**[Paul White: Page Free Space](http://sqlblog.com/blogs/paul_white/default.aspx)**

A technical SQL Server blog from New Zealand.

**Inside the Optimizer: Constructing a Plan - Part 1**

For today’s entry, I thought we might take a look at how the optimiser builds an executable plan using rules.  To illustrate the process performed by the optimiser, we’ll configure it to produce incrementally better plans by progressively applying the necessary rules.

Here’s a simple query (using the AdventureWorks sample database) that shows the total number of items in the warehouse for each of a small number of products:

SELECT

P.ProductNumber,

P.ProductID,

total\_qty = SUM(I.Quantity)

FROM Production.Product P

INNER JOIN Production.ProductInventory AS I ON

I.ProductID = P.ProductID

WHERE

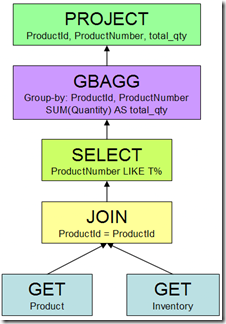
P.ProductNumber LIKE N'T%'

GROUP BY

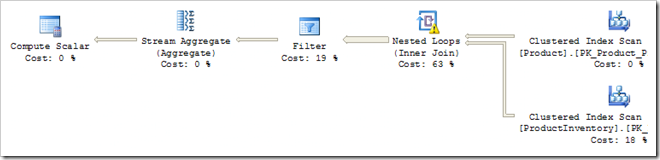
P.ProductID,

P.ProductNumber;

Heaps of work is done by SQL Server to parse and bind the query before it hits the query optimiser, but when it does, it arrives as a tree of logical relational operators, like this:

[](http://sqlblog.com/blogs/paul_white/LogicalTree_5CD711DE.png)

The optimiser needs to translate this into plan that can be executed by the Query Processor.  If the query optimiser did nothing more than translate the logical relational operators to the first valid form it found, we would get a plan like this:

[](http://sqlblog.com/blogs/paul_white/Literalplan_4159C1EE.png)

This executable plan features two full table scans, a Cartesian product, a Filter on the two predicates, and an Aggregate.  This is a long way from being an optimal query plan, but it does produce correct results.  (In case you are wondering, the Compute Scalar is there to ensure that the SUM aggregate returns NULL instead of zero when no rows are processed).

**Matching and Applying Rules**

The optimiser found this plan by replacing logical operations with one of the physical alternatives it knows about.  This type of transformation is performed by an *implementation rule* within the optimiser.  For example, the logical operation “Inner Join” was physically implemented by Nested Loops (other implementation rules exist for Merge and Hash join).

To get from the logical tree to the executable plan shown, a total of five implementation rules were successfully matched and run by the optimiser:

1. GET to Scan
2. JOIN to Nested Loops
3. SELECT to Filter
4. Group By Aggregate to Stream Aggregate
5. Group By Aggregate to Hash Aggregate

The first rule replaces the logical GETs with table scans.  The second rule implements the logical JOIN using Nested Loops as mentioned before.  The third converts all the predicates (including the join predicate) into a Filter operator.  The fourth and fifth rules represent two alternative strategies to physically perform the aggregation.  In this case, the optimiser chose a Stream Aggregate over a Hash Aggregate, based on costing.

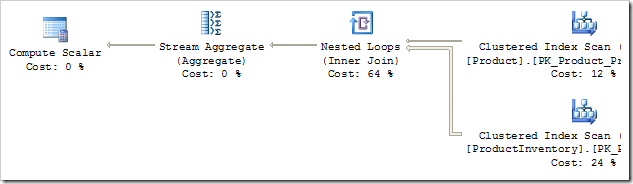
Ok, but no-one would buy SQL Server if it produced that sort of plan on a regular basis.  Luckily, there are many other rules that the optimiser can use; in fact there are nearly four hundred in total.  Rest assured that the query optimiser fought hard to resist producing such a dismal plan: more than one dozen separate rules had to be turned off to do so.

As well as more implementation rules, there are a number of *exploration* and *substitution* rules.  These transform some part of the logical request into an equivalent form, based on mathematical equivalences or heuristics.  A simple example of an exploration rule is 'Join Commute'.  This rule exploits the fact that A JOIN B is the same as B JOIN A (for inner joins).

