For most Data Definition Language (DDL) statements (such as `CREATE TABLE`, `CREATE PROC`, and so on), after passing through the algebrizer, the query is compiled directly for execution, since the optimizer need not choose among multiple processing strategies. For one DDL statement in particular, `CREATE INDEX`, the optimizer can determine an efficient processing strategy based on other existing indexes on the table, as explained in Chapter 8.

For this reason, you will never see any reference to `CREATE TABLE` in an execution plan, although you will see reference to `CREATE INDEX`. If the normalized query is a Data Manipulation Language (DML) statement (such as `SELECT`, `INSERT`, `UPDATE`, or `DELETE`), then the query processor tree is passed to the optimizer to decide the processing strategy for the query.

Optimization

Based on the complexity of a query, including the number of tables referred to and the indexes available, there may be several ways to execute the query contained in the query processor tree. Exhaustively comparing the cost of all the ways of executing a query can take a considerable amount of time, which may sometimes override the benefit of finding the most optimized query. Figure 15-4 shows that to avoid a high optimization overhead compared to the actual execution cost of the query, the optimizer adopts different techniques, namely, the following:

- Simplification
- Trivial plan match
- Multiple optimization phases
- Parallel plan optimization

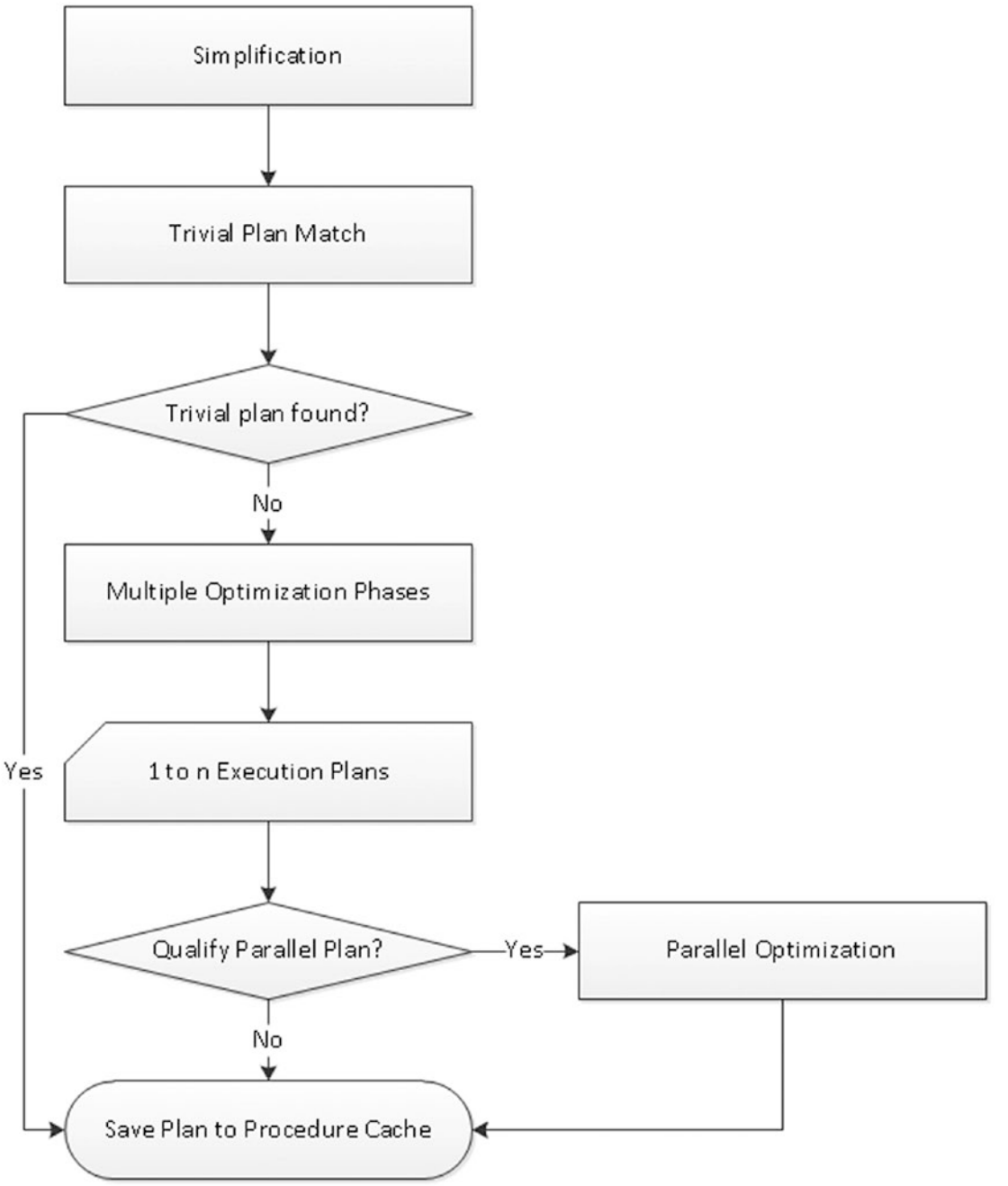


Figure 15-4. *Query optimization steps*

Simplification

Before the optimizer begins to process your query, the logical engine has already identified all the objects referenced in your database. When the optimizer begins to construct your execution plan, it first ensures that all objects being referenced are actually used and necessary to return your data accurately. If you were to write a query with a three-table join but only two of the tables were actually referenced by either the `SELECT` criteria or the `WHERE` clauses, the optimizer may choose to leave the other table out of the processing. This is known as the *simplification* step. It's actually part of a larger set of processing that gathers statistics and starts the process of estimating the cardinality of the data involved in your query. The optimizer also gathers the necessary information about your constraints, especially the foreign key constraints, that will help it later make decisions about the join order, which it can rearrange as needed to arrive at a good enough plan. Also during the Simplification process subqueries get transformed into joins. Other processes of simplification include the removal of redundant joins.

Trivial Plan Match

Sometimes there might be only one way to execute a query. For example, a heap table with no indexes can be accessed in only one way: via a table scan. To avoid the runtime overhead of optimizing such queries, SQL Server maintains a list of patterns that define a trivial plan. If the optimizer finds a match, then a similar plan is generated for the query without any optimization. The generated plans are then stored in the procedure cache. Eliminating the optimization phase means that the cost for generating a trivial plan is very low. This is not to imply that trivial plans are desired or preferable to more complex plans. Trivial plans are available only for extremely simple queries. Once the complexity of the query rises, it must go through optimization.

