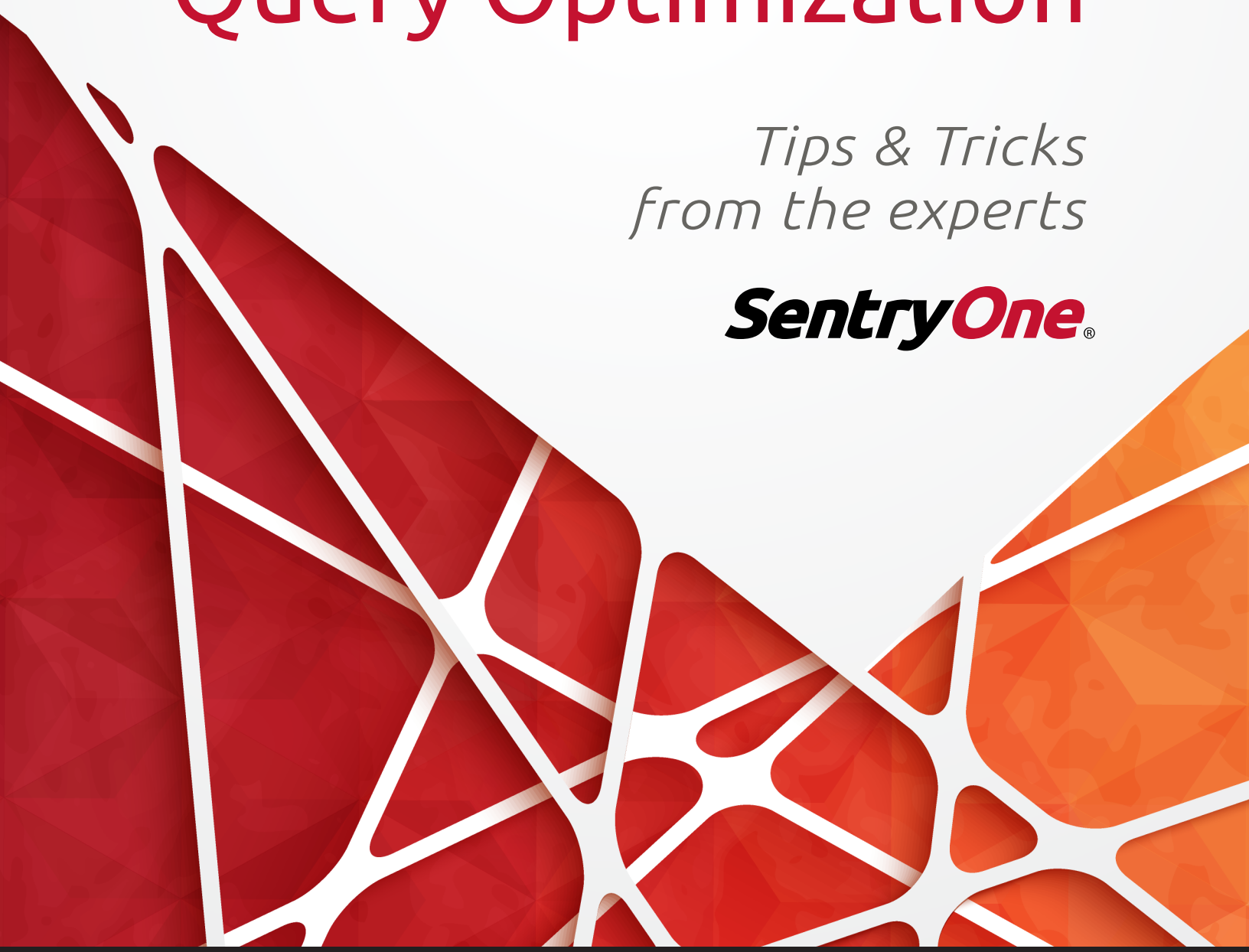
PLAN EXPLORER™

# Query Optimization

*Tips & Tricks from the experts*

SentryOne®



AUTHORS: Paul White | Greg Gonzalez | Aaron Bertrand | Rob Farley

> After recently becoming product manager of Plan Explorer, I continue to feel a sense of pride in helping SQL Server users throughout the community solve their most difficult query issues. This eBook is the first in a series, and will help you get the absolute most out of the additional insight the software offers.

- Aaron Bertrand

# Contents

# Performance Tuning the Whole Query Plan

Paul White

Execution plans provide a rich source of information that can help us identify ways to improve the performance of important queries. People often look for things like large scans and lookups as a way to identify potential data access path optimizations. These issues can often be quickly resolved by creating a new index or extending an existing one with more included columns.

We can also use post-execution plans to compare actual with expected row counts between plan operators. Where these are found to be significantly at variance, we can try to provide the optimizer by updating existing statistics, creating new statistics objects, utilizing statistics on computed columns, or perhaps by breaking a complex query up into less-complex component parts.

Beyond that, we can also look at expensive operations in the plan, particularly memory-consuming ones like sorting and hashing. Sorting can sometimes be avoided through indexing changes. Other times, we might have to refactor the query using syntax that favours a plan that preserves a particular desired ordering.

Sometimes, performance will still not be good enough even after all these performance tuning techniques are applied. A possible next step is to think a bit more about the plan as a whole. This means taking a step back, trying to understand the overall strategy chosen by the query optimizer, to see if we can identify an algorithmic improvement.

This article explores this latter type of analysis, using a simple example problem of finding unique column values in a moderately large data set. As is often the case in analogous real-world problems, the column of interest will have relatively few unique values, compared with the number of rows in the table. There are two parts to this analysis: creating the sample data, and writing the distinct-values query itself.

## Creating the Sample Data

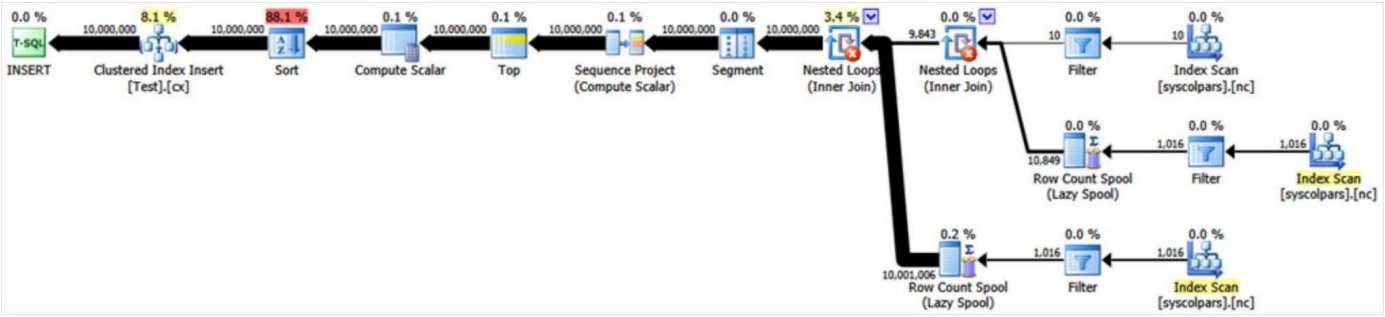
To provide the simplest possible example, our test table has just a single column with a clustered index (this column will hold duplicate values so the index cannot be declared unique):

```
CREATE TABLE dbo.Test
(
    data integer NOT NULL,
);
GO
CREATE CLUSTERED INDEX cx
ON dbo.Test (data);
```

To pick some numbers out of the air, we will choose to load ten million rows in total, with an even distribution over a thousand distinct values. A common technique to generate data like this is to cross join some system tables and apply the ROW_NUMBER function. We will also use the modulo operator to limit the generated numbers to the desired distinct values:

```
INSERT dbo.Test WITH (TABLOCK)
    (data)
SELECT TOP (10000000)
    (ROW_NUMBER() OVER (ORDER BY (SELECT 0)) % 1000) + 1
FROM master.sys.columns AS C1 WITH (READUNCOMMITTED)
CROSS JOIN master.sys.columns AS C2 WITH (READUNCOMMITTED)
CROSS JOIN master.sys.columns C3 WITH (READUNCOMMITTED);
```

The estimated execution plan for that query is as follows:

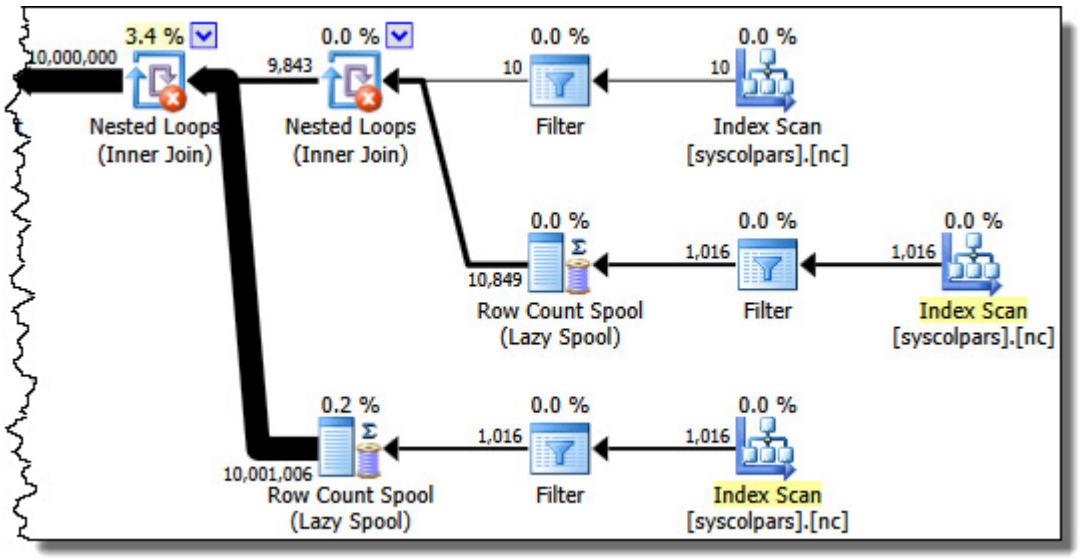



This takes around 30 seconds to create the sample data on my laptop. That is not an enormous length of time by any means, but it is still interesting to consider what we might do to make this process more efficient…

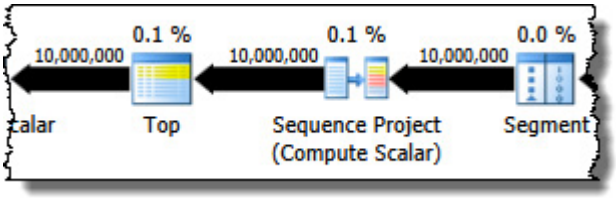
## Plan Analysis

We will start by understanding what each operation in the plan is there for.

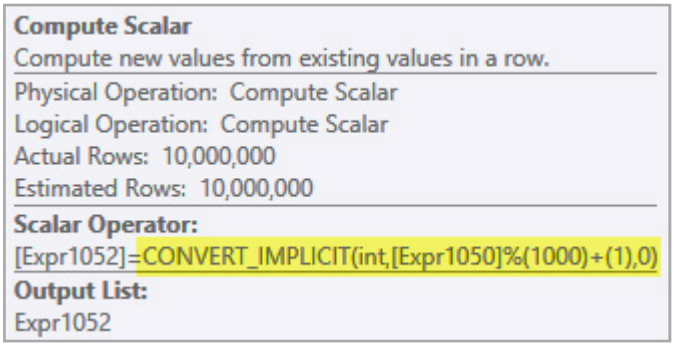
The section of the execution plan to the right of the Segment operator is concerned with manufacturing rows by cross joining system tables:



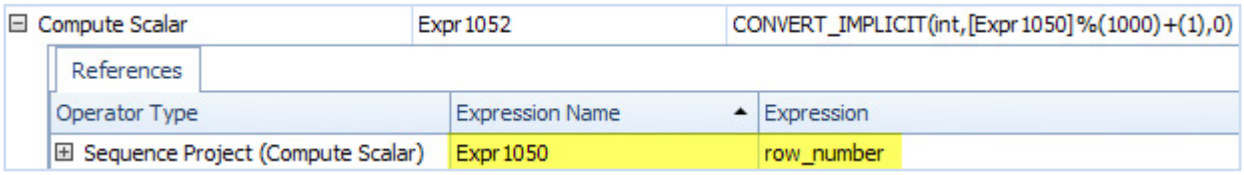

The Segment operator is there in case the window function had a PARTITION BY clause. That is not the case here, but it features in the query plan anyway. The Sequence Project operator generates the row numbers, and the Top limits the plan output to ten million rows:



The Compute Scalar defines the expression that applies the modulo function and adds one to the result:

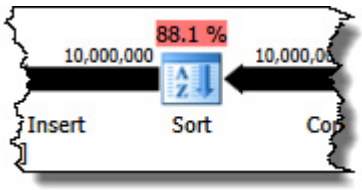



We can see how the Sequence Project and Compute Scalar expression labels relate using Plan Explorer's Expressions tab:



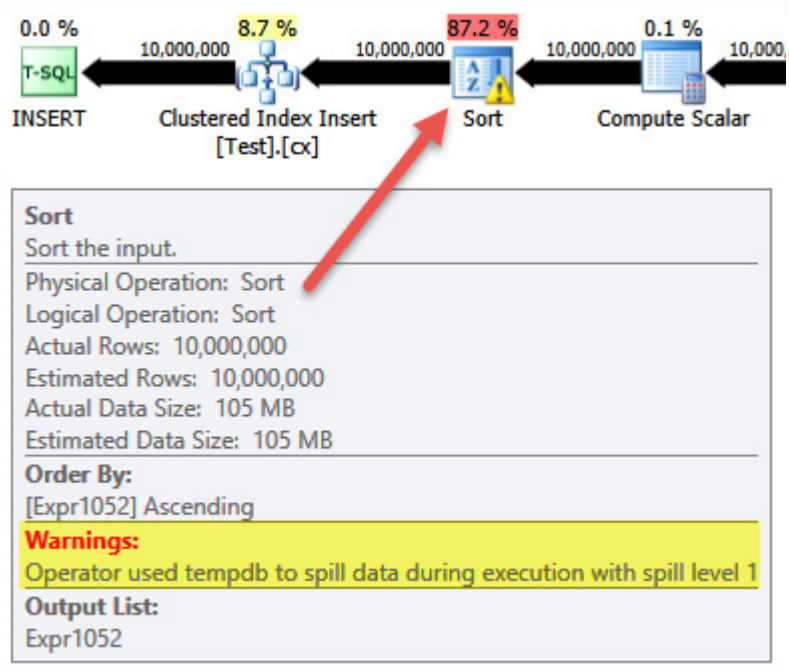

This gives us a more complete feel for the flow of this plan: the Sequence Project numbers the rows and labels the expression Expr1050; the Compute Scalar labels the result of the modulo and plus-one computation as Expr1052. Notice also the implicit conversion in the Compute Scalar expression. The destination table column is of type integer, whereas the ROW_NUMBER function produces a bigint, so a narrowing conversion is necessary.

The next operator in the plan is a Sort. According to the query optimizer's costing estimates, this is expected to be the most expensive operation (88.1% estimated):

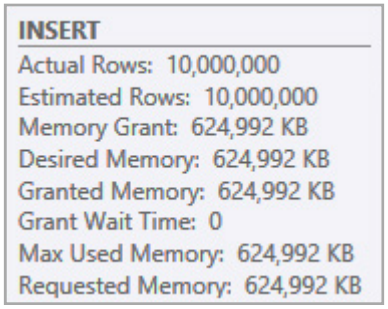



It might not be immediately obvious why this plan features sorting, since there is no explicit ordering requirement in the query. The Sort is added to the plan to ensure rows arrive at the Clustered Index Insert operator in clustered index order. This promotes sequential writes, avoids page splitting, and is one of the pre-requisites for minimally-logged INSERT operations.

These are all potentially good things, but the Sort itself is rather expensive. Indeed, checking the post-execution ("actual") execution plan reveals the Sort also ran out of memory at execution time and had to spill to physical tempdb disk:



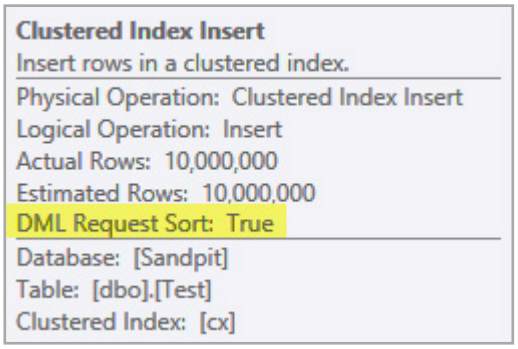

The Sort spill occurs despite the estimated number of rows being exactly right, and despite the fact the query was granted all the memory it asked for (as seen in the plan properties for the root INSERT node):




Sort spills are also indicated by the presence of IO_COMPLETION waits in the Plan Explorer wait stats tab:

| Wait Type | Wait Time | Signal Time |
|---|---|---|
| IO_COMPLETION | 3,913 | 0 |
| WRITELOG | 894 | 4 |
| PAGEIOLATCH_EX | 9 | 0 |
| SOS_SCHEDULER_YIELD | 1 | 1 |

Finally for this plan analysis section, notice the DML Request Sort property of the Clustered Index Insert operator is set true:



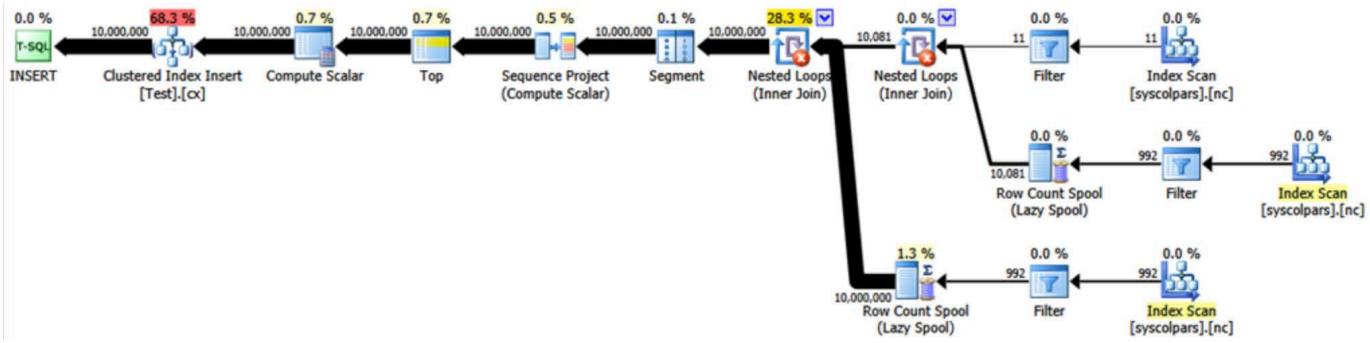

This flag indicates that the optimizer requires the sub-tree below the Insert to provide rows in index key sorted order (hence the need for the problematic Sort operator).

## Avoiding the Sort

Now that we know why the Sort appears, we can test to see what happens if we remove it. There are ways we could rewrite the query to "fool" the optimizer into thinking fewer rows would be inserted (so sorting would not be worthwhile) but a quick way to avoid the sort directly (for experimental purposes only) is to use undocumented trace flag 8795. This sets the DML Request Sort property to false, so rows are no longer required to arrive at the Clustered Index Insert in clustered key order:

```
TRUNCATE TABLE dbo.Test;
GO
INSERT dbo.Test WITH (TABLOCK)
    (data)
SELECT TOP (10000000)
    ROW_NUMBER() OVER (ORDER BY (SELECT 0)) % 1000
FROM master.sys.columns AS C1 WITH (READUNCOMMITTED)
CROSS JOIN master.sys.columns AS C2 WITH (READUNCOMMITTED)
CROSS JOIN master.sys.columns C3 WITH (READUNCOMMITTED)
OPTION (QUERYTRACEON 8795);
```

The new post-execution query plan is as follows (click the image to enlarge):



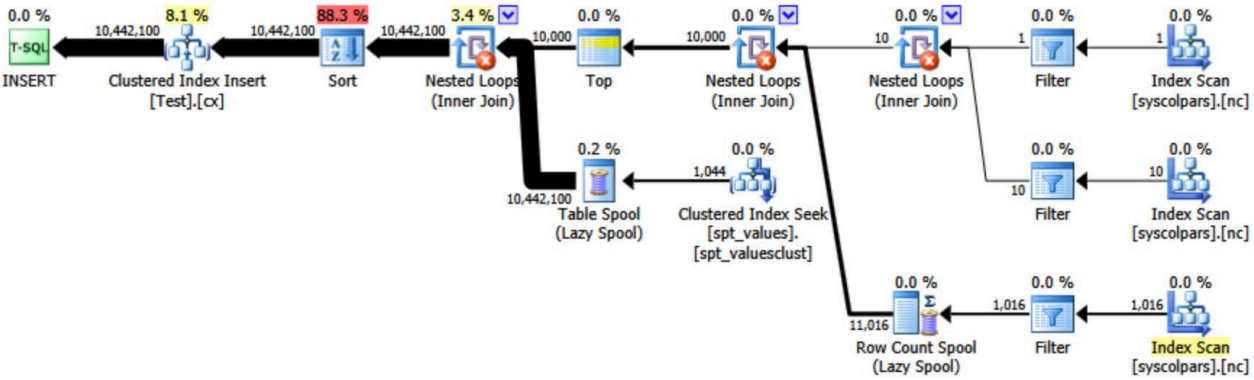
The Sort operator has gone, but the new query runs for over 50 seconds (compared with 30 seconds before). There are a couple of reasons for this. First, we lose any possibility of minimally-logged inserts because these require sorted data (DML Request Sort = true). Second, a large number of "bad" page splits will occur during the insert. In case that seems counter-intuitive, remember that although the ROW_NUMBER function numbers rows sequentially, the effect of the modulo operator is to present a repeating sequence of numbers 1…1000 to the Clustered Index Insert. The same fundamental issue occurs if we use T-SQL tricks to lower the expected row count to avoid the sort instead of using the unsupported trace flag.

## Avoiding the Sort II

Looking at the plan as a whole, it seems clear we would like to generate rows in a way that avoids an explicit sort, but which still reaps the benefits of minimal logging and bad page split avoidance. Put simply: we want a plan that presents rows in clustered key order, but without sorting. Armed with this new insight, we can express our query in a different way. The following query generates each number from 1 to 1000 and cross joins that set with 10,000 rows to produce the required degree of duplication. The idea is to generate an insert set that presents 10,000 rows numbered '1' then 10,000 numbered '2' … and so on.

```
TRUNCATE TABLE dbo.Test;
GO
INSERT dbo.Test WITH (TABLOCK)
    (data)
SELECT
    N.number
FROM
(
    SELECT SV.number
    FROM master.dbo.spt_values AS SV WITH (READUNCOMMITTED)
    WHERE SV.[type] = N'P'
    AND SV.number >= 1
    AND SV.number <= 1000
) AS N
CROSS JOIN
(
    SELECT TOP (10000)
        Dummy = NULL
    FROM master.sys.columns AS C1 WITH (READUNCOMMITTED)
    CROSS JOIN master.sys.columns AS C2 WITH (READUNCOMMITTED)
    CROSS JOIN master.sys.columns C3 WITH (READUNCOMMITTED)
) AS C;
```

Unfortunately, the optimizer still produces a plan with a sort:




There is not much to be said in the optimizer's defense here, this is just a daft plan. It has chosen to generate 10,000 rows then cross join those with numbers from 1 to 1000. This does not allow the natural order of the numbers to be preserved, so the sort cannot be avoided.

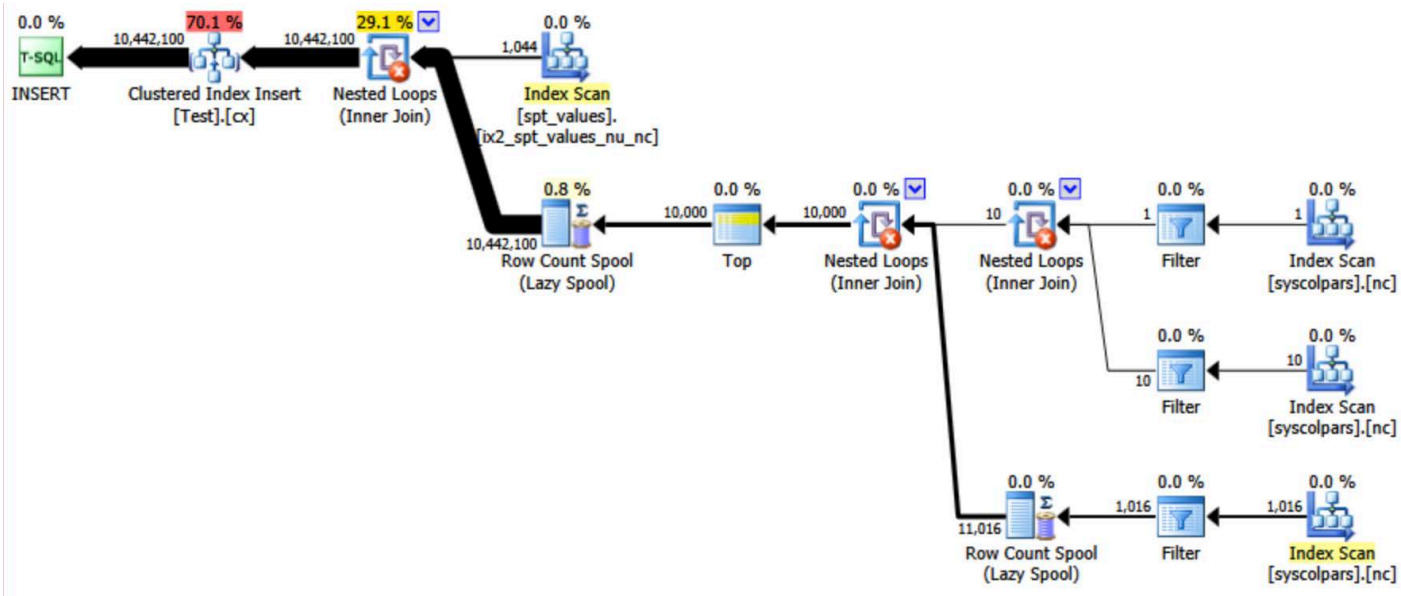
## Avoiding the Sort - Finally!