The optimiser also has *simplification* rules and *enforcer* rules.  Application of all these rules generates alternative strategies, the best of which will be incorporated in the final plan.  As an aside, a single simplification rule was responsible for the transformation shown in my [Segment Top](http://sqlblog.com/blogs/paul_white/archive/2010/07/28/the-segment-top-query-optimisation.aspx) post.  That amazing transformation goes by the understated name "Generate Group By Apply".

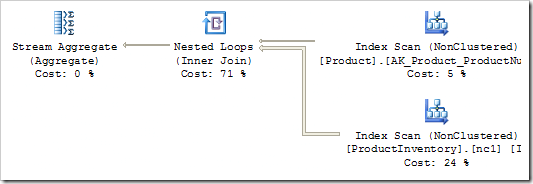
**Improving the Plan**

An obvious deficiency in the executable plan above is that it performs a Cartesian product of the two source tables followed by a Filter.  It would be much more efficient to evaluate the join predicate at the same time as performing the physical join.  The rule that performs this task is called SELonJN (SELECT on JOIN).  I should stress that this is not the T-SQL SELECT statement, it is the relational SELECT operation: a filter applied to rows.  By enabling the SELonJN optimisation rule, we get a better plan:

[](http://sqlblog.com/blogs/paul_white/SELonJN_on_3796EB27.png)

The Cartesian product has been replaced by a more normal-looking Nested Loops operator, which is a result of the join predicate being moved from the Filter into the join.  In fact, the Filter operator has disappeared completely - where has the predicate “ProductNumber LIKE 'T%'” gone?  It has also been pushed down the plan - all the way to the Product table scan.

That’s still not a great plan: we are scanning the whole Inventory table once for every row from the Product table that matches on the ProductNumber predicate.  We need rules that know about non-clustered indexes - the basic rule we have relied on so far simply translates a GET to a table scan.  Enabling a couple more rules produces:

[](http://sqlblog.com/blogs/paul_white/NCImpl_1_14698075.png)

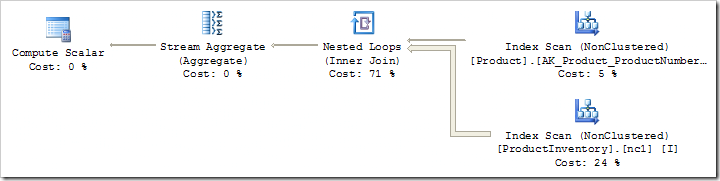
That's a little better, we are now using the correct non-clustered indexes but the predicates are being applied to a scan - not the index seek we might have hoped for.  There are quite a few rules left to apply before we get a really efficient plan.  I'll cover those next time.

**Inside the Optimizer: Constructing a Plan - Part 2**

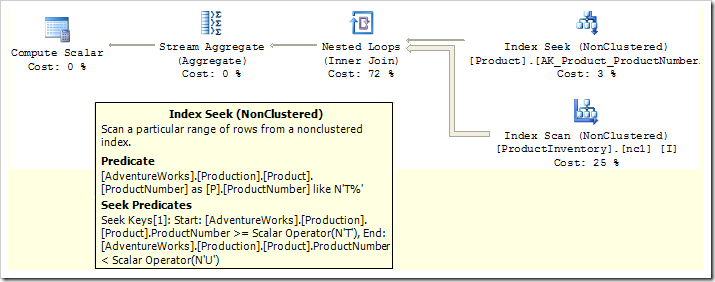
*Summary: Continuing the series of posts looking at how the optimiser matches and applies internal rules to refine a query plan.*

This post is part of a series: [Part1](http://sqlblog.com/blogs/paul_white/archive/2010/07/29/inside-the-optimiser-constructing-a-plan.aspx) [Part 2](http://sqlblog.com/blogs/paul_white/archive/2010/07/29/inside-the-optimiser-constructing-a-plan-ii.aspx) [Part3](http://sqlblog.com/blogs/paul_white/archive/2010/07/31/inside-the-optimiser-constructing-a-plan-part-3.aspx) [Part4](http://sqlblog.com/blogs/paul_white/archive/2010/07/31/inside-the-optimiser-constructing-a-plan-part-4.aspx)

The [last entry](http://sqlblog.com/blogs/paul_white/archive/2010/07/29/inside-the-optimiser-constructing-a-plan.aspx) ended with this query plan:

[](http://sqlblog.com/blogs/paul_white/NonclusteredScan_43262BF5.png)

The optimiser has pushed the predicate “ProductNumber LIKE 'T%'” down from a Filter iterator to the Index Scan on the product table, but it remains as a residual predicate.  We need to enable a new transformation rule (SelResToFilter) to allow the optimiser to rewrite the LIKE as an index seek:

[](http://sqlblog.com/blogs/paul_white/AfterSelResToFilter_3ED45683.png)

Notice that the LIKE is now expressed in a SARGable form, and the original LIKE predicate is now only evaluated on rows returned from the seek.