Multiple Optimization Phases

For a nontrivial query, the number of alternative processing strategies to be analyzed can be high, and it may take a long time to evaluate each option. Therefore, the optimizer goes through three different levels of optimizations. These are referred to as search 0, search 1, and search 2. But it's easier to think of them as *transaction*, *quick plan*, and *full optimization*. Depending on the size and complexity of the query, these different optimizations may be tried one at a time, or the optimizer might skip straight to full optimization. Each of the optimizations takes into account using different join

techniques and different ways of accessing the data through scans, seeks, and other operations.

The index variations consider different indexing aspects, such as single-column index, composite index, index column order, column density, and so forth. Similarly, the join variations consider the different join techniques available in SQL Server: nested loop join, merge join, and hash join. (Chapter 4 covers these join techniques in detail.) Constraints such as unique values and foreign key constraints are also part of the optimization decision-making process.

The optimizer considers the statistics of the columns referred to in the `WHERE`, `JOIN`, and `HAVING` clauses to evaluate the effectiveness of the index and the join strategies. Based on the current statistics, it evaluates the cost of the configurations in multiple optimization phases. The cost includes many factors, including (but not limited to) usage of CPU, memory, and disk I/O (including random versus sequential I/O estimation) required to execute the query. After each optimization phase, the optimizer evaluates the cost of the processing strategy. This cost is an estimation only, not an actual measure or prediction of behavior; it's a mathematical construct based on the statistics and the processes under consideration.

Note The cost estimates are just that, estimates. Further, any one set of estimates represented by an execution plan may or may not in actuality be costlier than another set of estimates, a different execution plan. Comparing the costs between plans can be a dangerous approach.

If the cost is found to be cheap enough, then the optimizer stops further iteration through the optimization phases and quits the optimization process. Otherwise, it keeps iterating through the optimization phases to determine a cost-effective processing strategy.

Sometimes a query can be so complex that the optimizer needs to extensively progress through the optimization phases. While optimizing the query, if it finds that the cost of the processing strategy is more than the cost threshold for parallelism, then it evaluates the cost of processing the query using multiple CPUs. Otherwise, the optimizer proceeds with the serial plan. You may also see that after the optimizer picks a parallel plan, that plan's cost may actually be less than the cost threshold for parallelism and less than the cost of the serial plan.