The strategy the optimizer missed is to take the numbers 1…1000 first, and cross join each number with 10,000 rows (making 10,000 copies of each number in sequence). The expected plan would avoid a sort by using a nested loops cross join that preserves the order of the rows on the outer input.

We can achieve this outcome by forcing the optimizer to access the derived tables in the order specified in the query, using the FORCE ORDER query hint:

```
TRUNCATE TABLE dbo.Test;
GO
INSERT dbo.Test WITH (TABLOCK)
    (data)
SELECT
    N.number
FROM
(
    SELECT SV.number
    FROM master.dbo.spt_values AS SV WITH (READUNCOMMITTED)
    WHERE SV.[type] = N'P'
    AND SV.number >= 1
    AND SV.number <= 1000
) AS N
CROSS JOIN
(
    SELECT TOP (10000)
        Dummy = NULL
    FROM master.sys.columns AS C1 WITH (READUNCOMMITTED)
    CROSS JOIN master.sys.columns AS C2 WITH (READUNCOMMITTED)
    CROSS JOIN master.sys.columns C3 WITH (READUNCOMMITTED)
) AS C
OPTION (FORCE ORDER);
```

Finally, we get the plan we were after:



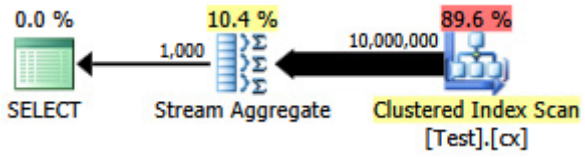

This plan avoids an explicit sort while still avoiding "bad" page splits and enabling minimally-logged inserts to the clustered index (assuming the database is not using the FULL recovery model). It loads all ten million rows in about 9 seconds on my laptop (with a single 7200 rpm SATA spinning disk). This represents a marked efficiency gain over the 30-50 second elapsed time seen before the rewrite.

## Finding the Distinct Values

Now we have the sample data created, we can turn our attention to writing a query to find the distinct values in the table. A natural way to express this requirement in T-SQL is as follows:

```
SELECT DISTINCT data
FROM dbo.Test WITH (TABLOCK)
OPTION (MAXDOP 1);
```

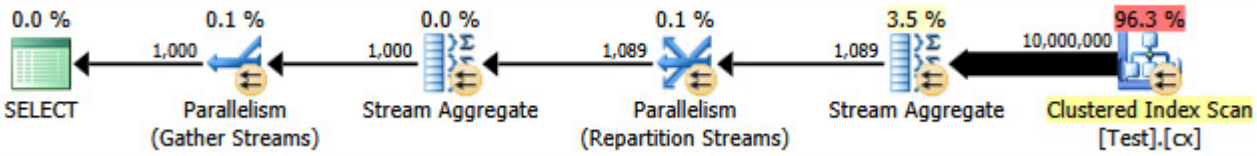
The execution plan is very simple, as you would expect:




This takes around 2900 ms to run on my machine, and requires 43,406 logical reads:

| Statement | Duration | CPU | Reads |
|---|---|---|---|
| SELECT DISTINCT data FROM dbo.Test WITH (TABLOCK) OPTION (MAXDOP 1) | 2,873 | 2,859 | 43,406 |

Removing the MAXDOP (1) query hint generates a parallel plan:




This completes in about 1500 ms (but with 8,764 ms of CPU time consumed), and 43,804 logical reads:

| Statement | Duration | CPU | Reads |
|---|---|---|---|
| SELECT DISTINCT data FROM dbo.Test WITH (TABLOCK) | 1,447 | 8,764 | 43,804 |

The same plans and performance result if we use GROUP BY instead of DISTINCT.

## A Better Algorithm

The query plans shown above read all values from the base table and process them through a Stream Aggregate. Thinking of the task as a whole, it seems inefficient to scan all 10 million rows when we know there are relatively few distinct values.

A better strategy might be to find the single lowest value in the table, then find the next highest, and so on until we run out of values. Crucially, this approach lends itself to singleton-seeking into the index rather than scanning every row.

We can implement this idea in a single query using a recursive CTE, where the anchor part finds the lowest distinct value, then the recursive part finds the next distinct value and so on.

A first attempt at writing this query is:

```
WITH RecursiveCTE
AS
(
    -- Anchor
    SELECT data = MIN(T.data)
    FROM dbo.Test AS T

    UNION ALL

    -- Recursive
    SELECT MIN(T.data)
    FROM dbo.Test AS T
    JOIN RecursiveCTE AS R
        ON R.data < T.data
)
SELECT data
FROM RecursiveCTE
OPTION (MAXRECURSION 0);
```

Unfortunately, that syntax does not compile:

```
Msg 467, Level 16, State 1, Line 1
GROUP BY, HAVING, or aggregate functions are not allowed
in the recursive part of a recursive common table expression 'RecursiveCTE'.
```

Ok, so aggregate functions are not allowed. Instead of using MIN, we can write the same logic using TOP (1) with an ORDER BY:

```
WITH RecursiveCTE
AS
(
    -- Anchor
    SELECT TOP (1)
        T.data
    FROM dbo.Test AS T
    ORDER BY
        T.data

    UNION ALL

    -- Recursive
    SELECT TOP (1)
        T.data
    FROM dbo.Test AS T
    JOIN RecursiveCTE AS R
        ON R.data < T.data
    ORDER BY T.data
)
SELECT
    data
FROM RecursiveCTE
OPTION (MAXRECURSION 0);
```

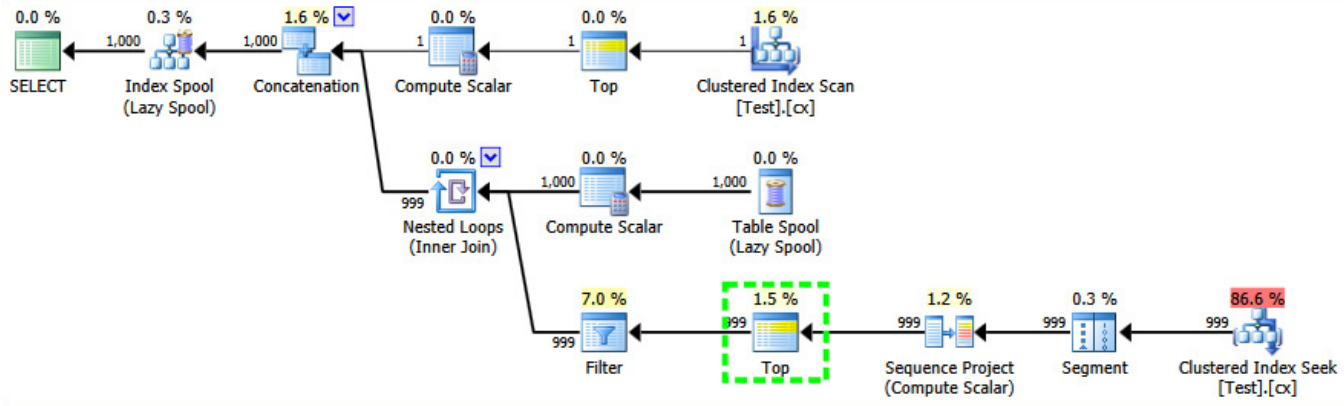
```
Msg 461, Level 16, State 1, Line 1
The TOP or OFFSET operator is not allowed in
the recursive part of a recursive common table expression 'RecursiveCTE'.
```

Still no joy.

It turns out that we can get around these restrictions by rewriting the recursive part to number the candidate rows in the required order, then filter for the row that is numbered 'one'. This might seem a little circuitous, but the logic is exactly the same:

```
WITH RecursiveCTE
AS
(
    -- Anchor
    SELECT TOP (1)
        data
    FROM dbo.Test AS T
    ORDER BY
        T.data

    UNION ALL

    -- Recursive
    SELECT R.data
    FROM
    (
        -- Number the rows
        SELECT
            T.data,
            rn = ROW_NUMBER() OVER (
                ORDER BY T.data)
        FROM dbo.Test AS T
        JOIN RecursiveCTE AS R
            ON R.data < T.data
    ) AS R
    WHERE
        -- Only the row that sorts lowest
        R.rn = 1
)
SELECT
    data
FROM RecursiveCTE
OPTION (MAXRECURSION 0);
```

This query does compile, and produces the following post-execution plan:




Notice the Top operator in the recursive part of the execution plan (highlighted). We cannot write a T-SQL TOP in the recursive part of a recursive common table expression, but that does not mean the optimizer cannot use one! The optimizer introduces the Top based on reasoning about the number of rows it will need to check to find the one numbered '1'.

The performance of this (non-parallel) plan is much better than the Stream Aggregate approach. It completes in around 50 ms, with 3007 logical reads against the source table (and 6001 rows read from the spool worktable), compared with the previous best of 1500ms (8764 ms CPU time at DOP 8) and 43,804 logical reads:

| Statement | Duration | CPU |
|---|---|---|
| WITH RecursiveCTE AS ( -- Anchor SELECT TOP (1) data FROM dbo.Test AS T O… | 50 | 31 |

| Table I/O | | |
|---|---|---|
| Table | Logical Reads | Scan Count |
| Test | 3,007 | 1,001 |
| Worktable | 6,001 | 2 |

## Conclusion

It is not always possible to achieve breakthroughs in query performance by considering individual query plan elements on their own. Sometimes, we need to analyse the strategy behind the whole execution plan, then think laterally to find a more efficient algorithm and implementation.

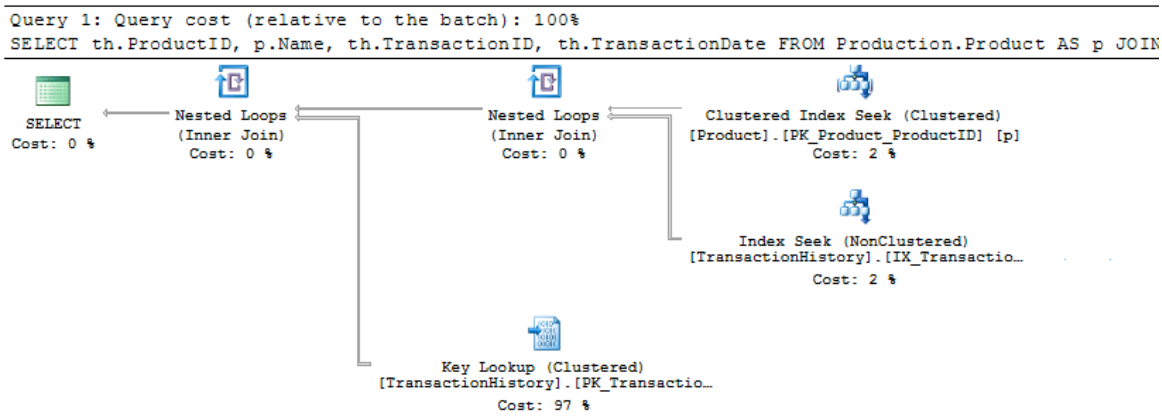
Aaron Bertrand

Over three years ago now, I posted about a fix to Plan Explorer regarding bad cardinality estimates that SQL Server's Showplan XML was producing, in the case of key/RID lookups with a filter predicate in SQL Server 2008 and above. I thought it would be interesting to look back and go into a little more detail about one of these plans and the iterations that we went through to ensure we were displaying correct metrics, regardless of what Management Studio shows. Again, this work was largely done by Brooke Philpott and Greg Gonzalez and with great input from Paul White.

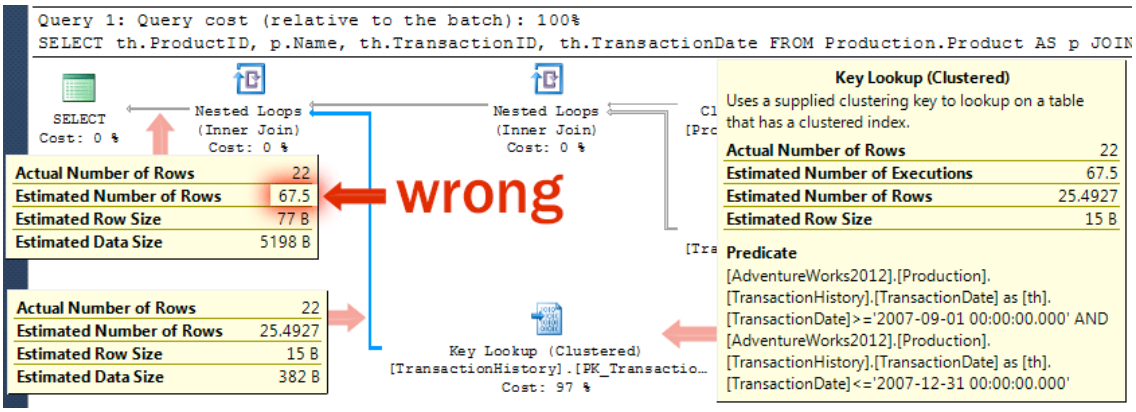
This is quite similar to the query I presented in my earlier post, which came from Paul (and which would take some work to reproduce exactly in modern versions of AdventureWorks, where at the very least transaction dates have changed):

```
SELECT
    th.ProductID,
    p.Name,
    th.TransactionID,
    th.TransactionDate
FROM Production.Product AS p
JOIN Production.TransactionHistory AS th ON
    th.ProductID = p.ProductID
WHERE
    p.ProductID IN (1, 2)
    AND th.TransactionDate BETWEEN '20070901' AND '20071231';
```

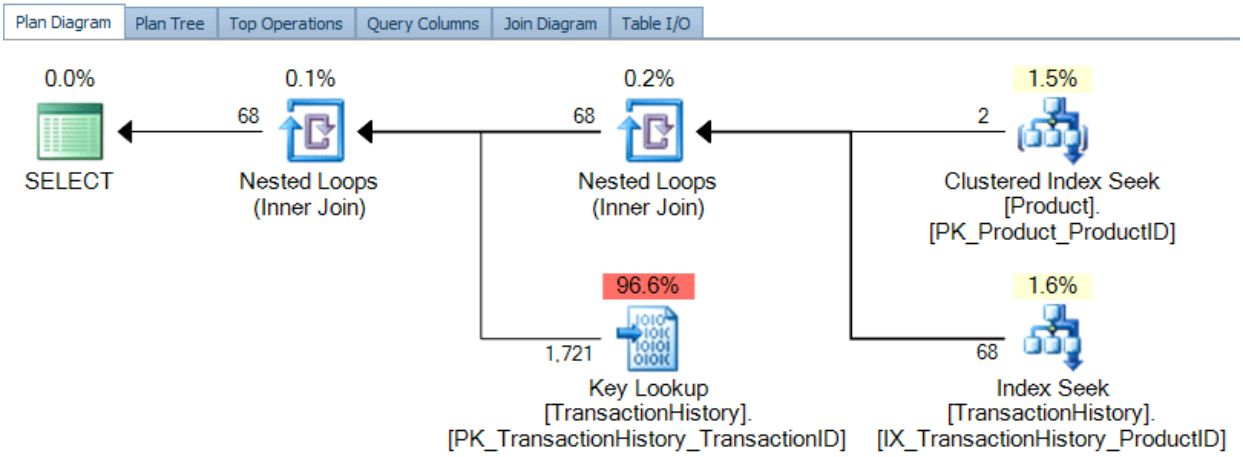
The plan from Management Studio looked correct enough:




However, if you look closer, it seems that the ShowPlan has pushed the estimated number of executions from the key lookup straight over to the estimated number of rows for the final exchange:



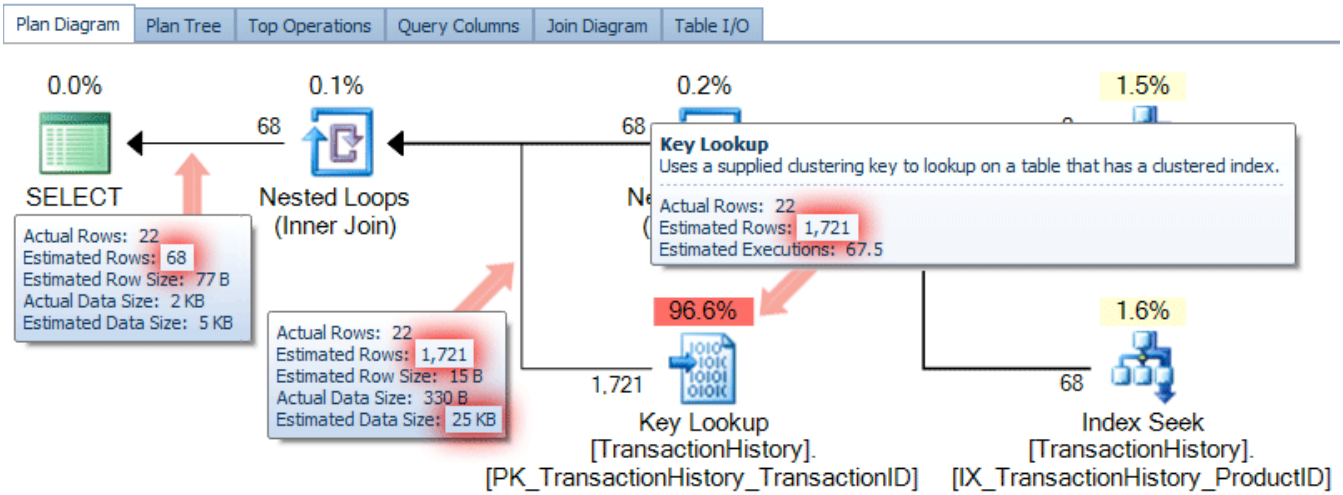
On first glance, the graphical plan diagram in Plan Explorer looks quite similar to the plan that SSMS produces:




## Cases

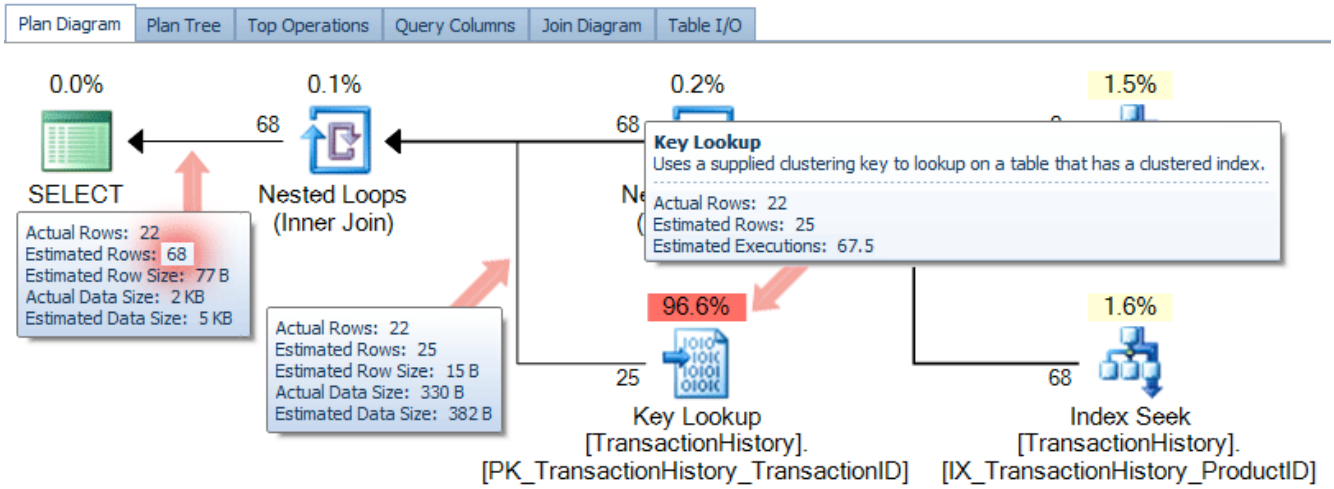
Now, in the process of developing Plan Explorer, we have discovered several cases where ShowPlan doesn't quite get its math correct. The most obvious example is percentages adding up to over 100%; we get this right in cases where SSMS is ridiculously off (I see this less often today than I used to, but it still happens).

Another case is where, starting in SQL Server 2008, SSMS started putting total estimated rows instead of rows per execution along with lookups, but only in cases where a predicate is pushed to the lookup (such as the case in this bug reported by Paul, and this more recent observation by Joey D'Antoni). In earlier versions of SQL Server (and with functions and spools), we would typically show estimated row counts coming out of a lookup by multiplying the estimated rows per execution (usually 1) by the estimated number of rows according to SSMS. But with this change, we would be over-counting, since the operator is now already doing that math. So, in earlier versions of Plan Explorer, against 2008+, you would see these details in the tooltips, connector lines, or in the various grids:



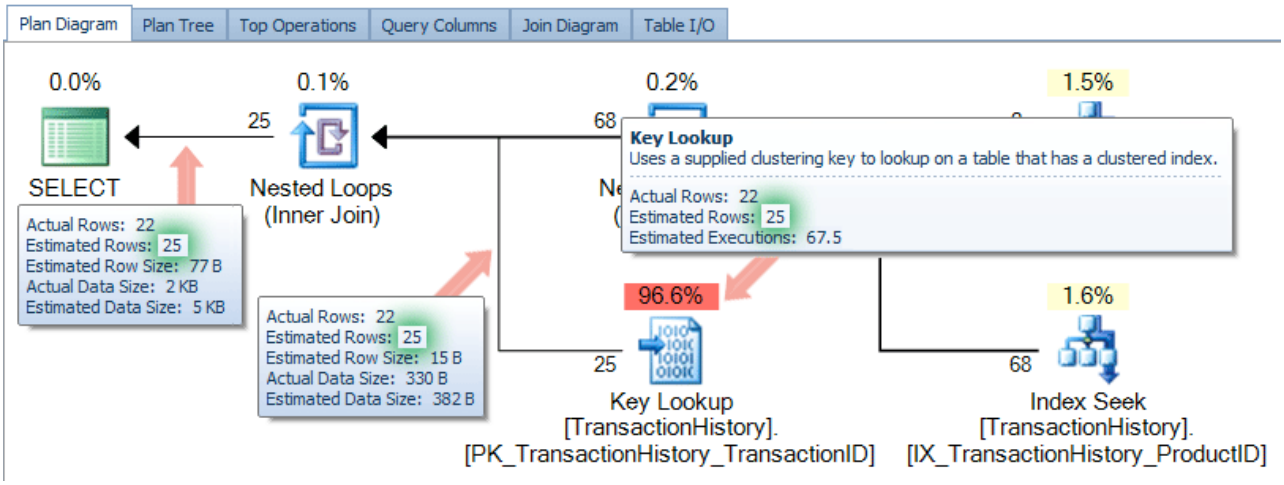

***Where does 1,721 come from? 67.5 estimated executions x 25.4927 estimated rows.***

Back in 2012, we fixed part of this issue by not performing this mathematical operation any longer, and relying solely on the estimated row counts coming out of the key lookup. This was almost correct, but we were still relying on the estimated row count ShowPlan was providing us for the final exchange:




## Fixing The Bug

We quickly addressed this issue as well, in version 7.2.42.0 (released on Halloween 2012), and now feel we are providing information that is much more accurate than Management Studio (though we will keep an eye on this bug from Paul):




This clearly happened a long time ago, but I still thought it would be interesting to share. We continue to make enhancements to Plan Explorer to provide you with the most accurate information possible, and I will be sharing a few more of these nuggets in upcoming posts.