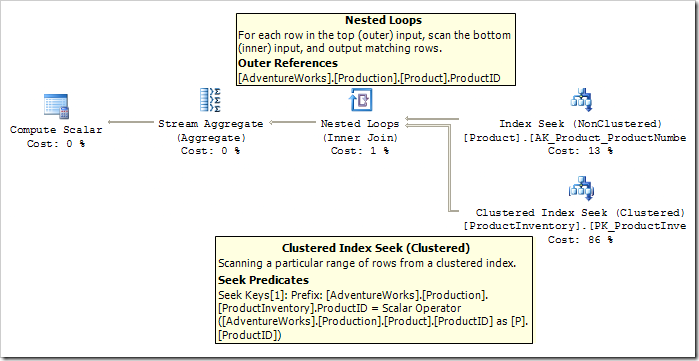
The remaining inefficiency is in scanning the whole Inventory table index for every row returned by our new seek operation.  At the moment, the JOIN predicate (matching ProductId between the two tables) is performed inside the Nested Loops operator.  It would be much more efficient to perform a seek on the Inventory table’s clustered index.

To achieve that, we need to do two things:

1. Convert the naive nested loops join to an index nested loops join (see [Understanding Nested Loops Joins](http://technet.microsoft.com/en-us/library/ms191318.aspx))
2. Drive each Inventory table seek using the current value of Product.ProductId

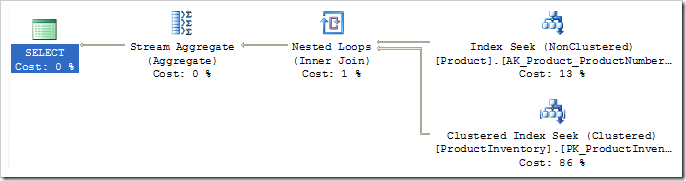
The first one is achieved by a rule called JNtoIdxLookup.  The second requirement is a correlated loops join - also known as an Apply.  The rule needed to transform our query to that form is AppIdxToApp.

With those two rules available to the optimiser, here’s the plan we get:

[](http://sqlblog.com/blogs/paul_white/IndexApply_3EF901D8.png)

We’re now pretty close to the optimal plan (for the specific value in this query).  The last step is to collapse the Compute Scalar into the Stream Aggregate.  You might remember that the purpose of the Compute Scalar is to ensure that the SUM aggregate returns NULL instead of zero if no rows are processed.

As it stands, the Compute Scalar is evaluating a CASE statement based on the result of a COUNT(\*) performed by the Stream Aggregate.  We can remove this Compute Scalar, and the need to compute COUNT(\*), by normalising the GROUP BY using a rule called ‘NormalizeGbAgg’.  Once that is done, we have the finished plan:

[](http://sqlblog.com/blogs/paul_white/FinalPlan_3DB98A5B.png)

In the next posts in the series, I’ll show you how to customise the rules available to the optimiser, and explore more of the internals of query optimisation.

# Inside the Optimizer: Constructing a Plan – Part 3

*Summary: This post presents an undocumented Dynamic Management View we can use to identify the optimisation rules involved in producing an executable plan.*

This post is part of a series: [Part1](http://sqlblog.com/blogs/paul_white/archive/2010/07/29/inside-the-optimiser-constructing-a-plan.aspx) [Part 2](http://sqlblog.com/blogs/paul_white/archive/2010/07/29/inside-the-optimiser-constructing-a-plan-ii.aspx) [Part3](http://sqlblog.com/blogs/paul_white/archive/2010/07/31/inside-the-optimiser-constructing-a-plan-part-3.aspx) [Part4](http://sqlblog.com/blogs/paul_white/archive/2010/07/31/inside-the-optimiser-constructing-a-plan-part-4.aspx)

##### Introduction

In order to fully explore the way the query optimiser uses rules to construct plan alternatives, we will need a way to identify the rules used to optimise a particular query.

SQL Server 2005 (later builds only) and SQL Server 2008 include an undocumented Dynamic Management View (DMV) that shows information about the internal rules used by the optimiser.  By taking a snapshot of that information before running a test query, and comparing it with the post-query DMV data, we can deduce the rules invoked by the optimiser for that query.