You can find out some detail of what occurred during the multiple optimization phases via two sources. Take, for example, this query:

```
SELECT soh.SalesOrderNumber,
       sod.OrderQty,
       sod.LineTotal,
       sod.UnitPrice,
       sod.UnitPriceDiscount,
       p.Name AS ProductName,
       p.ProductNumber,
       ps.Name AS ProductSubCategoryName,
       pc.Name AS ProductCategoryName
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
    JOIN Production.Product AS p
        ON sod.ProductID = p.ProductID
    JOIN Production.ProductModel AS pm
        ON p.ProductModelID = pm.ProductModelID
    JOIN Production.ProductSubcategory AS ps
        ON p.ProductSubcategoryID = ps.ProductSubcategoryID
    JOIN Production.ProductCategory AS pc
        ON ps.ProductCategoryID = pc.ProductCategoryID
WHERE soh.CustomerID = 29658;
```

When this query is run, the execution plan in Figure 15-5, a nontrivial plan for sure, is returned.

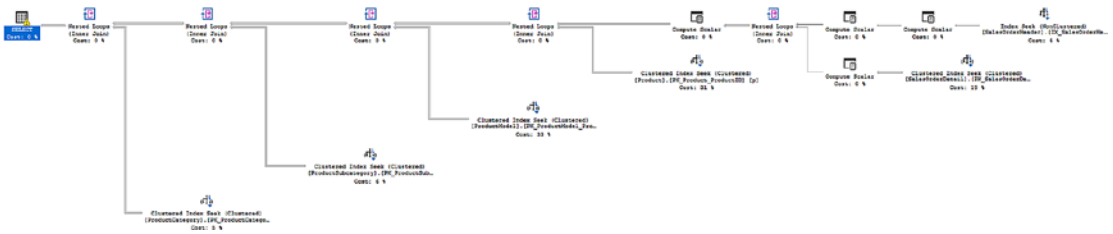
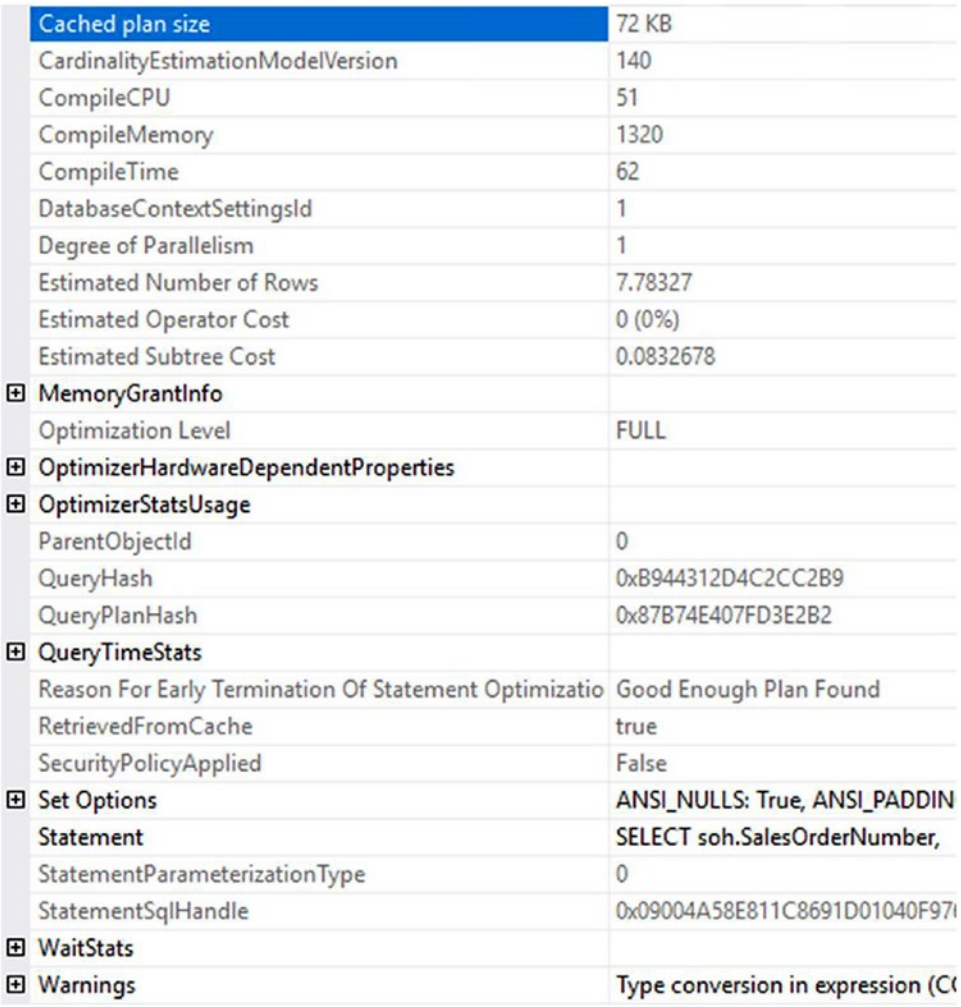