Rob Farley

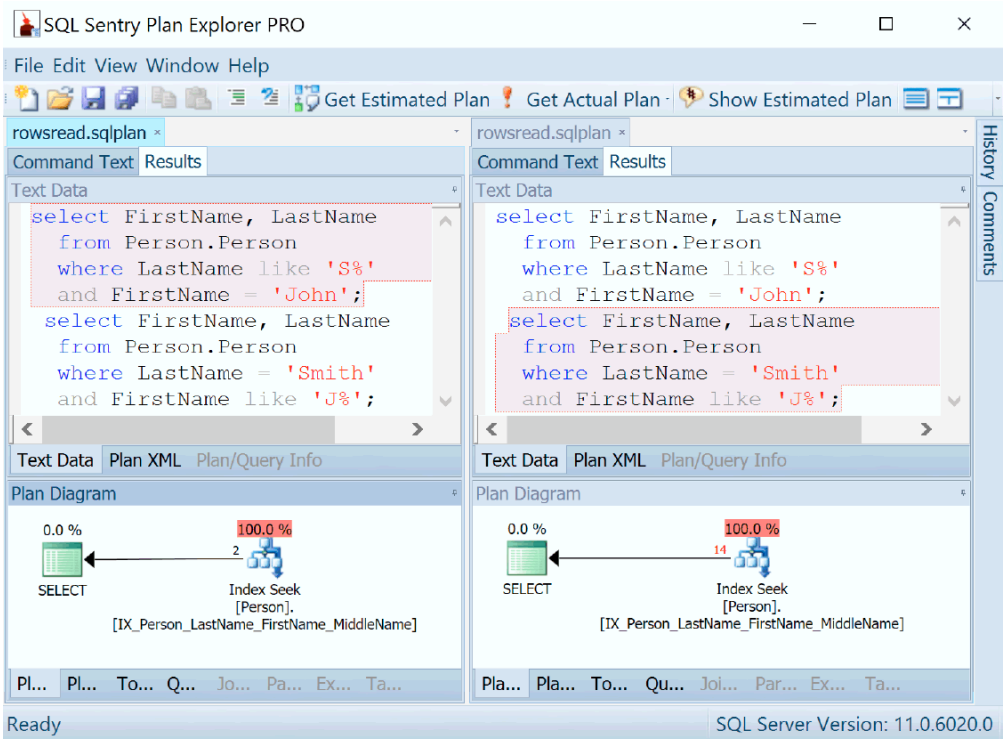
The new property "Actual Rows Read" in execution plans (which in SQL Server Management Studio is displayed as "Number of Rows Read") was a welcome addition to performance tuners. It's like having a new superpower, to be able to tell the significance of the Seek Predicate v the Residual Predicate within a Seek operator. I love this, because it can be really significant to querying. Let's look at two queries, which I'm running against AdventureWorks2012. They're very simple – one lists people called John S, and the other lists people called J Smith. Like all good phonebooks, we have an index on LastName,

```
select FirstName, LastName
  from Person.Person
  where LastName like 'S%'
  and FirstName = 'John';

select FirstName, LastName
  from Person.Person
  where LastName = 'Smith'
  and FirstName like 'J%';
```

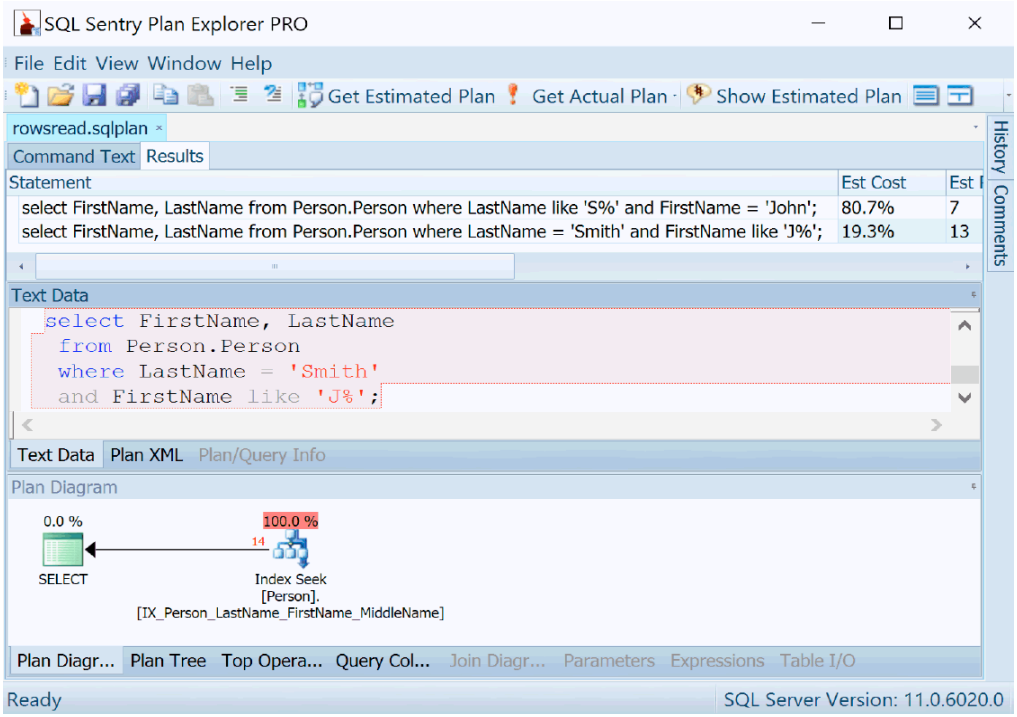
*In case you're curious, I get 2 rows back from the first one, and 14 rows back from the second. I'm not actually that interested in the results, I'm interested in the execution plans.*

## What's Going On?

I opened up an older copy of SQL Sentry Plan Explorer, and open my plans side by side. Incidentally – I had run both queries together and so both plans were in the same .sqlplan file. But I could open the same file twice in PE, and happily sit them side by side in tab groups.



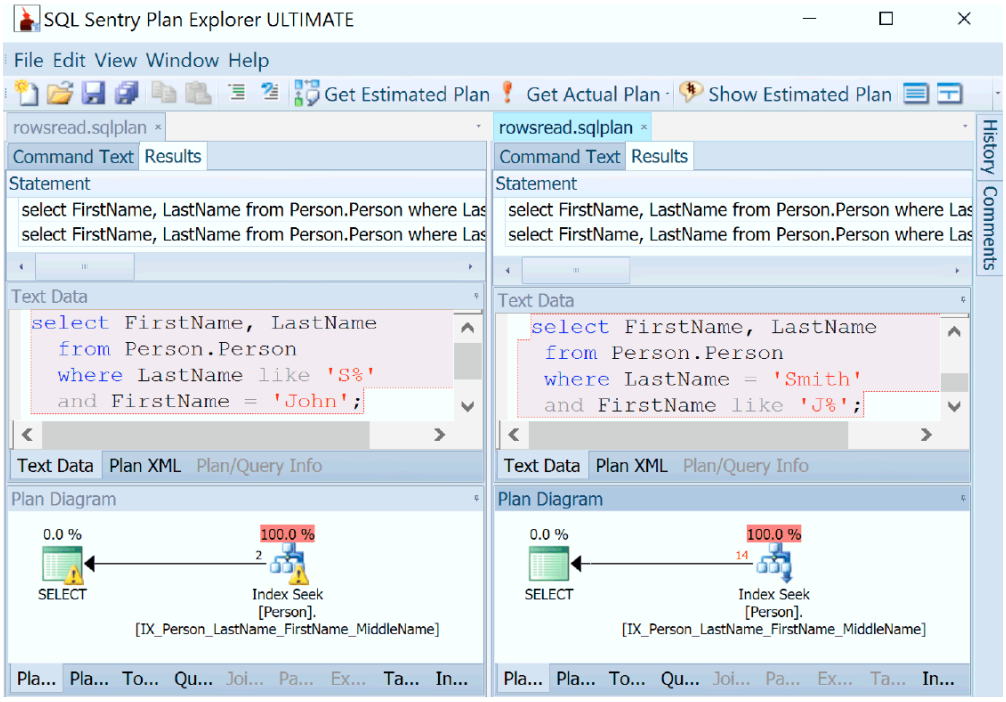
Great. They look the same! I can see that the Seek on the left is producing two rows instead of fourteen – obviously this is the better query. But with a larger window, I would've seen more information, and it's lucky that I had run the two queries in the same batch.




You can see that the second query, which produced 14 rows rather than 2 rows was estimated to take over 80% of the cost! If I'd run the queries separately, each would be showing me 100%.

## Let's Compare

***Now let's compare with the latest release of Plan Explorer.***



## Looking Closer

*The thing that jumps out to me immediately is the warning.*

Scan a particular range of rows from a nonclustered index.
Node ID: 0
Physical Operation: Index Seek
Logical Operation: Index Seek
Actual Rows: 2
Actual Rows Read: 2,130
Estimated Rows: 7
Estimated I/O Cost: 0.0112732
Estimated CPU Cost: 0.0025015
Actual Executions: 1
Estimated Executions: 1.0
Estimated Operator Cost: 0.0137747 (100.0%)
Estimated Subtree Cost: 0.0137747
Estimated Row Size: 35 B
Actual Data Size: 70 B
Estimated Data Size: 238 B
Actual Rebinds: 0
Actual Rewinds: 0
Ordered: True

Database: [AdventureWorks2012]
Table: [Person].[Person]
Index: [IX_Person_LastName_FirstName_MiddleName]

**Seek Predicates:**
[AdventureWorks2012].[Person].[Person].[LastName] >= N'S'
[AdventureWorks2012].[Person].[Person].[FirstName] >= N'John'
[AdventureWorks2012].[Person].[Person].[LastName] < N'T'

**Predicate:**
[AdventureWorks2012].[Person].[Person].[FirstName]=N'John'
AND [AdventureWorks2012].[Person].[Person].[LastName] like N'S%'

**Warnings:**
Operation caused residual IO. The actual number of rows read was 2,130, but the number of rows returned was 2.

**Output List:**
FirstName
LastName

The warning says “Operation caused residual IO. The actual number of rows read was 2,130, but the number of rows returned was 2.” Sure enough, further up we see “Actual Rows Read” saying 2,130, and Actual Rows at 2.

Whoa! To find those rows, we had to look through 2,130?

You see, the way that the Seek runs is to start by thinking about the Seek Predicate. That’s the one that leverages the index nicely, and which actually causes the operation to be a Seek. Without a Seek Predicate, the operation becomes a Scan. Now, if this Seek Predicate is guaranteed to be at most one row (such as when it has an equality operator on a unique index), then we have a Singleton seek. Otherwise, we have a Range Scan, and this range can have a Prefix, a Start, and an End (but not necessarily both a Start and an End). This defines the rows in the table that we’re interested in for the Seek.

## Residual Predicate

But ‘interested in’ doesn’t necessarily mean ‘returned’, because we might have more work to do. That work is described in the other Predicate, which is often known as the Residual Predicate.

Now that Residual Predicate might actually be doing most of the work. It certainly is here – it’s filtering things down from 2,130 rows to just 2.

The Range Scan starts in the index at “John S”. We know that if there is a “John S”, this must be the first row that can satisfy the whole thing. “Ian S” can’t. So we can search into the index at that point to start our Range Scan. If we look at the Plan XML we can see this explicitly.

Notice that we don't have a Prefix. That applies when you have an equality in the first column within the index. We just have StartRange and EndRange. The Start of the range is "Greater Than or Equal" (GE) ScanType, at the value "S, John" (the column references off-screen are LastName, FirstName), and the End of the range is "Less Than" (LT) the value T. When the scan hits T, it's done. Nothing more to do. The Seek has now completed its Range Scan. And in this case, it returns 2,130 rows!

Except that it doesn't actually return 2,130 rows, it just reads 2,130 rows. Names like Barry Sai and Ken Sánchez are read, but only the names that satisfy the next check are returned – the Residual Predicate that makes sure that the FirstName is John.

The Actual Rows Read entry in the Index Seek operator's properties shows us this value of 2,130. And while it's visible in earlier releases of Plan Explorer, we don't get a warning about it. That's relatively new.

Our second query (looking for J Smith) is much nicer, and there's a reason why it was estimated to be more than 4 times cheaper.

Here we know the LastName exactly (Smith), and the Range Scan is on the FirstName (J%).

## The Prefix


We see that our Prefix is an Equality operator (=, ScanType="EQ"), and that LastName must be Smith. We haven't even considered the Start or End of the range yet, but the Prefix tells us that the range is included within the portion of the index where LastName is Smith. Now we can find the rows >= J and < K.

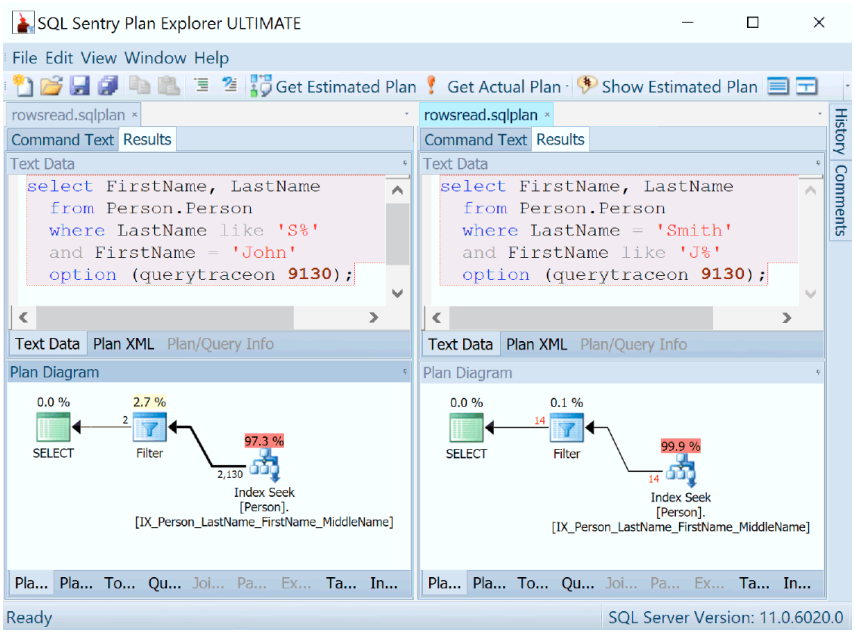
## No Waste

There is still a Residual Predicate here, but this is only make sure that "LIKE J%" is actually tested. While it seems intuitive to us that "LIKE J%" is exactly equivalent to ">= J and < K", the system doesn't guarantee that and wants to do an extra check. Importantly, we see the Actual Rows (returned) being the same as Actual Rows Read. They're both 14, and we're not wasting any resources looking at rows that we don't want.

**Index Seek**
Scan a particular range of rows from a nonclustered index.
Node ID: 0
Physical Operation: Index Seek
Logical Operation: Index Seek
Actual Rows: 14
Actual Rows Read: 14
Estimated Rows: 13
Estimated I/O Cost: 0.0031250
Estimated CPU Cost: 0.0001712
Actual Executions: 1
Estimated Executions: 1.0
Estimated Operator Cost: 0.0032962 (100.0%)
Estimated Subtree Cost: 0.0032962
Estimated Row Size: 35 B
Actual Data Size: 490 B
Estimated Data Size: 452 B
Actual Rebinds: 0
Actual Rewinds: 0
Ordered: True
Database: [AdventureWorks2012]
Table: [Person].[Person]
Index: [IX_Person_LastName_FirstName_MiddleName]
**Seek Predicates:**
[AdventureWorks2012].[Person].[Person].[FirstName] >= N'J'
[AdventureWorks2012].[Person].[Person].[FirstName] < N'K'
[AdventureWorks2012].[Person].[Person].[LastName] = N'Smith'
**Predicate:**
[AdventureWorks2012].[Person].[Person].[FirstName] like N'J%'
**Output List:**
FirstName
LastName

## TF Turned On

Before Service Pack 3 of SQL Server 2012, we didn't have this property, and to get a feel for the difference between the Actual Rows Read and the Actual Rows, we'd need to use trace flag 9130. Here are those two plans with that TF turned on.




You can see there's no warning this time, because the Seek operator is returning all 2130 rows. I think if you're using a version of SQL Server that supports this Actual Rows Read, you should stop using the trace flag 9130 in your investigations, and start looking at the warnings in Plan Explorer instead. But most of all, understand what how your operators do their stuff, because then you'll be able to interpret whether you're happy with the plan, or whether you need to take action.

In another post, I'll show you a situation when you may prefer to see Actual Rows Read be higher than Actual Rows.

***- @rob_farley***

Rob Farley

Don't get me wrong – I love the Actual Rows Read property that we saw arrive in SQL Server's execution plans in late 2015. But in SQL Server 2016 SP1, less than two months ago (and considering we've had Christmas in between, I don't think much of the time since then counts), we got another exciting addition – Estimated Number of Rows to be Read (oh, and this is somewhat down to the Connect item I submitted, both demonstrating that Connect Items are worth submitting and making this post eligible for this month's T-SQL Tuesday, hosted by Brent Ozar (@brento) on the topic of Connect items).

Let's recap a moment… when the SQL Engine access data in a table, it uses either a Scan operation or a Seek operation. And unless that Seek has a Seek Predicate that can access at most one row (because it's looking for an equality match on a set of columns – could be just a single column – which are known to be unique), then the Seek will perform a RangeScan, and behaves just like a Scan, just across the subset of rows that are satisfied by the Seek Predicate.

## Seek Predicate

The rows satisfied by a Seek Predicate (in the case of a Seek operation's RangeScan) or all the rows in the table (in the case of a Scan operation) are treated in essentially the same way. Both might get terminated early if no more rows are requested from the operator to its left, for example if a Top operator somewhere has already grabbed enough rows, or if a Merge Operator has no more rows to match against. And both might be filtered further by a Residual Predicate (shown as the 'Predicate' property) before the rows even get served up by the Scan/Seek operator. The "Number of Rows" and "Estimated Number of Rows" properties would tell us how many rows were expected to be produced by the operator, but we didn't have any information about how may rows would be filtered by just the Seek Predicate. We could see the TableCardinality, but this was only really useful for Scan operators, where there was a chance that the Scan might look through the whole table for the rows it needed. It wasn't useful at all for Seeks.

Top Operations

Drag a column header here to group by that column

| Operation | Table Cardinality | Object |
|---|---|---|
| Index Seek | 73,595 | [WideWorldImporters].[Sales].[Orders].[rf_Orders_SalesPeople_OrderDate].(NonClustered) |

***The query that I'm running here is against the WideWorldImporters database, and is:***

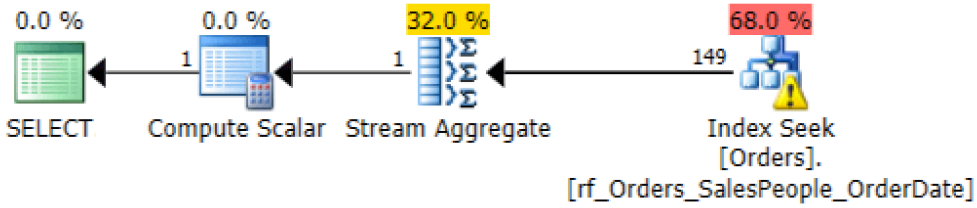
```
SELECT COUNT(*)
FROM Sales.Orders
WHERE SalespersonPersonID = 7
AND YEAR(OrderDate) = 2013
AND MONTH(OrderDate) = 4;
```

***Furthermore, I have an index in play:***

```
CREATE NONCLUSTERED INDEX rf_Orders_SalesPeople_OrderDate
  ON Sales.Orders (SalespersonPersonID, OrderDate);
```

This index is covering – the query doesn't need any other columns to get its answer – and has been designed so that a Seek Predicate can be used on SalespersonPersonID, quickly filtering the data down to a smaller range. The functions on OrderDate mean that those last two predicates can't be used within the Seek Predicate, so they are relegated to the Residual Predicate instead. A better query would filter those dates using OrderDate >= '20130401' AND OrderDate < '20130501', but I'm imagining a scenario here which is all too common…

Now, if I run the query, I can see the impact of the Residual Predicates. Plan Explorer even gives that useful warning that I'd written about before.




## Tooltip

I can see very clearly that the RangeScan is 7,276 rows, and that the Residual Predicate filters this down to 149. Plan Explorer shows more information about this on the tooltip:

**Index Seek**
Scan a particular range of rows from a nonclustered index.

Node ID: 3
Physical Operation: Index Seek
Logical Operation: Index Seek
Actual Rows: 149
Actual Rows Read: 7,276
Estimated Rows: 1,383
Estimated I/O Cost: 0.0137355
Estimated CPU Cost: 0.0081606
Actual Executions: 1
Estimated Executions: 1.0
Estimated Operator Cost: 0.0218961 (68.0%)
Estimated Subtree Cost: 0.0218961
Estimated Row Size: 10 B
Actual Data Size: 1 KB
Estimated Data Size: 14 KB
Actual Rebinds: 0
Actual Rewinds: 0
Ordered: True

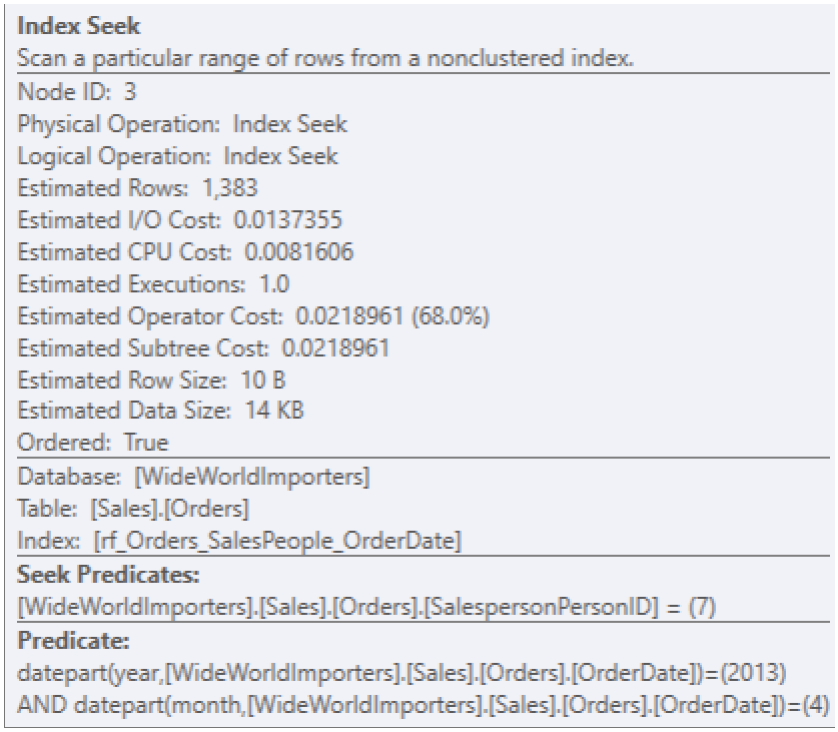
Database: [WideWorldImporters]
Table: [Sales].[Orders]
Index: [rf_Orders_SalesPeople_OrderDate]

**Seek Predicates:**
[WideWorldImporters].[Sales].[Orders].[SalespersonPersonID] = (7)

**Predicate:**
datepart(year,[WideWorldImporters].[Sales].[Orders].[OrderDate])=(2013)
AND datepart(month,[WideWorldImporters].[Sales].[Orders].[OrderDate])=(4)

**Warnings:**
Operation caused residual IO. The actual number of rows read was 7,276, but the number of rows returned was 149.

But without running the query, I can't see that information. It's simply not there. The properties in the estimated plan don't have it:




## The Cache

And I'm sure I don't need to remind you – this information is not present in the plan cache either. Having grabbed the plan from the cache using:

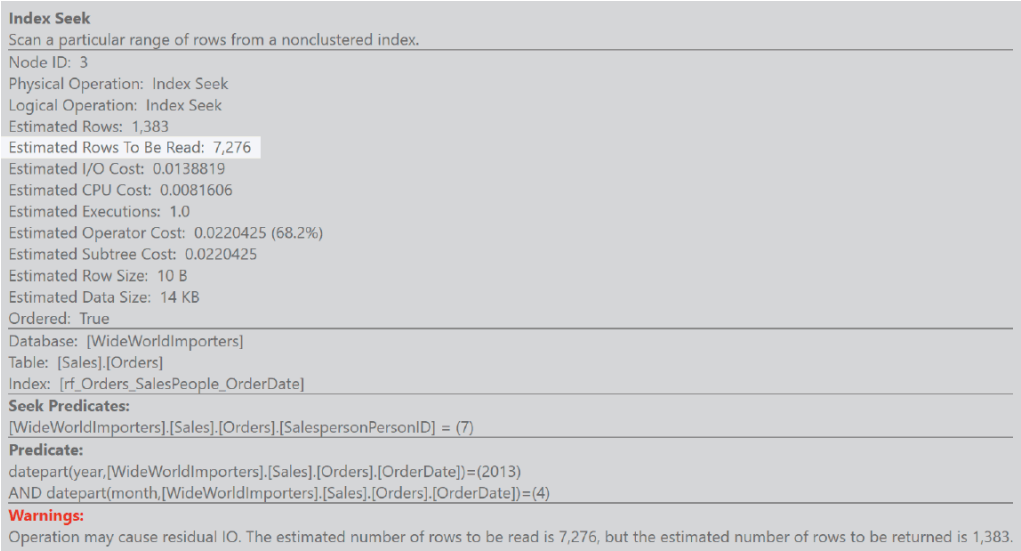
```
SELECT p.query_plan, t.text
FROM sys.dm_exec_cached_plans c
CROSS APPLY sys.dm_exec_query_plan(c.plan_handle) p
CROSS APPLY sys.dm_exec_sql_text(c.plan_handle) t
WHERE t.text LIKE '%YEAR%';
```

I opened it up, and sure enough, no sign of that 7,276 value. It looks just the same as the estimated plan I just showed.

Getting plans out of the cache is where the estimated values come into their own. It's not just that I'd prefer to not actually run potentially-expensive queries on customer databases. Querying the plan cache is one thing, but running queries to get the actuals – that's a lot harder.

## SQL 2016 SP1

With SQL 2016 SP1 installed, thanks to that Connect item, I can now see the Estimated Number of Rows to be Read property in estimated plans, and in the plan cache. The operator tooltip shown here is taken from the cache, and I can easily see that Estimated property showing 7,276, as well as the residual warning:




This is something which I could do on a customer box, looking in the cache for situations in problematic plans where the ratio of Estimated Number of Rows to be Read and Estimated Number of Rows isn't great. Potentially, someone could make a process that checked every plan in the cache, but it's not something that I've done.

Astute reading will have noticed that the Actual Rows that came out of this operator was 149, which was much smaller than the estimated 1382.56. But when I'm looking for Residual Predicates that are having to check too many rows, the ratio of 1,382.56 : 7,276 is still significant.

Now that we've found that this query is ineffective without even needing to run it, the way to fix it is to make sure that the Residual Predicate is sufficiently SARGable.

