minimum and maximum histogram steps, we get a less accurate join estimate in this example. In contrast, the step-by-step join alignment algorithm the legacy CE uses includes the exact frequency of each key value. This leads to an almost perfect join estimate.

Multiple Join Conditions

For joins with a conjunction of equality predicates, the legacy CE computes the selectivity of each equality predicate, assumes independence, and combines them. In contrast, the new CE estimates the join cardinality based on multi-column frequencies computed on the join columns.

The following example will show cardinality estimates without relying on unique keys. To demonstrate, the following command drops the primary key constraint for the FactProductInventory table:

```
USE [AdventureWorksDW2012];
GO
```

```
ALTER TABLE dbo.[FactProductInventory]
DROP CONSTRAINT [PK_FactProductInventory];
GO
```

Executing the following multi-predicate query shows the following row estimates for the new and legacy CE:

Query	New CE Estimated Rows – Hash Match	Legacy CE Estimated Rows – Hash Match	Actual Rows
<pre>SELECT [fs].[OrderDateKey], [fs].[DueDateKey], [fs].[ShipDateKey], [fi].[UnitsIn], [fi].[UnitsOut], [fi].[UnitsBalance] FROM dbo.[FactInternetSales] AS [fs] INNER JOIN dbo.[FactProductInventory] AS [fi] ON [fs].[ProductKey] = [fi].[ProductKey] AND [fs].[OrderDateKey] = [fi].[DateKey];</pre>	98,829 rows	51,560 rows	60,398 rows

The legacy CE uses the selectivity of individual join predicates and combines them using multiplication. The estimate is lower than the actual number of rows. In contrast, the new CE computes the join cardinality by estimating the number of distinct values for join columns from each side. The new CE takes the smaller of the two distinct counts and multiplies by the average frequency from both sides. The result, in this example, is an estimate that is higher than the actual number of rows. For both CEs (legacy and new), the query plan shape is the same.

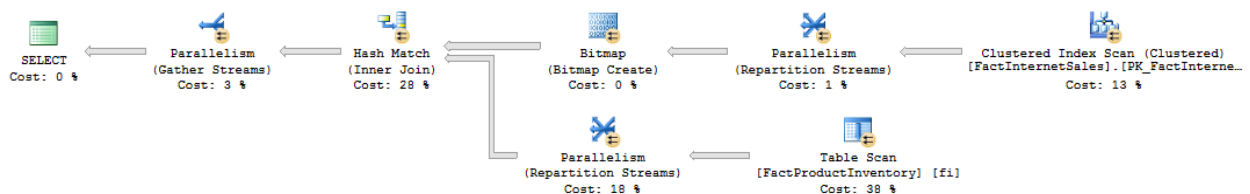


Figure 23

Continuing the example, the following command adds back the primary key constraint that was previously removed:

```

ALTER TABLE dbo.[FactProductInventory]
ADD CONSTRAINT [PK_FactProductInventory]
PRIMARY KEY CLUSTERED
(
[ProductKey] ASC,
[DateKey] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF,
IGNORE_DUP_KEY = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON
[PRIMARY];
GO

```

Re-executing the previous multi-join condition example using both CE versions provides a 100% accurate estimate of 60,398 rows for the Hash Match join operation. This is because the new and legacy CE leverage unique keys on join columns to help get accurate distinct-count estimates.

Joins with Equality and Inequality Predicates

With more complicated join conditions; for example, queries using a mix of equality and inequality join predicates, the legacy CE estimates the selectivity of individual join predicates and combines them using multiplication. The new CE, however, uses a simpler algorithm that assumes that there is a one-to-many join association between a large table and a small table. This assumes that each row in the large table matches exactly one row in the small table. This algorithm returns the estimated size of the larger input as the join cardinality.

The following example shows cardinality estimates without relying on unique keys. To demonstrate, the following command drops the primary key constraint for the FactProductInventory table.

```

USE [AdventureWorksDW2012];
GO

ALTER TABLE dbo.[FactProductInventory]
DROP CONSTRAINT [PK_FactProductInventory];
GO

```

The following query uses a mix of equality and inequality join predicates. The table shows row estimates for the new and legacy CE for the join algorithm operator (Hash Match).