Figure 15-5. *Nontrivial execution plan*

I realize that this execution plan is hard to read. The intent here is not to read through this plan. The important point is that it involves quite a few tables, each with indexes and statistics that all had to be taken into account to arrive at this execution plan. The first place you can go to look for information about the optimizer's work on this execution plan is the property sheet of the first operator, in this case the T-SQL SELECT operator, at the far left of the execution plan. Figure 15-6 shows the property sheet.



Cached plan size	72 KB
CardinalityEstimationModelVersion	140
CompileCPU	51
CompileMemory	1320
CompileTime	62
DatabaseContextSettingsId	1
Degree of Parallelism	1
Estimated Number of Rows	7.78327
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0832678
⊞ MemoryGrantInfo	
Optimization Level	FULL
⊞ OptimizerHardwareDependentProperties	
⊞ OptimizerStatsUsage	
ParentObjectId	0
QueryHash	0xB944312D4C2CC2B9
QueryPlanHash	0x87B74E407FD3E2B2
⊞ QueryTimeStats	
Reason For Early Termination Of Statement Optimizatio	Good Enough Plan Found
RetrievedFromCache	true
SecurityPolicyApplied	False
⊞ Set Options	ANSI_NULLS: True, ANSI_PADDIN
Statement	SELECT soh.SalesOrderNumber,
StatementParameterizationType	0
StatementSqlHandle	0x09004A58E811C8691D01040F97
⊞ WaitStats	
⊞ Warnings	Type conversion in expression (C

Figure 15-6. *SELECT operator property sheet*

Starting at the top, you can see information directly related to the creation and optimization of this execution plan.

- The size of the cached plan, which is 72KB
- The number of CPU cycles used to compile the plan, which is 51ms
- The amount of memory used, which is 1329KB
- The compile time, which is 62ms

The Optimization Level property (`StatementOptmLevel` in the XML plan) shows what type of processing occurred within the optimizer. In this case, `FULL` means that the optimizer did a full optimization. This is further displayed in the property Reason for Early Termination of Statement, which is Good Enough Plan Found. So, the optimizer took 62ms to track down a plan that it deemed good enough in this situation. You can also see the `QueryPlanHash` value, also known as the *fingerprint*, for the execution plan (you can find more details on this in the section "Query Plan Hash and Query Hash"). The properties of the `SELECT` (and the `INSERT`, `UPDATE`, and `DELETE`) operators are an important first stopping point when evaluating any execution plan because of this information.

Added in SQL Server 2017, you can also see the `QueryTimeStats` and `WaitStats` for any actual execution plan that you capture. This can be a useful way to capture query metrics.

The second source for optimizer information is the dynamic management view `sys.dm_exec_query_optimizer_info`. This DMV is an aggregation of the optimization events over time. It won't show the individual optimizations for a given query, but it will track the optimizations performed. This isn't as immediately handy for tuning an individual query, but if you are working on reducing the costs of a workload over time, being able to track this information can help you determine whether your query tuning is making a positive difference, at least in terms of optimization time. Some of the data returned is for internal SQL Server use only. Figure 15-7 shows a truncated example of the useful data returned in the results from the following query:

```
SELECT deqoi.counter,
       deqoi.occurrence,
       deqoi.value
FROM sys.dm_exec_query_optimizer_info AS deqoi;
```

	counter	occurrence	value
1	optimizations	3911	1
2	elapsed time	3905	0.00768245838668374
3	final cost	3905	34.4891834584044
4	trivial plan	611	1
5	tasks	3294	1687.9335154827
6	no plan	0	NULL
7	search 0	1538	1
8	search 0 time	1564	0.0444891304347826
9	search 0 tasks	1564	2743.25959079284
10	search 1	1746	1
11	search 1 time	1760	0.0159017045454545
12	search 1 tasks	1760	588.889204545455

Figure 15-7. *Output from sys.dm_exec_query_optimizer_info*

Running this query before and after another query can show you the changes that have occurred in the number and type of optimizations completed. However, if you can isolate your queries on a test box, you can be more assured that you get before and after differences that are directly related only to the query you're attempting to measure.