Before we get to the DMV itself, we need to nail down a few more things about the internals of the SQL Server query optimiser.  The next section builds on the ‘trees and rules’ information given in [part 1](http://sqlblog.com/blogs/paul_white/archive/2010/07/29/inside-the-optimiser-constructing-a-plan.aspx) of this series.

##### The Optimisation Process

Query optimisation is a recursive process that starts at the root of the logical operator tree, and ends with a physical representation suitable for execution.  The space of possible plans is explored by applying rules which result in a logical transformation of some part of the current plan, or a conversion to a physical implementation.

The optimiser does not try to match every available rule to every part of every query, in every possible combination.  That sort of exhaustive approach would guarantee the best plan possible, but you would not like the compilation times, or memory usage!

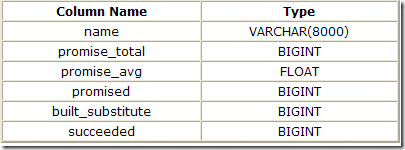
To find a good plan quickly, the optimiser uses a number of techniques.  I plan to cover these in some detail in future posts, but two of these tricks are immediately relevant to the DMV:

1. Every operator in the logical tree contains code to describe all the rules that are capable of matching with it.  This saves the optimiser from trying rules which have no chance of producing a lower-cost plan.
2. Every rule contains code to compute a value to indicate how promising the rule is (in context).  A rule has a higher promise value if it has a high potential to reduce the cost of the overall plan.  In general, commonly-used optimisations (like pushing a predicate) have a high ‘promise’ value.  More specialised rules, like those that match indexed views have a lower promise value.

When faced with several possible rule choices, the optimiser uses promise values as part of a pruning strategy.  This helps reduce compilation time, while still pursuing the most promising transformations.

##### sys.dm\_exec\_query\_transformation\_stats

This DMV contains one row for each rule, with the following columns:

[](http://sqlblog.com/blogs/paul_white/DMVInfo_07F70290.png)

The *name* column contains the internal name for the rule.  An example is ‘JNtoSM’ – an implementation rule to transform a logical inner join to a physical sort-merge join operator.

The *promised* column shows how many times the rule has been asked to provide a promise value to the optimiser.

The *promise\_total* column is a simple sum of all the promise values returned.

The *promise\_avg* column is just *promise\_total* divided by *promise*.

The *built\_substitute* column tracks how many times the rule has produced an alternative implementation.

The *succeeded* column tracks the number of times that a rule generated an transformation that was successfully added to the space of valid alternative strategies.  Not all transformations that result in an alternative will match the specific requirements of the current query plan (for example, the alternative may not preserve a required sort order, or some other property).

##### Using the DMV

The scripts included here were tested on SQL Server x86 Developer Edition, versions 10.0.2775 (2008 SP1 CU8) and 9.0.4294 (2005 SP3 CU9).  This DMV may not be available on all builds of SQL Server 2005.

Since the DMV contains server-scoped optimiser information, for correct results you will need to ensure that no other concurrent optimisation activity is occurring on the test server.  Working on a personal test SQL Server and ensuring yours is the only active connection is a good place to start with that.

First, we need to create the structure of a temporary table to hold our snapshot of the DMV contents prior to running our test query:

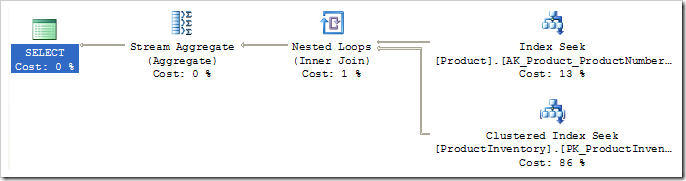
SELECT TOP (0)  
 name,  
 promise\_total,  
 promised,  
 built\_substitute,  
 succeeded  
INTO #Snapshot  
FROM sys.dm\_exec\_query\_transformation\_stats;

Now we can write a batch to capture a DMV snapshot, run our test query, and show the DMV differences afterward:

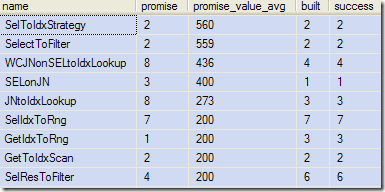
-- Clear the snapshot  
TRUNCATE TABLE #Snapshot;  
   