Query	New CE Estimated Rows – Hash Match	Legacy CE Estimated Rows – Hash Match	Actual Rows
<pre>SELECT [fs].[OrderDateKey], [fi].[UnitCost], [fi].[UnitsIn] FROM dbo.[FactInternetSales] AS [fs] INNER JOIN dbo.[FactProductInventory] AS [fi] ON [fs].[ProductKey] = [fi].[ProductKey] AND [fs].[OrderDateKey] = [fi].[DateKey] AND [fs].[OrderQuantity] > [fi].[UnitsBalance];</pre>	776,286 rows	324 rows	22,555 rows

The legacy cardinality estimate assumes independence among the join predicates. This results in an underestimate, whereas the new CE simply estimates the join cardinality using cardinality from the larger child-operator input, resulting in an overestimate.

The following command adds back the primary key constraint that was previously removed:

```
ALTER TABLE dbo.[FactProductInventory]
ADD CONSTRAINT [PK_FactProductInventory]
PRIMARY KEY CLUSTERED
(
[ProductKey] ASC,
[DateKey] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF,
IGNORE_DUP_KEY = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON
[PRIMARY];
GO
```

Join Containment Assumption Changes

The CE model for a join predicate involving an equijoin for two tables assumes that join columns will exist on both sides of the join. In the presence of additional non-join filter predicates against the join table, the legacy CE assumes *some* level of correlation. This implied correlation is called “simple containment”.

To illustrate the “simple containment” assumption, the following query joins two tables with an equijoin predicate on ProductID. It also uses one filter predicate that references product color from the Product table. The second filter predicate references record-modified date on the SalesOrderDetail table:

```
USE [AdventureWorks2012];
GO

SELECT [od].[SalesOrderID], [od].[SalesOrderDetailID]
FROM     Sales.[SalesOrderDetail] AS [od]
INNER JOIN Production.[Product] AS [p]
        ON [od].[ProductID] = [p].[ProductID]
WHERE    [p].[Color] = 'Red' AND
        [od].[ModifiedDate] = '2008-06-29 00:00:00.000'
OPTION (QUERYTRACEON 9481); -- CardinalityEstimationModelVersion 70
```


At a high level, the legacy CE join estimate assumes containment for any arbitrary inputs of the join. Any existing filter predicates load and scale down histograms before merging them using join predicates. The legacy CE behavior assumes that the non-join filter predicates are correlated. The legacy CE expects a higher estimate of row matches between the two tables. This expectation influences the cardinality estimate for the join operation as follows:

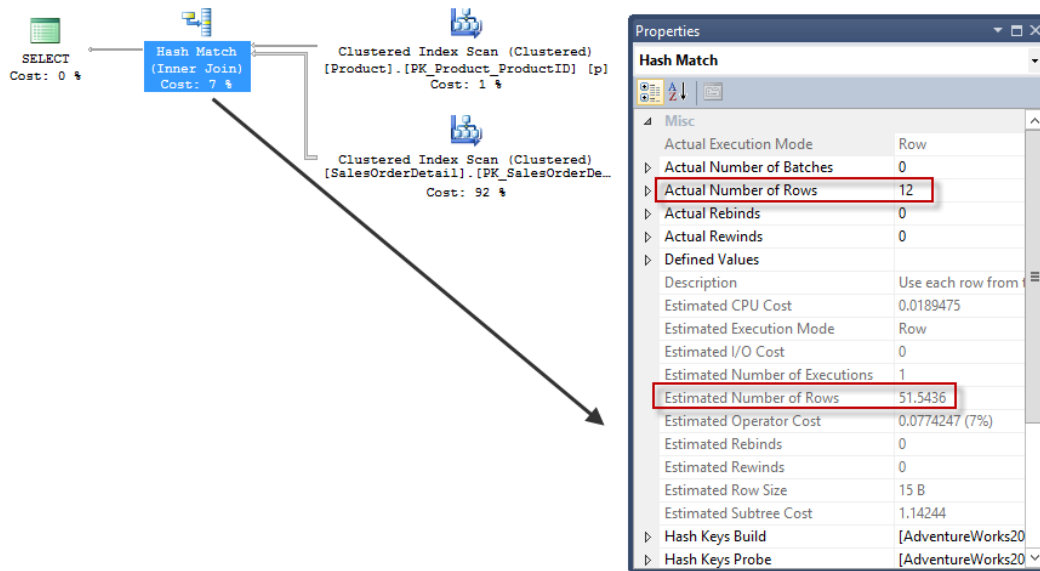


Figure 24

“Simple containment” assumes, given this query, that sales order rows modified on June 29th in 2008 are all for red-colored products. In this scenario, this assumption is *not* accurate. This results in an overestimate of 51.5436 rows versus the actual 12 rows.