Parallel Plan Optimization

The optimizer considers various factors while evaluating the cost of processing a query using a parallel plan. Some of these factors are as follows:

- Number of CPUs available to SQL Server
- SQL Server edition
- Available memory
- Cost threshold for parallelism
- Type of query being executed
- Number of rows to be processed in a given stream
- Number of active concurrent connections

If only one CPU is available to SQL Server, then the optimizer won't consider a parallel plan. The number of CPUs available to SQL Server can be restricted using the *affinity* setting of the SQL Server configuration. The affinity value is set either to specific CPUs or to specific NUMA nodes. You can also set it to ranges. For example, to allow SQL Server to use only CPUO to CPU3 in an eight-way box, execute these statements:

```
USE master;
EXEC sp_configure 'show advanced option','1';
RECONFIGURE;
ALTER SERVER CONFIGURATION SET PROCESS AFFINITY CPU = 0 TO 3;
GO
```

This configuration takes effect immediately. `affinity` is a special setting, and I recommend you use it only in instances where taking control away from SQL Server makes sense, such as when you have multiple instances of SQL Server running on the same machine and you want to isolate them from each other.

Even if multiple CPUs are available to SQL Server, if an individual query is not allowed to use more than one CPU for execution, then the optimizer discards the parallel plan option. The maximum number of CPUs that can be used for a parallel query is governed by the `max degree of parallelism` setting of the SQL Server configuration. The default value is 0, which allows all the CPUs (availed by the `affinity mask` setting) to be used for a parallel query. You can also control parallelism through the Resource Governor.
If you want to allow parallel queries to use no more than two CPUs out of CPUO to CPU3, limited by the preceding `affinity mask` setting, execute the following statements:

```
USE master;
EXEC sp_configure 'show advanced option','1';
RECONFIGURE;
EXEC sp_configure 'max degree of parallelism',2;
RECONFIGURE;
```

This change takes effect immediately, without any restart. The `max degree of parallelism` setting can also be controlled at a query level using the `MAXDOP` query hint.

```
SELECT  *
FROM    dbo.t1
WHERE   C1 = 1
OPTION  (MAXDOP 2);
```

Changing the `max degree of parallelism` setting is best determined by the needs of your application and the workloads on it. I will usually leave the max degree of parallelism set to the default value unless indications arise that suggest a change is necessary. I will usually immediately adjust the cost threshold for parallelism up from its default value of 5. However, it's really important to understand that the cost threshold is set for the server. Picking a single value that is optimal for all databases on the server may be somewhat tricky.

Since parallel queries require more memory, the optimizer determines the amount of memory available before choosing a parallel plan. The amount of memory required increases with the degree of parallelism. If the memory requirement of the parallel plan for a given degree of parallelism cannot be satisfied, then SQL Server decreases the degree of parallelism automatically or completely abandons the parallel plan for the query in the given workload context. You can see this part of the evaluation in the `SELECT` properties of Figure 15-6.

Queries with a very high CPU overhead are the best candidates for a parallel plan. Examples include joining large tables, performing substantial aggregations, and sorting large result sets, all common operations on reporting systems (less so on OLTP systems). For simple queries usually found in transaction-processing applications, the additional coordination required to initialize, synchronize, and terminate a parallel plan outweighs the potential performance benefit.

Whether a query is simple is determined by comparing the estimated execution cost of the query with a cost threshold. This cost threshold is controlled by the `cost threshold for parallelism` setting of the SQL Server configuration. By default, this setting's value is 5, which means that if the estimated execution cost (CPU and IO) of the serial plan is more than 5, then the optimizer considers a parallel plan for the query. For example, to modify the cost threshold to 35, execute the following statements:

```
USE master;
EXEC sp_configure 'show advanced option','1';
RECONFIGURE;
EXEC sp_configure 'cost threshold for parallelism',35;
RECONFIGURE;
```

This change takes effect immediately, without any restart. If only one CPU is available to SQL Server, then this setting is ignored. I've found that OLTP systems suffer when the cost threshold for parallelism is set this low. Usually increasing the value

to somewhere between 30 and 50 will be beneficial. A lower value can be better for analytical queries. Be sure to test this suggestion against your system to ensure it works well for you.