-- Save a snapshot of the DMV  
INSERT #Snapshot   
 (  
 name,   
 promise\_total,   
 promised,   
 built\_substitute,   
 succeeded  
 )  
SELECT name,   
 promise\_total,  
 promised,   
 built\_substitute,   
 succeeded  
FROM sys.dm\_exec\_query\_transformation\_stats  
OPTION (KEEPFIXED PLAN);  
   
-- Query under test  
-- Must use OPTION (RECOMPILE)  
SELECT P.ProductNumber,   
 P.ProductID,   
 total\_qty = SUM(I.Quantity)  
FROM Production.Product P  
JOIN Production.ProductInventory I  
 ON I.ProductID = P.ProductID  
WHERE P.ProductNumber LIKE N'T%'  
GROUP BY  
 P.ProductID,  
 P.ProductNumber  
OPTION (RECOMPILE);  
   
-- Results  
SELECT QTS.name,  
 promise = QTS.promised - S.promised,  
 promise\_value\_avg =   
 CASE  
 WHEN QTS.promised = S.promised  
 THEN 0  
 ELSE  
 (QTS.promise\_total - S.promise\_total) /  
 (QTS.promised - S.promised)  
 END,  
 built = QTS.built\_substitute - S.built\_substitute,  
 success = QTS.succeeded - S.succeeded  
FROM #Snapshot S  
JOIN sys.dm\_exec\_query\_transformation\_stats QTS  
 ON QTS.name = S.name  
WHERE QTS.succeeded != S.succeeded  
ORDER BY  
 promise\_value\_avg DESC  
OPTION (KEEPFIXED PLAN);

The query to test must have the OPTION (RECOMPILE) query hint added to ensure that a compilation occurs.  Other queries in the batch have OPTION (KEEPFIXED PLAN) to help avoid compilations that would skew the results.

The example above uses the AdventureWorks query we have been using in this series so far.  It produces the familiar, fully-optimised, plan:

[](http://sqlblog.com/blogs/paul_white/FinalQueryPlan_43A9C027.png)

Here are the (partial) results from a typical run:

[](http://sqlblog.com/blogs/paul_white/SampleDMVResults_79B0CFE5.png)

The output shows the rule name, the number of times a promise value was calculated, the average promise values produced, the number of times a transformed structure was built, and the number of times the new structure was successfully added to the optimiser’s list of valid choices.

Notice that rules can be invoked multiple times, since they may match more than one place in the query, and the compilation process is a recursive one.  You might also see rules with a promise value of zero, which simply means the promise-calculating code did not have enough information to produce a value.

##### Next time

Now that we have found a way to identify the rules used to optimise a given query, we can move on to the really fun stuff.  In the next part of the series, I will show two ways to affect the rules available to the optimiser, and present code to reproduce the ‘interesting’ partially-optimised query plans shown in parts 1 & 2.

In the meantime, I would appreciate any comments, ideas or feedback you may have, so I can improve future posts.

# Inside the Optimizer: Constructing a Plan – Part 4

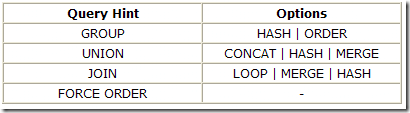
Summary: More undocumented ways to explore how the query optimiser works.

This post is part of a series: [Part1](http://sqlblog.com/blogs/paul_white/archive/2010/07/29/inside-the-optimiser-constructing-a-plan.aspx) [Part 2](http://sqlblog.com/blogs/paul_white/archive/2010/07/29/inside-the-optimiser-constructing-a-plan-ii.aspx) [Part3](http://sqlblog.com/blogs/paul_white/archive/2010/07/31/inside-the-optimiser-constructing-a-plan-part-3.aspx) [Part4](http://sqlblog.com/blogs/paul_white/archive/2010/07/31/inside-the-optimiser-constructing-a-plan-part-4.aspx)

##### Documented Rule-Affecting Options

It might surprise you to learn that most T-SQL users have written queries that disable one or more of the optimiser rules already.

[Join hints](http://msdn.microsoft.com/en-us/library/ms173815.aspx) { LOOP | HASH | MERGE | REMOTE } and the [query hints](http://msdn.microsoft.com/en-us/library/ms181714.aspx) shown below are all implemented by disabling one or more optimiser rules:

[](http://sqlblog.com/blogs/paul_white/Hints_5E03F261.png)

For example, a logical GROUP BY operation (including DISTINCT) can be physically implemented as either a Hash Aggregate or a Stream Aggregate.  The two implementation rules involved are GbAggToHS (Group By Aggregate to Hash) and GbAggToStrm (Group By Aggregate to Stream).  Normally both of these implementation options are available to the optimiser, and it might consider one or both when optimising a logical operator tree.