## The Query

```
SELECT COUNT(*)
FROM Sales.Orders
WHERE SalespersonPersonID = 7
AND OrderDate >= '20130401'
AND OrderDate <  '20130501';
```

…gives the same results, and doesn't have a Residual Predicate. In this situation, the Estimated Number of Rows to be Read value is identical to the Estimated Number of Rows, and the inefficiency is gone:

**Index Seek**
Scan a particular range of rows from a nonclustered index.

Node ID: 2
Physical Operation: Index Seek
Logical Operation: Index Seek
Actual Rows: 149
Actual Rows Read: 149
Estimated Rows: 534
Estimated Rows To Be Read: 534
Estimated I/O Cost: 0.0038657
Estimated CPU Cost: 0.0007442
Actual Executions: 1
Estimated Executions: 1.0
Estimated Operator Cost: 0.0046100 (93.5%)
Estimated Subtree Cost: 0.0046100
Estimated Row Size: 9 B
Actual Data Size: 1 KB
Estimated Data Size: 5 KB
Actual Rebinds: 0
Actual Rewinds: 0
Ordered: True

Database: [WideWorldImporters]
Table: [Sales].[Orders]
Index: [rf_Orders_SalesPeople_OrderDate]

**Seek Predicates:**
[WideWorldImporters].[Sales].[Orders].[OrderDate] >= CONVERT_IMPLICIT(date,[@2],0)
[WideWorldImporters].[Sales].[Orders].[OrderDate] < CONVERT_IMPLICIT(date,[@3],0)
[WideWorldImporters].[Sales].[Orders].[SalespersonPersonID] = CONVERT_IMPLICIT(int,[@1],0)

***- @rob_farley***

Aaron Bertrand

## A New Feature, You Say?

As SQL Server people first and foremost, we saw the pains often encountered by customers or colleagues troubleshooting queries and analyzing execution plans. We knew that many tuning efforts ultimately lead to more questions than answers; questions like...

- Was the chosen index the most optimal for this operation?
- How close was the chosen index to covering the query?
- What other indexes exist on this table?
- Would a missing or recommended index have fared better?
- Were estimates off because stats were out of date, or for a more elusive reason?
- Would different parameter values have yielded a different plan?
- What do the statistics histograms look like for the relevant columns?
- Do the parameter values help to indicate data skew or ascending key problems?

Our new Index Analysis tab can help answer these questions with much less manual legwork, enabling you to focus on your query's performance instead of wasting valuable time gathering facts and metadata.

## Where Do I Get It?

Before getting into any details, I should tell you how you can access this new feature, in case you want to follow along. Index Analysis has been made available first in SQL Sentry v10, but it will later be the flagship feature in our next version of Plan Explorer. If you haven't upgraded to v10 yet, you should check out Jason's post for a great overview, and even just browse the change list. Not a SQL Sentry customer (yet)? Take it for a spin today!

Currently, you need to generate a new actual or estimated plan in order to access the Index Analysis tab. Due to performance and storage overhead, it just wasn't feasible to capture all of this data with every single plan we capture, but we are investigating ways to collect this valuable information for you in the background, too. (In the new unified Plan Explorer, there also won't be any Index Analysis data if you open a plan generated in SSMS or a previous version of Plan Explorer, since that data wasn't collected at runtime.)

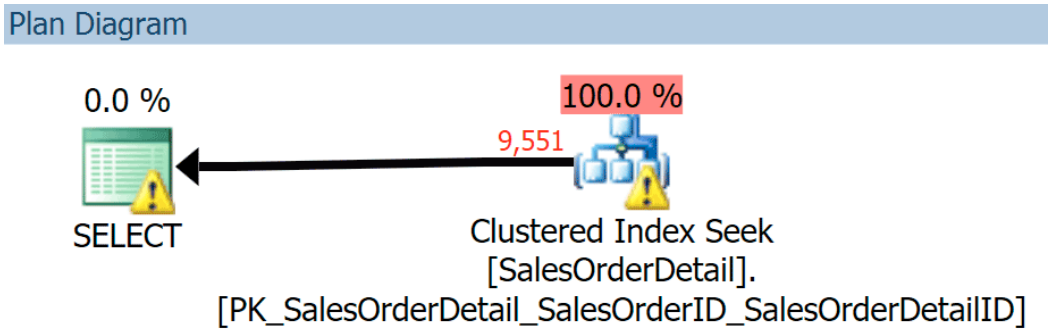
You can start a new integrated Plan Explorer session in the SQL Sentry Client by going to **File > New Plan Explorer Session.**

## What Does It Look Like?

This is just a quick example of the output in the Index Analysis tab. Using a pretty standard installation of the AdventureWorks sample database, I created the following stored procedure:

```
CREATE PROCEDURE dbo.IndexAnalysisDemo
  @MinProductID    int,
  @MaxProductID    int,
  @MinSalesOrderID int,
  @MaxSalesOrderID int
AS
BEGIN
   SET NOCOUNT ON;

  SELECT ProductID, SalesOrderID, SalesOrderDetailID,
         CarrierTrackingNumber, OrderQty, UnitPrice
  FROM Sales.SalesOrderDetail
  WHERE ProductID BETWEEN @MinProductID AND @MaxProductID
  AND SalesOrderID BETWEEN @MinSalesOrderID AND @MaxSalesOrderID;
END
GO
```

I experimented with calls to the procedure, choosing a variety of different values for each of the parameters. On several of the iterations, the following graphical plan was produced:



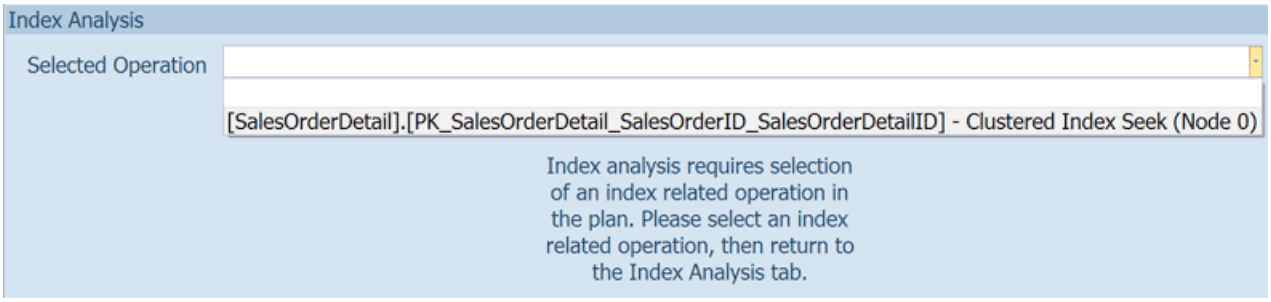

This looks like a pretty efficient plan; after all, it's a seek, right? Well, we can see by the red text for the row counts that our estimates are way off. And when the underlying SQL Server version supports it, our graphical plan now shows a handy warning for things like residual I/O – if you hover over to see the tooltip, you'll find this toward the bottom:

**Seek Predicates:**
[AdvWorks].[Sales].[SalesOrderDetail].[SalesOrderID] >= [@MinSalesOrderID]
[AdvWorks].[Sales].[SalesOrderDetail].[SalesOrderID] <= [@MaxSalesOrderID]

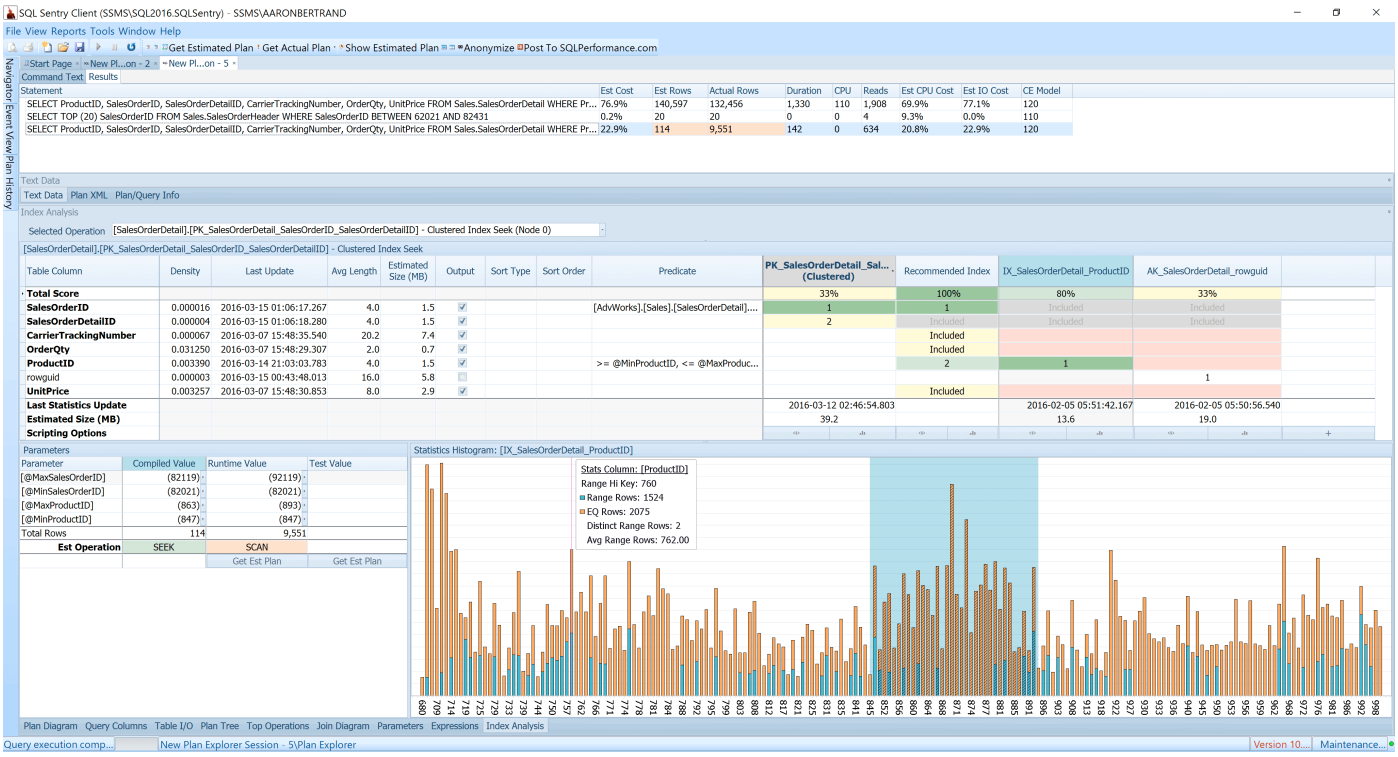
**Predicate:**
[AdvWorks].[Sales].[SalesOrderDetail].[ProductID]>=[@MinProductID]
AND [AdvWorks].[Sales].[SalesOrderDetail].[ProductID]<=[@MaxProductID]

**Warnings:**
Operation caused residual IO. The actual number of rows read was 46,007, but the number of rows returned was 9,551.

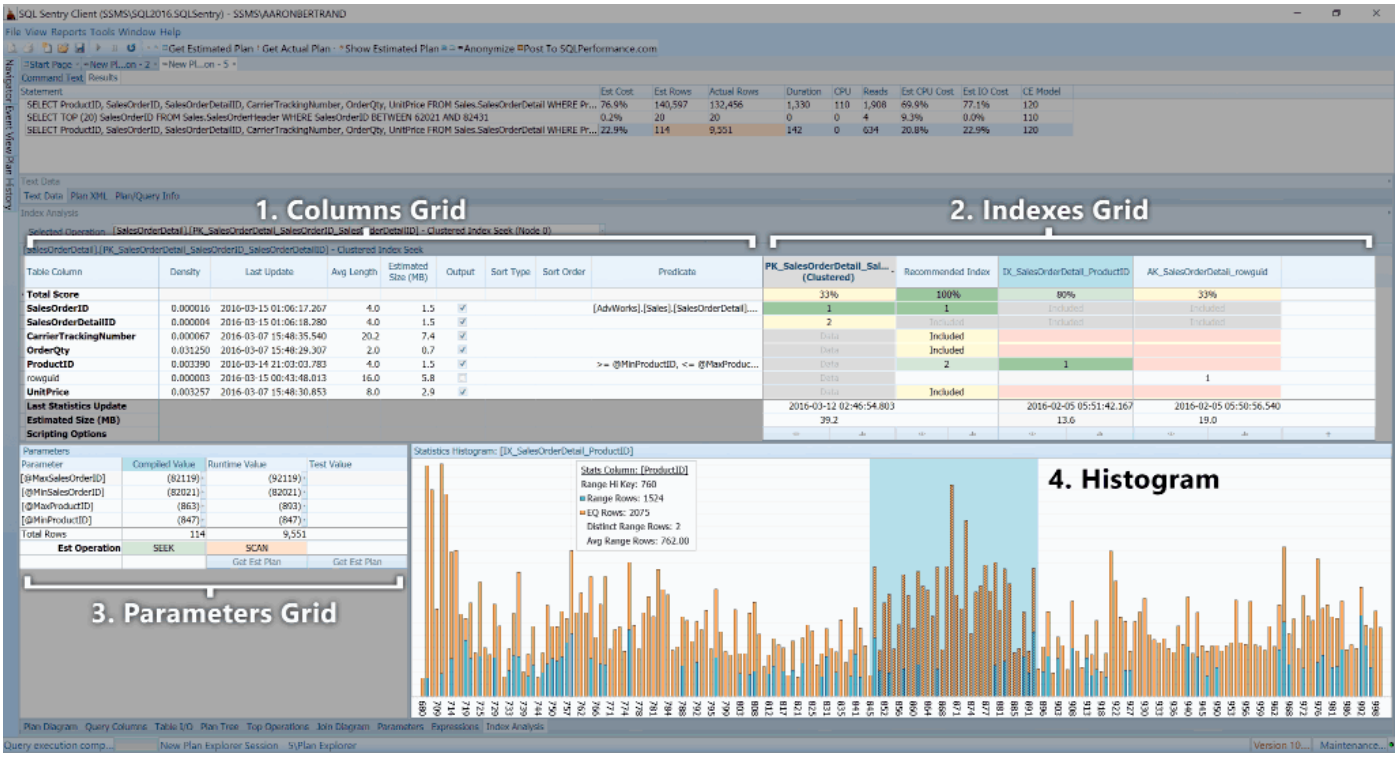
We'll treat this in more detail in a future post. For now, I started to wonder if my seek was such a good thing after all. Our new Index Analysis feature couldn't have come at a better time! I moved to the Index Analysis tab, and I saw this:




This is just telling me that I need to select a single index operation (Seek or Scan), either in the above drop-down or in another tab, in order to show relevant information in the Index Analysis tab (there is just way too much information to display multiple operations together). Like all of the grids in the integrated and standalone Plan Explorer product, this tab is also context sensitive, so its display depends on what might be selected on other tabs. I selected the clustered index seek, which happened to be the only relevant operation in this relatively simple plan, and the tab lit up.



It's a lot to take in on first glance, so let me first highlight the different functional areas on this tab, then I'll dig into each one.




## Columns Grid

Now I'll explain these four areas I've highlighted:

| Table Column | Density | Last Update | Avg Length | Estimated Size (MB) | Output | Sort Type | Sort Order | Predicate |
|---|---|---|---|---|---|---|---|---|
| **Total Score** | | | | | | | | |
| **SalesOrderID** | 0.000016 | 2016-03-15 01:06:17 | 4.0 | 1.5 | ☑ | | | [AdvWorks].[Sales].[S... |
| **SalesOrderDetailID** | 0.000004 | 2016-03-15 01:06:18 | 4.0 | 1.5 | ☑ | | | |
| **CarrierTrackingNumber** | 0.000067 | 2016-03-07 15:48:35 | 20.2 | 7.4 | ☑ | | | |
| **OrderQty** | 0.031250 | 2016-03-07 15:48:29 | 2.0 | 0.7 | ☑ | | | |
| **ProductID** | 0.003390 | 2016-03-14 21:03:03 | 4.0 | 1.5 | ☑ | | | >= @MinProductID, ... |
| rowguid | 0.000003 | 2016-03-15 00:43:48 | 16.0 | 5.8 | ☐ | | | |
| **UnitPrice** | 0.003257 | 2016-03-07 15:48:30 | 8.0 | 2.9 | ☑ | | | |

What I labeled as the Columns Grid and Indexes Grid are actually all part of the same grid control, but I thought I would explain them separately. The columns side of the grid shows all of the columns on the left side, with bold text for the columns used in the query in some way – as output columns, join criteria, or filters. Currently, this shows all columns involved with the query or at least one index, but in the future, we will show all columns in the table. When column-level statistics are available, we'll show details like density, the last update, and average length. (When not available, you'll see ? in their place.) We'll also compute the estimated size for you, and indicate whether the column is used for output. The next two data points, if the columns are involved in a sort of any kind, show sort direction (ASC or DESC) and ordinal position of the column within the sort. For example, if we ran with a bad idea and added the following to the above query:

```
ORDER BY SalesOrderID DESC, SalesOrderDetailID;
```

We would see data for those first two rows in the sort columns as follows:

| Table Column | Density | Sort Type | Sort Order |
|---|---|---|---|
| **SalesOrderID** | 0.000016 | DESC | 1 |
| **SalesOrderDetailID** | 0.000004 | ASC | 2 |
| **CarrierTrackingNumber** | 0.00006 | | |

The last column in this grid shows the predicate(s) used against each column, if any; this can include both join and filter conditions, and if more than one predicate exists, they will be comma-separated.

## Indexes Grid

| PK_SalesOrderDetail_Sal... (Clustered) | Recommended Index | IX_SalesOrderDetail_ProductID | AK_SalesOrderDetail_rowguid |
|---|---|---|---|
| 33% | 100% | 80% | 33% |
| 1 | 1 | Included | Included |
| 2 | Included | Included | Included |
| Data | Included | | |
| Data | Included | | |
| Data | 2 | 1 | |
| Data | | | 1 |
| Data | Included | | |
| 2016-03-12 02:46:54.803 | | 2016-02-05 05:51:42.167 | 2016-02-05 05:50:56.540 |
| 39.2 | | 13.6 | 19.0 |

Along the top you can see that there is a display column for each index (my screen shot doesn't show the column names; they're at the left of the entire grid). If there is a clustered index, it will be listed first, on the left.

Following that will be the index that was actually used for the currently selected operation (if it wasn't the clustered index). In either case, the selected index will have its name in bold.

Next, we'll show any missing indexes if they were suggested; in some cases, we will recommend an index, but we'll always defer to SQL Server if it provides a missing index suggestion (that wasn't the case here). Sometimes there won't be a missing index suggestion from SQL Server *or* a recommended index from us.

After that, we'll list all the other indexes on the table, ordered by score: highest on the left, lowest on the right.

***Wait, what is this score you speak of?***

I thought I would introduce that quietly to make sure you were still paying attention, even though this is really one of the defining concepts of this feature. :-) With a tremendous amount of input from Paul White (@SQL_Kiwi), we have developed an algorithm to score each index, based on a number of factors – density/selectivity, whether they cover the query, whether they support a seek, and a variety of others. The color behind the score is scaled in an intuitive way, even if you're not already familiar with our software: green is good, yellow is okay, red is bad. You may note that the only decent score here has been assigned to an index that doesn't even exist.

It is also important to note that a score of 100% is not always attainable, and that the scoring algorithm can change over time.

Beneath the score there are rows for each column in the table; therefore there is a cell for each column in each index. We attempt to relay a lot of information visually here, partly through color, and partly through data. The colors have the following meanings:

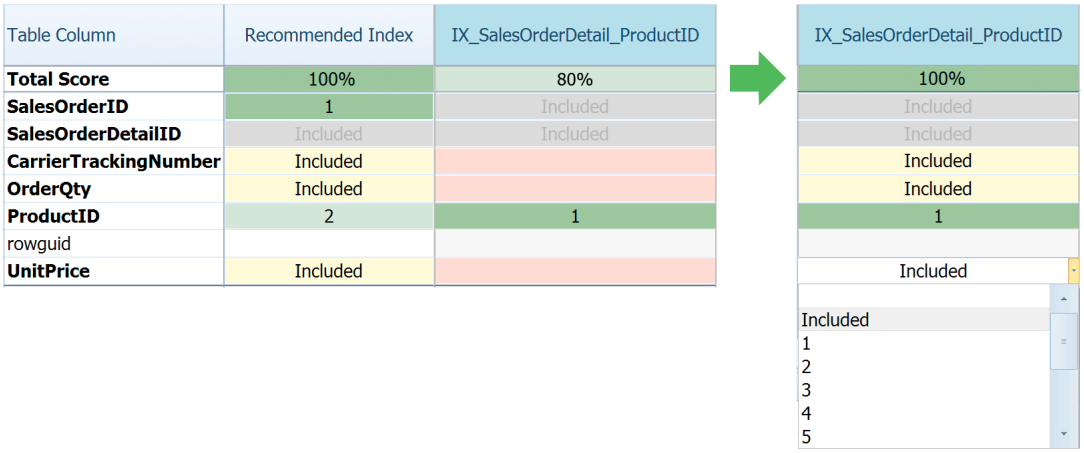
| Color | | Description |
|---|---|---|
| | Grey | The column is either part of the clustering key (so it is automatically included in all non-clustered indexes, and will be indicated as such with the word **Included**), or the column is part of the clustered index (in which case, under the clustered index, it will be indicated as such using the word **Data**). |
| | Green | The column is used in the query, the current index key covers the column, the ordinal position is favorable, and a seek is likely. The darker green usually means it's the leading column in the index **and** it has a predicate applied. |
| | Yellow | The column is used in the query and is either an `INCLUDE` column or is covered by the current index key but is either (a) not in the left-most subset or (b) not sorted in the desired order. |
| | Red | The column is required by the query, either as an output column, a filter, or part of an expression, but is not covered by either the key or the `INCLUDE` list for the current index. |
| | White | The column is not required by the query. |

Text in the cell could be a number, which indicates ordinal position within the index key. If the cell says "Included" then it is part of the INCLUDE list. If the cell is blank, then the column isn't part of the index.

Note that you can change that. In the indexes grid you can make hypothetical or real changes to existing indexes, test the feasibility of new indexes, or even update stats directly without ever leaving this tab.

## Improve Your Score

In any cell you can change the ordinal position of a column, move it to or from the INCLUDE list, or remove it from the index altogether (by choosing the blank option in the drop-down). Don't worry, these changes don't go in and muck with your indexes as you click around in real time; in fact that's not really the purpose. You can experiment with the estimated impact of changing existing indexes right here in the grid, to see if minor tweaks to those indexes could improve their score. As you make beneficial changes to an index, you will see its score at the top of the column change. You can then consider making actual changes to the database by scripting out a drop and re-create of an index with your changes included. For example, let's say I wanted to "improve" the IX_SalesOrderDetail_ProductID index (currently scored at 80%) by including the three other columns required by the query. This improves the index score to 100%, as you can see to the right (from a separate screen shot of course):



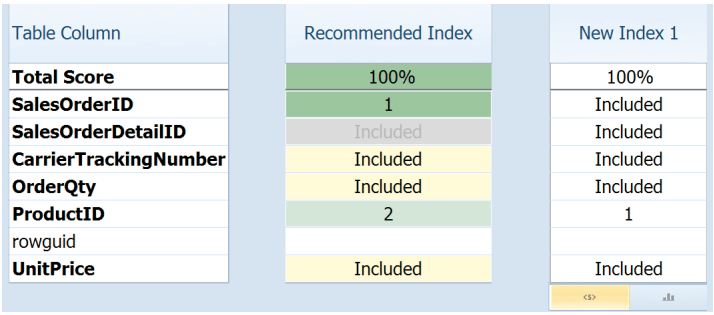
| Table Column | Recommended Index | IX_SalesOrderDetail_ProductID |
|---|---|---|
| **Total Score** | 100% | 80% |
| **SalesOrderID** | 1 | Included |
| **SalesOrderDetailID** | Included | Included |
| **CarrierTrackingNumber** | Included | |
| **OrderQty** | Included | |
| **ProductID** | 2 | 1 |
| rowguid | | |
| **UnitPrice** | Included | |

| IX_SalesOrderDetail_ProductID |
|---|
| 100% |
| Included |
| Included |
| Included |
| Included |
| 1 |
| |
| Included |

Included
1
2
3
4
5

## Create New Index with Better Score

The last index is entirely blank except for a + button at the bottom; this column is here so you can create your own index. In this case, there is not much point – it's not possible to do better than the index SQL Sentry has already recommended. But still, for the sake of experimentation, let's hit the + button and create a new index that mimics the recommended index to some degree, with a different key column. Here it is easy to obtain that 100% score as well, and I can hit the <s> button at the bottom of the column in order to generate a script for the new index:

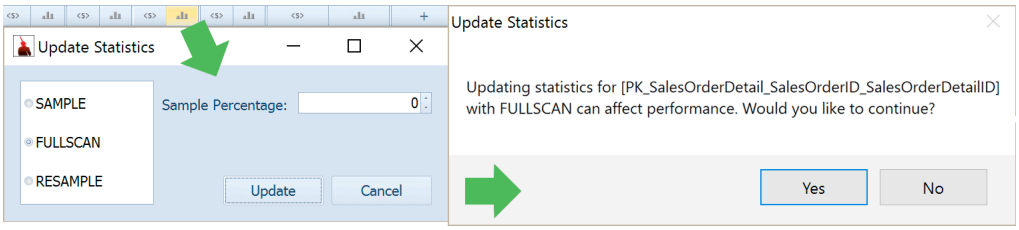


| Table Column | Recommended Index | New Index 1 |
|---|---|---|
| **Total Score** | 100% | 100% |
| **SalesOrderID** | 1 | Included |
| **SalesOrderDetailID** | Included | Included |
| **CarrierTrackingNumber** | Included | Included |
| **OrderQty** | Included | Included |
| **ProductID** | 2 | 1 |
| rowguid | | |
| **UnitPrice** | Included | Included |

Once I've scripted the index, I can make changes to it, and I can either copy it from this dialog to run elsewhere, or I can immediately execute it. Note that I can use the same <s> button at the bottom of any index in the grid to inspect or directly change the index after I've made any changes.

## Update Stats Directly

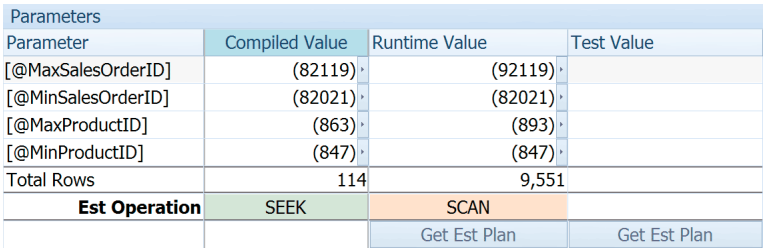
There is also an Update Statistics button, which looks like this: ( ). Under any index I can choose to manually update statistics if I notice that the last stats update was a long time ago, or if I know about recent underlying data change that might not get picked up by auto-update statistics. Depending on the options you choose, you may get a warning about performance impact:



At the bottom of the grid, above the buttons, are two other pieces of information: when the index statistics were last updated, and the estimated size of the index.