Alternatively, the new CE uses “base containment” as an assumption. This means that the new CE assumes that the filter predicates on separate tables are *not* correlated with each other. At a high level, the new CE derives the join selectivity from base-table histograms without scaling down using the associated filter predicates. Instead the new CE computes join selectivity using base-table histograms *before* applying the selectivity of non-join filters.

Figure 25 shows the new CE plan estimates.

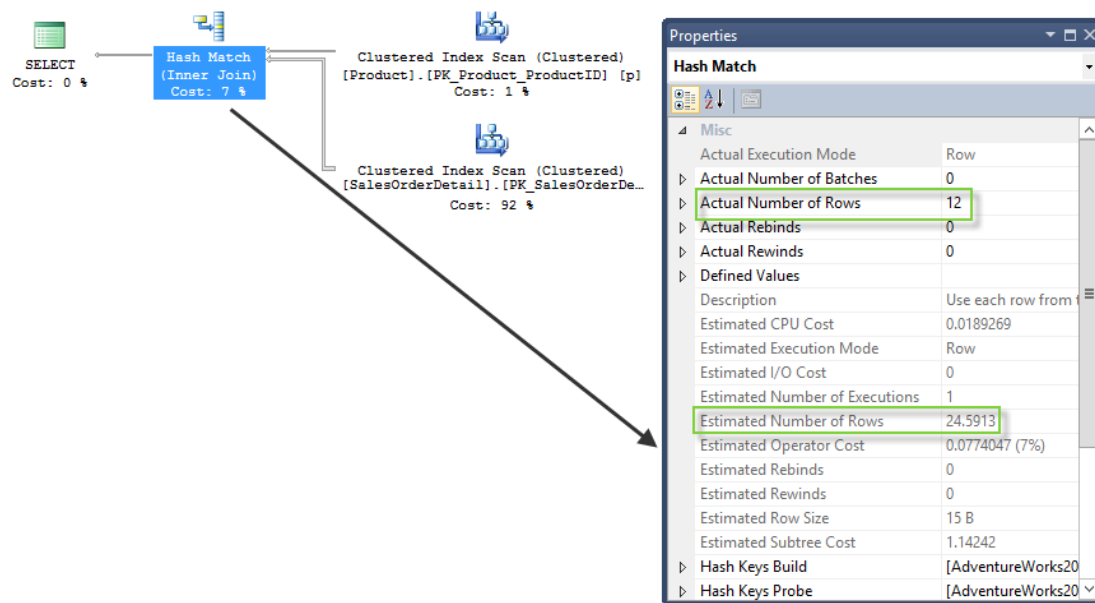


Figure 25

The new estimated number of rows for the Hash Match operation using “base containment” is 24.5913, down from 51.5437 rows estimated using “simple containment”. Removing the implied filter correlation moves the estimate closer to the actual number of rows.

Distinct Value Count Estimation Changes

For queries where many-to-many join operations are not involved, distinct value count estimation differences between the legacy and new CE are minimal. For example, the following query joins two tables and groups using one column from each table:

Query	New CE Estimated Rows – Distinct Sort	Legacy CE Estimated Rows – Distinct Sort	Actual Rows – Distinct Sort
<pre>USE [AdventureWorksDW2012]; GO SELECT [f].[ProductKey], [d].[DayNumberOfYear] FROM dbo.[FactInternetSales] AS [f] INNER JOIN dbo.[DimDate] AS [d] ON [f].[OrderDateKey] = [d].[DateKey] WHERE [f].[SalesTerritoryKey] = 8 GROUP BY [f].[ProductKey], [d].[DayNumberOfYear];</pre>	5,625 rows	4,784 rows	4,718 rows

There were minimal estimation differences for a Sort (Distinct Sort) operator, and the plan shape was identical for both the legacy and new CE.