When the query hint OPTION (HASH GROUP) appears in a query, the GbAggToStrm implementation rule is disabled.  This means that all GROUP BY operations are implemented by the GbAggToHS rule, and so appear as Hash Aggregates.

The join hint “INNER MERGE JOIN” works by disabling the implementation rules for nested loops and hash join.  (An interesting side-effect is that using a join hint makes the query act as if OPTION (FORCE ORDER) had also been specified.)

If we want the optimiser to only consider using merge and hash join strategies for a query, we could use the query hint OPTION (MERGE JOIN, HASH JOIN).  This works by disabling the JNtoNL implementation rule (JOIN to Nested Loops), leaving the optimiser the choice of sort-merge or hash joins (rules JNtoSM and JNtoHS).  Hinting joins in a query hint also implies OPTION (FORCE ORDER).

#### Undocumented Features

***Warning: This information is provided AS IS and for educational purposes only.  Never use these tools on production systems.  No documentation or support is available.***

Join hints and query hints don’t expose options to selectively enable or disable the full range of rules available to the optimiser.  We can, however, use a couple of undocumented DBCC commands and the undocumented dynamic management view *sys.dm\_exec\_query\_transformation\_stats* (covered in [part 3](http://sqlblog.com/blogs/paul_white/archive/2010/07/31/inside-the-optimiser-constructing-a-plan-part-3.aspx)) to explore the way the optimiser uses rules.

The techniques shown below work best in SQL Server 2008, but will also function in SQL Server 2005 (with a number of important caveats).  The specific versions used in writing this entry were x86 Developer Editions 10.0.2775 (2008 SP1 CU8) and 9.0.4294 (2005 SP3 CU9).

##### Trace Flags and DBCC commands

As with [other](http://blogs.msdn.com/b/sqlserverstorageengine/archive/2006/12/13/more-undocumented-fun_3a00_-dbcc-ind_2c00_-dbcc-page_2c00_-and-off_2d00_row-columns.aspx) undocumented DBCC options, we need to enable trace flag 3604 for the current session, so that any output is returned to the client (Management Studio, for example).

DBCC TRACEON (3604);

We can disable one or more optimiser rules using *DBCC RULEOFF*.  This command takes one or more rule names (or numbers) as its parameters.  To disable the rules that implement a logical JOIN as a sort-merge or hash, we would execute:

DBCC RULEOFF('JNtoSM', 'JNtoHS');

To enable the rules again, we can use *DBCC RULEON*, with the same syntax:

DBCC RULEON('JNtoSM', 'JNtoHS');

Both RULEON and RULEOFF return confirmation messages (with trace flag 3604 on).  The direct effects on the optimiser only apply to the current session, but the (sub-optimal) plans produced will be cached as normal – another great reason to only play with this stuff on a personal test system!

To reset to normal operation, enable any disabled rules, or simply disconnect and reconnect to the server.  It is also a good idea to run DBCC FREEPROCCACHE to remove any sub-optimal plans from cache.

To see which rules are currently enabled or disabled, use the *DBCC SHOWONRULES* and *DBCC SHOWOFFRULES* commands.  Neither of these commands take any parameters.

##### SQL Server 2005 Bugs

In SQL Server 2005, SHOWOFFRULES displays a list of rules that are ON, and SHOWONRULES displays rules that are OFF, which is actually quite funny.

DBCC SHOWONRULES also does not return any output unless you call DBCC SHOWOFFRULES immediately afterward.  The list of disabled rules will be prefixed to the output of SHOWOFFRULES (which shows enabled rules, remember).

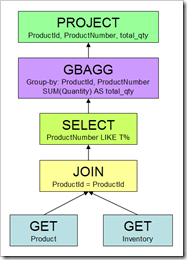
The list of disabled rules is also not formatted correctly in 2005: all rule names are concatenated without any separator.

Both commands work perfectly in SQL Server 2008.