Between the ability to update statistics and the immediate feedback of the index score, this can be a very powerful way to gauge the potential impact of new or modified indexes. This can effectively serve as a sandboxing environment where you can consider the effect of these changes without actually having to create new or hypothetical indexes (or touch the underlying instance of SQL Server at all). The exercise can also help you identify the most expensive columns in your index (which prevent you from getting a better score), and consider removing those from the query; or the least selective columns in your index (which can guide you in moving them away from the key).

## Parameters Grid

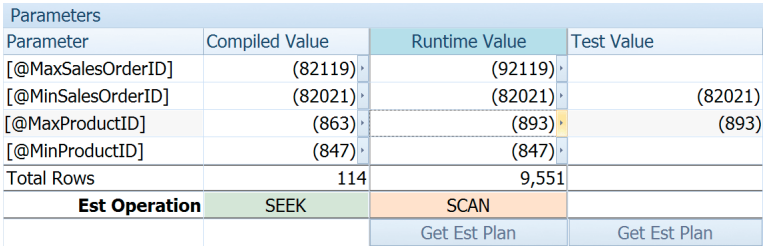


Parameters

| Parameter | Compiled Value | Runtime Value | Test Value |
|---|---|---|---|
| [@MaxSalesOrderID] | (82119) | (92119) | |
| [@MinSalesOrderID] | (82021) | (82021) | |
| [@MaxProductID] | (863) | (893) | |
| [@MinProductID] | (847) | (847) | |
| Total Rows | 114 | 9,551 | |
| **Est Operation** | SEEK | SCAN | |
| | | Get Est Plan | Get Est Plan |

If you're already familiar with Plan Explorer, this grid will look a little familiar, as it will show the compiled and runtime values of every parameter. But there is some extra functionality here as well, including the total number of rows predicted, and the estimated operation that will take place with those values. In some cases, the compiled parameters may expect to yield a seek, while the runtime parameters may expect to yield a scan, or vice versa, due to factors such as "the tipping point." Note that these are educated guesses, not guarantees of what will happen when a plan gets generated – in fact when the values fall within the grey area of the tipping point, we'll place a ? there instead.

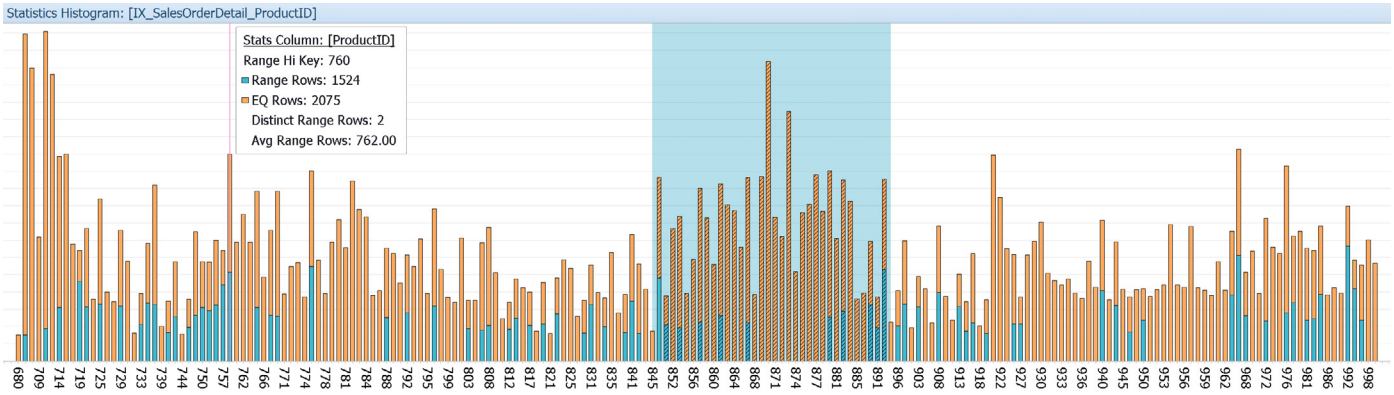
There is a third column called "Test Values" which, you may have guessed, allows you to test completely different parameter values, and generate a new estimated execution plan (this will refresh the entire session with the new, estimated plan). If you only want to change a single parameter value, you can populate the rest with either the compiled or runtime value by clicking the small arrow next to each value:



Parameters

| Parameter | Compiled Value | Runtime Value | Test Value |
|---|---|---|---|
| [@MaxSalesOrderID] | (82119) | (92119) | |
| [@MinSalesOrderID] | (82021) | (82021) | (82021) |
| [@MaxProductID] | (863) | (893) | (893) |
| [@MinProductID] | (847) | (847) | |
| Total Rows | 114 | 9,551 | |
| **Est Operation** | SEEK | SCAN | |
| | | Get Est Plan | Get Est Plan |

## Histogram




Finally, the Histogram allows you to visualize data skew and easily identify potential problems with the data distribution for the leading key column of the selected index. The above shows the distribution of values for the ProductID column, and you can see how the runtime parameter values are shown through a range overlay.

Handy tooltips show you all the information you're used to parsing endlessly from the output of various DBCC commands and DMVs. You will be able to easily see when a NULL or other token value represents a large portion of your data set, and know at a glance which parameter values you should be testing for the best chance at capturing parameter sniffing, ascending key, or other plan variation problems.

## Conclusion

This is a value-packed feature, and in upcoming posts, I plan to dig deeper and give you more detailed and practical examples to work from. And as a reminder, all of this functionality will be available in a future version of Plan Explorer.

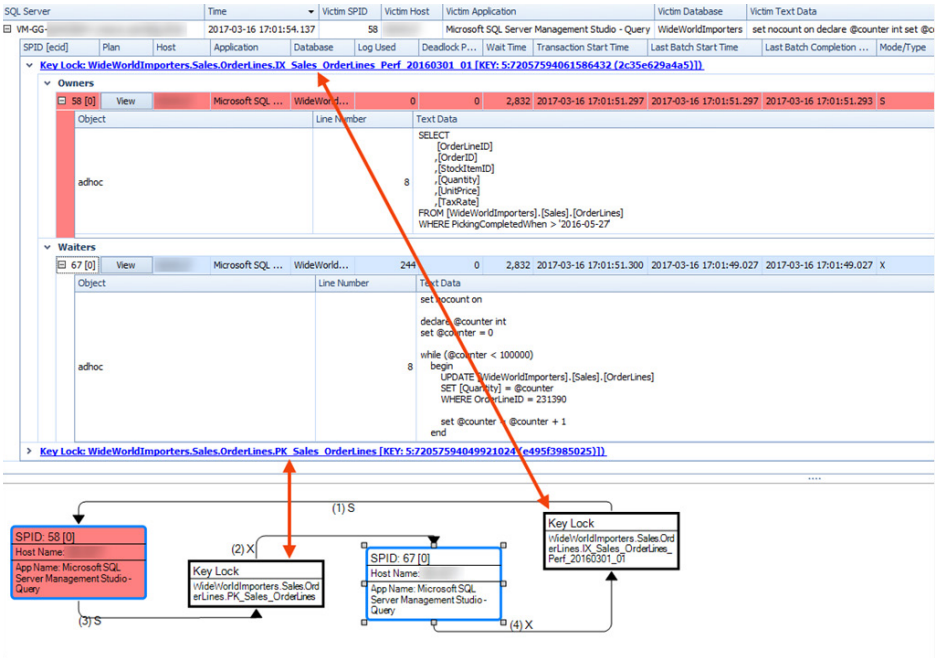
Greg Gonzalez

One of the sleeper features of our free Plan Explorer is its ability to open and view SQL Server deadlocks. The view is designed to clearly present all of the details you need to troubleshoot a deadlock, without overwhelming you with less helpful elements like owner IDs and transaction descriptors. The deadlock diagram is synchronized with a treeview showing query text, call stack, lock details, owners & waiters, and more. Clicking on any process node on the diagram instantly takes you to the associated query, and sequence indicators on the connectors let you quickly ascertain the order of events that led to the deadlock.

The view is mostly the same between the standalone Plan Explorer and the integrated Plan Explorer in the full SQL Sentry software. The primary difference is that SQL Sentry automatically captures the deadlocks and associated queries and query plans for you, and lets you jump directly into the query plan for further analysis, index updates, etc. With standalone Plan Explorer you must capture the deadlock xml via other means, and then you can open the .xdl file manually.

I would estimate that at least 2/3 of the deadlocks I have run across in my career working with SQL Server involve key lookups. They seem to be everywhere on busy OLTP systems, and are most common when SELECTs with key lookups are regularly operating within the same range of rows as many UPDATEs and DELETEs. The good news is that they are often one of the easier deadlocks to identify and resolve.

If you open a deadlock with Plan Explorer, the telltale sign a key lookup is involved is a key lock against a clustered index, and a key lock against a non-clustered index. These are visible on both the diagram and treeview as shown below.




With integrated Plan Explorer, you can quickly confirm by clicking the "View" button on the non-clustered index row (shown above) and it will open the query plan captured automatically when the deadlock occurred. If no plan was captured, it will auto-request an estimated plan. With standalone Plan Explorer, simply copy the query text via right-click Copy -> Cell, and paste it into a new session tab and request the estimated or actual query plan. If it turns out that the SELECT doesn't use a key lookup but rather a clustered index scan, it's likely because statistics and/or the cached plan have changed, and the optimizer now thinks a scan would be more efficient. (For more details on how and why this happens, see Kimberly Tripp's tipping point series).

## Let's Try it Out

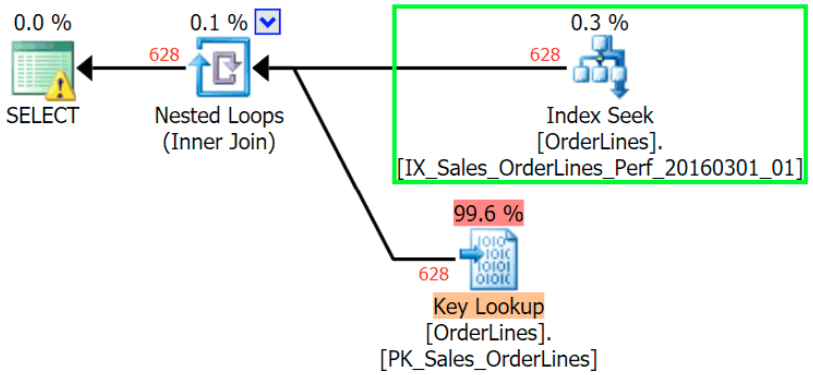
To create a simulated deadlock, I'll use two queries against the Wide World Importers database, a SELECT and an UPDATE, both looped to simulate heavy activity. If you run these in separate tabs in SSMS, a deadlock should result within a few seconds.

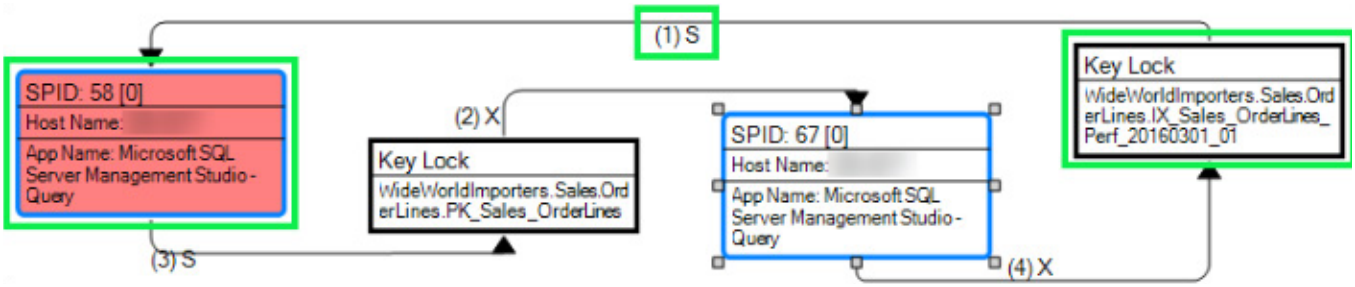
```
-- Run in SSMS session 1:

USE WideWorldImporters;
SET NOCOUNT ON;

declare @counter int = 0;
while (@counter < 100000)
        begin
                UPDATE [WideWorldImporters].[Sales].[OrderLines]
                SET [Quantity] = @counter
                WHERE OrderLineID = 231390;
                set @counter = @counter + 1;
        end

---------------------------
-- Run in SSMS session 2:

USE WideWorldImporters;
SET NOCOUNT ON;

declare @counter int = 0;
while (@counter < 100000)
        begin
                SELECT
                        [OrderLineID]
                       ,[OrderID]
                       ,[StockItemID]
                       ,[Quantity]
                       ,[UnitPrice]
                       ,[TaxRate]
                FROM [WideWorldImporters].[Sales].[OrderLines]
                WHERE PickingCompletedWhen > '2016-05-31';
                set @counter = @counter + 1;
        end
```

## Visualize The Problem

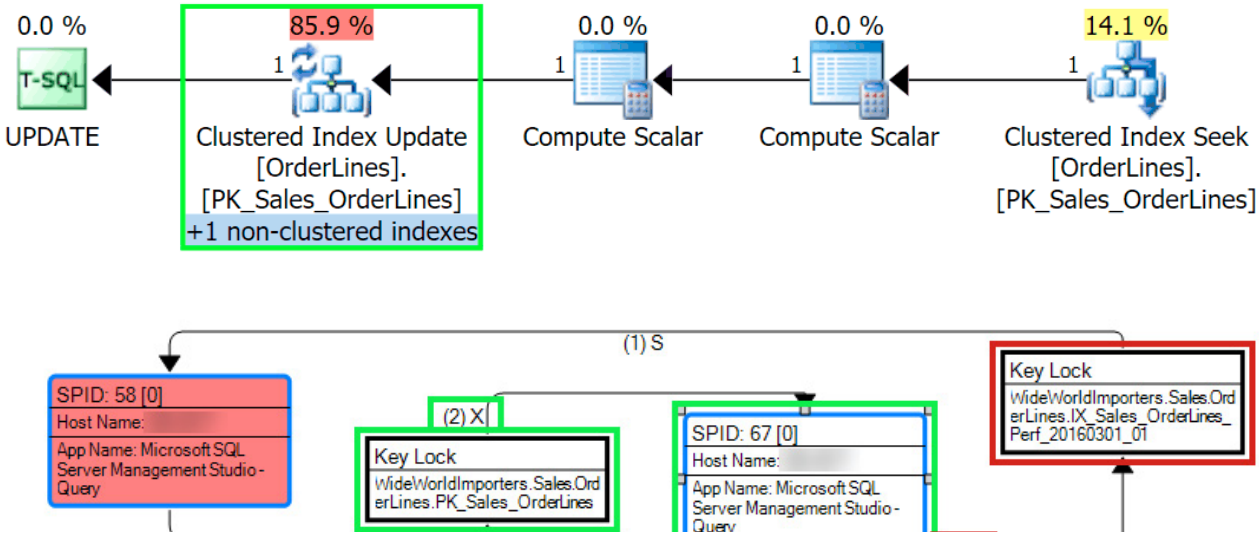
I will walk through the sequence of events leading to the deadlock, using Plan Explorer's deadlock and plan diagrams to illustrate each step. The nodes outlined in green (or red) are those that are relevant for that step, and the number in parens on each deadlock connector line represents the order in which the lock was acquired.

1. The SELECT retrieves data from the non-clustered index via a Seek. A shared lock is taken on one or more pages.

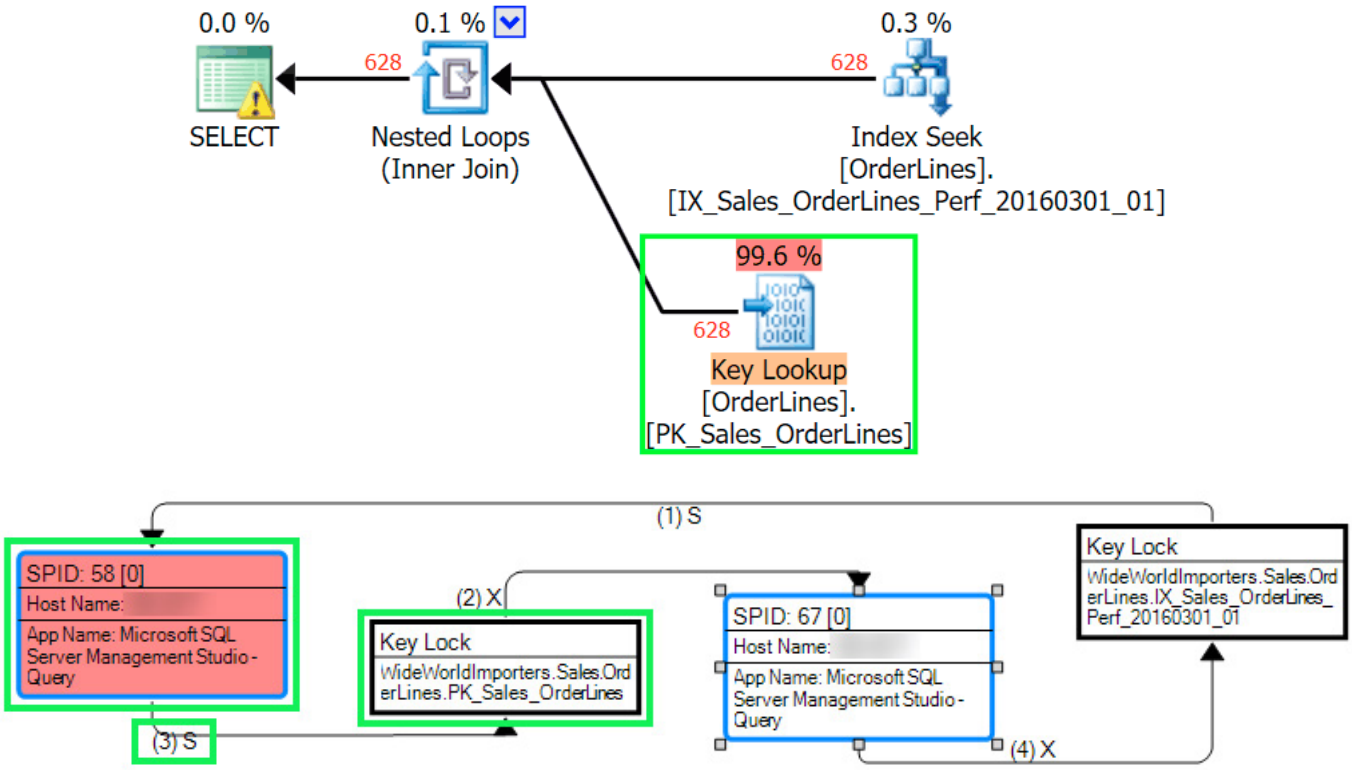


2. The UPDATE takes an exclusive lock on the clustered index row. Because at least one of the columns in the non-clustered index is being updated, an exclusive lock is immediately attempted on the non-clustered index row, but it can't be acquired because of the shared lock owned by the SELECT.

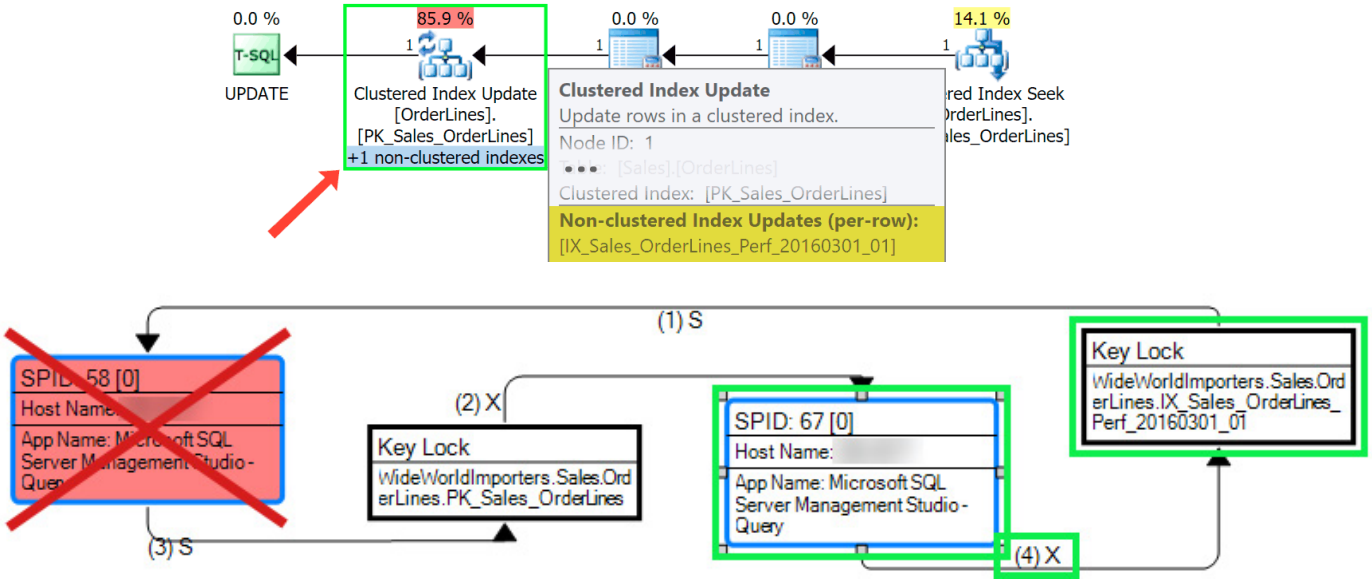



3. The Key Lookup tries to acquire a shared lock on the same same row in the clustered key currently locked by the UPDATE, but it can't be granted because of its exclusive lock.




At this point a stalemate exists until the lock monitor thread detects the deadlock, usually within 5 seconds. Inevitably the SELECT will be the loser because the UPDATE will have done more work. (The amount of work done by each process is shown in the Log Used column in the treeview.)

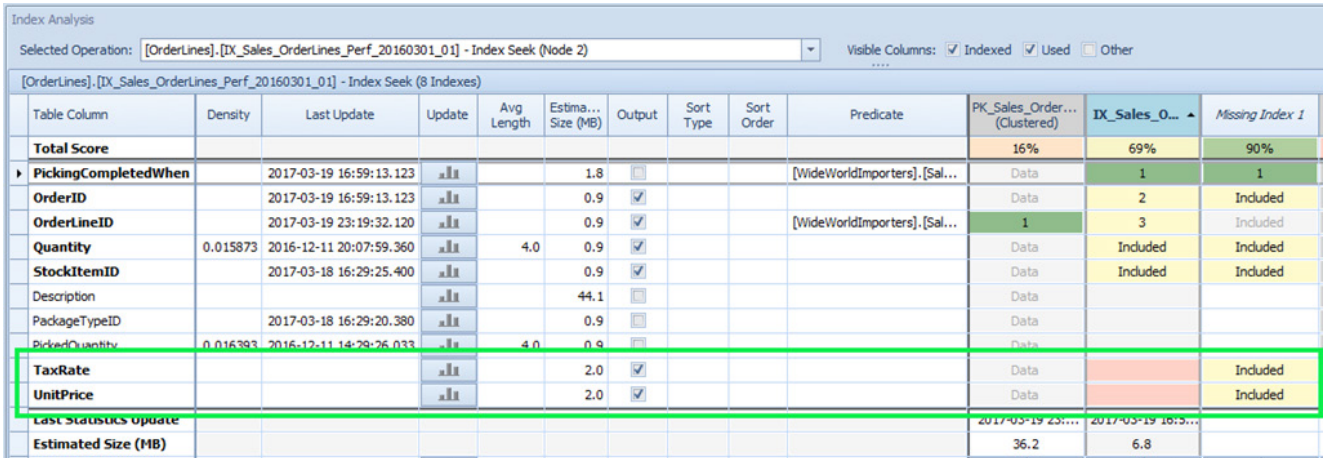
4. Once the SELECT thread has been chosen as the deadlock victim and terminated, the UPDATE can successfully acquire the exclusive lock on the non-clustered row and update it.




In the plan diagram above, note that the number of non-clustered index updates associated with an UPDATE or DELETE is always highlighted in blue below the Clustered Index Update operator, with the list of affected index names shown in the tooltip.