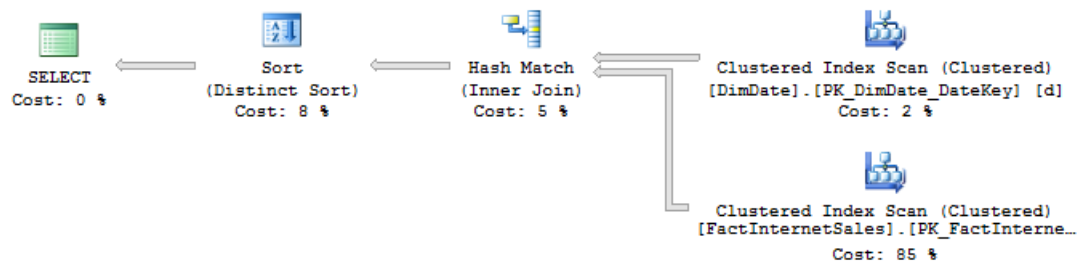


Figure 26

If a join condition amplifies the cardinality of either side of the join using a many-to-many join operation, the legacy CE may provide inaccurate estimates. The new CE establishes common-sense boundaries based on the chosen distinct values and the participating join or filter predicates defined in a query. The new CE uses “ambient cardinality”, which is the cardinality of the smallest set of joins that contains the GROUP BY or DISTINCT columns. This effectively reduces the distinct count when the overall join cardinality itself is large.

The following example extends the previous example by adding an additional INNER JOIN to the FactProductInventory table (estimates are for the Distinct Sort operator).

Query	New CE Estimated Rows – Distinct Sort	Legacy CE Estimated Rows – Distinct Sort	Actual Rows – Distinct Sort
<pre> SELECT [f].[ProductKey], [d].[DayNumberOfYear] FROM dbo.[FactInternetSales] AS [f] INNER JOIN dbo.[DimDate] AS [d] ON [f].[OrderDateKey] = [d].[DateKey] INNER JOIN dbo.[FactProductInventory] AS fi ON [fi].[DateKey] = [d].[DateKey] WHERE [f].[SalesTerritoryKey] = 8 GROUP BY [f].[ProductKey], [d].[DayNumberOfYear]; </pre>	5,350 rows	36,969 rows	4,718 rows

The new CE continues to use the 5,350 row estimate based on the smallest join that contains the GROUP BY columns. However, the legacy CE significantly overestimates the number of rows because of the presence of a many-to-many join to the FactProductInventory table.

Advanced Diagnostic Output

SQL Server 2014 introduces additional diagnostic output. When used in conjunction with the new CE, this output can be useful for identifying statistics that were used to calculate cardinality estimates. This output also helps to identify how estimates were derived. The additional diagnostic output can also be used for advanced cardinality estimate troubleshooting scenarios.

Note: Not all output is publicly documented. The detailed output can provide useful troubleshooting diagnostic data when provided to a Microsoft Support engineer in the context of a support case.

The new *query_optimizer_estimate_cardinality* XEvent fires when the Query Optimizer estimates cardinality on a relational expression.

Warning: As with other debug-channel events, care should be taken when using *query_optimizer_estimate_cardinality* on a production system. It can significantly degrade query workload performance. You should only use this event when troubleshooting or monitoring specific problems for only brief periods of time.

The following is a sample XEvent session with creation, session startup, and stopping of a session after the workload has been executed:

```
CREATE EVENT SESSION [CardinalityEstimate] ON SERVER
ADD EVENT sqlserver.query_optimizer_estimate_cardinality
ADD TARGET package0.event_file ( SET filename = N'S:\CE\CE_Data.xel' ,
                                max_rollover_files = ( 2 ) )
WITH ( MAX_MEMORY = 4096 KB ,
        EVENT_RETENTION_MODE = ALLOW_SINGLE_EVENT_LOSS ,
        MAX_DISPATCH_LATENCY = 30 SECONDS ,
        STARTUP_STATE = OFF );
GO

-- Start the session
ALTER EVENT SESSION [CardinalityEstimate] ON SERVER STATE=START;

--
-- Your workload to be analyzed executed here (or in another session)
--

-- Stop the session after the workload is executed
ALTER EVENT SESSION [CardinalityEstimate] ON SERVER STATE=STOP;
```