Another option is to simply look at the plans in your cache and then make an estimate, based on the queries there and the type of workload they represent to arrive at a specific number. You can separate your OLTP queries from your reporting queries and then focus on the reporting queries most likely to benefit from parallel execution. Take an average of those costs and set your cost threshold to that number.

The DML action queries (`INSERT`, `UPDATE`, and `DELETE`) are executed serially. However, the `SELECT` portion of an `INSERT` statement and the `WHERE` clause of an `UPDATE` or a `DELETE` statement can be executed in parallel. The actual data changes are applied serially to the database. Also, if the optimizer determines that the estimated cost is too low, it does not introduce parallel operators.

Note that, even at execution time, SQL Server determines whether the current system workload and configuration information allow for parallel query execution. If parallel query execution is allowed, SQL Server determines the optimal number of threads and spreads the execution of the query across those threads. When a query starts a parallel execution, it uses the same number of threads until completion. SQL Server reexamines the optimal number of threads before executing the parallel query the next time.

Once the processing strategy is finalized by using either a serial plan or a parallel plan, the optimizer generates the execution plan for the query. The execution plan contains the detailed processing strategy decided by the optimizer to execute the query. This includes steps such as data retrieval, result set joins, result set ordering, and so on. A detailed explanation of how to analyze the processing steps included in an execution plan is presented in Chapter 4. The execution plan generated for the query is saved in the plan cache for future reuse.

With all that then, we can summarize the process. The optimizer starts by simplifying and normalizing the input tree. From there it generates possible logical trees that are the equivalent of that simplified tree. Then the optimizer transforms the logical trees into possible physical trees, costs them, and selects the cheapest tree. That's the optimization process in a nutshell.

Execution Plan Caching

The execution plan of a query generated by the optimizer is saved in a special part of SQL Server's memory pool called the *plan cache*. Saving the plan in a cache allows SQL Server to avoid running through the whole query optimization process again when the same query is resubmitted. SQL Server supports different techniques such as *plan cache aging* and *plan cache types* to increase the reusability of the cached plans. It also stores two binary values called the *query hash* and the *query plan hash.*

Note I discuss the techniques supported by SQL Server for improving the effectiveness of execution plan reuse in this Chapter 15.

Components of the Execution Plan

The execution plan generated by the optimizer contains two components.

- *Query plan*: This represents the commands that specify all the physical operations required to execute a query.
- *Execution context*: This maintains the variable parts of a query within the context of a given user.

I will cover these components in more detail in the next sections.

Query Plan

The query plan is a reentrant, read-only data structure, with commands that specify all the physical operations required to execute the query. The reentrant property allows the query plan to be accessed concurrently by multiple connections. The physical operations include specifications on which tables and indexes to access, how and in what order they should be accessed, the type of join operations to be performed between multiple tables, and so forth. No user context is stored in the query plan.

Execution Context

The execution context is another data structure that maintains the variable part of the query. Although the server keeps track of the execution plans in the procedure cache, these plans are context neutral. Therefore, each user executing the query will have a separate execution context that holds data specific to their execution, such as parameter values and connection details.

Aging of the Execution Plan

The plan cache is part of SQL Server's buffer cache, which also holds data pages. As new execution plans are added to the plan cache, the size of the plan cache keeps growing, affecting the retention of useful data pages in memory. To avoid this, SQL Server dynamically controls the retention of the execution plans in the plan cache, retaining the frequently used execution plans and discarding plans that are not used for a certain period of time.

SQL Server keeps track of the frequency of an execution plan's reuse by associating an age field to it. When an execution plan is generated, the age field is populated with the cost of generating the plan. A complex query requiring extensive optimization will have an age field value higher than that for a simpler query.

At regular intervals, the current cost of all the execution plans in the plan cache is examined by SQL Server's lazy writer process (which manages most of the background processes in SQL Server). If an execution plan is not reused for a long time, then the current cost will eventually be reduced to 0. The cheaper the execution plan was to generate, the sooner its cost will be reduced to 0. Once an execution plan's cost reaches 0, the plan becomes a candidate for removal from memory. SQL Server removes all plans with a cost of 0 from the plan cache when memory pressure increases to such an extent that there is no longer enough free memory to serve new requests. However, if a system has enough memory and free memory pages are available to serve new requests, execution plans with a cost of 0 can remain in the plan cache for a long time so that they can be reused later, if required.