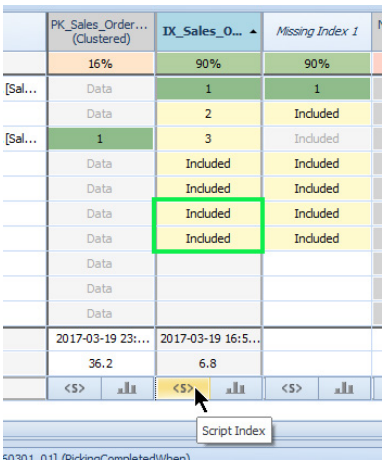
## Possible Solutions

The key lookup happens because the non-clustered index doesn't include, or cover, all of the columns used by the query. This is apparent on the Index Analysis tab – note the light red cells for Tax Rate and Unit Price shown below.




| Table Column | Density | Last Update | Update | Avg Length | Estima... Size (MB) | Output | Sort Type | Sort Order | Predicate | PK_Sales_Order... (Clustered) | IX_Sales_O... | Missing Index 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Total Score** | | | | | | | | | | 16% | 69% | 90% |
| **PickingCompletedWhen** | | 2017-03-19 16:59:13.123 | | | 1.8 | | | | [WideWorldImporters].[Sal... | Data | 1 | 1 |
| **OrderID** | | 2017-03-19 16:59:13.123 | | | 0.9 | | | | | Data | 2 | Included |
| **OrderLineID** | | 2017-03-19 23:19:32.120 | | | 0.9 | | | | [WideWorldImporters].[Sal... | 1 | 3 | Included |
| **Quantity** | 0.015873 | 2016-12-11 20:07:59.360 | | 4.0 | 0.9 | | | | | Data | Included | Included |
| **StockItemID** | | 2017-03-18 16:29:25.400 | | | 0.9 | | | | | Data | Included | Included |
| Description | | | | | 44.1 | | | | | Data | | |
| PackageTypeID | | 2017-03-18 16:29:20.380 | | | 0.9 | | | | | Data | | |
| PickedQuantity | 0.016393 | 2016-12-11 14:29:26.033 | | 4.0 | 0.9 | | | | | Data | | |
| **TaxRate** | | | | | 2.0 | | | | | Data | | Included |
| **UnitPrice** | | | | | 2.0 | | | | | Data | | Included |
| **Last Statistics Update** | | | | | | | | | | 2017-03-19 23:... | 2017-03-19 16:5... | |
| **Estimated Size (MB)** | | | | | | | | | | 36.2 | 6.8 | |

## Include The Columns




The easy solution is to add these as included columns to the existing index, which will eliminate the lookup. I can do this by selecting the dropdowns for both red cells and selecting the "Included" option, then clicking the Script Index (<S>) button (see image at right).

We also show a missing index recommendation found by the optimizer, but there's no point in creating an entirely new index when only these two columns are needed.

NOTE: Before adding included columns, you should first consider the number of rows in the index, the total size of the columns, and the level of DML activity on the table to determine whether the additional overhead of a larger index will be justified.

## Force a Scan

If adjusting the non-clustered index isn't a viable option, and the clustered index is small enough where it will easily fit entirely in buffer (maybe a few thousand pages max), a FORCESCAN hint can be used with the SELECT to force a clustered index scan.

If you don't control the SQL, such as when it is being sent by a 3rd party app, a plan guide can be used to apply the hint. When using QDS (Query Data Store) on SQL Server 2016+, the scan plan can be "forced" instead. You should of course adequately test to ensure that shifting those lookups over to scans isn't going to cause a significant performance hit.

## Use RCSI

Another option would be to enable RCSI (Read Committed Snapshot Isolation), which will effectively prevent the blocking that causes these deadlocks. I would not recommend RCSI for resolving key lookup deadlocks alone, but rather more as a possibility to consider for systems suffering from heavy reader-writer contention in general. This is not something to be done lightly, so if you're not familiar with RCSI, I'd encourage you to read this post by Paul White, and this one by Kendra Little, before making such a move.

## Summing It Up

Hopefully by now you are excited to squash some key lookup deadlocks. There are two ways get started with SentryOne software:

- Download the free Plan Explorer and open any deadlock .xdl file.
- Download the SentryOne trial and start monitoring your SQL Servers with SQL Sentry. You'll be alerted via email whenever a deadlock occurs, and you can simply click the link in the email to jump directly to the deadlock.

***Happy Hunting!***

Aaron Bertrand

Last month we released the new, all-free, Plan Explorer 3.0, and now seemed the right time to follow that up with the 3rd version of the demo kit, too. Please consider earlier versions of the demo kit deprecated; I won't remove them, but I will add a note to those pages soon.

Now, you might ask, "What is a demo kit?" We came up with this concept back in 2011, after being asked by multiple presenters to help them share Plan Explorer with their own audiences.

One purpose is to serve as a starting point to build your own presentations about Plan Explorer specifically, or even about query tuning in SQL Server in general.

The other purpose I see is to act as a training aid – haven't used Plan Explorer before? Don't feel you've taken advantage of all of the features? The demos here are designed to showcase most of the features, and to demonstrate how we make execution plan problems much more obvious and easier to solve. We hope it makes you more productive with the software. Click here to visit our website and download Plan Explorer for free.

## Before You Get Started

Make sure you have the most recent version of Plan Explorer installed – you can always download it from our web site.

The session and other files can be opened directly in Plan Explorer and, for the most part, used without ever connecting to a database. But I also decided as a part of this update to abandon plans that used various versions of AdventureWorks or our own databases, and use the sample database Microsoft is committing to going forward: WideWorldImporters.

I went to great lengths to make a copy of that database that can be restored on any edition and on any version, from SQL Server 2008 Express all the way up to SQL Server 2016. Ripping out features added after 2008 turned out to be a much more complex undertaking than I thought; I'll blog about that separately, and update this space with links. Without going into a bunch of detail of what's missing from the full version of the database, just think of it as roughly the same schema and data, but without modern or Enterprise features. I am trying to get it (or something very close) incorporated into the official GitHub repo and, again, will update this space when that happens.

You'll need this backup restored to an instance of SQL Server 2008 or greater, any edition, if you want to run any of the queries interactively. You are free to run these same queries against the official WideWorldImporters database, and they should "work" in all cases, however you will not necessarily see the same plans (for example, you may see ColumnStore operations if you are using 2016 Enterprise, but not if you have a different edition, and hints to block those would fail in lower versions).

Here is a RESTORE command you can modify for your environment:

```
RESTORE DATABASE WideWorldImporters_Legacy
  FROM DISK = N'<location of BAK file here>' WITH REPLACE, RECOVERY,
  MOVE N'WWI_Legacy_Data' TO N'<location of data files here>\WWI_Legacy.mdf',
  MOVE N'WWI_Legacy_Log'  TO N'<location of log files here>\WWI_Legacy.ldf';
```

Note that when you run the queries interactively, you'll need to go to Edit > Connection and change GORDIE\SQL2008 to whatever you use to connect to your instance, as my local development connection details are the ones that are stored with the session file. If you change the name of the database, you'll need to modify that information in the connection dialog as well.

## PEDemo.pesession

This .pesession file can be opened in Plan Explorer, and you can move to the different steps of the demo using the History window (each step in the demo is a "version"). There are comments to help identify which version in the History corresponds to which step in the demo below. Note that I ran through these demos multiple times, so the numbers you see for very precise metrics like duration or CPU might not match exactly between the session file you download and the screen shot I attached to the blog post. Also note that if you generate new actual or estimated plans, you will change the History window too, so make sure to save off a backup of the .pesession file before you start, or save it as a new .pesession file after you've opened it.

History

| Version | Type | Comments |
|---|---|---|
| 1 | A | Reinitialize (restore original indexes, clear caches) |
| 2 | A | Initial query with lookups |
| 3 | A | Index creation & re-query |
| 4 | A | Histogram |
| 5 | E | - estimated for different histogram range |
| 6 | E | - estimated for out of range warning |
| 7 | A | Join diagram & missing indexes |
| 8 | A | Live query capture and replay |
| 10 | A | Reinitialize again |

You can move to any History Version # referenced below simply by clicking on that row in the History window.

## History V1- Reinitialize

This entry isn't used for anything plan- or demo-related, but simply to reset the system to the beginning of your demo – it clears the procedure cache and buffer pool, and changes indexes back to how they started. It uses this procedure.

```
CREATE PROCEDURE PlanExplorerDemo.Reinitialize
AS
BEGIN;
  SET NOCOUNT ON;

  CREATE INDEX [FK_Sales_InvoiceLines_InvoiceID]
    ON [Sales].[InvoiceLines]([InvoiceID] ASC)
        WITH (DROP_EXISTING = ON)

  CREATE INDEX [FK_Sales_Invoices_CustomerID]
    ON [Sales].[Invoices]([CustomerID] ASC)
    WITH (DROP_EXISTING = ON);

  DECLARE @db int;
  SET @db = DB_ID();
  DBCC FLUSHPROCINDB(@db) WITH NO_INFOMSGS;

  DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;
END;
GO
```

## HV2 - Initial Query

Virtually any demo can be used to show the better use of screen real estate, the plan layout controls, the use of color to highlight problems, how we point out discrepancies between estimated and actual row counts, and the ability to view costs by I/O, CPU, or both.

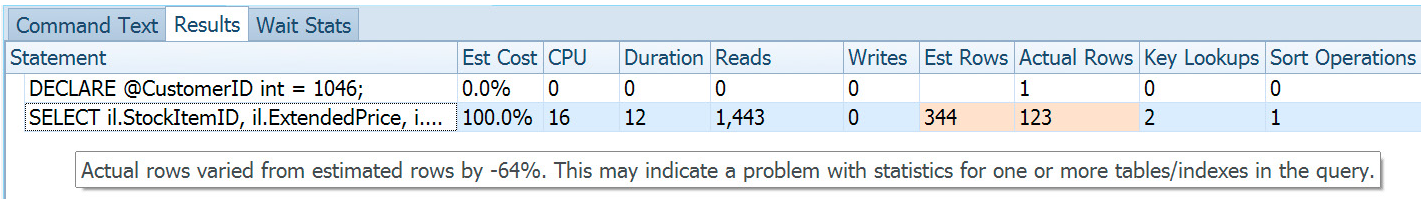
In the first demo, we run a query that attempts a join between Invoices and InvoiceLines, intentionally constructed to yield two key lookups and a sort:

```
DECLARE @CustomerID int = 1046;

SELECT
     il.StockItemID, il.ExtendedPrice,
     i.CustomerID, i.InvoiceID, 0 - il.Quantity, i.InvoiceDate
  FROM Sales.InvoiceLines AS il
  INNER JOIN Sales.Invoices AS i
  ON il.InvoiceID = i.InvoiceID
  WHERE i.CustomerID = @CustomerID
  ORDER BY il.InvoiceID, il.StockItemID;
```

After running this query (or moving to item #2 in the session file), you can show the information in the statement grid that you won't get from Management Studio by default, such as CPU, Duration, and Reads. You can also show that we highlight the difference between estimated rows and actual rows when they differ by a large enough percent. This discrepancy can be caused by a variety of things, including:

Out-of-date statistics – usually the most common cause is that the current statistics do not reflect the actual data. You can resolve this using UPDATE STATISTICS (and you can see more information about statistics on the Index Analysis tab, described later).
No histogram – perhaps there are no stats already and auto-create statistics is disabled, or columns are not in the leading key of the index, the optimizer may need to use density or average distribution information instead. This can lead to very vague, ballpark guesses.
Sheer complexity – in some cases the optimizer just has too much work to do trying to determine estimates, for example if filtering is occurring against many columns or using many predicates.
Statistics discrepancies are important to note at the statement level, but they are important at the individual operator level as well. We highlight these discrepancies in red on the plan diagram for an actual plan, so you can see exactly where incorrect estimates are happening. These can cascade from one operator throughout an entire plan, causing the optimizer to make sub-optimal decisions.

The grid also includes certain columns dynamically, only when they're relevant (in this case, Key Lookups and Sort Operations). You can show the ability to add other columns at will by right-clicking the column header, choosing Column Chooser, and dragging any column onto the grid. The metrics you see should look something like this in this case:

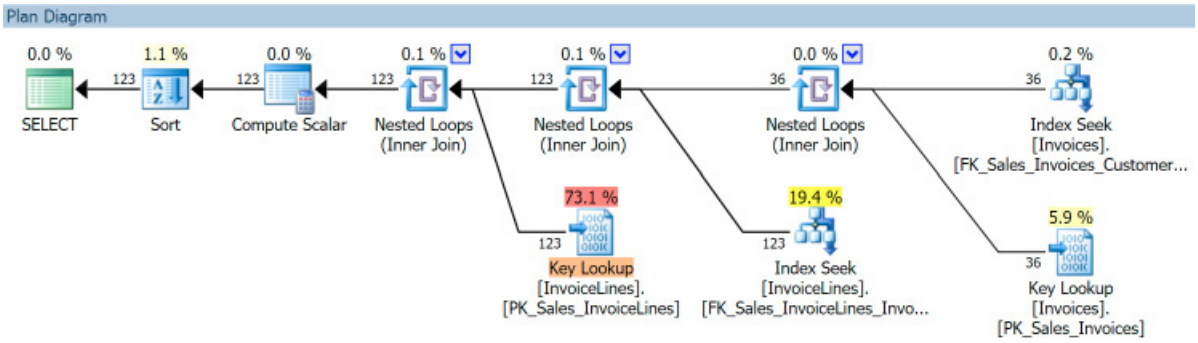

Command Text | Results | Wait Stats

| Statement | Est Cost | CPU | Duration | Reads | Writes | Est Rows | Actual Rows | Key Lookups | Sort Operations |
|---|---|---|---|---|---|---|---|---|---|
| DECLARE @CustomerID int = 1046; | 0.0% | 0 | 0 | 0 | 0 | | 1 | 0 | 0 |
| SELECT il.StockItemID, il.ExtendedPrice, i.... | 100.0% | 16 | 12 | 1,443 | 0 | 344 | 123 | 2 | 1 |

Actual rows varied from estimated rows by -64%. This may indicate a problem with statistics for one or more tables/indexes in the query.

Moving to the Plan Diagram, you can show that color is used to indicate problem areas. There are two Key Lookups in this plan, but one clearly has an overwhelming estimated cost.

You can stress that these costs are estimated and may not reflect reality at all (something we'll see in a different plan below), but they can still be used as a rough guideline on where to focus.




The Plan Tree and Top Operations tabs have some useful information (for example, the Actual Executions column shows part of the reason the highlighted Key Lookup is so expensive). They show essentially the same information; the Plan Tree fixes the grid into a structure like you would see in SHOWPLAN_TEXT, while Top Operations presents a grid sortable by any column.

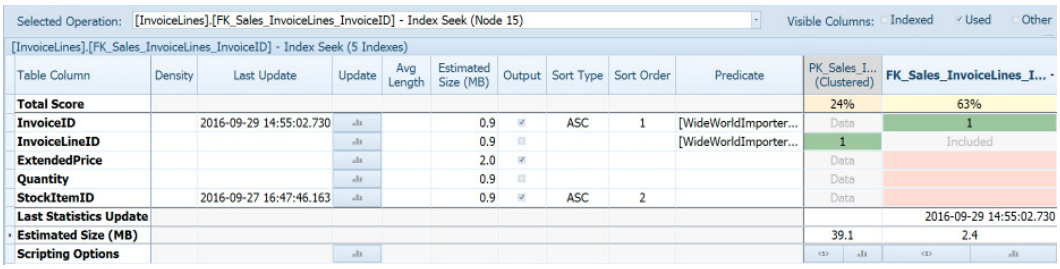
If you switch to the Query Columns tab, you can show how we highlight the columns that made the more expensive Key Lookup necessary. This is to provide some evidence that changing the index to either have these columns in the key or include list will probably help the performance of this query.

Query Columns

| Database | Schema | Table | Column | Operation | Index | Actual Rows | Est Rows |
|---|---|---|---|---|---|---|---|
| [WideWorldImporters_Legacy] | [Sales] | [InvoiceLines] | InvoiceID | Sort | | 123 | 344 |
| [WideWorldImporters_Legacy] | [Sales] | [InvoiceLines] | StockItemID | Sort | | 123 | 344 |
| [WideWorldImporters_Legacy] | [Sales] | [InvoiceLines] | InvoiceLineID | Index Seek | [FK_Sales_InvoiceLines_InvoiceID] | 123 | 344 |
| [WideWorldImporters_Legacy] | [Sales] | [InvoiceLines] | InvoiceID | Index Seek | [FK_Sales_InvoiceLines_InvoiceID] | 123 | 344 |
| [WideWorldImporters_Legacy] | [Sales] | [InvoiceLines] | StockItemID | Key Lookup | [PK_Sales_InvoiceLines] | 123 | 344 |
| [WideWorldImporters_Legacy] | [Sales] | [InvoiceLines] | Quantity | Key Lookup | [PK_Sales_InvoiceLines] | 123 | 344 |
| [WideWorldImporters_Legacy] | [Sales] | [InvoiceLines] | ExtendedPrice | Key Lookup | [PK_Sales_InvoiceLines] | 123 | 344 |
| [WideWorldImporters_Legacy] | [Sales] | [Invoices] | InvoiceID | Index Seek | [FK_Sales_Invoices_CustomerID] | 36 | 106 |
| [WideWorldImporters_Legacy] | [Sales] | [Invoices] | CustomerID | Index Seek | [FK_Sales_Invoices_CustomerID] | 36 | 106 |
| [WideWorldImporters_Legacy] | [Sales] | [Invoices] | InvoiceDate | Key Lookup | [PK_Sales_Invoices] | 36 | 106 |

Note that when I'm talking about this, I always stress the importance of considering both sides of the workload – while making an index wider can help this one specific query, it might not be run enough to justify the change, and the change may not benefit any other queries, either. Most importantly, changing the index can create a lot more work for all of your write queries, too. (I talk about some of the decision factors, which are relevant both for creating new indexes and changing existing indexes, in the post, Don't just blindly create those "missing" indexes!).

A new feature in Plan Explorer 3.0 is Index Analysis, designed to vastly improve the way you look at queries and consider index improvements. Let's move to that tab and see what we have. The problematic Key Lookup is the one associated with the Index Seek on InvoiceLines, so let's select that operation (Node 15) in the Selected Operation list, and uncheck Other and Indexed from Visible Columns so we can focus on the columns used in this query. You should see something very close to this:

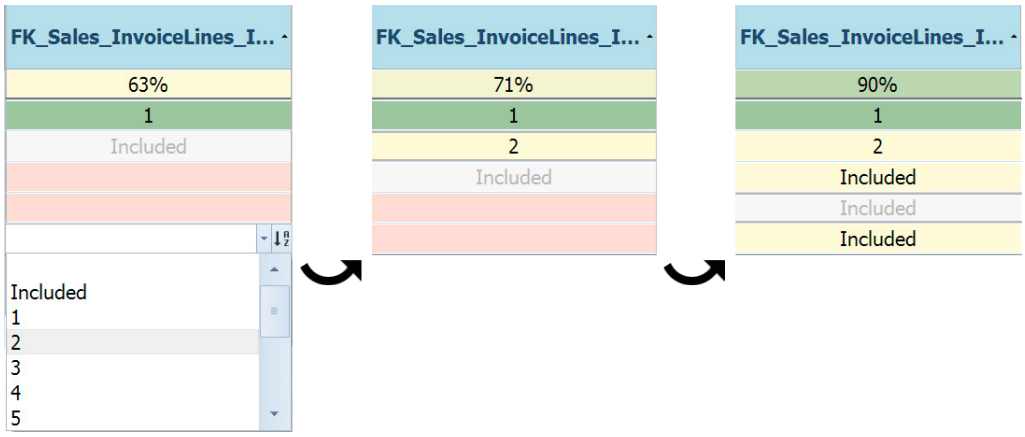


There's a lot of detail here – essentially on the left you have information about the columns, including statistics, estimated size, and whether they are sorted or are included in a predicate.

On the right, you have all of the indexes on the table, starting with the clustered index (if there is one). Next will be the index selected for this specific operation (though sometimes that will be the clustered index). After that, though clipped from the screen shot above, you will see the rest of the indexes on the table, ordered from left to right by what we call "Score." The algorithm for determining an index's score is not public, but you can assume that, for example, a covering index with a leading key column that is used in both a predicate and a sort will score much higher than a non-covering index with a leading key column that is sorted in the wrong direction.

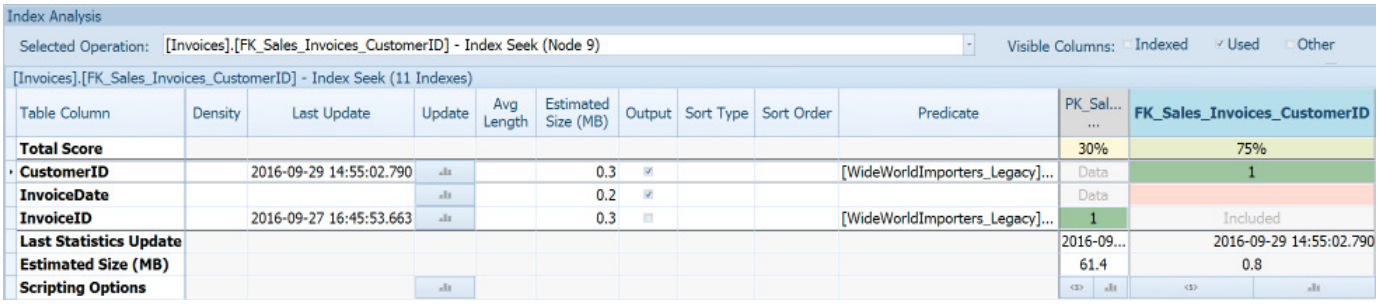
Now, if we look closer at the index that was chosen for this query, we see that it only scored 63%. The reason? It isn't covering. Meaning there are columns missing from the index, causing the key lookup. In the grid, we highlight those for you in salmon, to make it obvious which columns would help make this index better (or to use in a different index, if you wanted to go that way). The functionality we provide here to allow you to fix this, though, is quite a different way to think about index improvement. You can click into any of those missing column cells, make them a part of the index (hypothetically), and see how it improves the score of the index.

Since you know from the left side of the grid (or from the query text) that StockItemID is both part of the output and part of the ORDER BY, the first thing you do is change that column so that it is second in the key list. You do that by clicking on the cell and selecting the number 2 (and if the sort order were descending, you'd click the sort icon to change that order). This makes the index score 71%. The InvoiceLineID column says it's involved in a predicate, but that's the XML playing tricks on us (if you expand the predicate, you'll see that it's just validating against itself, as part of the nested loops operation). As the clustering key, it's included in this non-clustered index anyway, so you can move on to the other two columns. If key add those to the include list, by clicking the cell and choosing "Included," we see the score change again – this time to 90%.



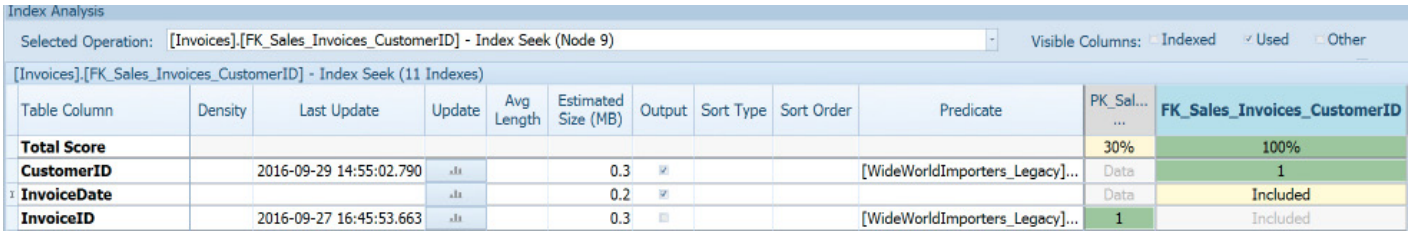

Not perfect, and keep in mind it won't always be possible to hit 100%, but certainly something we can try to improve the performance of the query. Click on the script button at the bottom left of that column (<s>), and it will give you a new dialog with a batch of SQL involving a DROP INDEX / CREATE INDEX. You don't have to do anything with this (it will be in the next entry in the History), but mention that you can run this directly here, copy it to a query window to modify before running, or just save it to a text file for later consideration. Click Close.

Before moving on, let's see if you can't improve this query further by removing the other Key Lookup. At the top of the tab, switch the Selected Operation to the other Index Seek (Node 9). You should see something like this:




| Table Column | Density | Last Update | Update | Avg Length | Estimated Size (MB) | Output | Sort Type | Sort Order | Predicate | PK_Sal... | FK_Sales_Invoices_CustomerID |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Total Score** | | | | | | | | | | 30% | 75% |
| **CustomerID** | | 2016-09-29 14:55:02.790 | | | 0.3 | ☑ | | | [WideWorldImporters_Legacy]... | Data | 1 |
| **InvoiceDate** | | | | | 0.2 | ☑ | | | | Data | |
| **InvoiceID** | | 2016-09-27 16:45:53.663 | | | 0.3 | ☐ | | | [WideWorldImporters_Legacy]... | 1 | Included |
| **Last Statistics Update** | | | | | | | | | | 2016-09... | 2016-09-29 14:55:02.790 |
| **Estimated Size (MB)** | | | | | | | | | | 61.4 | 0.8 |
| **Scripting Options** | | | | | | | | | | | |

Again, InvoiceID is shown with a predicate because of the nested loop, not because it's needed for the index, so you can focus on the other column that makes the Key Lookup necessary – InvoiceDate. This column is clearly required for output but not involved in any sort operations or predicates, so we can change the column to be included in the index, and watch the score jump from 75% to 100%.

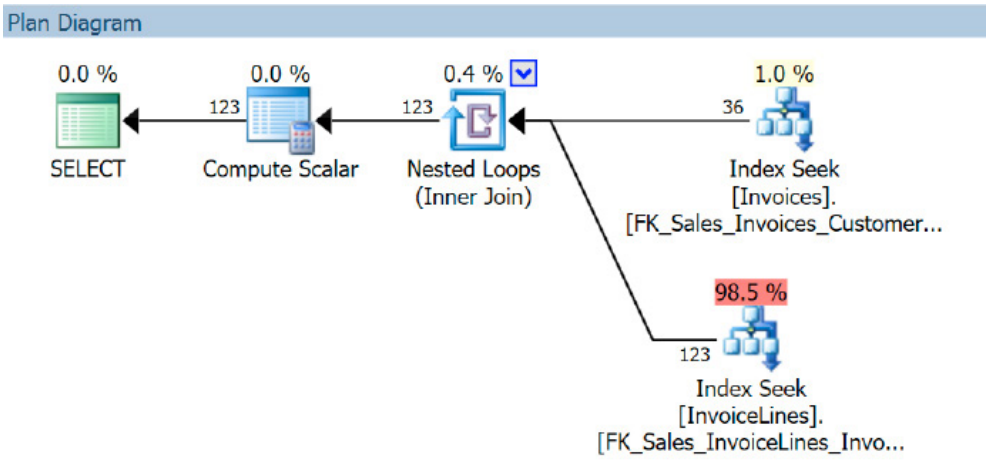



| Table Column | Density | Last Update | Update | Avg Length | Estimated Size (MB) | Output | Sort Type | Sort Order | Predicate | PK_Sal... | FK_Sales_Invoices_CustomerID |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Total Score** | | | | | | | | | | 30% | 100% |
| **CustomerID** | | 2016-09-29 14:55:02.790 | | | 0.3 | ☑ | | | [WideWorldImporters_Legacy]... | Data | 1 |
| **InvoiceDate** | | | | | 0.2 | ☑ | | | | Data | Included |
| **InvoiceID** | | 2016-09-27 16:45:53.663 | | | 0.3 | ☐ | | | [WideWorldImporters_Legacy]... | 1 | Included |

You can show the output in the <s> dialog, but know that both index scripts will be included in History version 3.