The *query_optimizer_estimate_cardinality* event outputs the following information:

Event Data	Description
query_hash	Hash of the query that is being compiled. This is the same query hash that appears in an execution plan.
creation_time	Query compilation time; this uses the local server time and matches the creation_time for the plan in sys.dm_exec_query_stats.
input_relation	Input relation on which the cardinality is being estimated, shown in XML format.
calculator	The cardinality estimation strategy being used, shown in XML format.
stats_collection	The statistics collection generated for the input that shows the result of the estimate in XML format. For tables with multiple statistics objects, this information can be used to answer, “What statistics object was used for calculating the cardinality estimate?”
stats_collection_id	Identifier for the statistics collection that will appear in the “RelOp” element for a query execution plan. This value can be used to map to non-scan related plan operators.

While the query_optimizer_estimate_cardinality XEvent is being captured, the stats_collection_id is also added to the associated query execution plan. For example, the following query's execution plan uses a Hash Match algorithm that has a StatsCollectionId of 5:

```
USE [AdventureWorks2012];
GO

SELECT [od].[SalesOrderID], [od].[SalesOrderDetailID]
FROM Sales.[SalesOrderDetail] AS [od]
INNER JOIN Production.[Product] AS [p]
    ON [od].[ProductID] = [p].[ProductID]
WHERE p.[Color] = 'Red' AND
[od].[ModifiedDate] = '2008-06-29 00:00:00.000'
OPTION (RECOMPILE);
GO
```

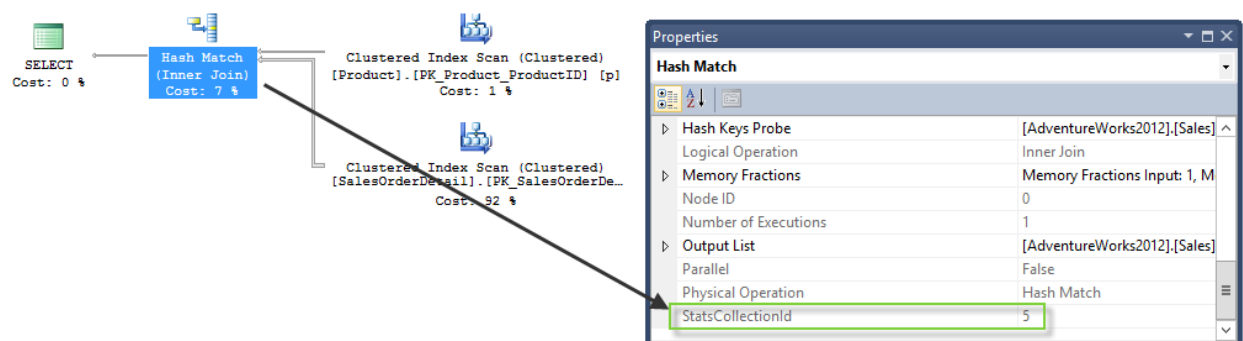


Figure 27

This StatsCollectionID can then be associated to the specific query_optimizer_estimate_cardinality XEvent output. (Figure 28 shows the “Watch Live Data” viewer for XEvents in SQL Server Management Studio).

Displaying 22 Events. SQL server events session stopped or no longer available.			
	name	timestamp	stats_collection_id
	query_optimizer_estimate_cardinal...	2014-02-08 13:06:37.6453772	3
	query_optimizer_estimate_cardinal...	2014-02-08 13:06:37.6475800	2
	query_optimizer_estimate_cardinal...	2014-02-08 13:06:37.6523260	4
	query_optimizer_estimate_cardinal...	2014-02-08 13:06:37.6701946	5
	query_optimizer_estimate_cardinal...	2014-02-08 13:06:37.6935605	7
	query_optimizer_estimate_cardinal...	2014-02-08 13:06:37.7191153	8

Figure 28

The event highlighted in Figure 28 has additional calculator, creation_time, input_relation, query_hash, and stats_collection output. You can use this output to troubleshoot cardinality estimate issues. The following table shows this information for stats_collection_id 5:

Event Data	Description
query_hash	8987645114783459866
creation_time	2014-02-08 13:06:37.3866666
input_relation	<pre> <Operator Name="LogOp_Join" ClassNo="14"> <StatsCollection Name="CStCollFilter" Id="3" Card="172.14" /> <StatsCollection Name="CStCollFilter" Id="4" Card="38.00" /> <Operator Name="ScaOp_Comp " ClassNo="100"> <CompInfo CompareOp="EQ" /> <Operator Name="ScaOp_Identifier " ClassNo="99"> <IdentifierInfo TableName="[p]" ColumnName="ProductID" /> </Operator> <Operator Name="ScaOp_Identifier " ClassNo="99"> <IdentifierInfo TableName="[od]" ColumnName="ProductID" /> </Operator> </Operator> </Operator> </pre>
calculator	<pre> <CalculatorList> <JoinCalculator CalculatorName="CSelCalcExpressionComparedToExpression" Selectivity="0.004" SelectivityBeforeAdjustmentForOverPopulatedDimension="0.002" /> </CalculatorList> </pre>
stats_collection	<pre> <StatsCollection Name="CStCollJoin" Id="5" Card="24.59"> <LoadedStats> <StatsInfo DbId="5" ObjectId="1154103152" StatsId="3" /> <StatsInfo DbId="5" ObjectId="1973582069" StatsId="1" /> </LoadedStats> </StatsCollection> </pre>
stats_collection_id	5

New CE Troubleshooting Methods

This section summarizes troubleshooting options to consider if new CE-related cardinality estimates are incorrect and are affecting plan quality and the associated workload performance.

Changing the Database Compatibility Level

If the new CE functionality is negatively influencing critical workloads, you can disable the use of the new CE. You disable the new CE by reverting to a database compatibility level below 120. The database context of the session will determine the CE version. Therefore, connecting to the legacy database compatibility level means that workloads will revert to the legacy CE behavior.

Using Trace Flags

You can use the QUERYTRACEON query-level option to enable a plan-affecting trace flag applicable to a single-query compilation. QUERYTRACEON supported trace flags are documented in the KB article, ["Enable plan-affecting SQL Server query optimizer behavior that can be controlled by different trace flags on a specific-query level."](http://support.microsoft.com/kb/2801413) (<http://support.microsoft.com/kb/2801413>)

You may want to use new features that are tied to the SQL Server 2014 database compatibility level. However, you may not want to use the new CE. To use these new features without using the CE, enable

a trace flag at the server-level. Use DBCC TRACEON with the -1 argument to enable the trace flag globally. Alternatively, use the -T startup option to enable the trace flag during SQL Server startup. Use trace flag 9481 to revert to the legacy CE behavior within the context of the SQL Server 2014 database compatibility level. Use trace flag 2312 to enable the new CE within the context of a pre-SQL Server 2014 database compatibility level.

Fundamental Troubleshooting Methods

The following troubleshooting methods apply to the new and legacy CE behavior. These are fundamental areas to consider, regardless of the new CE changes. While we do not discuss all known CE-issue scenarios in this section, we do discuss the more common customer issues.

Missing Statistics

Missing statistics in association with specific query join or filter predicates should appear as a warning in the query execution plan. If statistics are missing and not generated automatically, check if automatic creation of statistics is enabled for the database. Check sys.databases and see if the is_auto_create_stats_on option is selected. If automatic creation of statistics is disabled, consider enabling it. If it is disabled for supportability or other reasons, consider creating the required statistics manually. Alternatively, if appropriate for navigation purposes, create the associated index.

Stale Statistics

You may determine that the statistics are outdated. This is not necessarily an issue if the data distribution within the statistics object histogram and density vector still represent the current state of the data. Stale statistics can be problematic, however, if they no longer accurately represent the current data distribution. Check the statistics object using DBCC SHOW_STATISTICS to verify if the lack of recent updates is actually causing bad estimates. Additionally, verify the histogram and density vector. If stale statistics are an issue, verify that automatic statistics updates are enabled for the database using sys.databases and the is_auto_update_stats_on column. If these are disabled, verify that manual statistics maintenance is scheduled to occur on an appropriate schedule. Check if individual statistics objects are set to “no recompute”; you can verify this by querying sys.stats and the no_recompute column.