As well as changing the costs downward, execution plans can also find their costs increased to the max cost of generating the plan every time the plan is reused (or to the current cost of the plan for ad hoc plans). For example, suppose you have two execution plans with generation costs equal to 100 and 10. Their starting cost values will therefore

be 100 and 10, respectively. If both execution plans are reused immediately, their age fields will be set back to that maximum cost. With these cost values, the lazy writer will bring down the cost of the second plan to 0 much earlier than that of the first one, unless the second plan is reused more often. Therefore, even if a costly plan is reused less frequently than a cheaper plan, because of the effect of the initial cost, the costly plan can remain at a nonzero cost value for a longer period of time.

Summary

SQL Server's cost-based query optimizer decides upon an effective execution plan based not on the exact syntax of the query but on evaluating the cost of executing the query using different processing strategies. The cost evaluation of using different processing strategies is done in multiple optimization phases to avoid spending too much time optimizing a query. Then, the execution plans are cached to save the cost of execution plan generation when the same queries are reexecuted.

In the next chapter, I will discuss how the plans get reused from the cache in different ways depending on how they're called.

CHAPTER 16

Execution Plan Cache Behavior

Once all the processing necessary to generate an execution plan has been completed, it would be crazy for SQL Server to throw away that work and do it all again each time a query gets called. Instead, it saves the plans created in a memory space on the server called the *plan cache*. This chapter will walk through how you can monitor the plan cache to see how SQL Server reuses execution plans.

In this chapter, I cover the following topics:

- How to analyze execution plan caching
- Query plan hash and query hash as mechanisms for identifying queries to tune
- Ways to improve the reusability of execution plan caching
- Interactions between the Query Store and the plan cache

Analyzing the Execution Plan Cache

You can obtain a lot of information about the execution plans in the plan cache by accessing various dynamic management objects. The initial DMO for working with execution plans is `sys.dm_exec_cached_plans`.

```
SELECT decp.refcounts,
       decp.usecounts,
       decp.size_in_bytes,
       decp.cacheobjtype,
```


G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_16

```
        decp.objtype,
        decp.plan_handle
FROM sys.dm_exec_cached_plans AS decp;
```

Table 16-1 shows some of the useful information provided by `sys.dm_exec_cached_plans`.

Table 16-1. *sys.dm_exec_cached_plans*

Column Name	Description
`refcounts`	This represents the number of other objects in the cache referencing this plan.
`usecounts`	This is the number of times this object has been used since it was added to the cache.
`size_in_bytes`	This is the size of the plan stored in the cache.
`cacheobjtype`	This specifies what type of plan this is; there are several, but of particular interest are these: Compiled plan: A completed execution plan Compiled plan stub: A marker used for ad hoc queries (you can find more details in the "Ad Hoc Workload" section of this chapter) Parse tree: A plan stored for accessing a view
`Objtype`	This is the type of object that generated the plan. Again, there are several, but these are of particular interest: Proc Prepared Adhoc View

Using the DMV `sys.dm_exec_cached_plans` all by itself gets you only a small part of the information. DMOs are best used in combination with other DMOs and other system views. For example, using the dynamic management function `sys.dm_exec_query_plan(plan_handle)` in combination with `sys.dm_exec_cached_plans` will also bring back the XML execution plan itself so that you can display it and work with it. If you then bring in `sys.dm_exec_sql_text(plan_handle)`, you can also retrieve the

original query text. This may not seem useful while you're running known queries for the examples here, but when you go to your production system and begin to pull in execution plans from the cache, it might be handy to have the original query. To get aggregate performance metrics about the cached plan, you can use `sys.dm_exec_query_stats` for batches, `sys.dm_exec_procedure_stats` for procedures and in-line functions, and `sys.dm_exec_trigger_stats` for returning that same data for triggers. Among other pieces of data, the query hash and query plan hash are stored in this DMF. Finally, to find your way to execution plans for queries that are currently executing, you can use `sys.dm_exec_requests`.

In the following sections, I'll explore how the plan cache works with actual queries of these DMOs.

Execution Plan Reuse

When a query is submitted, SQL Server checks the plan cache for a matching execution plan. If one is not found, then SQL Server performs the query compilation and optimization to generate a new execution plan. However, if the plan exists in the plan cache, it is reused with the private execution context. This saves the CPU cycles that otherwise would have been spent on the plan generation. In the event that a plan is not in the cache but that plan is marked as forced in the Query Store, optimization proceeds as normal, but the forced plan is used instead, assuming it's still a valid plan.