## HV3 - Index Creation & Re-Query

If you move to history item #3, you'll see that the indexes have been re-created with the adjustments we scripted out in step 2, and then the same query is run again. This time you see a much better plan, with two Key Lookups removed and, as a side effect of adding a column to the key of one of the indexes, a Sort has been removed as well:

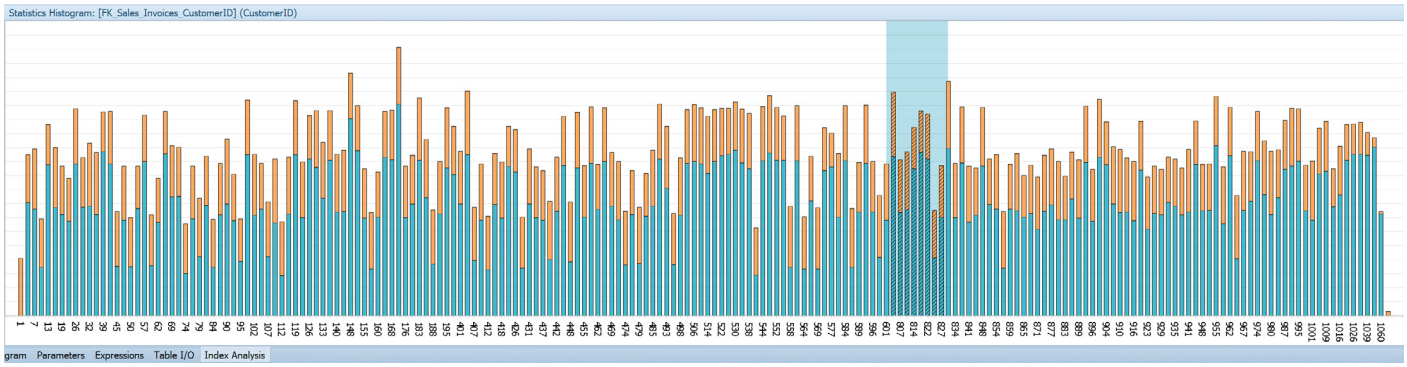


## HV4/5/6 Histogram

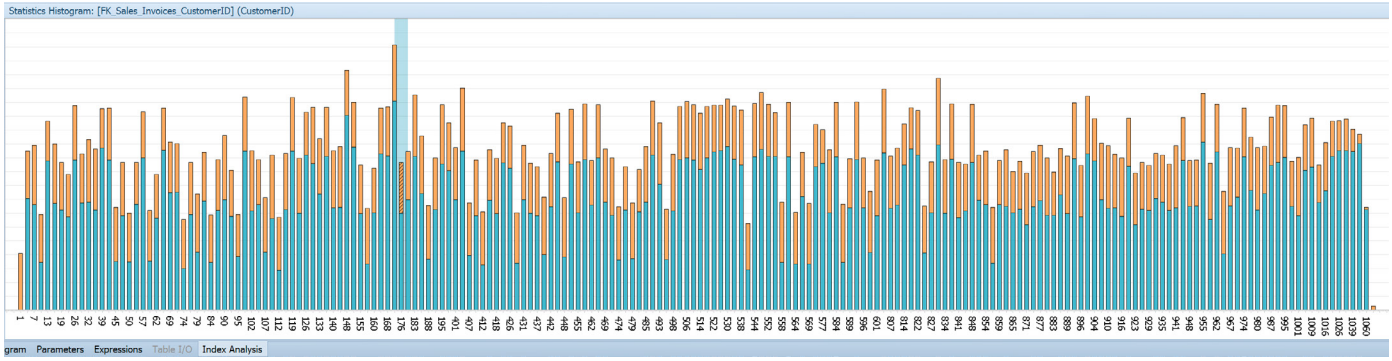
Next we can take a look at the histogram, which can be a useful source of information as well. For the foreign key index on Invoices, the distribution is fairly uniform. This query is slightly different in that we're looking for a range of customers. We'll start with a low of 800 and a high of 830 (yielding about 10,000 rows):

```
DECLARE @CustomerIDLow int = 800, @CustomerIDHigh int = 830;
SELECT
     il.StockItemID, il.ExtendedPrice,
     i.CustomerID, i.InvoiceID, 0 - il.Quantity, i.InvoiceDate
  FROM Sales.InvoiceLines AS il
  INNER JOIN Sales.Invoices AS i
  ON il.InvoiceID = i.InvoiceID
  WHERE i.CustomerID >= @CustomerIDLow
    AND i.CustomerID <= @CustomerIDHigh
  ORDER BY il.InvoiceID, il.StockItemID;
```

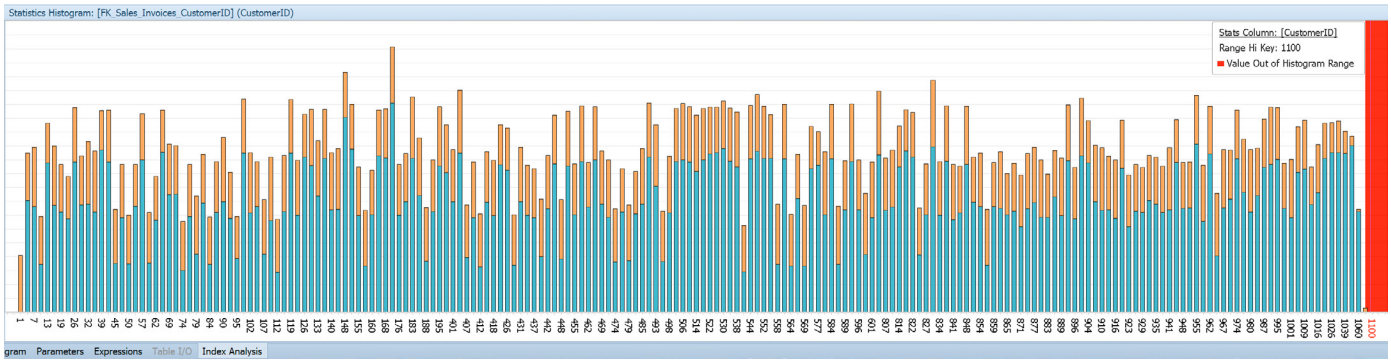
Look at the Index Analysis tab and make sure that the Index Seek is selected – in this case there is only a seek against Invoices, while a scan was chosen for InvoiceLines. You will see that there is a histogram below, with a rather uniform and uninteresting distribution, with a range of values highlighted – this represents the range of the runtime parameters (allowing you to spot parameter sniffing problems, say, where a scan is chosen instead of a seek or vice versa):




In the Parameters tab next to the Histogram, we can also test other values. For example, you can put 176 as high and low end of the range, click Get Est Plan, and you will end up with a slightly different looking histogram (as shown in History version 5):

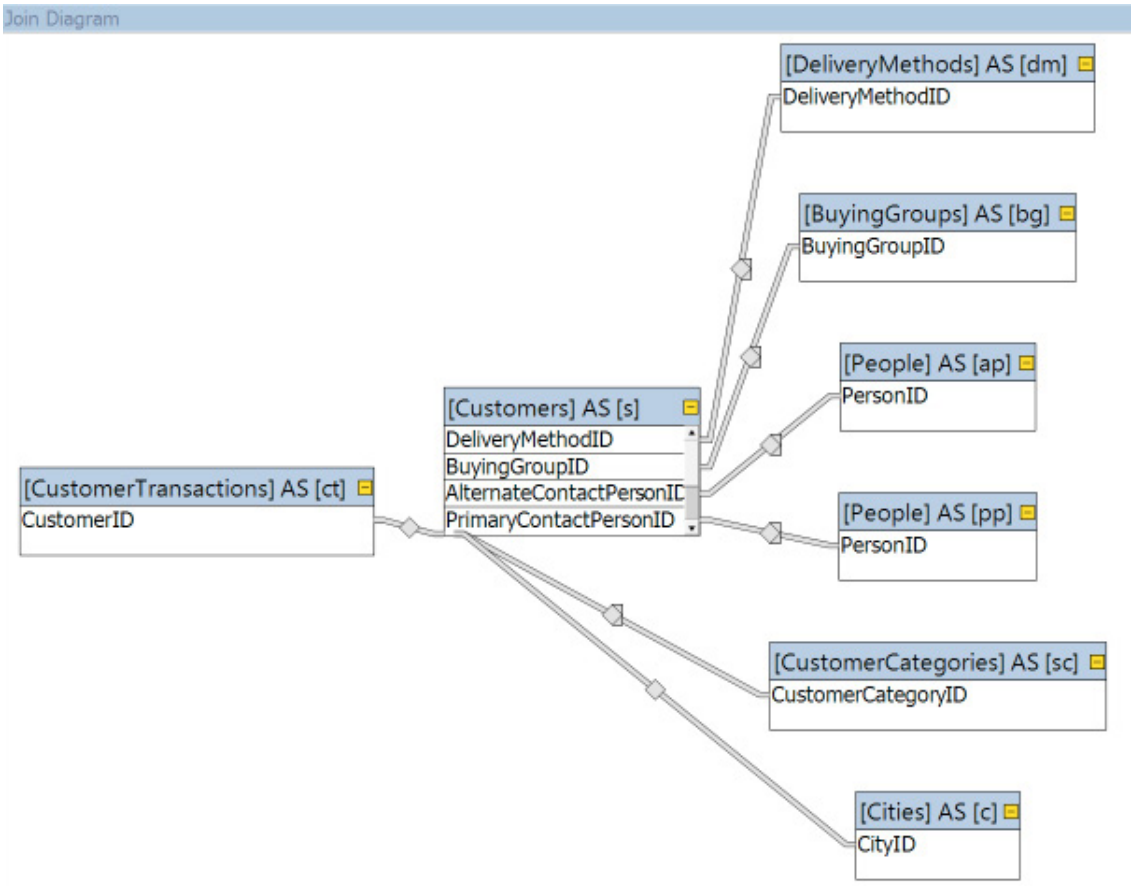


And if you repeat that process with values outside of the range, say 1100 and 1120 (to simulate an ascending key problem), you can see we paste a big red warning label there, with a tooltip that describes the problem (this is visible under History version 6):

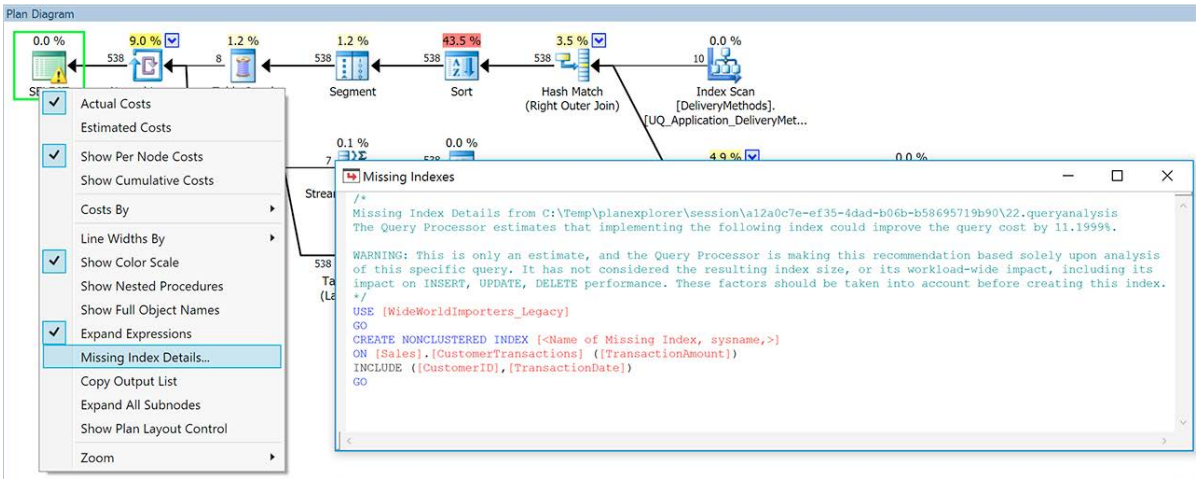



## HV7 - Join Diagrams & Missing Indexes

This is a rather lengthy query adapted from one of the views in the sample database. It demonstrates how you can visualize the tables and relationships involved in a query by moving to the Join Diagram tab. This can be quite useful for detangling and understanding queries involving views, especially when they are nested, as we show the underlying relationships between the base tables:



It also shows our Missing Indexes feature, which you can access by right-clicking the SELECT node on the Plan Diagram (and I'll repeat the warning here, don't just create every missing index recommended to you):




And finally, the Table I/O tab shows how we organize SET STATISTICS IO ON; data for you in a sortable grid.

Table I/O

| Table | LOB Read-Ahead Reads | LOB Physical Reads | LOB Logical Reads | Read-Ahead Reads | Physical Reads | Logical Reads | Scan Count |
|---|---|---|---|---|---|---|---|
| Worktable | 0 | 0 | 0 | 0 | 0 | 1,189 | 3 |
| CustomerTransactions | 0 | 0 | 0 | 0 | 0 | 1,028 | 1 |
| Customers | 0 | 0 | 0 | 0 | 0 | 37 | 1 |
| Cities | 0 | 0 | 0 | 0 | 0 | 314 | 1 |
| CustomerCategories | 0 | 0 | 0 | 0 | 0 | 2 | 1 |
| People | 0 | 0 | 0 | 0 | 0 | 20 | 2 |
| BuyingGroups | 0 | 0 | 0 | 0 | 0 | 2 | 1 |
| DeliveryMethods | 0 | 0 | 0 | 0 | 0 | 2 | 1 |

## HV8 - Live Query Capture

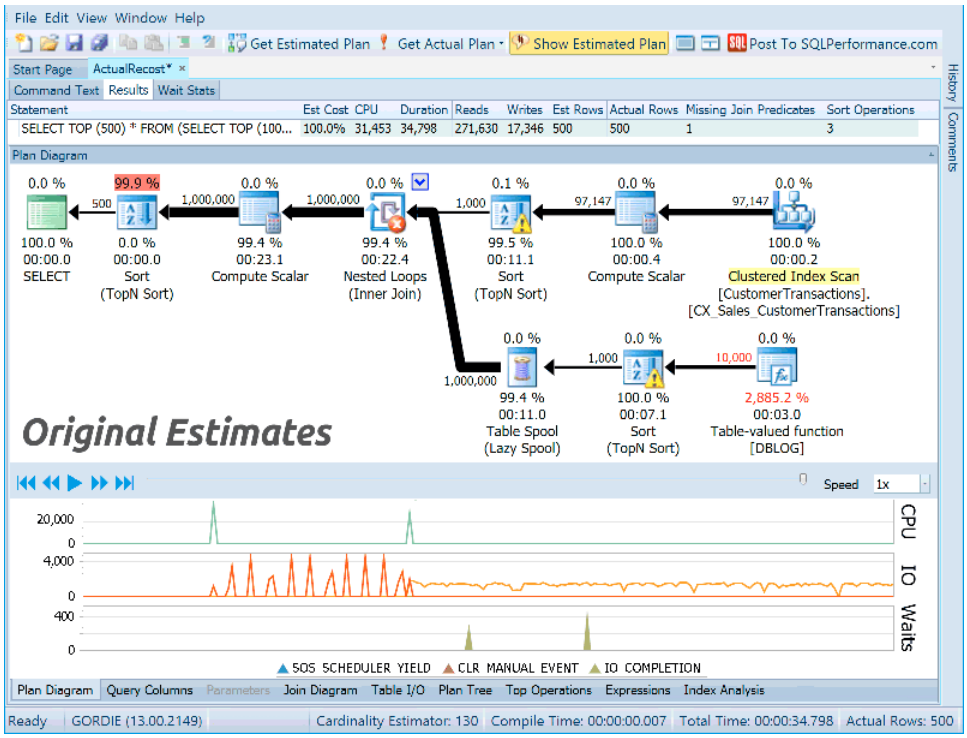
This has a complex query that runs long enough to show off our Live Query Capture, replay, and actual plan recosting features. Note that it can only be run interactively if you restore the database (or have an existing WideWorldImporters database) on an instance of SQL Server 2014 SP1 or newer, and if you enable "With Live Query Profile" under Get Actual Plan, but you can still show all of the details – even without connecting to a database.

If you open this history version, and move to the Plan Diagram tab, you will see it has some new "stuff" – a chart showing the progression of CPU, I/O, and Waits used throughout the execution of the statement, and a little set of play controls in the middle. The play controls let you replay the execution of the query – without executing against the server – to help you pinpoint exactly when spikes in any resource occurred (in this case, you'll see that they correlate to the activity from sys.fn_dblog()).

What's also interesting about this plan is that you can show how different estimated costs and actual costs can be – in many cases the estimated costs have no relevance whatsoever to what will happen when the query is executed. If you turn on "Show Estimated Plan" on the toolbar (or right-click the plan and select "Estimated Costs"), you see that SQL Server guessed that the Sort would be the most expensive operation in this plan. If you switch back to Actual, because we've collected per-operator resource usage, we can actually tell you very accurately what the actual costs were.

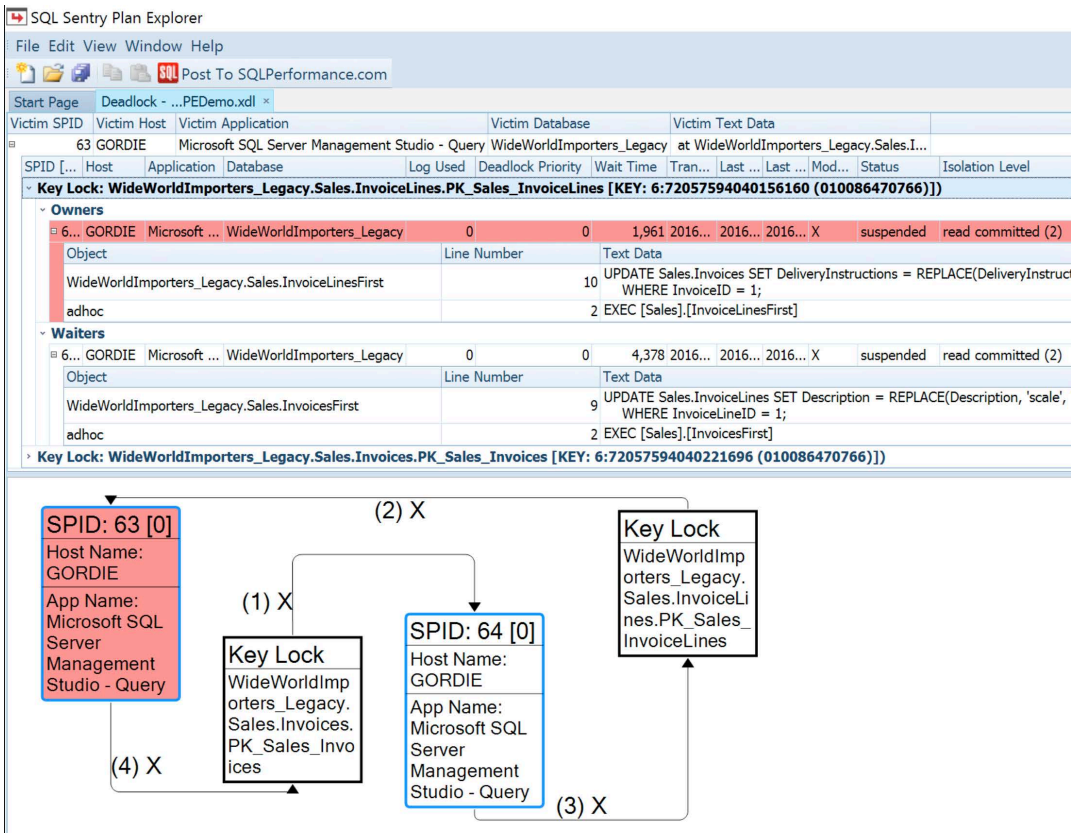
Since most SQL Server deployments are I/O-bound, we find it very useful to display the costs by I/O, which is not possible in Management Studio. If you right-click the plan again and choose Costs By > I/O, and then Line Widths By > Data Size (MB), you'll see that almost all of the I/O cost of this query – while still estimated to be in the Sort – is actually in the Table Spool related to sys.fn_dblog().

Again, the actual costs involved in the plan you have (especially if you run it again) are unlikely to match this graphic exactly, but you can see as you switch between these three views the difference between how SQL Server estimated the costs would be distributed, how we observed them being distributed in practice, and how the costs stacked up when we took a focus on I/O:



## PEDemo.xdl

I created two stored procedures in the sample database to make it easy to show how Plan Explorer can provide a lot of insight about a deadlock. In the screen shot below you can see that we include a grid with information such as isolation level, the statements that deadlocked, and even the procedures they came from – without having to reverse engineer things like ObjectIDs and HobtIDs. Below that there is a deadlock graph with a more intuitive layout than the one you get by default – again no reverse engineering of object names, and the flow clearly indicates the order in which locks were taken – which can help you solve the deadlock quicker.




In this case, the problem is simple – the two stored procedures named in the grid have batches that update two tables in the opposite order. If you wrap those procedure calls in outer transactions, SQL Server has no alternative – it needs to make one batch the victim and let the other one succeed.

## Other Features

This is just scratching the surface, but there are multiple other things you can show to see how much quicker it is to analyze plans within Plan Explorer, from just running queries to get runtime metrics more conveniently, comparing plans before and after changes using the History feature, spotting residual I/O, seeing the impact of updates on non-clustered indexes, and many other things. This kit is meant as a starting point, but you are more than welcome to generate plans from your own queries and make your demos your own.

For some ideas, see these other resources:

- Plan Explorer 3.0 Webinar
- Plan Explorer 3.0 Webinar – Samples and Q&A
- An updated Plan Explorer demo kit (v2)
- A demo kit for SQL Sentry Plan Explorer (v1)
- Plan Explorer User Guide
- Greg Gonzalez blogs about multiple PRO features

And I can't forget to mention that we have a Q & A site dedicated to solving execution plan issues using Plan Explorer, where you can get advice from query tuning experts like Paul White. You can upload plans and questions to the site right from within the application:

Post query plan issues to Answers.SQLPerformance.com.

I'm not at all ashamed to admit that many of the things I know about query tuning, execution plans, and the optimizer have come from reading (and re-reading many times) some of Paul's 300+ answers there.

## What Else is There?

The kit includes a short PowerPoint presentation – one slide on the evolution of Plan Explorer, a set of slides listing high-level features, and then descriptions of our newest features. The deck is built with our branding font, Ubuntu – a free, multi-platform download. If you don't like the typeface or don't want to download or install it, I can't make any claims about how every slide might look with a replacement font that is sized or kerned differently. There is a PDF version of the deck as well, where this won't matter; but I like to include the PowerPoint deck in case you want to make changes for your event. And please, feel free to make changes.

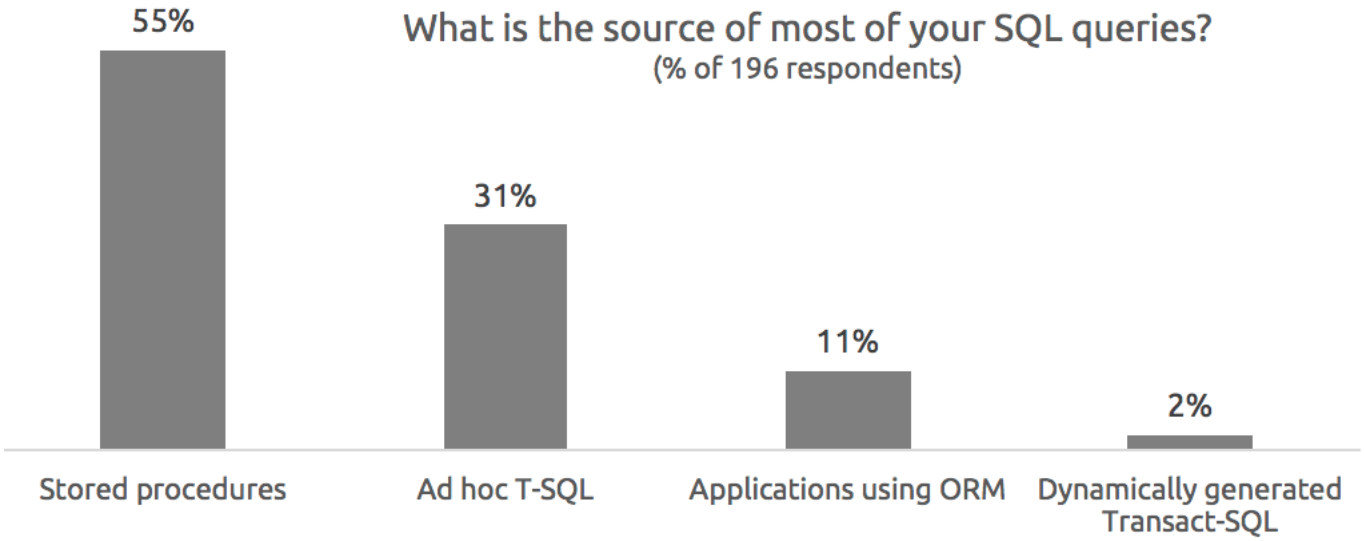
## Questions? Comments?