If your table is very large, the statistics update threshold and the frequency of the associated automatic update of statistics may not be enough to maintain useful statistics. Consider a more frequent manual scheduling of statistics maintenance or consider activating trace flag 2371. Trace flag 2371 uses a decreasing, dynamic statistics update threshold, calculated as the square root of 1,000 multiplied by the table cardinality.

Statistic Object Sampling Issues

Tables with very “jagged” (uneven occurrence) data distributions may not have adequate representation within the 200 maximum steps of a statistics object histogram. Validate if sampling is occurring for the statistics generation by comparing the “rows sampled” versus “rows” in DBCC SHOW_STATISTICS or sys.dm_db_stats_properties. Consider testing a higher percent sampling. You can also use FULLSCAN during manual statistics updates to see if the histogram quality has improved and the data distributions

are properly represented. FULLSCAN scans *all* rows in the table, so this can sometimes cause unacceptable performance overhead for very large tables.

Filtered Statistics

Filtered statistics may help address statistics quality issues for very large tables that contain uneven data distributions. Be careful to consider the filtered statistics update threshold, however, which is based on overall table thresholds and *not* the filter predicate. In the absence of a RECOMPILE hint, filtered indexes and statistics will not be used in conjunction with parameterization that refers to the filter column.

Multi-column Statistics

Automatic creation of statistics only apply to *single*-column statistics. Multi-column statistics are not automatically created. You must create them manually or implicitly in conjunction with a multi-column index. You can use Database Tuning Advisor to recommend multi-column statistics. If you know in advance that columns for a table are correlated, creating multi-column statistics can sometimes improve estimates. This is true if the overall data distribution is relatively uniform across the entire table based on the density vector “all density” values.

Warning: As of SQL Server 2014 RTM, with the new CE, multi-column statistics are not used to estimate filter predicates on multiple columns. Microsoft will address this issue in the next release of SQL Server.

Parameter Sensitivity

Initial compilation of a parameterized plan is based on the first passed value, called the parameter compiled value. If the parameter compiled value results in a plan that is not optimal for other potential run-time values, this can result in performance issues. If different parameter values benefit from significantly different plan shapes, there are several workaround solutions to consider, each with varying tradeoffs. These solutions include procedure branching for parameter run-time value ranges, OPTIMIZE FOR usage, and RECOMPILE directives that are applicable at varying scopes. If you suspect parameter sensitivity as the root cause of your performance issue, check the compilation versus the run-time values within the actual query execution plan. Perform this comparison by checking the ParameterCompiledValue and ParameterRuntimeValue attributes.

Note: Trace flag 4136 is used to disable parameter sniffing, which is equivalent to adding an OPTIMIZE FOR UNKNOWN hint to each query that references a parameter. We describe this trace flag in [KB article 980653](#). This trace flag is only appropriate for very specific circumstances and should not be used without thorough performance testing and benchmarking.

Table Variables

Table variables do not have statistics associated with them as of SQL Server 2014 RTM. Unless a statement-level recompilation occurs, table variables assume a fixed, single-row cardinality estimate for filter and join predicates. If the table variable contains a small number of rows, performance of the overall query may not be affected. But for larger result sets with higher complexity operations, row underestimates can significantly impair query plan quality and performance. For scenarios where table variable underestimates directly impact performance, consider using temporary tables instead.

Multi-Statement User-Defined Functions

Similar to table variables, multi-statement table-valued functions use a fixed cardinality. For large cardinality estimate skews in a multi-statement table-valued function, consider using an inline table valued function instead. As an alternative, you can execute the associated multi-statement table-valued function outside the context of a function. While this practice may go against the encapsulation benefits that functions provide, the query performance improvements may be worth the tradeoff.

Note: The default fixed cardinality estimate for multi-statement table-valued functions in the new CE is now 100 instead of 1.

XML Reader Table-Valued Function Operations

Using the nodes() XQuery method without an associated XML index can lead to bad cardinality estimates. The Table-Valued Function XML Reader operator uses a fixed row cardinality estimate that is only slightly adjusted based on multiple pushed path filters. One method for addressing this issue is to create a supporting primary XML index for the referenced XML column.