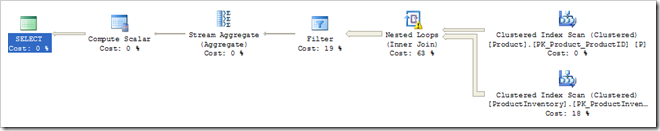
#### Putting it all together

We now have all the tools we need to produce the partially-optimised plans seen in previous posts in this series.  We can use the *sys.dm\_exec\_query\_transformation\_stats* DMV to identify rules invoked by the optimiser, and the new DBCC commands to selectively disable them to see the effect on the final plan.

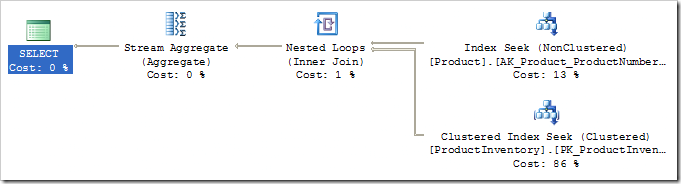
Here’s our sample AdventureWorks query’s logical relational tree representation again (click to enlarge):

[](http://sqlblog.com/blogs/paul_white/RelationalTree_630B5172.png)

…and the very basic plan originally shown in [part 1](http://sqlblog.com/blogs/paul_white/archive/2010/07/29/inside-the-optimiser-constructing-a-plan.aspx) of the series:

[](http://sqlblog.com/blogs/paul_white/BasicPlan_570BF1EF.png)

That query plan has an estimated cost of 3.59557 – compare that to the fully-optimised plan’s cost of 0.0248906:

[](http://sqlblog.com/blogs/paul_white/FullyOptimised_260ED1F3.png)

##### Producing Partially-Optimised Plans

Using the DMV and query from [part 3](http://sqlblog.com/blogs/paul_white/archive/2010/07/31/inside-the-optimiser-constructing-a-plan-part-3.aspx) we saw that the optimiser used more than twenty different rules in finding a fully-optimised plan.  To create the very basic plan, just four core implementation rules are needed (five in SQL Server 2005):

-- Route messages to the client  
DBCC TRACEON (3604);  
   
-- Ensure the four core implementation rules are available  
DBCC RULEON ('GetToScan');  
DBCC RULEON ('JNtoNL');  
DBCC RULEON ('SelectToFilter');  
DBCC RULEON ('GbAggToStrm')  
   
-- Required by SQL Server 2005 only  
DBCC RULEON ('ReduceGbAgg');

We need to disable the other rules normally considered:

-- Alternative join implementations  
DBCC RULEOFF ('JNtoHS');  
DBCC RULEOFF ('JNtoSM');  
   
-- Index-related transformations  
DBCC RULEOFF ('GetIdxToRng');  
DBCC RULEOFF ('GetToIdxScan');  
DBCC RULEOFF ('SelIdxToRng');  
DBCC RULEOFF ('SelToIdxStrategy');  
DBCC RULEOFF ('SELonJN');  
DBCC RULEOFF ('JNtoIdxLookup');  
DBCC RULEOFF ('AppIdxToApp');  
DBCC RULEOFF ('SelResToFilter');  
DBCC RULEOFF ('WCJNonSELtoIdxLookup');  
   
-- Exploration rules  
DBCC RULEOFF ('GbAggToHS')  
DBCC RULEOFF ('JoinCommute');  
DBCC RULEOFF ('GbAggBeforeJoin');  
DBCC RULEOFF ('GenLGAgg');  
DBCC RULEOFF ('BuildSpool');  
DBCC RULEOFF ('ImplRestrRemap');  
DBCC RULEOFF ('EnforceSort');  
DBCC RULEOFF ('NormalizeGbAgg');

We can now get the basic query plan by executing the AdventureWorks query on its own (or by requesting an estimated plan):

SELECT P.ProductNumber,   
 P.ProductID,   
 total\_qty = SUM(I.Quantity)  
FROM Production.Product P  
JOIN Production.ProductInventory I  
 ON I.ProductID = P.ProductID  
WHERE P.ProductNumber LIKE N'T%'  
GROUP BY  
 P.ProductID,  
 P.ProductNumber  
OPTION (RECOMPILE);

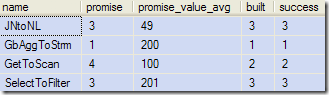
We have disabled so many important rules that we can no longer run the DMV batch code from part 3.  The optimiser is unable to produce any correct plan with the restricted range of rules now available to it, so the following error is returned:

.Net SqlClient Data Provider: Msg 8624, Level 16, State 1, Line 1   
Internal Query Processor Error: The query processor could not produce a query plan.   
For more information, contact Customer Support Services.