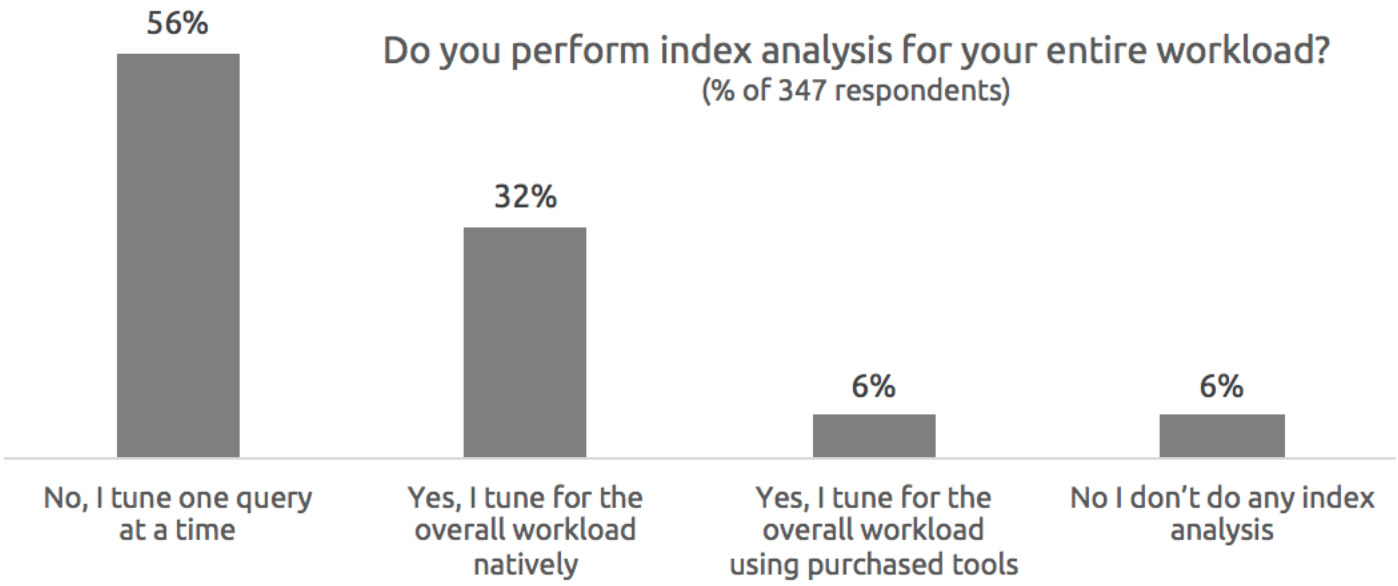
If you have any questions or comments about the demo kit, or Plan Explorer in general, or anything really, feel free to hit us up at community@sentryone.com or on twitter at @SentryOne.

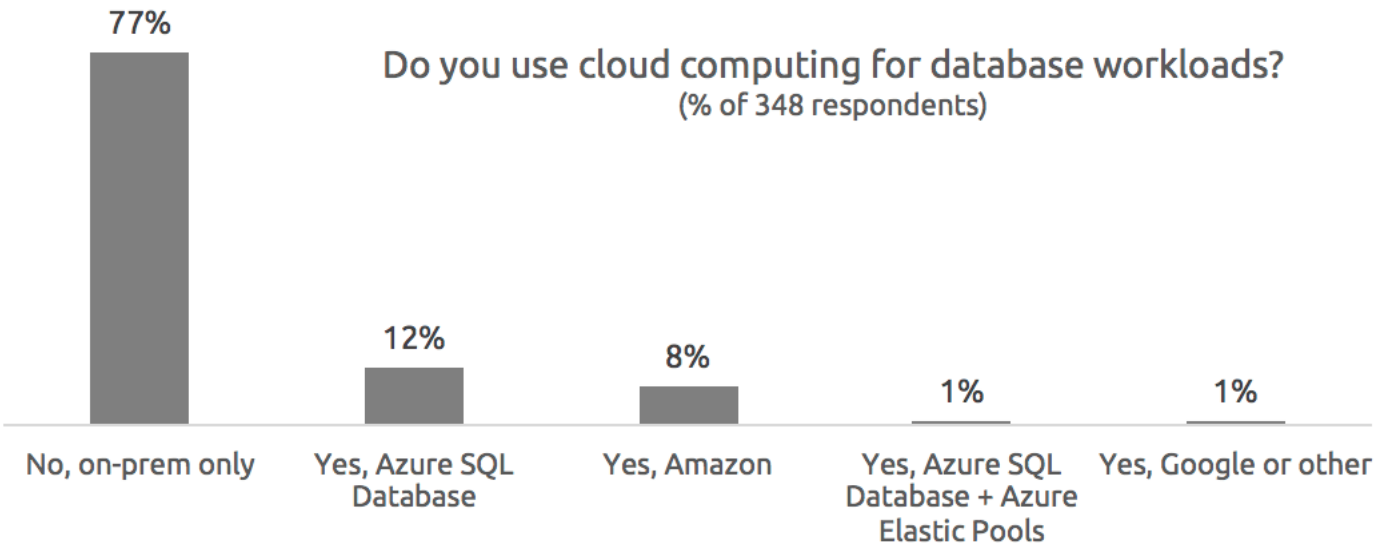
Aaron Bertrand

Recently I gave a webinar about Plan Explorer 3.0, the new features, and why we decided to eliminate the PRO edition and give away all the features for **free**. If you missed it, you can watch the webinar here.

There were many great questions submitted, and I will try to address those here. We also asked a few of our own questions at different points during the presentation, and users asked for details of those, so I'll start with the survey questions. We had a peak of 502 attendees, and I will indicate on the charts below how many people answered each question. Since the first question was asked before the webinar technically started, a smaller number of people answered that one.






## Audience Questions

**Q: Are the code samples available?**

**A:** Yes, the three session files I used for my demos are available here.

You can open these in the latest build of Plan Explorer, but if you want to run any of the queries again locally, you'll need AdventureWorks2014 (with the enlarging script from Jonathan Kehayias) and/or the new Wide World Importers sample database.

---

**Q: So everything shown today is in the new, unified, free Plan Explorer? If so, what is your company's new revenue model?**

**A:** I'm always surprised when I come across people who think that all we offer is Plan Explorer (I see these in person, and there were several similar comments on Greg's blog post as well). Our real bread and butter is in our monitoring platform, and we're hoping that your positive experience with Plan Explorer will lead you to try out our other solutions, too.

---

**Q: We're still using SQL Server 2008. Are there benefits to using PE vs SSMS?**

**A:** Yes, while you will miss out on some of the functionality (such as Live Query Profile), there is a lot more information available to you compared to SSMS, and we go out of our way to make specific issues much more discoverable.

---

**Q: Will Live Query Profile work for SQL Server 2014?**

**A:** Yes, as long as Service Pack 1 is applied, as the feature relies on a DMV that was added in SQL Server 2014 SP1.

**Q: What are the limitations with respect to SQL Server 2012? Can I use this tool at all?**

**A:** Absolutely. The limitation I brought up during the webinar about SQL Server 2012 and lower is that they are unable to capture Live Query Profile data.

---

**Q: Is the data only collected for SQL Server 2014 and higher? What if SQL Server 2014 is installed but the compatibility is set to 2012?**

**A:** Yes, Live Query Profile (and the resource charts) works in SQL Server 2014 (with at least SP1), SQL Server 2016, and Azure SQL Database. It is unaffected by compatibility level.

---

**Q: Which version of SQL Server is needed to get the wait stats information back?**

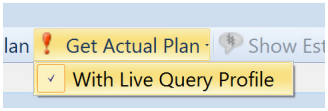
**A:** Wait statistics collection relies on an Extended Events session, so you need to be running against SQL Server 2008 or higher, and execute in the context of a user or login with enough permissions to create and drop an Extended Events session (CONTROL SERVER in SQL Server 2008 and 2008 R2, and ALTER ANY EVENT SESSION in SQL Server 2012 and above).

---

**Q: How do I get Index Analysis or the Live Query Profile charts to display?**

**A:** There were many variations on these two questions, and from the sounds of it, people were actively playing with the new version during the webinar, and not seeing either the Index Analysis data or the Live Query Profile data. If you have an existing plan captured from SSMS or an earlier version of Plan Explorer, there won't be any information to display.

In order to collect Index Analysis data, you must generate an estimated or actual plan from within Plan Explorer. In order to see a columns and indexes grid, you must choose a Selected Operation: in the dropdown at the top of the Index Analysis tab.




In order to collect Live Query Profile data, you must generate an actual plan from within Plan Explorer, and be running against 2014 SP1 or better. You also need to ensure you have selected the option "With Live Query Profile" (see image at right), and wait for the query execution to finish before the charts will render. In a future version, the charts will render in real time, but in this release we do that after all of the data has been collected.

---

**Q: Does the Live Query Profile function against cloned databases in SQL Server 2014 SP2?**

**A:** Yes, this will work, however it won't provide much information since a cloned database is empty – you will see the right estimates in the plan, but the actuals will all be 0, and so the runtime metrics won't represent any realistic or meaningful bottlenecks. Unless you are populating the clone with alternate data, as Erin Stellato promotes in an earlier post. Also note that if you want query plans to reflect real production data sizes, you'll want to make sure all forms of auto-stats are off, otherwise they will be updated as you run queries, and then all estimates will be 0.

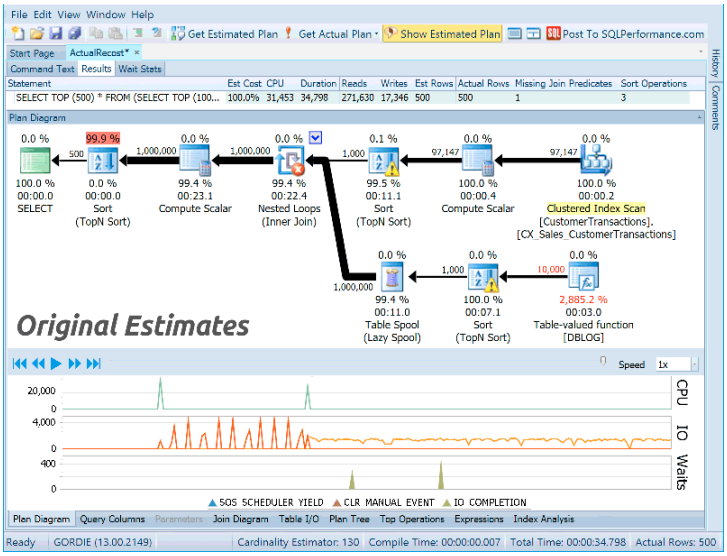
**Q: Does the new version of Plan Explorer work with SQL Server 2016?**

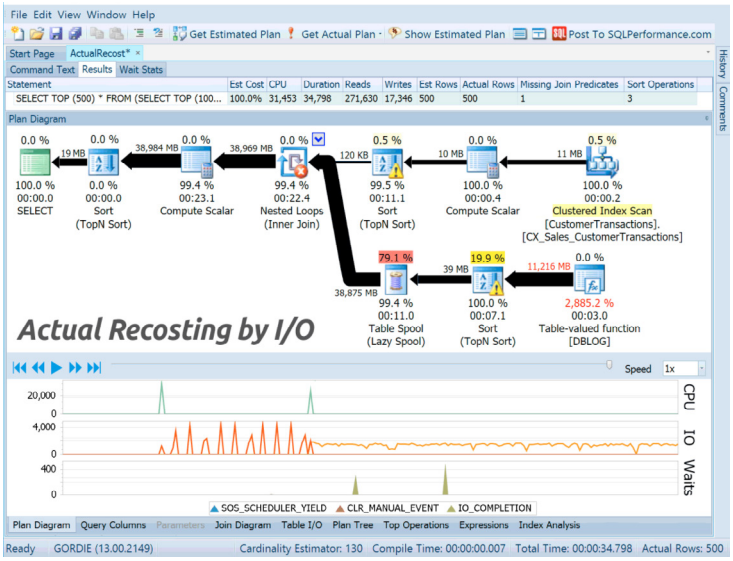
**A:** Yes. We support all of the new SQL Server 2016 plan operators and other showplan changes (see my post, "Plan Explorer Support for SQL Server 2016"), and the add-in works with the latest version of SSMS as well (see my post, "Announcing Plan Explorer Add-In Support for SSMS 2016").

---

**Q: So even an actual execution plan in SSMS is labeled with estimated costs?**

**A:** Yes, that's right. When you capture Live Query Profile data, we can change the cost percentages for all of the operators, because we know with a significant degree of accuracy how much actual work each operation performed (the query needs to run longer than a threshold, however). This can be especially useful if you are troubleshooting an I/O problem, because the estimates never seem to take I/O bottlenecks into account. The following graphic cycles through the original estimates (we can always show you what SSMS would have told you), the actuals after re-costing, and the actuals after re-costing and changing costs to "by I/O" and line widths to "by data size":



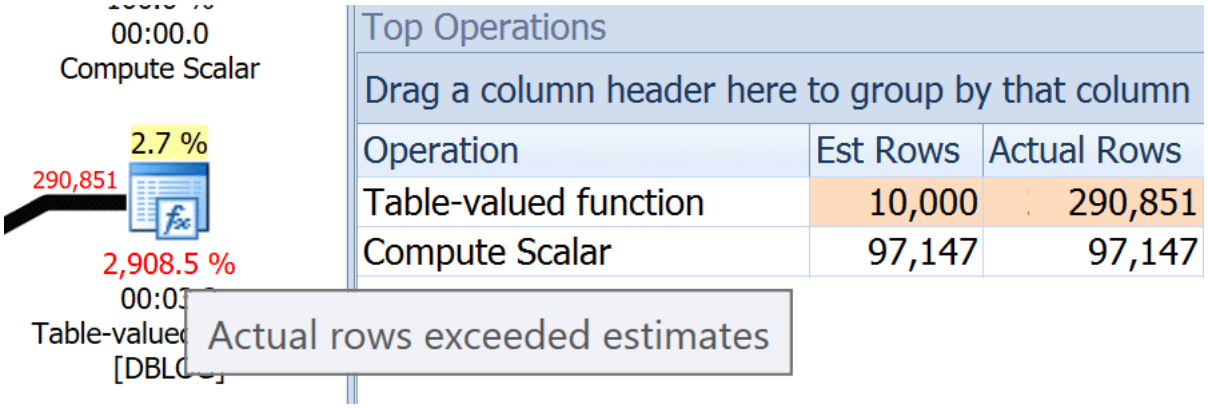
**Q: I used to open my plan produced by SSMS in Plan Explorer but from what Aaron just showed, did I understand correctly that I should run my queries (while tuning) from Plan Explorer?**

**A:** I addressed this question in the webinar, but to be clear, I think there are two steps in the evolution of a query: (1) ensuring correct results, and (2) performance optimization. I am a strong believer that currently you should be using SSMS for (1) and Plan Explorer for (2). I've long promoted that once people are sure they have correct results, they should tune by generating actual execution plans from within Plan Explorer, because we collect a lot more runtime information for you. This runtime information is particularly helpful if you share your plans on our Q & A site, because it makes all of the metrics and potential bottlenecks much more apparent.

---

**Q: What are the percentages below the operator… for example that 2,885% below the function?**

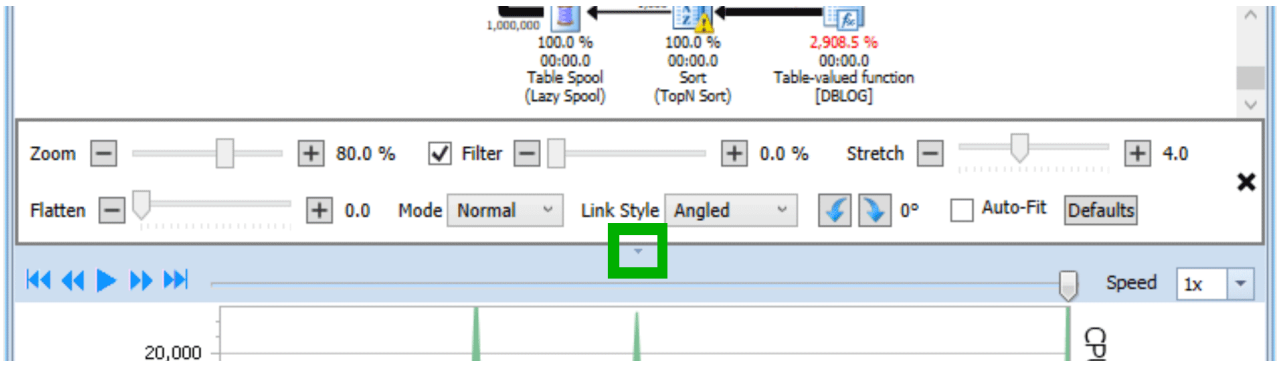
**A:** That percentage is not a cost but rather the % of rows that were actually processed compared to the estimate. In this case, SQL Server estimated that the function would return 10,000 rows, but at runtime it returned close to 300,000! You can see a tooltip if you hover only on that % number, and you can see the row count estimate differences in the tooltip for the operator, or in other grids like Top Operations (the function returns a different number of rows now than it did during the demo):




---

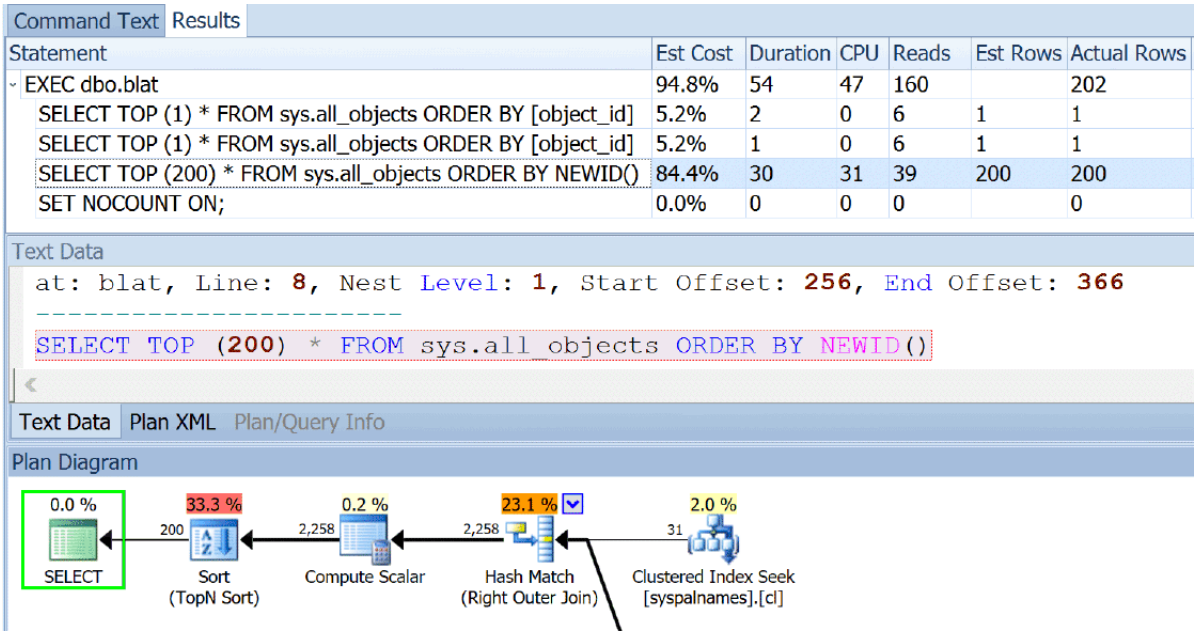
**Q: Can you minimize or hide the replay portion to have more real estate for the plan itself?**

**A:** Yes, all of our panels are adjustable; many have a push-pin which toggles between static and auto-hide, most panels can be dragged around (just like in Visual Studio, SSMS, etc.), and the replay panel in particular has a little arrow at top center that allows you to quickly show/hide:



**Q: Can you see the offending block of code directly from the plan?**

**A:** I'm not sure if I'm interpreting the question correctly, but all of our panels are context sensitive, and the statement for the plan currently being examined is shown both in the Statement grid and on the Text Data panel:

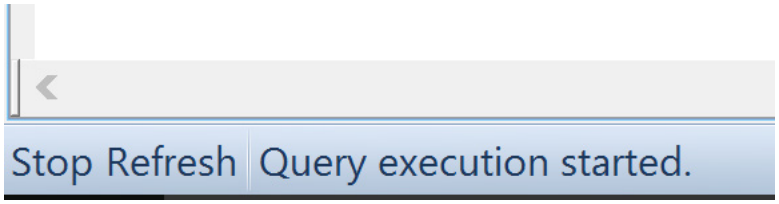



*If the statement text isn't fully visible due to length, you can always right-click that cell and choose Copy Statement to Command Text Copy, and then switch to that tab. Or, if you don't wantt o overwrite the current contents of the Command Text tab, choose Copy > Cell and paste into a new session, SSMS, or another editor.*

---

**Q: How can I stop a "Get actual plan" if I started a 1-hour-query by mistake?**

**A:** If a query is currently running, there is a Stop button on the status bar, bottom left:




---

**Q: Would not it be better to use DROP_EXISTING = ON instead of dropping an index first and creating a new one?**

**A:** We definitely have plans to make the index scripting more robust in the future, including options like DROP_EXISTING and ONLINE.

**Q: Does this tie into SentryOne?**

**A:** All of the functionality in Plan Explorer is also available in the SentryOne Client. You do not technically need to install Plan Explorer if you have the client, except that updates are pushed on a different schedule, so in many cases it may make sense to have both installed.

Keep in mind that plans that we collect for you during monitoring activities are estimated plans, due to the high cost of collecting actual plans for all queries running against a server. This means that if you drill down to a collected plan in the client, it won't have additional information such as Index Analysis and Live Query Profile data. You can always run the query again, interactively, in order to get that additional runtime data.

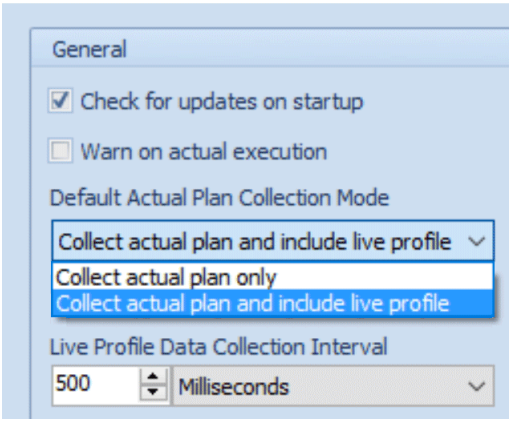
---

**Q: What is the performance overhead of these new features?**

**A:** Most of the information we collect is no more expensive than if you ran the same queries and collected the same runtime data from Management Studio (e.g. with SHOWPLAN, STATISTICS TIME, and STATISTICS IO on). Much of this is offset, though, by our default behavior of discarding results, so we don't burden the server with the effort of transmitting results to our application.

For extremely complex plans running against databases with very complex schemas and a LOT of indexes, the index and stats collection could be less efficient, but this will be extremely unlikely to cause any noticeable impact on existing workloads. This will not be affected by the number of rows in a table, which was mentioned in one variation of this question.

Preferences




For really long-running or resource-intensive queries, my biggest concern would be our Live Query Profile collection. We have two preferences that can help with this: whether to include Live Query Profile with all actual plan generation by default, and what interval to collect data from the DMV. While I still feel the overhead of this collection should never come anywhere near the overhead of the query itself, you can tweak these settings to make the collection less aggressive.

That all said, with the disclaimer that everything should be done in moderation, I haven't observed any issues related to the overhead of collecting the data, and would not hesitate to use the full functionality against a production instance.

---

**Q: Is there anything in there for helping build filtered indexes?**

**A:** Currently we don't have any functionality that recommends filtered indexes, but it is definitely on our radar.

**Q: Any plans to add a query plan compare feature to Plan Explorer?**

**A:** Yes, this has certainly been on our roadmap since long before this functionality was introduced in SSMS. :-) We're going to take our time and build out a feature set that you've hopefully come to expect from us.

---

**Q: Could you use with SSIS packages to figure out performance of a package?**

**A:** I suppose you could, if you invoke the package or job through T-SQL against a server (Plan Explorer doesn't have the ability to launch things like SSIS packages directly). But the application will only show the performance aspects that are made visible through SQL Server – if there are inefficiencies within the SSIS package that aren't related to execution against SQL Server (say, an infinite loop in a script task), we're not going to be able to pick those up, because we have no visibility and aren't performing any code analysis.

---

**Q: Can you quickly show how to use the deadlock analysis feature?**

**A:** I missed this question during the webinar, but I talk about this functionality in my Demo Kit, Jonathan Kehayias has blogged about it here, Steve Wright has a video about it on SQLSentry.TV, and the official documentation can be reviewed in the PE User Guide.

---

**Q: Can this be used like Profiler? Can I analyze an entire workload?**

**A:** Plan Explorer is designed to help analyze individual queries and their execution plans. We have a fully-featured monitoring platform for larger scoped efforts, and there are several 3rd party workload analysis tools out there as well.

---

**Q: I'm very new to query tuning – could you suggest tools and articles for deeper understanding?**

**A:** There are a lot of resources for getting better at query tuning:

- Any T-SQL book by Itzik Ben-Gan or Grant Fritchey;
- Any blog post by Paul White or Rob Farley;
- Q & A over at dba.stackexchange.com;
- The latest Plan Explorer Demo Kit; and,
- **Practice.** Seriously. You can read all the books and articles you want, but without practical, hands-on work troubleshooting and improving problematic queries with real performance issues, it will be tough to become an expert. IMHO.

# Thank you!

We hope you enjoyed this eBook. Follow SentryOne for updates and new releases of Plan Explorer.