Data Type Conversions

Mismatched data type usage within join or filter predicates can affect cardinality estimates and the associated plan selection. To avoid cardinality estimate issues caused by data type mismatches, use identical data types in search and join conditions.

Intra-column Comparison

Cardinality estimate issues can occur when performing comparisons between columns in the same table. If this is an issue, consider creating computed columns. The computed column can then be referenced in a filter predicate. You can automatically generate the computed column's associated statistics (by activating automatic statistics creation at the database level) to improve overall estimates. If intra-table column comparison cardinality estimation is a frequent issue, consider normalization techniques (separate tables), derived tables, or common table expressions.

Query Hints

In general, you should avoid using query hints to override the query optimization process without first attempting less invasive troubleshooting. However, cardinality estimate issues that persist and lead to poor query optimization choices may require their use. In that situation, use query hints to override various decisions for join order and physical operator algorithm choice. For examples of the available query hints, see the Books Online topic [Query Hints \(Transact-SQL\)](http://technet.microsoft.com/en-us/library/ms181714.aspx) (<http://technet.microsoft.com/en-us/library/ms181714.aspx>).

Distributed Queries

Linked-server distributed queries can have cardinality estimate skew issues if the linked-server definition does not have the minimum permissions needed to gather statistics for referenced objects. An example of such a permission is the db_ddladmin fixed database role permission. SQL Server 2012 Service Pack 1 reduces the required permissions. It allows DBCC SHOW_STATISTICS to work with SELECT permissions on all columns in the statistics object and a filter condition, if one is designated.

Recursive Common Table Expressions

Recursive Common Table Expressions may have cardinality estimate skew issues under some circumstances (e.g., if non-unique parent/child keys are used for recursion). We discuss this issue and its associated workaround techniques in the SQL Server Customer Advisory Team blog post “[Optimize Recursive CTE Query](http://blogs.msdn.com/b/sqlcat/archive/2011/04/28/optimize-recursive-cte-query.aspx?Redirected=true).” (<http://blogs.msdn.com/b/sqlcat/archive/2011/04/28/optimize-recursive-cte-query.aspx?Redirected=true>)

Predicate Complexity

Avoid embedding predicate column references in functions or complex expressions. This can prevent the Query Optimizer from correctly estimating cardinality for columns referenced in the join or filter predicate.

Query Complexity

SQL Server can often handle complex queries very well. If a large and complex single-statement query has significant cardinality estimate issues, one potential solution is to break it into smaller, less complex statements. Smaller statements can give the optimizer a better chance of creating high quality query plans. Examples of such a solution are using UNION ALL with simpler query statements or creating intermediate result sets that are built upon sequentially. For an overview of these techniques, see the Microsoft Tech Note, “[When To Break Down Complex Queries](http://blogs.msdn.com/b/sqlcat/archive/2013/09/09/when-to-break-down-complex-queries.aspx).” (<http://blogs.msdn.com/b/sqlcat/archive/2013/09/09/when-to-break-down-complex-queries.aspx>)

Summary

SQL Server 2014 marks the first significant redesign of the SQL Server Query Optimizer cardinality estimation process since version 7.0. Use of the new CE can result in an overall improvement in average query performance for a wide range of application workloads. The new CE also provides diagnostic output for use in troubleshooting cardinality estimate issues. As described in this paper, some workloads may encounter degraded performance with the new CE. We recommend that you thoroughly test existing applications before migrating. When using the new CE, users can leverage trace flags to use the legacy model for queries that regress. This allows you to still benefit from queries that improved under the new model.

References

- [Statistics \(SQL Server\)](#)
- [Cardinality Estimation \(SQL Server\)](#)
- [Statistics Used by the Query Optimizer in Microsoft SQL Server 2008](#)
- [When To Break Down Complex Queries](#)

For more information:

<http://www.microsoft.com/sqlserver/>: SQL Server Web site

<http://technet.microsoft.com/en-us/sqlserver/>: SQL Server TechCenter

<http://msdn.microsoft.com/en-us/sqlserver/>: SQL Server DevCenter

Did this paper help you? Please give us your feedback. Tell us, on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

- Are you rating it high because it has good examples, excellent screen shots, clear writing, or another reason?
- Are you rating it low because of poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of the white papers we release.