If you want to see the DMV statistics, you’ll need to clear the plan cache using DBCC FREEPROCCACHE before running the following modified code:

-- Clear the snapshot  
TRUNCATE TABLE #Snapshot;  
   
-- Save a snapshot of the DMV  
INSERT #Snapshot   
 (  
 name,   
 promise\_total,   
 promised,   
 built\_substitute,   
 succeeded  
 )  
SELECT name,   
 promise\_total,  
 promised,   
 built\_substitute,   
 succeeded  
FROM sys.dm\_exec\_query\_transformation\_stats  
OPTION (KEEPFIXED PLAN);  
   
-- Query under test  
-- Must use OPTION (RECOMPILE)  
SELECT P.ProductNumber,   
 P.ProductID,   
 total\_qty = SUM(I.Quantity)  
FROM Production.Product P  
JOIN Production.ProductInventory I  
 ON I.ProductID = P.ProductID  
WHERE P.ProductNumber LIKE N'T%'  
GROUP BY  
 P.ProductID,  
 P.ProductNumber  
OPTION (RECOMPILE);  
GO  
-- Results  
SELECT QTS.name,  
 promise = QTS.promised - S.promised,  
 promise\_value\_avg =   
 CASE  
 WHEN QTS.promised = S.promised  
 THEN 0  
 ELSE  
 (QTS.promise\_total - S.promise\_total) /  
 (QTS.promised - S.promised)  
 END,  
 built = QTS.built\_substitute - S.built\_substitute,  
 success = QTS.succeeded - S.succeeded  
FROM #Snapshot S  
JOIN sys.dm\_exec\_query\_transformation\_stats QTS  
 ON QTS.name = S.name  
WHERE QTS.succeeded != S.succeeded  
OPTION (KEEPFIXED PLAN);

You’ll see results similar to these:

[](http://sqlblog.com/blogs/paul_white/ModifiedDMVOutput_30CEB9F9.png)

Refer back to part 3 for further details about that output.

##### A Spool Rule

Let’s explore one more alternative plan by enabling the rule ‘BuildSpool’.  This is one of the many rules that can introduce a Table Spool operator into the plan to improve efficiency:

DBCC RULEON ('BuildSpool');

The estimated plan for our test query now looks like this (again, click to enlarge):

[](http://sqlblog.com/blogs/paul_white/LazySpoolPlan_6DF2757E.png)

It’s still a terrible overall plan of course, but the introduction of a Lazy Spool on the inner side of the nested loops join has reduced the estimated cost of the plan from 3.59557 to 3.12199 – a worthwhile improvement.

##### Clean Up

Don’t forget to reset your session by re-enabling the disabled rules:

DBCC RULEON ('JNtoHS');  
DBCC RULEON ('JNtoSM');  
DBCC RULEON ('GetIdxToRng');  
DBCC RULEON ('GetToIdxScan');  
DBCC RULEON ('SelIdxToRng');  
DBCC RULEON ('SelToIdxStrategy');  
DBCC RULEON ('SELonJN');  
DBCC RULEON ('JNtoIdxLookup');  
DBCC RULEON ('AppIdxToApp');  
DBCC RULEON ('SelResToFilter');  
DBCC RULEON ('WCJNonSELtoIdxLookup');  
DBCC RULEON ('GbAggToHS')  
DBCC RULEON ('JoinCommute');  
DBCC RULEON ('GbAggBeforeJoin');  
DBCC RULEON ('GenLGAgg');  
DBCC RULEON ('BuildSpool');  
DBCC RULEON ('ImplRestrRemap');  
DBCC RULEON ('EnforceSort');  
DBCC RULEON ('NormalizeGbAgg');  
DBCC RULEON ('ReduceGbAgg');

You can check that all rules are now enabled again by running DBCC SHOWOFFRULES (but see my previous remarks concerning the bugs in SQL Server 2005).

##### Final thoughts

You might be wondering whether all this has any practical application.  For me, a better understanding of optimiser internals enables me to write better queries, and more quickly debug poor plans.

There are also very rare (and advanced) uses where we can capture a ‘customised’ plan for use in a USE PLAN hint.

That’s all for this post, as always your comments and feedback are welcome below.  If you’re a twitter user, you’ll find me there as [@SQL\_Kiwi](http://twitter.com/SQL_Kiwi)

© Paul White   
email: [SQLkiwi@gmail.com](mailto:SQLkiwi@gmail.com)