Queries are often submitted to SQL Server with filter criteria to limit the size of the result set. The same queries are often resubmitted with different values for the filter criteria. For example, consider the following query:

```
SELECT soh.SalesOrderNumber,
       soh.OrderDate,
       sod.OrderQty,
       sod.LineTotal
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE soh.CustomerID = 29690
      AND sod.ProductID = 711;
```

When this query is submitted, the optimizer creates an execution plan and saves it in the plan cache to reuse in the future. If this query is resubmitted with a different filter criterion value—for example, `soh.CustomerlD = 29500`—it will be beneficial to reuse the existing execution plan for the previously supplied filter criterion value (unless this is a bad parameter sniffing scenario). Whether the execution plan created for one filter criterion value can be reused for another filter criterion value depends on how the query is submitted to SQL Server.

The queries (or workload) submitted to SQL Server can be broadly classified into two categories that determine whether the execution plan will be reusable as the value of the variable parts of the query changes.

- Ad hoc
- Prepared

Tip To test the output of `sys.dm_exec_cached_plans` for this chapter, it will be necessary to remove the plans from cache on occasion by executing `DBCC FREEPROCCACHE`. Do not run this on your production server except when you use the methods outlined here, passing a plan handle. Otherwise, you will flush the cache and will require all execution plans to be rebuilt as they are executed, placing a serious strain on your production system for no good reason. You can use `DBCC FREEPROCCACHE(plan_handle)` to target specific plans. Retrieve the `plan_handle` using the DMOs I've already talked about and as demonstrated later. You can also flush the cache for a single database using `ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE CACHE`. However, here again, I do not recommend running this on a production server except when you have intention of removing all plans for that database.

Ad Hoc Workload

Queries can be submitted to SQL Server without explicitly isolating the variables from the query. These types of queries executed without explicitly converting the variable parts of the query into parameters are referred to as *ad hoc workloads* (or queries). Most of the examples in the book so far are ad hoc queries, such as the previous listing.

If the query is submitted as is, without explicitly converting either of the hard-coded values to a parameter (that can be supplied to the query when executed), then the query is an ad hoc query. Setting the values to local variables using the `DECLARE` statement is not the same as parameters.

In this query, the filter criterion value is embedded in the query itself and is not explicitly parameterized to isolate it from the query. This means you cannot reuse the execution plan for this query unless you use the same values and all the spacing and carriage returns are identical. However, the places where values are used in the queries can be explicitly parameterized in three different ways that are jointly categorized as a prepared workload.

Prepared Workload

Prepared workloads (or queries) explicitly parameterize the variable parts of the query so that the query plan isn't tied to the value of the variable parts. In SQL Server, queries can be submitted as prepared workloads using the following three methods:

- *Stored procedures*: Allows saving a collection of SQL statements that can accept and return user-supplied parameters.
- *`sp_executesql`*: Allows executing a SQL statement or a SQL batch that may contain user-supplied parameters, without saving the SQL statement or batch.
- *Prepare/execute model*: Allows a SQL client to request the generation of a query plan that can be reused during subsequent executions of the query with different parameter values, without saving the SQL statements in SQL Server. This is the most common practice for ORM tools such as Entity Framework.

For example, the `SELECT` statement shown previously can be explicitly parameterized using a stored procedure as follows:

```
CREATE OR ALTER PROC dbo.BasicSalesInfo
    @ProductID INT,
    @CustomerID INT
```

```
AS
SELECT soh.SalesOrderNumber,
       soh.OrderDate,
       sod.OrderQty,
       sod.LineTotal
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE soh.CustomerID = @CustomerID
      AND sod.ProductID = @ProductID;
```

The plan of the SELECT statement included within the stored procedure will embed the parameters (@ProductID and @Customerld), not variable values. I will cover these methods in more detail shortly.

Plan Reusability of an Ad Hoc Workload

When a query is submitted as an ad hoc workload, SQL Server generates an execution plan and stores that plan in the cache, where it can be reused if the same ad hoc query is resubmitted. Since there are no parameters, the hard-coded values are stored as part of the plan. For a plan to be reused from the cache, the T-SQL must match exactly. This includes all spaces and carriage returns plus any values supplied with the plan. If any of these change, the plan cannot be reused.

To understand this, consider the ad hoc query you've used before, shown here:

```
SELECT  soh.SalesOrderNumber,
        soh.OrderDate,
        sod.OrderQty,
        sod.LineTotal
FROM    Sales.SalesOrderHeader AS soh
JOIN    Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE   soh.CustomerID = 29690
        AND sod.ProductID = 711;
```